```
#!/usr/bin/env python3
# -*- 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
from collections import OrderedDict
from datetime import datetime
import matplotlib.pyplot as plt

import numpy as np
import imutils
import time
import cv2

import math
from math import radians

###############################################################################################################################################
# Visualization/Plotting Helper Functions

# 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(clearance_color, obstacle_space_color, free_space_color, robot_clearance, robot_radius, map_height, map_width):

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

    # Set obstacle color
    obstacle_color = obstacle_space_color

    # Define total clearance value in pixels using user robot radius and robot clearance inputs
    c = robot_clearance + robot_radius # 5 + 5 or 5 + 0

    #############################################################
    # 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):

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

            # 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

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

            # Plot horizontal walls bloated by c
            if (x >= 0 and x < map_width and y >= 0 and y < c) or (x >= 0 and x < map_width and y >=
map_height - c  and y < map_height):
                map_grid[y,x] = clearance_color

            # Plot vertical walls bloated by c
            if (x >= 0 and x < c and y >= 0 and y < map_height) or (x >= map_width - c and x < map_width
and y >= 0 and y < map_height):
                map_grid[y,x] = clearance_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[int(child_node_y)][int(child_node_x)] == -1

# Function checks if node is within map_grid bounds
def check_node_in_map(child_node_x, child_node_y, map_height, map_width):
    return 0 <= child_node_x < map_width and 0 <= child_node_y < map_height
```

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

# Function handles negative angles and angles greater than 360 degrees
def handle_theta_angle(input_angle):
    return int(input_angle % 360.0)

# 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, goal_node_coord,
action, map_grid, map_height, map_width, step_size):

    valid_move = False # boolean truth value of valid swap
    is_node_obstacle = False # boolean to check if node is in obstacle space

    child_node_x, child_node_y, child_theta, cost_of_action, angle_to_add = 0, 0, 0, 0, 0

    parent_node_x = parent_node[0][0] # parent x coordinate
    parent_node_y = parent_node[0][1] # parent y coordinate
    parent_theta = parent_node[1][4] # parent theta value
    parent_cost_to_come = parent_node[1][5] # parent cost to come (not total cost)

    # Action logic using dictionary
    action_dict = {1: -60, 2: -30, 3: 0, 4: 30, 5: 60}

    # Set angle_to_add based on the action value
    angle_to_add = action_dict.get(action, 0)
    child_theta = parent_theta + angle_to_add
    child_theta = handle_theta_angle(child_theta)

    child_node_x = parent_node_x + step_size * np.cos(radians(child_theta))
    child_node_y = parent_node_y + step_size * np.sin(radians(child_theta))
    # Round coordinates to nearest half to reduce node search
    child_node_x = round(2*child_node_x)/2
    child_node_y = round(2*child_node_y)/2

    # Check if node is within map bounds and if it is in the obstacle space for later computations
    is_node_in_map = check_node_in_map(child_node_x, child_node_y, map_height, map_width)
    if is_node_in_map:
        is_node_obstacle = check_node_in_obstacle_space(child_node_x, child_node_y, obstacle_matrix)
    else:
        is_node_obstacle = False
    valid_move = is_node_in_map and not is_node_obstacle

    ########### COST LOGIC ###################

    # Compute child node's cost to come value -> CostToCome(x') = CostToCome(x) + L(x,u)
    cost_of_action = step_size
    child_cost_to_come = round(cost_of_action + parent_cost_to_come, 1)
```

```
    pta = (child_node_x, child_node_y)
    ptb = (goal_node_coord[0], goal_node_coord[1])

    # Euclidean Distance Heuristic
    cost_to_go = np.linalg.norm(np.array(pta)-np.array(ptb))

    weight = 1 # Can change this value to further bias the results toward the goal node ex. enter 4
    total_cost = child_cost_to_come + (weight*cost_to_go)

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

    # returned node is the resulting child node of the requested action
    return child_cost_to_come, total_cost, valid_move, child_node_x, child_node_y, child_theta,
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,
closest_node_to_goal_x, closest_node_to_goal_y):
    path_list = [] # list to store coordinates of optimal path


    closest_node_to_goal = (closest_node_to_goal_x, closest_node_to_goal_y)

    first_parent_coord = visited_queue[closest_node_to_goal][3] # set first parent to search for equal to
the goal node's parent node
    curr_elem_x = closest_node_to_goal[0] # get x coordinate of the goal node, first node added to the
path list
    curr_elem_y = closest_node_to_goal[1] # get y coordinate of the goal node, first node added to the
path list

    path_list.append(closest_node_to_goal) # 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)) # Initiate all cells to 0
    map_boundary_matrix[free_space] = np.inf # Set free space to infinty
    map_boundary_matrix[obstacle_space] = -1 # Set obstacle space to -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 -60")
    elif i == 2:
        print("Action -30")
    elif i == 3:
        print("Action 0")
    elif i == 4:
        print("Action 30")
    elif i == 5:
        print("Action 60")

