```python
# -*- coding: utf-8 -*-

"""
Created on Tue Mar 21 16:05:08 2023

@author: caitlin.p.conn

"""
# Required imported libraries
# Additional packages were installed, tried to capture requirements in readme file
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
import time
import math
import cv2
import imutils

# Function finds the third point of a triangle given two points
# Resource: https://stackoverflow.com/questions/69671976/python-function-to-find-a-point-of-an-
equilateral-triangle
def find_equal_triangle_coordinate(pt1, pt2):

    pt3_x = (pt1[0] + pt2[0] + np.sqrt(3) * (pt1[1] - pt2[1])) / 2
    pt3_y = (pt1[1] + pt2[1] + np.sqrt(3) * (pt1[0] - pt2[0])) / 2

    unknown_pt = [pt3_x, pt3_y]

    return np.array(unknown_pt)

# Function computes the a,b,c constants of the line passing between two points
# Resource: https://www.geeksforgeeks.org/program-find-line-passing-2-points/
def compute_line_abc(pt_a, pt_b):

    a_val = pt_b[1] - pt_a[1]
    b_val = pt_a[0] - pt_b[0]
    c_val = (a_val*(pt_a[0])) + (b_val*(pt_a[1]))

    return a_val, b_val, c_val

# Function creates the map grid image with the appriopriate obstacle and freespace boundaries
# Resource: quora.com/How-do-you-find-the-distance-between-the-edges-and-the-center-of-a-regular-
hexagon-if-you-know-the-length-of-its-sides
def create_map_grid(obstacle_space_color, free_space_color):

    # Define map grid shape
    map_height = 250
    map_width = 600
    map_grid = np.ones((map_height,map_width,3), dtype = np.uint8)

    # Define obstacle and wall color
    obstacle_color = obstacle_space_color
    # Define obstacle clearance color
    clearance_color = free_space_color

    c = 5 # clearance value in pixels

    #################################################################
    # Compute hexagon logic

    hexagon_x_center = 300 # 100 + 50 + 150
    hexagon_y_center = 125
    hex_edge_length = 75
```

```
    hex_dist_center_to_edge = hex_edge_length * math.sqrt(3) / 2

    # Hexagon Vertex 1 - Top
    v1_x = int(100 + 50 + 150)
    v1_y = int(125 + hex_dist_center_to_edge)

    vertex1 = [hexagon_x_center, hexagon_y_center]
    vertex2 = [v1_x,v1_y]
    result = find_equal_triangle_coordinate(vertex1, vertex2)

    # Hexagon Center Coordinate
    # map_grid = cv2.circle(map_grid, (hexagon_x_center,hexagon_y_center), radius=5, color=(255,0,0),
thickness=-1)

    # Hexagon Vertex 2
    v2_x = 100 + 50 + 150 + hex_dist_center_to_edge
    v2_y = int(result[1])

    # Hexagon Vertex 6
    v6_x = v1_x - hex_dist_center_to_edge
    v6_y = int(result[1])

    # Hexagon Vertex 3
    v3_x = int(v2_x)
    v3_y = int(result[1]) - hex_edge_length

    # Hexagon Vertex 4
    v4_x = int(v1_x)
    v4_y = int(125 - hex_dist_center_to_edge)

    # Hexagon Vertex 5
    v5_x = int(v6_x)
    v5_y = int(result[1])-hex_edge_length

    pt1 = [v1_x,v1_y+c]
    pt2 = [v2_x+c,v2_y+c]
    pt3 = [v3_x+c,v3_y-c]
    pt4 = [v4_x,v4_y-c]
    pt5 = [v5_x-c,v5_y-c]
    pt6 = [v6_x-c,v6_y+c]

    l1a, l1b, l1c = compute_line_abc(pt1, pt2)
    l2a, l2b, l2c = compute_line_abc(pt2, pt3)
    l3a, l3b, l3c = compute_line_abc(pt3, pt4)
    l4a, l4b, l4c = compute_line_abc(pt4, pt5)
    l5a, l5b, l5c = compute_line_abc(pt5, pt6)
    l6a, l6b, l6c = compute_line_abc(pt6, pt1)

    pt1_i = [v1_x,v1_y]
    pt2_i = [v2_x,v2_y]
    pt3_i = [v3_x,v3_y]
    pt4_i = [v4_x,v4_y]
    pt5_i = [v5_x,v5_y]
    pt6_i = [v6_x,v6_y]

    l1a_i, l1b_i, l1c_i = compute_line_abc(pt1_i, pt2_i)
    l2a_i, l2b_i, l2c_i = compute_line_abc(pt2_i, pt3_i)
    l3a_i, l3b_i, l3c_i = compute_line_abc(pt3_i, pt4_i)
    l4a_i, l4b_i, l4c_i = compute_line_abc(pt4_i, pt5_i)
    l5a_i, l5b_i, l5c_i = compute_line_abc(pt5_i, pt6_i)
    l6a_i, l6b_i, l6c_i = compute_line_abc(pt6_i, pt1_i)

