```python
# Github link: https://github.com/vijaydevmasters/ENPM661-PRJ3_PHASE2

import numpy as np
import cv2
import math
from queue import PriorityQueue


# Function to map coordinates to the bottom left of the image
def map_to_bottom_left(point):
    """
    Maps the given coordinates to the bottom left corner of a rectangle.

    Parameters:
    x (int): The x-coordinate of the point.
    y (int): The y-coordinate of the point.
    width (int): The width of the rectangle.
    height (int): The height of the rectangle.

    Returns:
    tuple: A tuple containing the x and y coordinates of the point mapped to the bottom left corner.
    """
    height, width = 400, 1200
    bottom_left_x = point[0]
    bottom_left_y = height - point[1]
    return (bottom_left_x, bottom_left_y)


# Function to check if a point is a valid neighbor
def is_valid_neighbor(point):
    global Robot_radius
    global clearance
    #print("clearance: ", clearance)
    #print("Robot Radius: ", Robot_radius)
    """
    Checks if a given point is a valid neighbor based on the obstacle map.

    Args:
        point (tuple): The coordinates of the point to check.
        obstacles (numpy.ndarray): The obstacle map.

    Returns:
        bool: True if the point is a valid neighbor, False otherwise.
    """
    x, y,_ ,f ,i= point

    x,y=map_to_bottom_left((x,y))
    #print("x: "+str(x)+" y: "+str(y))
    if clearance+Robot_radius <= x < width-clearance-Robot_radius and clearance+Robot_radius <= y <
height-clearance-Robot_radius:
        if y > (200-clearance-Robot_radius) and (x > (300-clearance-Robot_radius) and x <
(350+clearance+Robot_radius)):
            #print("2")
            return False
        elif y < (200+clearance+Robot_radius) and (x > (500-clearance-Robot_radius) and x <
(550+clearance+Robot_radius)):
            #print("3")
            return False
        elif (x - 840) ** 2 + (y - 240) ** 2 <= (120+clearance+Robot_radius) ** 2:
           # print("4")
            return False

        return True
    return False



def get_neighbors(point, obstacles, r1, r2):
    """
    Get the neighboring points for a given point on the grid.

    Args:
        point (tuple): The coordinates of the point (x, y, theta).
        obstacles (list): List of obstacles on the grid.
        r1 (int): RPM of the left wheel.
        r2 (int): RPM of the right wheel.

    Returns:
        list: List of neighboring points, each represented as a tuple (x, y, theta, cost, action).
```

```
    """
    global print_interval

    R = 33/5  # Radius of the wheels
    L = 287/5  # Distance between the wheels (wheelbase)
    dt = 0.01  # Time step
    neighbors_l = []
    Xi, Yi, Thetai = point

    R_over_2 = R / 2
    R_over_L = R / L

    actions = np.array([[r2, r2], [0, r1], [r1, 0], [r1, r1], [0, r2], [r2, 0], [r1, r2], [r2, r1]])
    for action in actions:
        rpm1, rpm2 = action
        omega1 = rpm1 * (np.pi / 30)
        omega2 = rpm2 * (np.pi / 30)

        v = R_over_2 * (omega1 + omega2)
        omega = R_over_L * (omega2 - omega1)
        t = 0
        a = Xi
        b = Yi
        x, y, theta = Xi, Yi, Thetai * np.pi / 180
        cost = 0
        flag = 0
        i_n = []
        i_n2 = []
        while True:
            if t >= 1.0:
                break
            theta = theta + (omega * dt)

            cos_theta_dt = np.cos(theta) * dt
            sin_theta_dt = np.sin(theta) * dt

            a = x
            b = y
            x = x + v * cos_theta_dt
            y = y + v * sin_theta_dt
            i_n.append((int(math.floor(x)), int(math.floor(y))))
            i_n2.append((int(math.floor(a)), int(math.floor(b))))

            if is_valid_neighbor((x, y, 0, 0, 0)) == False:
                flag = 1
                break
            t += dt
            if rpm1 == rpm2:
                cost += euclidean_distance((a, b), (x, y))
            else:
                cost += euclidean_distance((a, b), (x, y))

        if flag == 0:
            for i in range(len(i_n)):
                cv2.line(obstacle_map, i_n2[i], i_n[i], (0, 0, 0), 1)
        neighbor = (round(x, 3), round(y, 3), round(np.degrees(theta) % 360, 3), round(cost, 3), (action[0],
action[1]))

        if is_valid_neighbor(neighbor) and flag == 0:
            if print_interval%1000 == 0:
                cv2.imshow("Shortest Path", obstacle_map)
                out.write(obstacle_map)
                cv2.waitKey(1)
            neighbors_l.append(neighbor)


    return neighbors_l


# Function to draw a line on the image
def draw_line(img, start_point, end_point, color, thickness):
    """
    Draw a line on the given image from the start point to the end point.

