```python
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 12 21:03:05 2023

Code for solving grid world problem using Djikstra's algorithm.

@author: Tharun V Puthanveettil
"""
# Importing libraries
import numpy as np
import time
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.animation import FuncAnimation
import cv2



class node:
    """Node class for storing node state, node index and parent node
index.
    """
    def __init__(self, node_state, node_cost, p_node):
        self.node_state = node_state # State of the node.
        self.cost = node_cost # Index of the node.
        self.p_node = p_node # Parent node index.

class Puzzle_Solver:
    """Puzzle_Solver class for solving 8-puzzle problem.
    """
    def __init__(self, start, goal, grid):
        """Constructor for Puzzle_Solver class.

        Args:
            start (numpy array): Initial state of the puzzle.
            goal (numpy array): Goal state of the puzzle.
        """
        self.initial_state =  start
        self.node_state = start # Current state of the puzzle.
        self.explored = {} # Dictionary to store explored nodes.
        self.not_explored = {} # Dictionary to store not explored
nodes.
        #self.index = 0 # Index for nodes.
```

```
        self.current_node = node(self.node_state, 0, None) # member to
store current node.
        self.explored[str(self.node_state)] = self.current_node #
Adding current node to not explored dictionary.
        self.goal_state = goal # Array to store goal state.
        self.goal_found = False # Boolean to store goal found status.
        self.path = None # List to store path from start to goal.
        self.grid = grid

    # def return_blank_tile(self):
    #     """ Function to return position of blank tile.

    #     Returns:
    #         tuple: Position of blank tile.
    #     """
    #     b = np.where(self.current_node.node_state == 0) # Returns
position of blank tile.
    #     return b[0][0],b[1][0]

    def return_current_pos(self):
        """ Function to return current node state.

        Returns:
            numpy array: Current node state.
        """
        return
self.current_node.node_state[0],self.current_node.node_state[1]

    def valid_move(self, move):
        """ Function to check if move is valid.

        Args:
            move (tuple): Move to be checked.
        """
        current_node_pos = self.return_current_pos() # Returns current
position
        next_tile_pos = current_node_pos + move # Returns position of
next tile.
        if next_tile_pos[0] < 0 or next_tile_pos[0] >=
self.grid.shape[0] or next_tile_pos[1] < 0 or next_tile_pos[1] >=
self.grid.shape[1]: # Checks if next tile is within the puzzle.
            return False
        if self.grid[next_tile_pos[0], next_tile_pos[1]] == 1:
```

```
            return False
        return True


    def check_goal(self):
        """ Function to check if goal is reached.

        Returns:
            bool: Goal found status.
        """
        if (self.current_node.node_state == self.goal_state).all():
            self.goal_found = True
            # Add current node to explored.
            self.explored[str(self.current_node.node_state)] =
self.current_node


    def ActionMoveRight(self):
        """Function to move blank tile right.

        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile right, if possible
        cost = 1
        current_node_pos = self.return_current_pos() # Returns position
of blank tile.
        move = np.array([0, 1]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position
of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos,
next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and
state of next node.
        return False, None # Returns status of move and state of next
node.

    def ActionMoveLeft(self):
        """function to move blank tile left.

        Returns:
```

```
                tuple: Status of move and state of next node.
        """
        # Moves blank tile left, if possible
        cost = 1
        current_node_pos = self.return_current_pos() # Returns position of blank tile.
        move = np.array([0, -1]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos, next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and state of next node.
        return False, None

    def ActionMoveUp(self):
        """Function to move blank tile up.

        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile up, if possible
        cost = 1
        current_node_pos = self.return_current_pos() # Returns position of blank tile.
        move = np.array([-1, 0]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move    # Returns position of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos, next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and state of next node.
        return False, None # Returns status of move and state of next node.

    def ActionMoveDown(self):
        """Function to move blank tile down.

        Returns:
```

```
            tuple: Status of move and state of next node.
        """
        # Moves blank tile down, if possible
        cost = 1
        current_node_pos = self.return_current_pos() # Returns position
of blank tile.
        move = np.array([1, 0]) # Move to be checked.
        if self.valid_move(move):   # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position
of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos,
next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and
state of next node.
        return False, None


    def ActionMoveTopRight(self):
        """Function to move blank tile top right.