    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_total_cost(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, total_cost,
node_idx, curr_parent_idx, curr_node_coord, child_cost_to_come):
    goal_found = True

    node_idx = node_idx + 1

    child_node = ((child_node_x_valid, child_node_y_valid),(total_cost, node_idx, curr_parent_idx,
curr_node_coord, child_cost_to_come))
```

```
    visited_queue[(child_node_x_valid, child_node_y_valid)] = (total_cost, node_idx, curr_parent_idx,
curr_node_coord, child_cost_to_come)

    closest_node_to_goal_x = child_node_x_valid
    closest_node_to_goal_y = child_node_y_valid

    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, closest_node_to_goal_x, closest_node_to_goal_y

#########################################################################################
########################################################
# 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) + ": "))
        th_initial = eval(input("Enter the orientation of the robot at the start point in degrees" + ": "))

        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.")
                print()
                continue

            if (th_initial % 30) != 0:
                print("Re-enter initial theta, angle not multiple of 30. (k * 30), i.e. {.., -60,-30, 0, 30, 60, ..}")
                print()
```

```
            continue
        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) + ": "))
        th_goal = eval(input("Enter the orientation of the robot at the goal point in degrees" + ": "))

        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.")
                print()
                continue

            if (th_goal % 30)!=0:
                print("Re-enter goal theta, angle not multiple of 30. (k * 30), i.e. {.., -60,-30, 0, 30, 60, ..}")
                print()
                continue
            else:
                break

    # Get user defined step size
    while True:
        step_size = eval(input("Enter step size of the robot in units (1 <= L <= 10)" + ": "))

        if not(1 <= step_size <= 10):
            print("Re-enter step size of the robot in units (1 <= L <= 10).")
            print()

        else:
            print("Step Size L:", step_size)
            print()
            break

    # Make sure input initial and goal angles are non-negative and are <= 360 degrees
    th_initial = handle_theta_angle(th_initial)
```

```
    th_goal = handle_theta_angle(th_goal)


print("_____________________________________________________________________________
__________")
    print()

    # Just for debugging and quick code execution

    # x_initial = 10
    # y_initial = 11
    # th_initial = 0 #-60
    # x_goal =  150 #200 #250 #280 #225 #220 #215 #210 #205 #200 #170 #62
    # y_goal = 15
    # th_goal = 0 #270 # 30
    # step_size = 10

    return x_initial, y_initial, x_goal, y_goal, th_initial, th_goal, step_size

# Function gets the user defined input to define robot radius and clearance
def get_user_robot_input():

    # Get user defined robot radius
    while True:
        robot_radius = eval(input("Enter robot radius (ex. 5)" + ": "))

        if not(isinstance(robot_radius, int)):
            print("Re-enter robot radius as integer value")
            print()
            continue

        else:
            print("Robot Radius:", robot_radius)
            print()
            break

    # Get user defined robot clearance
    while True:
        robot_clearance = eval(input("Enter robot clearance (ex. 5)" + ": "))

        if not(isinstance(robot_clearance, int)):
            print("Re-enter robot clearance as integer value")
            print()
            continue

        else:
            print("Robot Clearance:", robot_clearance)
            print()
            break
```

```
    return robot_radius, robot_clearance

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

# Function takes numerical input and rounds it to the nearest half
def round_num_nearest_half(num, thresh_val):
    return (2*(round(num / thresh_val) * thresh_val))

# Function checks if node is duplicate node in visited_matrix by using i,j,k mapping
def is_node_visited_duplicate(visited_matrix, child_node_x_valid, child_node_y_valid, child_theta):
    # Thresholds defined per problem statement
    dist_thresh = 0.5
    theta_thresh = 30

    # Compute the indices of the visited region for the node
    i = int(round_num_nearest_half(child_node_x_valid, dist_thresh))
    j = int(round_num_nearest_half(child_node_y_valid, dist_thresh))
    k = int(child_theta / theta_thresh)

