A_star_Varun_Lakshmanan_Sai_Jagadeesh_Muralikrishnan.py

```python
import numpy as np
from queue import PriorityQueue
import cv2
import time

#--------------------------------------------------Creating the Canvas------------------------------------------------------#
height = 500
width = 1200

Graph_map = np.ones((height, width, 3), dtype=np.uint8)*255

#--------------------------------------------------Creating the User Interface-------------------------------------------------#

## Taking input from the user for start and goal nodes.
# User  input for x and y coordinates of start node.
def start_node(width, height, canvas):
    while True:
        try:
            Xs = int(input("Enter the x-coordinate of the start node(Xs): "))
            start_y = int(input("Enter the y-coordinate of the start node(Ys): "))
            Ys = height - start_y
            start_theta = int(input("Enter the angle of the start_node: "))

            if Xs < 0 or Xs >= width or Ys < 0 or Ys >= height:
                print("The x and y coordinates of the start node is out of range.Try again!!!")
            elif np.any(canvas[Ys, Xs] != [255, 255,255]):
                print("The x or y or both coordinates of the start node is on the obstacle.Try again!!!")
            elif start_theta % 30 != 0:
                print("The angle of the start node is out of range.Try again!!!")
            else:
                return Xs, Ys, start_theta
        except ValueError:
            print("The x and y coordinates of the start node is not a number. Try again!!!")


def goal_node(width, height, canvas):
    while True:
        try:
            Xg = int(input("Enter the x-coordinate of the goal node(Xg): "))
            goal_y = int(input("Enter the y-coordinate of the goal node(Yg): "))
            Yg = height - goal_y
            goal_theta = int(input("Enter the angle of the goal node: "))

            if Xg < 0 or Xg >= width or Yg < 0 or Yg >= height:
                print("The x and y coordinates of the goal node is out of range.Try again!!!")
            elif np.any(canvas[Yg,Xg] != [255,255,255]):
                print("The x or y or both coordinates of the goal node is on the obstacle.Try again!!!")
            elif goal_theta % 30 != 0:
                print("The angle of the goal node is out of range.Try again!!!")
```

```python
            else:
                return Xg, Yg, goal_theta
        except ValueError:
            print("The x and y coordinates of the goal node is not a number. Try again!!!")

# User input for step size.
def step_size_function():
    while True:
        try:
            step_size = int(input("Enter the step size between 1 and 10(inclusive): "))
            if 1 <= step_size <= 10:
                return step_size
            else:
                print("The step size is not between 1 and 10. Try again!!.")
        except ValueError:
            print("The step size is not a number. Try again!!!")

def print_a_star_ascii():
    print("""
    ___                 _____  __                              __    __       ____ _ __
   /   |               / ___/ / /_ ____ _ _____   ____   ____ _ / /_ / /_     / __/(_)____ ____/ /___   _____
  / /| |    ______    \__ \ / __// __ `// ___/  / __ \ / __ `// __// __ \   / /_ / // __ \ / __  // _ \ / ___/
 / ___ |   /_____/   ___/ // /_ / /_/ // /     / /_/ // /_/ // /_ / / / /  / __// // / / // /_/ //  __// /
/_/  |_|            /____/ \__/ \__,_//_/     / .___/ \__,_/ \__//_/ /_/  /_/  /_//_/ /_/ \__,_/ \___//_/
                                             /_/
""")

print_a_star_ascii()
# User input for radius of the robot.
radius_of_robot = int(input("Enter the radius of the robot: "))
clearance = int(input("Enter the clearance of the robot: "))
step_size = step_size_function()
Total_clearance = radius_of_robot + clearance

# Creating a matrix to store the visited nodes.
G = np.zeros((1000, 2400, 12), dtype=np.uint8)

# Creating a cache to store the heuristic values.
heuristic_cache = {}

#---------------------------------------------------Creating the Hexagon-----------------------------------------------#
# Center of the hexagon.
center_h = (650,250)
# Side of hexagon.
side = 150
# radius from thhe center.
r = np.cos(np.pi/6) * side
# Center Coordinates of hexagon.
c_x,c_y = center_h

angles = np.linspace(np.pi / 2, 2 * np.pi + np.pi / 2, 7)[:-1]
v_x = c_x + r * np.cos(angles) # x_coordinate_vertices.
```

```python
v_y = c_y + r * np.sin(angles) # y_coordinate_vertices.
radius_clearance = r + Total_clearance # Clearance from radius.
v_x_c = c_x + radius_clearance * np.cos(angles) # x_coordinate_clearance_vertices.
v_y_c= c_y + radius_clearance * np.sin(angles) # y_coordinate_clearance_vertices.
vertices = np.vstack((v_x, v_y)).T # storing x and y vertices in a tuple.
clearance_verticies = np.vstack((v_x_c, v_y_c)).T # storing clearance x and y vertices.