        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile top right, if possible
        cost = 1.4
        current_node_pos = self.return_current_pos() # Returns position
of blank tile.
        move = np.array([-1, 1]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position
of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos,
next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and
state of next node.
        return False, None # Returns status of move and state of next
node.

    def ActionMoveTopLeft(self):
        """Function to move blank tile top left.
```

```
        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile top left, if possible
        cost = 1.4
        current_node_pos = self.return_current_pos() # Returns position of blank tile.
        move = np.array([-1, -1]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos, next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and state of next node.
        return False, None # Returns status of move and state of next node.

    def ActionMoveBottomRight(self):
        """Function to move blank tile bottom right.

        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile bottom right, if possible
        cost = 1.4
        current_node_pos = self.return_current_pos() # Returns position of blank tile.
        move = np.array([1, 1]) # Move to be checked.
        if self.valid_move(move): # Checks if move is valid.
            next_tile_pos = current_node_pos + move # Returns position of next tile.
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos, next_tile_pos) # Returns state of next node.
            return True, NextNode_state # Returns status of move and state of next node.
        return False, None # Returns status of move and state of next node.

    def ActionMoveBottomLeft(self):
        """Function to move blank tile bottom left.
```

```
        Returns:
            tuple: Status of move and state of next node.
        """
        # Moves blank tile bottom left, if possible
        cost = 1.4
        current_node_pos = self.return_current_pos()
        move = np.array([1, -1])
        if self.valid_move(move):
            next_tile_pos = current_node_pos + move
            NextNode_state = [next_tile_pos, cost]
            #NextNode_state = self.swap_blank_tile(blank_tile_pos,
next_tile_pos)
            return True, NextNode_state
        return False, None


    def generate_potential_moves(self):
        """Function to generate potential moves for blank tile.

        Returns:
            list: List of potential moves.
        """
        # Generates potential moves for blank tile.
        potential_nodes_states = [] # List of potential moves.

        status, NextNode_state = self.ActionMoveTopRight() # Returns
status of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveTopLeft() # Returns
status of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveBottomRight() # Returns
status of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.
```

```
        status, NextNode_state = self.ActionMoveBottomLeft() # Returns
status of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveLeft() # Returns status
of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveRight() # Returns
status of move and state of next node.
        if status:  # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveUp() # Returns status
of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.

        status, NextNode_state = self.ActionMoveDown() # Returns status
of move and state of next node.
        if status: # Checks if move is valid.
            potential_nodes_states.append(NextNode_state) # Appends
state of next node to list of potential moves.
        return potential_nodes_states # Returns list of potential
moves.

        return potential_nodes_states # Returns list of potential
moves.


    def string_to_array(self, node):
        """Function to convert string to array.

        Args:
            node (string): String to be converted.
        Returns:
```

```
            numpy array: Array of node.
        """
        array = np.array([[int(node[2]), int(node[4]), int(node[6])],
                          [int(node[11]), int(node[13]),
int(node[15])],
                          [int(node[20]), int(node[22]),
int(node[24])]])
        return array



    def get_possible_moves(self):
        """Function to get possible moves.
        """
        potential_nodes_states = self.generate_potential_moves() #
Generates potential moves for blank tile.
        #print("Potential Nodes States: ", potential_nodes_states)
        for node_state in potential_nodes_states: # Iterates through
potential node states.
            # Check if node has been explored or not explored.
            if str(node_state[0]) not in self.explored:
                if str(node_state[0]) not in self.not_explored:
                        new_node = node(node_state[0],
self.current_node.cost + node_state[1], self.current_node)
                        self.not_explored[str(node_state[0])] =
new_node
            else:
                if  self.explored[str(node_state[0])].cost >
self.current_node.cost + node_state[1]:
                    self.explored[str(node_state[0])].cost =
self.current_node.cost + node_state[1]
                    self.explored[str(node_state[0])].p_node =
self.current_node



        ## Add current node to explored.
        self.explored[str(self.current_node.node_state)] =
self.current_node



    def explore_next_move(self):
```

```
        """Function to explore next move.
        """
        try:
            # Determine next move to explore.
            sorted_not_explored = sorted(self.not_explored.items(),
key=lambda x:x[1].cost) # Sort not explored by node index.