    #############################################################
    # Compute triangle logic
```

```
    tri_low_pt = [460,25]
    tri_up_pt = [460,225]
    tri_right_pt = [510,125]

    t1a, t1b, t1c = compute_line_abc(tri_low_pt, tri_up_pt)
    t2a, t2b, t2c = compute_line_abc(tri_low_pt, tri_right_pt)
    t3a, t3b, t3c = compute_line_abc(tri_up_pt, tri_right_pt)

    obstacle_triangle_coordinates = [tri_low_pt, tri_up_pt, tri_right_pt]

    center_pt_triangle_x = (tri_low_pt[0] + tri_up_pt[0] + tri_right_pt[0]) / 3
    center_pt_triangle_y = (tri_low_pt[1] + tri_up_pt[1] + tri_right_pt[1]) / 3
    center_pt_triangle = [center_pt_triangle_x, center_pt_triangle_y]

    outer_triangle_pts = []

    for vertex_coord in obstacle_triangle_coordinates:

        coord_shifted = np.array(vertex_coord) - np.array(center_pt_triangle)
        computed_length = np.sqrt(coord_shifted.dot(coord_shifted))
        norm_vec = coord_shifted / computed_length if computed_length != 0 else 
np.zeros_like(coord_shifted)

        scaled_pt = (c * norm_vec) + vertex_coord

        outer_triangle_pts.append(scaled_pt)

    tri_low_pt = outer_triangle_pts[0]
    tri_up_pt = outer_triangle_pts[1]
    tri_right_pt = outer_triangle_pts[2]

    t1aa, t1bb, t1cc = compute_line_abc(tri_low_pt, tri_up_pt)
    t2aa, t2bb, t2cc = compute_line_abc(tri_low_pt, tri_right_pt)
    t3aa, t3bb, t3cc = compute_line_abc(tri_up_pt, tri_right_pt)

    #############################################################
    # Change image pixels to reflect map boundaries

    for y in range(map_height):
        for x in range(map_width):

            # Plot horizontal walls clearance
            if (x >= 0 and x < map_width and y >= 5 and y < 10) or (x >= 0 and x < map_width and y >= 
240  and y < 245):
                map_grid[y,x] = clearance_color

            # Plot horizontal walls
            if (x >= 0 and x < map_width and y >= 0 and y < 5) or (x >= 0 and x < map_width and y >= 
245  and y < map_height):
                map_grid[y,x] = obstacle_color

            # Plot vertical walls clearance
            if (x >= 5 and x < 10 and y >= 0 and y < map_height) or (x >= 590 and x < 595 and y >= 0 
and y < map_height):
                map_grid[y,x] = clearance_color

            # Plot vertical walls
            if (x >= 0 and x < 5 and y >= 0 and y < map_height) or (x >= 595 and x < map_width and y 
>= 0 and y < map_height):
                map_grid[y,x] = obstacle_color

            # Display rectangles
            # Plot lower rectange obstacle space
            if x >= 100 - c and x < 150 + c and y >= 0 - c and y <= 100 + c:
                map_grid[y,x] = clearance_color
```

```
            # Plot lower rectange clearance
            if x >= 100 and x <= 150 and y >= 0 and y <= 100:
                map_grid[y,x] = obstacle_color

            # Plot upper rectange clearance
            if x >= 100 - c and x <= 150 + c and y >= 150 - c and y <= 250 + c:
                map_grid[y,x] = clearance_color

            # Plot upper rectange obstacle space
            if x >= 100 and x <= 150 and y >= 150 and y <= 250:
                map_grid[y,x] = obstacle_color

            # Display hexagon
            if ( ((l1b*y)+(l1a*x)-l1c) >= 0  and ((l2b*y)+(l2a*x)-l2c) >= 0) and ((l3b*y)+(l3a*x)-
l3c) >= 0 and ((l4b*y)+(l4a*x)-l4c) >= 0 and ((l5b*y)+(l5a*x)-l5c) >= 0 and ((l6b*y)+(l6a*x)-l6c) >=
0:
                map_grid[y,x] = clearance_color

            if ( ((l1b_i*y)+(l1a_i*x)-l1c_i) >= 0  and ((l2b_i*y)+(l2a_i*x)-l2c_i) >= 0) and
((l3b_i*y)+(l3a_i*x)-l3c_i) >= 0 and ((l4b_i*y)+(l4a_i*x)-l4c_i) >= 0 and ((l5b_i*y)+(l5a_i*x)-l5c_i)
>= 0 and ((l6b_i*y)+(l6a_i*x)-l6c_i) >= 0:
                map_grid[y,x] = obstacle_color

            # Display triangle clearance