    Parameters:
    img (numpy.ndarray): The image on which the line will be drawn.
    start_point (tuple): The coordinates of the start point of the line (x, y).
    end_point (tuple): The coordinates of the end point of the line (x, y).
    color (tuple): The color of the line in BGR format (blue, green, red).
```

```
    thickness (int): The thickness of the line.

    Returns:
    None
    """
    cv2.line(img, start_point, end_point, color, thickness)

# Function to draw obstacles using half-plane equations
def draw_obstacles(obstacle_map, obstacles):
    """
    Draw obstacles on the given obstacle map.

    Parameters:
    - obstacle_map: The map on which obstacles will be drawn.
    - obstacles: A list of dictionaries representing the obstacles. Each dictionary should have the following
keys:
        - 'vertices': A list of vertices (points) that define the shape of the obstacle.
        - 'color' (optional): The color of the obstacle. Default is black.
        - 'thickness' (optional): The thickness of the lines used to draw the obstacle. Default is 1.

    Returns:
    None
    """

    for obstacle in obstacles:
        shape = obstacle.get('shape')

        if shape == 'rectangle':
            color = obstacle.get('color', (0, 0, 0))  # Default color is black
            thickness = obstacle.get('thickness', 1)  # Default thickness is 1
            vertices = obstacle['vertices']
            for i in range(len(vertices)):
                draw_line(obstacle_map, map_to_bottom_left(vertices[i]), map_to_bottom_left(vertices[(i + 1) %
len(vertices)]), color, thickness)

            cv2.fillPoly(obstacle_map, np.array([[map_to_bottom_left(point) for point in vertices]]), color)

        if shape == 'circle':
            vertices = obstacle['vertices']
            center, radius = vertices[0], vertices[1]
            color = obstacle.get('color', (0, 0, 0))  # Default color is black
            thickness = obstacle.get('thickness', -1)  # Default thickness is 1

            cv2.circle(obstacle_map, map_to_bottom_left(center), radius, color, thickness)


# Function to calculate the Euclidean distance between two points
def euclidean_distance(p1, p2):
    """
    Calculates the Euclidean distance between two points in a two-dimensional space.

    Parameters:
        p1 (tuple): The coordinates of the first point (x1, y1).
        p2 (tuple): The coordinates of the second point (x2, y2).

    Returns:
        float: The Euclidean distance between the two points.


    """

    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)





def ask_for_start_point(message, default=None):
    """
    Asks the user to input a point and validates its validity based on the obstacle map.

    Args:
        message (str): The message to display when asking for the point.
        default (tuple, optional): The default point to use if the user does not provide any input.
                                   Defaults to None.

    Returns:
        tuple: The validated point (x, y).
```

```
    Raises:
        ValueError: If no default value is provided and the user does not provide any input.
    """

    while True:
        user_input = input(f"{message} (default: {default[0]},{default[1]},{default[2]}): ")
        default[0], default[1] = default[0]/5, default[1]/5
        if user_input.strip() == "":
            if default is None:
                raise ValueError("No default value provided.")
            else:
                x, y,theta = default
                x, y = map_to_bottom_left((x, y))
        else:
            x, y, theta = map(int, user_input.split(','))
            x, y = map_to_bottom_left((x, y))

        if 0 <= x < width and 0 <= y < height and is_valid_neighbor((x,y,0,0,0)) and theta%30==0 and theta%30
== 0:
            return x, y,theta
        elif theta%30 !=0:
            print("Enter angle in multiples of 30 degrees")

        else:
            print("Point is invalid.")

# Function to ask for a point from the user
def ask_for_goal_point(message, default=None):
    """
    Asks the user to input a point and validates its validity based on the obstacle map.

    Args:
        message (str): The message to display when asking for the point.
        default (tuple, optional): The default point to use if the user does not provide any input.
                                   Defaults to None.

    Returns:
        tuple: The validated point (x, y).

    Raises:
        ValueError: If no default value is provided and the user does not provide any input.