            # Determine the next node to explore.
            next_node = sorted_not_explored[0][1] # Get next node.
        except:
            print("No path found. Target state not reachable.")
            quit()
        self.current_node = next_node # Set current node to next node.
        self.check_goal() # Check if goal has been found.
        # # # Remove current node from not_explored.
        self.not_explored.pop(str(self.current_node.node_state))


    def generate_path(self):
        """Function to generate path from goal to initial state.
        """
        # Generate path from goal to initial state.
        path = []
        current_node = self.current_node
        while(current_node.p_node != None):
            path.append(current_node.node_state)
            current_node = current_node.p_node
        path.append(self.initial_state)
        self.path = path


    def plot_path(self, type = 'path'):
        """Function to plot path.
        """
        # Plot path.
        if type == 'path':
            path = np.array(self.path)
            plt.plot(path[:,1], path[:,0], 'r')
        elif type == 'nodes':
            for i in self.explored.values():
                plt.scatter(i.node_state[1], i.node_state[0], c = 'b')
                plt.pause(0.01)
        elif type == 'grid':
```

```
            plt.imshow(grid, cmap = 'gray')


def grid_builder(height, width, clearance = 5):
    grid = np.zeros((height, width), dtype=np.uint8)

    #Note: (0,0) is top left corner of image hence the x and y index
are reversed as j and i respectively.

    # Triangle - Puffed
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            if ((i <= -2*j + 1145 - clearance) and (i >= 2*j-895 +
clearance) and (j >= 460 - clearance) and (j <= 510 + clearance) and (i
>= 25 - clearance) and (i <= 225 + clearance)):
                # print(i, j)
                grid[i, j] = 1

    # # Triangle - Original
    # for i in range(grid.shape[0]):
    #     for j in range(grid.shape[1]):
    #         if ((i <= -2*j + 1145) and (i >= 2*j-895) and (j >= 460)
and (j <= 510) and (i >= 25) and (i <= 225)):
    #             print(i, j)
    #             grid[i, j] = 0

    # Rectangle Bottom - Puffed
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            if ((j >= 100-clearance) and (j <= 150+clearance) and (i >=
0) and (i <= 100 + clearance)):
                # print(i, j)
                grid[i, j] = 1


    # # Rectangle Bottom - Original
    # for i in range(grid.shape[0]):
    #     for j in range(grid.shape[1]):
    #         if ((j >= 100) and (j <= 150) and (i >= 0) and (i <=
100)):
    #             print(i, j)
    #             grid[i, j] = 1
```

```
    # Rectangle Top - Puffed
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            if ((j >= 100 - clearance) and (j <= 150 + clearance) and
(i >= 150-clearance) and (i <= 250)):
                # print(i, j)
                grid[i, j] = 1

    # # Rectangle Top - Original
    # for i in range(grid.shape[0]):
    #     for j in range(grid.shape[1]):
    #         if ((j >= 100) and (j <= 150) and (i >= 150) and (i <=
250)):
    #             print(i, j)
    #             grid[i, j] = 1

    # Hexagon - Puffed
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            if (j >= (235 - clearance)) and (j <= (365 + clearance))
and ((j + 2*i) >= 395) and ((j - 2*i) <= 205) and ((j - 2*i) >= -105)
and ((j + 2*i) <= 705):
                # print(i, j)
                grid[i, j]=1

    # # Hexagon - original
    # for i in range(grid.shape[0]):
    #     for j in range(grid.shape[1]):
    #         if ((i <= -0.58*j + 373) and (j <= 365) and (i >=
0.58*j-123.08) and (i >= -0.58*j+223.08) and (j >= 235) and (i <=
0.58*j+27.38)):
    #             print(i, j)
    #             grid[i, j]=1

    # # Wall - Puffed
    # # Draw a line from (0,0) to (0,249) with a thickness of 5 pixels
    grid = cv2.line(grid, (0,0), (0,249), 1, 8) # Given 8 as thickness
to make it 5 pixel thick due to resolution
    # # Draw a line from (0,0) to (599,0) with a thickness of 5 pixels
    grid = cv2.line(grid, (0,0), (599,0), 1, 8) # Given 8 as thickness
to make it 5 pixel thick due to resolution
```

```
    # # Draw a line from (599,0) to (599,249) with a thickness of 1
pixel
    grid = cv2.line(grid, (599,0), (599,249), 1, 8) # Given 8 as
thickness to make it 5 pixel thick due to resolution