    # Check if the node resides in the visited region already within the visited_matrix
    node_visited_status = False

    if (0 <= i < 500) and (0 <= j < 1200) and (0 <= k < 12):
        node_visited_status = visited_matrix[i][j][k] == 1
        # Mark node resides in visited region in visited_matrix
        visited_matrix[i][j][k] == 1
        return node_visited_status, True # True if ijk are valid
    else:
        return node_visited_status, False # False if ijk are not valid

# Function checks if two numbers are comparable within a tolerance
def compare_nums(numa, numb):
    tolerance = 0.1 #0.5
    return abs(numa - numb) < tolerance

# Function checks if node already exists in open_dict
def is_node_in_open_dict(compare_x, compare_y, open_dict):
    for node in open_dict:
        if compare_nums(compare_x, node[0]) and compare_nums(compare_y, node[1]):
            return True, node[0], node[1]

    return False, None, None

##########################################################################################
############################################################
# 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 astar_approach_alg(obstacle_matrix, map_boundary_matrix, initial_node_coord, goal_node_coord,
map_grid, map_height, map_width, th_initial, th_goal, step_size):

    # Thresholds defined per problem statement
    euclid_dist_threshold = 1.5
    theta_threshold = 30

    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
    # debug_counter2 = 0

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

    visited_matrix = np.zeros((500, 1200, 12))

    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, initial_node_coord, th_initial, 0]

    # Check if the initial node is the goal node, if so stop search
    pt1 = (initial_node_coord[0], initial_node_coord[1])
    pt2 = (goal_node_coord[0], goal_node_coord[1])
    euclidean_dist = np.linalg.norm(np.array(pt1)-np.array(pt2))

    if (initial_node_coord[0] == goal_node_coord[0] and initial_node_coord[1] == goal_node_coord[1]
and (abs(th_initial - th_goal) <= theta_threshold)):

        curr_node_coord = (initial_node_coord[0], initial_node_coord[1])
        visited_queue, goal_found, closest_node_to_goal_x, closest_node_to_goal_y =
goal_node_found(goal_found, visited_queue, initial_node_coord[0], initial_node_coord[1], 0,
node_idx, curr_parent_idx, curr_node_coord, 0)
        return visited_queue, goal_found, fig, ax, closest_node_to_goal_x, closest_node_to_goal_y

    ##############################################################################
    print("Main code execution has started...")
    print()
```

```
    # Process next node in the open dictionary with the lowest cost to come
    while (len(open_dict) != 0): # Stop search when node queue is empty

        debug_counter = debug_counter + 1 # Debug variable

        curr_node = get_node_lowest_total_cost(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

        debug_runs = 5000

        if show_grid == True and debug_counter % debug_runs == 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/move/action

        while i < 6: # Iterate for 5 times -> 5 moves

            # CAN PUT DEBUG STATEMENT HERE TO MONITOR CHILD NODE CREATION

            # Generate child node
            child_cost_to_come, total_cost, valid_move, child_node_x_valid, child_node_y_valid, 
child_theta, is_node_obstacle = generate_child_node(obstacle_matrix, map_boundary_matrix, 
curr_node, goal_node_coord, i, map_grid, map_height, map_width, step_size)
```

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

        if valid_move == False:
            i = i + 1 # Update action variable to evaluate the next move
            continue

        # Check if child node is a duplicate node in visited matrix
        is_node_duplicate, is_mapping_valid = is_node_visited_duplicate(visited_matrix,
child_node_x_valid, child_node_y_valid, child_theta)
        if is_node_duplicate or not(is_mapping_valid):
            i = i + 1 # Update action variable to evaluate the next move
            continue

        # Check if child node is in the visited queue
        explored = (child_node_x_valid, child_node_y_valid) in set(visited_queue)
        if explored:
            i = i + 1 # Update action variable to evaluate the next move
            continue

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

        # Check if child node is in open dictionary
        is_in_open, x_replace, y_replace = is_node_in_open_dict(child_node_x_valid,
child_node_y_valid, open_dict)
        key_replace = (x_replace, y_replace)

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

            # Check if the initial node is the goal node, if so stop search
            pt1 = [child_node_x_valid, child_node_y_valid]
            pt2 = goal_node_coord
            euclidean_dist = np.linalg.norm(np.array(pt1)-np.array(pt2))

            # Check if current child node is goal node
            if (euclidean_dist <= euclid_dist_threshold) and (abs(child_theta - th_goal) <=
theta_threshold): # Child node equals goal node
                visited_queue, goal_found, closest_node_to_goal_x, closest_node_to_goal_y =
goal_node_found(goal_found, visited_queue, child_node_x_valid, child_node_y_valid, total_cost,
node_idx, curr_parent_idx, curr_node_coord, child_cost_to_come)
                return visited_queue, goal_found, fig, ax, closest_node_to_goal_x,
closest_node_to_goal_y

            else: # Goal node/state not found yet

                if (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)]=(total_cost, node_idx,
curr_parent_idx, curr_node_coord, child_theta, child_cost_to_come)
```