            if ( ((t1bb*y)+(t1aa*x)-(t1cc-0)) >= 0  and ((t2bb*y)+(t2aa*x)-(t2cc+0)) <= 0 and
((t3bb*y)+(t3aa*x)-(t3cc-0)) >= 0):
                map_grid[y,x] = clearance_color

            # Display triangle obstacle space
            if ( ((t1b*y)+(t1a*x)-t1c) >= 0  and ((t2b*y)+(t2a*x)-t2c) <= 0 and ((t3b*y)+(t3a*x)-t3c)
>= 0):
                map_grid[y,x] = obstacle_color

    return map_grid, map_height, map_width

#####################################################################################################
######################################
# Function checks if node is in the defined obstacle space
def check_node_in_obstacle_space(child_node_x, child_node_y, obstacle_matrix):

    return obstacle_matrix[child_node_y][child_node_x] == -1

#####################################################################################################
######################################
# Function determines the validity of the action to produce the child node and checks if the
resulting action is in the obstacle space
def generate_child_node(obstacle_matrix, map_boundary_matrix, parent_node, action, map_grid,
map_height, map_width):

    valid_move = False # boolean truth value of valid swap
    parent_cost_to_come = parent_node[1][0]
    parent_node_x = parent_node[0][0]
    parent_node_y = parent_node[0][1]
    child_node_x = 0
    child_node_y = 0

    is_node_obstacle = False # boolean to check if node is in obstacle space

    # Action logic
    if action == 1: #left (-1,0)
        cost_to_move = 1
        if parent_node_x != 0:
            child_node_x = parent_node_x - 1
            child_node_y = parent_node_y
```

```
    elif action == 2: #up (0,1)
        cost_to_move = 1
        if parent_node_y != map_height - 1:
            child_node_x = parent_node_x
            child_node_y = parent_node_y + 1

    elif action == 3: # right (1,0)
        cost_to_move = 1
        if parent_node_x != map_width - 1:
            child_node_x = parent_node_x + 1
            child_node_y = parent_node_y

    elif action == 4: # down (0,-1)
        cost_to_move = 1
        if parent_node_y != 0:
            child_node_x = parent_node_x
            child_node_y = parent_node_y - 1

    elif action == 5: # right & up (1,1)
        cost_to_move = 1.4
        if parent_node_x != map_width - 1 and parent_node_y != map_height - 1:
            child_node_x = parent_node_x + 1
            child_node_y = parent_node_y + 1

    elif action == 6: # left & up (-1,1)
        cost_to_move = 1.4
        if parent_node_x != 0 and parent_node_y != map_height - 1:
            child_node_x = parent_node_x - 1
            child_node_y = parent_node_y + 1

    elif action == 7: # right & down (1,-1)
        cost_to_move = 1.4
        if parent_node_x != map_width - 1 and parent_node_y != 0:
            child_node_x = parent_node_x + 1
            child_node_y = parent_node_y - 1

    elif action == 8: # left & down (-1,-1)
        cost_to_move = 1.4
        if parent_node_x != 0 and parent_node_y != 0:
            child_node_x = parent_node_x - 1
            child_node_y = parent_node_y - 1

    is_node_obstacle = check_node_in_obstacle_space(child_node_x, child_node_y, obstacle_matrix)

    valid_move = not is_node_obstacle

    # Compute child node's cost to come value
    cost_to_move = round(cost_to_move + parent_cost_to_come,1)

    # returned node is the resulting child node of the requested action
    return cost_to_move, valid_move, child_node_x, child_node_y, is_node_obstacle

####################################################################################################
######################################
# Function takes the visited queue as an input and computes the optimal path from the start to end
goal