    # # Draw a line from (0,249 to (600,249) with a thickness of 5
pixels
    grid = cv2.line(grid, (0,249), (599,249), 1, 8)  # Given 8 as
thickness to make it 5 pixel thick due to resolution

    return grid




if __name__ == "__main__":
    # Define the initial and goal state

########################################################################
##########
    # Github Repository Link:
https://github.com/tvpian/ENPM661---Project-2/tree/main

########################################################################
##########

    # Test Case #1
    #initial_state = np.array([6,6])
    #goal_state = np.array([200, 200])

    val = input("Enter the starting position as x,y [eg: for [0,0] -
0,0]: ")
    x = int(val.split(',')[0])
    y = int(val.split(',')[1])
    initial_state = np.array([x,y])


    val = input("Enter the final position as x,y [eg: for [0,0] - 0,0]:
")
    x = int(val.split(',')[0])
    y = int(val.split(',')[1])
    goal_state = np.array([x,y])
```

```
    height = 250 # Height of the grid
    width = 600 # Width of the grid
    grid = grid_builder(height, width, clearance = 5)

    # Visualize the grid
    plt.imshow(grid, cmap = "gray")
    plt.show()

    # Get the coordinates of the obstacles
    obstacle_space = np.where(grid == 1)


    steps = 0 # Counter for number of steps
    if(str(initial_state) != str(goal_state)):
        solver = Puzzle_Solver(initial_state, goal_state, grid) #
Create instance of Puzzle_Solver class
        start = time.time() # Start the timer
        while(True):
            steps = steps+1
            print("No. of steps: ",steps)
            test = solver.get_possible_moves() # Get possible moves
            solver.explore_next_move() # Explore next move
            if(solver.goal_found):  # Check if goal has been found
                print("Goal Found in ", steps, " steps") # Print number
of steps
                print("Initial State: ", solver.initial_state) # Print
initial state
                print("Goal State: ", solver.goal_state) # Print goal
state
                break
        print("Performing Backtracking to find path....")
        solver.generate_path() # Generate path from intial to goal
state
        print("Done")
        print("Path: ", solver.path)
        # print("Writing to file....")
        # solver.write_node_nodes_file() # Write nodes to file
        # solver.write_node_nodesinfo_file() # Write nodes info to file
        # solver.write_node_nodespath_file() # Write nodes path to file
        # print("Done")
        end = time.time() # Stop the timer
        print("Time taken for finding path: ", end - start, " seconds")
# Print time taken
```

```
        start = time.time() # Start the timer
        print("Recording video....Please wait until said done")
        result = cv2.VideoWriter('Final.avi',
                                    cv2.VideoWriter_fourcc(*'MJPG'),
                                    30, (grid.shape[1], grid.shape[0])) #
Create a video writer object

        nodes = solver.explored.values()
        path = solver.path # Get the path
        path = path[::-1] # Reverse the path
        nodes = [i.node_state for i in nodes] # Get the nodes
        frame = np.zeros((grid.shape[0], grid.shape[1], 3), dtype =
np.uint8) # Create a frame

        # print(obstacle_space)

        x,y = obstacle_space
        for i in range(len(x)):
            frame[x[i], y[i]] = [0,0,255]


        cv2.imshow("Obstacle Space", frame)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        for i in nodes:
            frame[grid.shape[0]-i[0], i[1]] = [0,255,0]
            # cv2.imshow("Graph Exploration", frame)
            # cv2.waitKey(1)
            result.write(frame)

        # cv2.imshow("Frame", frame)
        # cv2.waitKey(0)
        # cv2.destroyAllWindows()

        for i in path:
            frame[grid.shape[0]-i[0], i[1]] = [255,0,0]
            cv2.imshow("Backtracing", frame)
            cv2.waitKey(3)
            result.write(frame)
```

```
        # cv2.imshow("Frame", frame)
        # cv2.waitKey(0)
        cv2.destroyAllWindows()
        result.release()
        end = time.time() # Stop the timer
        print("Path planning and video recording complete")
        print("Time taken for recording video: ", end - start, "
seconds") # Print time taken
    else:
        print("Goal Found as Initial State equals Goal State")
```