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

                        child_total_cost_stored = open_dict[(x_replace, y_replace)][0]

                        # Update node
                        if total_cost < child_total_cost_stored: #and (abs(child_theta - child_theta_val_stored)
<= theta_threshold):
                            total_cost_new = total_cost
                            existing_child_idx = open_dict[(x_replace, y_replace)][1]

                            del open_dict[key_replace]

                            child_node_list = (total_cost_new, existing_child_idx, curr_parent_idx,
curr_node_coord, child_theta, child_cost_to_come)
                            open_dict[(child_node_x_valid, child_node_y_valid)] = child_node_list

            #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_func():

    # Map/grid dimensions defined per problem statements
    map_height = 250
    map_width = 600

    # Define RGB colors and attributes for map grid image
    obstacle_space_color = (156,14,38)
    free_space_color = (0,0,0)
    clearance_color = (102, 0, 0)
```

```
    optimal_path_color = (0,204,204)
    start_node_color = (255, 255, 0)
    goal_node_color = (0,153,0)
    explored_node_color = (255, 255, 255) # White arrows


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

    start_goal_pt_thickness = 3
    traversal_thickness = 3

    robot_clearance, robot_radius = 0, 0
    robot_radius, robot_clearance = get_user_robot_input()

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

    # Get user defined initial and goal node coordinates
    x_initial, y_initial, x_goal, y_goal, th_initial, th_goal, step_size = 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)

    print("Map Created and User Input Saved")
    print()

    # 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()


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

    # 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, closest_node_to_goal_x, closest_node_to_goal_y =
astar_approach_alg(obstacle_matrix, map_boundary_matrix, initial_node_coord, goal_node_coord,
map_grid, map_height, map_width, th_initial, th_goal, step_size)

    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,
closest_node_to_goal_x, closest_node_to_goal_y)


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

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


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

    # Plots start and goal nodes
    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

    last_parent_x = next(iter(visited_queue))[0]
    last_parent_y = next(iter(visited_queue))[1]
    last_parent_x = int(last_parent_x)
    last_parent_y = int(last_parent_y)

    for visited_node in visited_queue:

        # Add arrow to the plot
        arrow_img = np.copy(map_grid)
        pc = visited_queue[visited_node][3]
        px = int(pc[0])
        py = int(pc[1])
        cv2.arrowedLine(arrow_img, (px, py), (int(visited_node[0]), int(visited_node[1])),
explored_node_color, thickness=1, tipLength=0.05)

        # Overlay the arrow on the original map
        map_grid = cv2.addWeighted(map_grid, 0.5, arrow_img, 0.5, 0)

        ## Debug Statement
        # 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()

        # Plot start and end goal nodes
        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

        # Necessary image processing for proper color display
        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

        # Keep track of start nodes for each arrow
        last_parent_x = visited_node[0]
        last_parent_y = visited_node[1]
        last_parent_x = int(last_parent_x)
        last_parent_y = int(last_parent_y)


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

    last_parent_x = optimal_path[0][0]
    last_parent_y = optimal_path[0][1]

    last_parent_x = int(last_parent_x)
    last_parent_y = int(last_parent_y)

    for optimal_node in optimal_path:

        # Define the color of the arrow
        color = optimal_path_color

        # Add the arrow to the plot
        arrow_img = np.copy(map_grid)

        cv2.line(arrow_img, (last_parent_x, last_parent_y), (int(optimal_node[0]), int(optimal_node[1])), 
color, thickness=1)
        map_grid = cv2.circle(map_grid, (int(optimal_node[0]), int(optimal_node[1])), radius = 0 , 
color=(38, 25, 216), thickness = traversal_thickness) # start node

        # Overlay the arrow on the original map
        map_grid = cv2.addWeighted(map_grid, 0.5, arrow_img, 0.5, 0)
```

```
        map_grid[int(optimal_node[1]),int(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

        last_parent_x = optimal_node[0]
        last_parent_y = optimal_node[1]

        last_parent_x = int(last_parent_x)
        last_parent_y = int(last_parent_y)


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

    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 to terminal/console
    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
    plt.figure()
    plt.title('Final Map Grid')
    plt.imshow(map_grid.astype(np.uint8), origin="lower")
    plt.show()

    # Display last frame to video
    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_func()

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