def compute_optimal_path(visited_queue, initial_node_coord, goal_node_coord):
    path_list = [] # list to store coordinates of optimal path

    first_parent_coord = visited_queue[goal_node_coord][3] # set first parent to search for equal to
the goal node's parent node
    curr_elem_x = goal_node_coord[0] # get x coordinate of the goal node, first node added to the
path list
```

```
    curr_elem_y = goal_node_coord[1] # get y coordinate of the goal node, first node added to the
path list

    path_list.append(goal_node_coord) # add goal node to the path list
    parent_coord = first_parent_coord # set variavle equal to current node's coordinates

    # Continue while loop logic until the initial node is reached
    while(not((curr_elem_x == initial_node_coord[0]) and (curr_elem_y == initial_node_coord[1]))):
        for visited_elem in visited_queue: # loop through each node in the visited queue;
visited_elem returns just the nodes coordinates (x,y)
            curr_elem_x = visited_elem[0] # current node's x coordinate
            curr_elem_y = visited_elem[1] # current node's y coordinate
            curr_coord = (curr_elem_x, curr_elem_y) # store coordinate as variable
            if curr_coord == parent_coord: # check if the current node is the node being searched for
                path_list.append(visited_elem) # add the current element to the path list
                parent_coord = visited_queue[visited_elem][3] # search for the current node's parent
node
                break

    # Debug Statements
    # for elem in visited_queue:
    #     print("Visited Queue Current Element: ", elem)
    # print()

    # for p in path_list:
    #     print("Path List Current Element: ", p)
    # print()

    path_list = np.flipud(path_list)

    return path_list

#####################################################################################################
######################################
# Function keeps track of obstacle and free space boundaries of the map
# Matrix is only created or edited once at the beginning of the code file

def map_obstacle_freespace_matrix(map_grid, map_height, map_width):

    # Create boolean arrays to represent the various regions of the map
    free_space = np.mean(map_grid, axis=-1) == 1
    obstacle_space = np.logical_not(free_space)

    # Create map_boundary_matrix using the boolean arrays
    map_boundary_matrix = np.zeros((map_height, map_width))
    map_boundary_matrix[free_space] = np.inf
    map_boundary_matrix[obstacle_space] = round(-1,1)

    # Set the starting point to 0
    map_boundary_matrix[0, 0] = round(0,1)

    return map_boundary_matrix

#####################################################################################################
######################################
# Debugging functions

# Display current state of the map grid to the IDE
def display_map_grid_plot(map_grid, x, y, point_thickness, goal_found, goal_x, goal_y, curr_x,
curr_y):

    plt.figure()
    plt.title('Map State')
    plt.imshow(map_grid.astype(np.uint8), origin="lower")
    plt.show()
```

```
    return

# Function prints debugging statements to the terminal
def print_function(i, valid_move, is_node_obstacle, plot_fig, map_grid):

    # i identifies the move action that results in a child node
    if i == 1:
        print("Action Left (-1,0)")
    elif i == 2:
        print("Action Up (0,1)")
    elif i == 3:
        print("Action Right (1,0)")
    elif i == 4:
        print("Action Down (0,-1)")
    elif i == 5:
        print("Action Right & Up (1,1)")
    elif i == 6:
        print("Action Left & Up (-1,1)")
    elif i == 7:
        print("Action Right & Down (1,-1)")
    elif i == 8:
        print("Action Left & Down (-1,-1)")