    """
    while True:
        user_input = input(f"{message} (default: {default[0]},{default[1]}): ")
        default[0], default[1] = default[0]/5, default[1]/5
        if user_input.strip() == "":
            if default is None:
                raise ValueError("No default value provided.")
            else:
                x, y = default
                x, y = map_to_bottom_left((x, y))
        else:
            x, y = map(int, user_input.split(','))
            x, y = map_to_bottom_left((x, y))

        if 0 <= x < width and 0 <= y < height and is_valid_neighbor((x,y,0,0,0)):
            return x, y
        else:
            print("Point is invalid.")
def ask_for_rpm(message, default=None):
    """
    Asks the user to input a point and validates its validity based on the obstacle map.

    Args:
        message (str): The message to display when asking for the point.
        default (tuple, optional): The default point to use if the user does not provide any input.
                                   Defaults to None.

    Returns:
        tuple: The validated point (x, y).

    Raises:
        ValueError: If no default value is provided and the user does not provide any input.
    """
    while True:
        user_input = input(f"{message} (default: {default[0]},{default[1]}): ")
        if user_input.strip() == "":
            if default is None:
                raise ValueError("No default value provided.")
```

```python
            else:
                rpm1, rpm2 = default
        else:
            rpm1, rpm2 = map(int, user_input.split(','))

        if rpm1 > 0 and rpm2 > 0:
            return rpm1, rpm2
        else:
            print("Enter positive values for RPMs.")

def ask_clearence(message, default=None):
    # Function to ask for clearance from the user
    def ask_clearence(message, default=None):
        """
        Asks the user to input the clearance value and validates its validity.

        Args:
            message (str): The message to display when asking for the clearance.
            default (int, optional): The default clearance value to use if the user does not provide any input.
                                     Defaults to None.

        Returns:
            int: The validated clearance value.

        Raises:
            ValueError: If no default value is provided and the user does not provide any input.
        """
        while True:
            user_input = input(f"{message} (default: {default}): ")
            if not user_input:  # If the user just clicks enter, use the default value
                return default
            try:
                clearance = int(user_input)
                if clearance > 0:
                    return clearance
                else:
                    print("Enter a positive value for clearance")
            except ValueError:
                print("Invalid input. Please enter a number or press Enter for default.")
    print("Click ENTER for entering default value ")
    while True:
        user_input = user_input = input(f"{message} (default: {default}): ")
        if not user_input:  # If the user just clicks enter, use the default value
            return default
        try:
            clearance = int(user_input)
            if clearance > 0:
                return clearance
            else:
                print("Enter a positive value for clearance")
        except ValueError:
            print("Invalid input. Please enter a number or press Enter for default.")




def a_star(start, goal, obstacles, threshold, rpm1, rpm2):
    """
    A* algorithm implementation to find the shortest path from start to goal.

    Parameters:
    - start: Tuple representing the start node coordinates (x, y, theta).
    - goal: Tuple representing the goal node coordinates (x, y, theta).
    - obstacles: List of obstacles in the environment.
    - threshold: Threshold value for considering the goal reached.
    - step_size: Step size for generating neighboring nodes.

    Returns:
    - path: List of nodes representing the shortest path from start to goal.
    """

    frontier = PriorityQueue()
    frontier.put((0, start))

    cost_so_far = {(start[0],start[1]): 0}
    came_from = {(start[0],start[1]): None}

    while not frontier.empty():
```

```
        current_cost, current_node = frontier.get()


        if (current_node[0] > goal[0] - threshold and current_node[0] < goal[0] + threshold) and
(current_node[1] > goal[1] - threshold and current_node[1] < goal[1] + threshold):
                print("Goal Threshold reached orientation: " + "(" + str(current_node[0]) + "," + str(width -
current_node[1]) + "," + str(360 - current_node[2]) + ")")
                break