#--------------------------------------Creating the Rectangles using half planes------------------------------------------------#
for x in range(1200):
    for y in range(500):
        y_transform = 500 - y

        # Wall clearance.
        if (x <= 0 + Total_clearance or x >= 1200 - Total_clearance or y_transform <= 0 + Total_clearance or y_transform >= 500 - Total_clearance):
            Graph_map[y,x] = [0,255,0]

        # object 1(rectangle)
        if (x >= 100 and x <= 175  and y_transform >= 100 and y_transform <= 500 ):
            Graph_map[y,x] = [0,0,0]
        elif (x >= 100 - Total_clearance  and x <= 175 + Total_clearance and y_transform >= 100 - Total_clearance and y_transform <= 500 + Total_clearance):
            Graph_map[y,x] = [0, 255, 0]

        # object 2(rectangle)
        if (x >= 275 and x <= 350 and y_transform >= 0 and y_transform <= 400):
            Graph_map[y,x] = [0,0,0]
        elif(x >= 275 - Total_clearance and x <= 350 + Total_clearance and y_transform >= 0 - Total_clearance and y_transform <= 400 + Total_clearance):
             Graph_map[y,x] = [0, 255, 0]

        # object 3 (combination of 3 rectangles)
        if (x >= 1020 - Total_clearance and x <= 1100 + Total_clearance and y_transform>= 50 - Total_clearance  and y_transform <= 450 + Total_clearance):
            Graph_map[y,x] = [0,255,0]
        elif (x >= 900 - Total_clearance  and x <= 1100 + Total_clearance  and y_transform >= 50 - Total_clearance and y_transform <= 125 + Total_clearance):
            Graph_map[y,x] = [0, 255, 0]
        elif (x >= 900 - Total_clearance and x <= 1100 + Total_clearance and y_transform >= 375 - Total_clearance and y_transform <= 450 + Total_clearance):
            Graph_map[y,x] = [0,255,0]

        if (x >= 1020 and x <= 1100 and y_transform>= 50  and y_transform <= 450):
            Graph_map[y,x] = [0,0,0]
        elif (x >= 900 and x <= 1100  and y_transform >= 50 and y_transform <= 125):
            Graph_map[y,x] = [0,0,0]
        elif (x >= 900 and x <= 1100 and y_transform >= 375 and y_transform <= 450):
            Graph_map[y,x] = [0,0,0]

# object 4 (hexagon)
def hexagon(x, y, vertices): # Defining a function to calucalate cross product of vertices inside hexagon.
    result = np.zeros(x.shape, dtype=bool)
    num_vertices = len(vertices)
    for i in range(num_vertices):
        j = (i + 1) % num_vertices
```

```
            cross_product = (vertices[j, 1] - vertices[i, 1]) * (x - vertices[i, 0]) - 
(vertices[j, 0] - vertices[i, 0]) * (y - vertices[i, 1])
            result |= cross_product > 0
        return ~result

# Creating a meshgrid.
x, y = np.meshgrid(np.arange(1200), np.arange(500))

# Hexagon and its clearance.
hexagon_original = hexagon(x, y, vertices)
hexagon_clearance = hexagon(x, y,clearance_verticies) & ~hexagon_original

# Drawing hexagon and its clearance on the graph_map.
Graph_map[hexagon_clearance] = [0, 255, 0]
Graph_map[hexagon_original] = [0, 0, 0]

# Creating a video file to store the output.
output = cv2.VideoWriter('A_star_Varun_Lakshmanan_Sai_Jagadeesh_Muralikrishnan.mp4', 
cv2.VideoWriter_fourcc(*'mp4v'), 30, (width, height))

#-----------------------------------------Creating the Action sets---------------------------
------#
# Move straight forward
def movement_1(node, step_size):
    x, y, theta = node
    new_node = (int(x + step_size * np.cos(np.radians(theta))), y + step_size * 
np.sin(np.radians(theta)), theta)
    x, y, theta = new_node
    return x,y,theta
# Move 30 degrees to the right
def movement_2(node, step_size):
    x, y, theta = node
    theta_i = (theta + 30) % 360
    new_node = (x + step_size * np.cos(np.radians(theta_i)), y + step_size * 
np.sin(np.radians(theta_i)), theta_i)
    x, y, theta = new_node
    return x, y, theta
# Move 60 degrees to the right
def movement_3(node, step_size):
    x, y, theta = node
    theta_i = (theta + 60) % 360
    new_node = (x + step_size* np.cos(np.radians(theta_i)), y + step_size * 
np.sin(np.radians(theta_i)), theta_i)
    x, y, theta = new_node
    return x, y, theta
# Move 30 degrees to the left
def movement_4(node, step_size):
    x, y, theta = node
    theta_i = (theta - 30) % 360
    new_node = (x + step_size*np.cos(np.radians(theta_i)), y + step_size * 
np.sin(np.radians(theta_i)), theta_i)
    x, y, theta = new_node
    return x, y, theta
# Move 60 degrees to the left
def movement_5(node, step_size):
    x, y, theta = node
    theta_i = (theta - 60) % 360
```

```python
    new_node = (x + step_size * np.cos(np.radians(theta_i)), y + step_size * np.sin(np.radians(theta_i)), theta_i)
    x, y, theta = new_node
    return x, y, theta
#----------------------------------Function to check the possible nodes-----------------------------#
def possible_node(node):
    new_nodes = []
    action_set = {movement_1:step_size,
                  movement_2:step_size,
                  movement_3:step_size,
                  movement_4:step_size,
                  movement_5:step_size}
    rows, columns, _ = Graph_map.shape