    print("Is Valid Move Boolean -> ? : ", valid_move)

    print("is_node_obstacle: ", is_node_obstacle)

    print()

    if plot_fig == True:
        plt.figure()
        plt.title('Map State')
        plt.imshow(map_grid.astype(np.uint8), origin="lower")
        plt.show()

    return

####################################################################################################
######################################
# Function returns node with the lowest cost to come in the visited queue

def get_node_lowest_cost_to_come(open_list):

    node, min_c2c = min(open_list.items(), key=lambda x: x[1][0])
    return node

####################################################################################################
######################################
# Function computes final logic if the goal node has been found, goal node is added to the visited
queue

def goal_node_found(goal_found, visited_queue, child_node_x_valid, child_node_y_valid, cost_to_move,
node_idx, curr_parent_idx, curr_node_coord):
    goal_found = True

    node_idx = node_idx + 1

    child_node = ((child_node_x_valid, child_node_y_valid),(cost_to_move, node_idx, curr_parent_idx,
curr_node_coord))

    visited_queue[(child_node_x_valid, child_node_y_valid)] = (cost_to_move, node_idx,
curr_parent_idx, curr_node_coord)

    print("Last Child Node (Goal Node): \n", child_node)
```

```
    print()
    print("Problem solved, now backtrack to find optimal path!")
    print()
    print("_______________________________________________________________________________________")
    print()

    return visited_queue, goal_found

####################################################################################################
######################################
# Function gets the user defined input to define the initial and goal nodes

def get_user_input(map_width, map_height, check_node_in_obstacle_space, obstacle_matrix):

    # Get user defined initial node
    while True:
        x_initial = eval(input("Enter start node's x coordinate. x coordinate can range from 0 to " +
str(map_width - 1) + ": "))
        y_initial = eval(input("Enter start node's y coordinate. y coordinate can range from 0 to " +
str(map_height - 1) + ": "))

        if not(0 <= x_initial <= map_width - 1) or (not(0 <= y_initial <= map_height - 1)):
            print("Re-enter initial coordinates, coordinates not within bounds.")
            print()

        else:
            print("Start node x-coordinate:", x_initial)
            print("Start node y-coordinate:", y_initial)
            print()

            is_initial_obstacle = check_node_in_obstacle_space(x_initial, y_initial, obstacle_matrix)
            if (is_initial_obstacle == True):
                print("Re-enter initial node, coordinates are within bounds but are not within
freespace.")
            else:
                break

    # Get user defined goal node
    while True:
        x_goal = eval(input("Enter goal node's x coordinate. x coordinate can range from 0 to " +
str(map_width - 1) + ": "))
        y_goal = eval(input("Enter goal node's y coordinate. y coordinate can range from 0 to " +
str(map_height - 1) + ": "))

        if not(0 <= x_goal <= map_width - 1) or (not(0 <= y_goal <= map_height - 1)):
            print("Re-enter goal coordinates, coordinates not within bounds.")
            print()

        else:
            print("Goal node x-coordinate:", x_goal)
            print("Goal node y-coordinate:", y_goal)
            print()

            is_goal_obstacle = check_node_in_obstacle_space(x_goal, y_goal, obstacle_matrix)
            if (is_goal_obstacle == True):
                print("Re-enter goal node, coordinates are within bounds but are not within
freespace.")
            else:
                break

    print("_______________________________________________________________________________________")
    print()

    return x_initial, y_initial, x_goal, y_goal
```

```
########################################################################################################
######################################
# Function uses backtracking logic to find the traversal pathway from the initial node to goal node
# Function that calls subfunctions to perform search operations
# Resource: https://numpy.org/doc/stable/reference/generated/numpy.fliplr.html
# Must use flipud function to ensure using a forward search strategy!!

def dijkstra_approach_alg(obstacle_matrix, map_boundary_matrix, initial_node_coord, goal_node_coord,
map_grid, map_height, map_width):

    fig, ax = plt.subplots() # keeps track of figure and axis for map grid image

    curr_parent_idx = 0 # Parent index
    node_idx = 1 # Current node index

    debug_counter = 0

    # Create empty data structures
    visited_queue = {} # explored/visited/closed, valid nodes
    open_dict = {} # keeps track of the node queue to be processed

    show_grid = True # Debug boolean
    goal_found = False # When true, stop search

    ##############################################################################
    # Add initial node to the open node dictionary, initial node has no parent
    open_dict[initial_node_coord] = [0, node_idx, None, (0,0)]

    # Check if the initial node is the goal node, if so stop search
    if (initial_node_coord[0] == goal_node_coord[0] and initial_node_coord[1] == goal_node_coord[1]):
        curr_node_coord = (initial_node_coord[0], initial_node_coord[1])
        visited_queue, goal_found = goal_node_found(goal_found, visited_queue, initial_node_coord[0],
initial_node_coord[1], 0, node_idx, curr_parent_idx, curr_node_coord)
        return visited_queue, goal_found, fig, ax

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

    # Process next node in the open dictionary with the lowest cost to come
    # When all children are evaluated of the current parent node, remove the next node from the open
dictionary
    while (len(open_dict) != 0): # Stop search when node queue is empty

        debug_counter = debug_counter + 1 # Debug variable

        curr_node = get_node_lowest_cost_to_come(open_dict)

        curr_node_x = curr_node[0]
        curr_node_y = curr_node[1]
        curr_coord = (curr_node_x,curr_node_y) # Returns current node's (x,y) coordinates
        curr_node_list = open_dict.pop(curr_coord) # Returns (cost_to_come, current_node_idx,
parent_node_idx, parent_node_coordinates)
        curr_node_coord = (curr_node_x, curr_node_y)

        curr_node = (curr_coord, curr_node_list) # Creates a tuple, first element is the node's
coordinates, 2nd element is curr_node_list

        visited_queue[curr_node_coord] = curr_node_list

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

        # Debug statements
        if show_grid == True and debug_counter % 5000 == 0:

            print("debug_counter: ", debug_counter)
            print("Current Parent Node:")
```

```
            print(curr_node)
            print()

            # display_map_grid_plot(map_grid, curr_node[3][0], curr_node[3][1], point_thickness, 
goal_found, goal_node_coord[0], goal_node_coord[1],  curr_node[3][0],  curr_node[3][1])

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

        # Evaluate children
        curr_parent_idx = curr_node_list[1] # Update parent node index
        i = 1 # Start with first child or move action

        while i < 9: # Iterate for 8 times -> 8 moves

            # Generate child node
            cost_to_move, valid_move, child_node_x_valid, child_node_y_valid, is_node_obstacle = 
generate_child_node(obstacle_matrix, map_boundary_matrix, curr_node, i, map_grid, map_height, 
map_width)

            # Check if child node is in the visited queue
            explored = (child_node_x_valid, child_node_y_valid) in set(visited_queue)

            # Check if child node is in open dictionary
            is_in_open = (child_node_x_valid, child_node_y_valid) in set(open_dict)

            if valid_move == True and explored == False: # Child node is valid but has not been 
explored yet

                is_equal = (child_node_x_valid == goal_node_coord[0] and child_node_y_valid == 
goal_node_coord[1]) # check if goal node reached

                if (is_equal == True): # Child node equals goal node

                    visited_queue, goal_found = goal_node_found(goal_found, visited_queue, 
child_node_x_valid, child_node_y_valid, cost_to_move, node_idx, curr_parent_idx, curr_node_coord)

                    return visited_queue, goal_found, fig, ax

                else: # Goal state not found yet

                    if (explored == False and is_in_open == False): # New child, child has not been 
expored and is not is in open dictionary

                        node_idx = node_idx + 1 # Create new child index

                        open_dict[(child_node_x_valid, child_node_y_valid)]=(cost_to_move, node_idx, 
curr_parent_idx, curr_node_coord)

                    elif (explored == False and is_in_open == True): # Child has not been explored 
but is in open dictionary

                        cost_to_move_new = cost_to_move
                        child_c2c_val_stored = open_dict[(child_node_x_valid, child_node_y_valid)][0]

                        if cost_to_move_new < child_c2c_val_stored: # update cost to come value and 
parent node
                            existing_child_idx = open_dict[(child_node_x_valid, child_node_y_valid)]
[1]
                            child_node_list = (cost_to_move_new, existing_child_idx, curr_parent_idx, 
curr_node_coord)
                            open_dict[(child_node_x_valid, child_node_y_valid)] = child_node_list