    for action, cost in action_set.items():
        new_node = action(node, step_size)
        cost = step_size
        next_x, next_y, new_theta = new_node
        if 0 <= next_x <= columns and 0 <= next_y < rows and np.all(Graph_map[int(next_y), int(next_x)] == [255, 255, 255]) and not visited_check(new_node):
            new_nodes.append((cost, new_node))
    return new_nodes

#------------------------------------Creating the Heuristic Function--------------------------------#
def heuristic(node, goal):
    if node in heuristic_cache:
        return heuristic_cache[node]
    else:
        heuristic_value = np.sqrt((node[0] - goal[0])**2 + (node[1] - goal[1])**2)
        heuristic_cache[node] = heuristic_value
        return heuristic_value

#----------------------------------------- the A* Algorithm-----------------------------------------#
def A_star(start_node, goal_node):
    parent = {}
    cost_list = {start_node:0}
    closed_list = set()
    open_list = PriorityQueue()
    open_list.put(((0 + heuristic(start_node, goal_node)), start_node))
    map_visualization = np.copy(Graph_map)
    marking_visited(start_node)
    step_count = 0

    # While loop to check the open_list is empty or not.
    while not open_list.empty():
        current_cost, current_node = open_list.get()
        closed_list.add(current_node)

        # If the current node is equal to goal node, then it will break the loop and return the path along with writing the path to the video.
        if heuristic(current_node, goal_node) < 1.5 and current_node[2] == goal_node[2]:
            path = A_star_Backtracking(parent, start_node, current_node, map_visualization, step_count)
            for _ in range(80):
               output.write(map_visualization)
            return path
```

```python

        # If the current node is not equal to goal node, then it will check the possible nodes and add it to the open_list along with visulizing the node exploration.
        for cost, new_node in possible_node(current_node):
            cost_to_come = cost_list[current_node] + cost
            if new_node not in cost_list or cost_to_come < cost_list[new_node]:
                cost_list[new_node] = cost_to_come
                parent[new_node] = current_node
                cost_total = cost_to_come + heuristic(new_node, goal_node)
                open_list.put((cost_total, new_node))
                marking_visited(new_node)
                cv2.arrowedLine(map_visualization, (int(current_node[0]), int(current_node[1])), (int(new_node[0]), int(new_node[1])), (0, 0, 255), 1, tipLength=0.3)
                if step_count % 2000 == 0:
                    output.write(map_visualization)
                step_count += 1

    output.release()
    return None
#-------------------------Creating the Matrix using second method----------------------------------#
# Getting the indices of the matrix.
def matrix_indices(node):
    x, y, theta = node
    x = round(x)
    y = round(y)
    i = int(2 * y)
    j = int(2 * x)
    k = int(theta / 30) % 12
    return i, j, k

# Marking the visited nodes.
def marking_visited(node):
    i, j, k = matrix_indices(node)
    if 0 <= i < 1000 and 0 <= j < 2400:
        G[i, j, k] = 1

# Checking the visited nodes.
def visited_check(node):
    i, j, k = matrix_indices(node)
    return G[i, j, k] == 1

#---------------------------Creating the Backtracking Function--------------------------------------#
def A_star_Backtracting(parent, start_node, end_node, map_visualization, step_count):
    path = [end_node] # Adding end node to the path
    while end_node != start_node: # If the end node is not equal to start_node, parent of the end_node is added to path and continues.
        path.append(parent[end_node])
        end_node = parent[end_node] # The parent of end node becomes the current node.
    path.reverse()
    for i in range(len(path) - 1):
        start_point = (int(path[i][0]), int(path[i][1]))  # Converting the coordinates for visualization.
        end_point = (int(path[i + 1][0]), int(path[i + 1][1]))
        cv2.arrowedLine(map_visualization, start_point, end_point, (255, 0, 0), 1, tipLength=0.3)
        if step_count % 5 == 0:
```

```
            output.write(map_visualization)
        step_count += 1
    return path

Xs, Ys, start_theta = start_node(width, height, Graph_map) # Getting the start node from the user
Xg, Yg, goal_theta = goal_node(width, height, Graph_map) # Getting the goal node from the user


#--------------------------------------Initializing the nodes------------------------------------#
start_node = (Xs, Ys, start_theta)
goal_node = (Xg, Yg, goal_theta)

start_time = time.time()   # Starting to check the runtime.
path = A_star(start_node, goal_node)

if path is None:
    print("No optimal path found")
else:
    print("Path found")

end_time = time.time()    # end of runtime
print(f'Runtime : {((end_time-start_time)/60):.2f} Minutes')

#--------------------------------------------------End of the Program-------------------------------------------------#
```