            # else:
            #     print("Move not valid.")
```

```
            plot_fig = True

            # print_function(i, valid_move, is_node_obstacle, plot_fig, map_grid)

            i = i + 1 # Update action variable to evaluate the next move

    # End of outer while loop
    print("Goal Found Boolean: ", goal_found)
    print()

    return visited_queue, goal_found, fig, ax

####################################################################################################
#################################################################
# Function gets user input, prints results of dijkstra_approach_alg function, optimal path is also
computed and animation video is created if solution is found
# Resource for time functions: https://stackoverflow.com/questions/27779677/how-to-format-elapsed-
time-from-seconds-to-hours-minutes-seconds-and-milliseco
# Resouce for RBG map grid color selection: https://www.rapidtables.com/web/color/RGB_Color.html
# Resource for creation animation video:
https://docs.opencv.org/3.4/dd/d9e/classcv_1_1VideoWriter.html

def main():

    # Define RGB colors and attributes for map grid image
    explored_color = (255,255,255)
    obstacle_space_color = (156,14,38)
    free_space_color = (0,0,0)
    optimal_path_color = (0,204,204)
    start_node_color = (38, 25, 216)
    goal_node_color = (0,153,0)

    text_font = cv2.FONT_HERSHEY_SIMPLEX
    plt_origin = (85, 10)
    video_origin = (85, 25)
    font_scale = 0.5
    color = (255, 0, 0)
    thickness = 1 # pixels

    # Create map grid and store original width and height of map image
    map_grid, map_height, map_width = create_map_grid(obstacle_space_color, free_space_color)
    original_map_height = map_height
    original_map_width = map_width

    # Resize map image using imutils resize function to speed up processing time, can alter width
value
    # associated height will automatically be computed to maintain aspect ratio
    map_grid = imutils.resize(map_grid, width = 300)
    map_height, map_width, _ = np.shape(map_grid)

    # Set up matrix to keep track of obstacle and free spaces
    map_boundary_matrix = map_obstacle_freespace_matrix(map_grid, map_height, map_width)
    obstacle_matrix = map_boundary_matrix.copy()

    # Plot the resized map grid with the obstacle and free space
    plt.figure()
    plt.title('Resized Map Grid')
    plt.imshow(map_grid.astype(np.uint8), origin="lower")
    plt.show()

    # Get user defined initial and goal node coordinates
    x_initial, y_initial, x_goal, y_goal = get_user_input(map_width, map_height,
check_node_in_obstacle_space, obstacle_matrix)
    initial_node_coord = (x_initial, y_initial)
    goal_node_coord = (x_goal, y_goal)
```

```
##########################################################################################
######

    # Use two different ways to compute the time the dijkstra algorithm takes to solve the given
problem space
    start1 = time.time()
    start2 = datetime.now()

    visited_queue, goal_found, fig, ax = dijkstra_approach_alg(obstacle_matrix, map_boundary_matrix,
initial_node_coord, goal_node_coord, map_grid, map_height, map_width)

    end1 = time.time()
    end2 = datetime.now()

    print("Was goal found ? -> ", goal_found)
    print()

    if goal_found == False:
        print("No solution.")
        print()
        return

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

    # Execution Time Computed - Method 1
    hrs, remain = divmod(end1 - start1, 3600)
    mins, secs = divmod(remain, 60)
    print("- Problem solved in (hours:min:sec:milliseconds) (Method 1): {:0>2}:{:0>2}:
{:05.2f}".format(int(hrs),int(mins),secs))

    # Execution Time Computed - Method 2
    runtime=end2-start2
    print("- Problem solved in (hours:min:sec:milliseconds) (Method 2): " + str(runtime))
    print()

    print("Start node coordinate input:", (x_initial,y_initial))
    print("Goal node coordinate input:", (x_goal,y_goal))
    print()

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

    # Call function to compute optimal path
    optimal_path = compute_optimal_path(visited_queue, initial_node_coord, goal_node_coord)

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

    # Create animation visualization video
    out = cv2.VideoWriter('conn_dijkstra_algorithm_video.avi', cv2.VideoWriter_fourcc(*'XVID'), 50,
(original_map_width,original_map_height))

    start_goal_pt_thickness = 3
    map_grid = cv2.circle(map_grid, (x_initial,y_initial), radius = 0, color=start_node_color,
thickness = start_goal_pt_thickness) # start node
    map_grid = cv2.circle(map_grid, (x_goal,y_goal), radius = 0, color=goal_node_color, thickness =
start_goal_pt_thickness) # goal node

    for visited_node in visited_queue:

        map_grid[visited_node[1],visited_node[0]] = explored_color # explored node
        map_grid = cv2.circle(map_grid, (x_initial,y_initial), radius =0 , color=start_node_color,
thickness = start_goal_pt_thickness) # start node
        map_grid = cv2.circle(map_grid, (x_goal,y_goal), radius = 0, color=goal_node_color, thickness
= start_goal_pt_thickness) # goal node
```

```
        output_frame = cv2.flip(map_grid, 0) # change y axis of image
        output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR) # change color space to reflect
proper colors

        output_frame = cv2.resize(output_frame, (original_map_width, original_map_height)) # resize
image back to original image shape

        out.write(output_frame) # write image framr to animation video

    for optimal_node in optimal_path:
        map_grid[optimal_node[1],optimal_node[0]] = optimal_path_color # optimal path node
        map_grid = cv2.circle(map_grid, (x_initial,y_initial), radius=0, color=start_node_color,
thickness=start_goal_pt_thickness) # start node
        map_grid = cv2.circle(map_grid, (x_goal,y_goal), radius=0, color=goal_node_color,
thickness=start_goal_pt_thickness) # goal node

        output_frame = cv2.flip(map_grid, 0) # change y axis
        output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR) # change color space to reflect
proper colors

        output_frame = cv2.resize(output_frame, (original_map_width, original_map_height)) # resize
image back to original image shape

        out.write(output_frame) # write image framr to animation video

    output_frame = cv2.flip(map_grid, 0) # change y axis
    output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR) # change color space to reflect
proper colors
    if goal_found:
        output_frame = cv2.putText(output_frame, 'Solution Found', video_origin, text_font,
font_scale, color, thickness, cv2.LINE_AA)
        map_grid = cv2.putText(map_grid, 'Solution Found', plt_origin, text_font, font_scale, (0, 0,
255), thickness, cv2.LINE_AA, bottomLeftOrigin= True)

    else:
        output_frame = cv2.putText(output_frame, 'No Solution', video_origin, text_font, font_scale,
color, thickness, cv2.LINE_AA)
        map_grid = cv2.putText(map_grid, 'No Solution', plt_origin, text_font, font_scale, (0, 0,
255), thickness, cv2.LINE_AA, bottomLeftOrigin = True)

    output_frame = cv2.resize(output_frame, (original_map_width, original_map_height)) # resize image
back to original image shape
    out.write(output_frame) # write image framr to animation video

    out.release() # release video object, done writing frames to video

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

    # Display last frame to Python IDE

    ax.set_title('Final Map Grid')
    ax.axis('off')
    ax.imshow((map_grid).astype(np.uint8), animated=True, origin="lower")

    print("Code Script Complete.")

    # End of algorithm

#######################################################################################################
######################################
# Call main function

main()
```

```
# End of code file
#####################################################################################################
```