        for next_node_with_cost in get_neighbors(current_node, obstacles,rpm1,rpm2):

            next_node = next_node_with_cost[:3]
            current_node_int = (int(current_node[0]),int(current_node[1]))
            new_cost = cost_so_far[current_node_int] + next_node_with_cost[3]
            new_cost_check = new_cost + 10*euclidean_distance(next_node, goal)
            next_node_int = (int(next_node[0]),int(next_node[1]))

            if next_node_int not in cost_so_far or new_cost_check <
cost_so_far[(int(next_node[0]),int(next_node[1]))]:
                cost_so_far[(int(next_node[0]),int(next_node[1]))] = new_cost

                priority = round(new_cost + 10*euclidean_distance(next_node, goal), 3)  # A* uses f = g + h
                frontier.put((priority, next_node))

                came_from[(int(math.floor(next_node[0])),int(math.floor(next_node[1])))] =
(int(current_node[0]),int(current_node[1]),next_node_with_cost[4])        #exit(0)
    path = []
    start_int=(int(start[0]),int(start[1]))
    print("Start: ", start_int)
    print("Current Node Int: ", current_node_int)
    while True:
        current_node_int=(int(current_node[0]),int(current_node[1]))

        if current_node_int == start_int:
            break


path.append(((int(current_node[0]),int(current_node[1])),came_from[(int(current_node[0]),int(current_node[1]))]))

        current_node = came_from[(int(current_node[0]),int(current_node[1]))]

    path.reverse()
    print("Path: ", path)

    return path



# Define image dimensions
width = 1200
height = 400

# Create a blank image filled with white
obstacle_map = np.ones((height, width, 3), dtype=np.uint8) * 255


clearance = ask_clearence("Enter clearance in mm: ", (75))
clearance=int(clearance/5)
Robot_radius = 220/5

path_interval = 0




obstacles = [
    {'shape': 'rectangle', 'vertices': [(300-clearance, 200-clearance), (300-clearance, 400+clearance),
(350+clearance, 400+clearance), (350+clearance, 200-clearance)], 'color': (128, 128, 128), 'thickness': 1},  #
Rectangle obstacle 2
    {'shape': 'rectangle', 'vertices': [(300, 200), (300, 400), (350, 400), (350, 200)], 'color': (0, 0, 0),
'thickness': 1},  # Rectangle obstacle 2
    {'shape': 'rectangle', 'vertices': [(500-clearance, 0), (500-clearance, 200+clearance), (550+clearance,
200+clearance), (550+clearance, 0)], 'color': (128, 128, 128), 'thickness': 1},  # Rectangle obstacle 3
    {'shape': 'rectangle', 'vertices': [(500, 0), (500, 200), (550, 200), (550, 0)], 'color': (0, 0, 0),
'thickness': 1},  # Rectangle obstacle 4
    {'shape': 'rectangle', 'vertices': [(0, 0), (0, clearance), (1200, clearance), (1200, 0)], 'color': (128,
128, 128), 'thickness': 1},  # Rectangle obstacle 11
    {'shape': 'rectangle', 'vertices': [(0, 0), (0, 400), (clearance, 400), (clearance, 0)], 'color': (128,
```

```
128, 128), 'thickness': 1},  # Rectangle obstacle 12
    {'shape': 'rectangle', 'vertices': [(1200-clearance, 0), (1200-clearance, 400), (1200, 400), (1200, 0)],
'color': (128, 128, 128), 'thickness': 1},  # Rectangle obstacle 13
    {'shape': 'rectangle', 'vertices': [(0, 400-clearance), (0, 400), (1200, 400), (1200, 400-clearance)],
'color': (128, 128, 128), 'thickness': 1},  # Rectangle obstacle 14
    {'shape': 'circle', 'vertices': [(840, 240), 120+clearance], 'color': (128, 128, 128), 'thickness': -1},  #
Rectangle obstacle 14

    {'shape': 'circle', 'vertices': [(840, 240), 120], 'color': (0, 0, 0), 'thickness': -1},  # Rectangle
obstacle 14


]
print_interval=0
# Draw obstacles on the obstacle map
draw_obstacles(obstacle_map, obstacles)

# Ask for start and end points
start = ask_for_start_point("Enter start point (x, y,theta): ", [500, 1000,0])
goal = ask_for_goal_point("Enter goal point (x, y,theta): ", [5700, 1200])

rpm1,rpm2=ask_for_rpm("Enter RPM1 and RPM2 separated by comma: ", (50,100))

cv2.circle(obstacle_map, (int(goal[0]), int(goal[1])), 3, (0, 0, 255), -1)  # Explored nodes in green

fourcc = cv2.VideoWriter_fourcc(*'mp4v')

# Save the obstacle map with the shortest path as a video

out = cv2.VideoWriter('Shortest_Path.mp4', fourcc, 60.0, (width, height))

threshold =int( (0.5*Robot_radius))
cv2.circle(obstacle_map, (int(goal[0]), int(goal[1])), int(threshold), (0, 0, 255), 1)  # Explored nodes in
green
# Find the shortest path using Dijkstra's algorithm
shortest_path = a_star(start, goal, obstacles,threshold,rpm1,rpm2)

theta_draw = 0
# Mark the shortest path on the obstacle map
for point in shortest_path:
        R = 33/5  # Radius of the wheels
        L = 287/5  # Distance between the wheels (wheelbase)
        dt = 0.01  # Time step
        R_over_2 = R / 2
        R_over_L = R / L
        rpm1, rpm2 = point[1][2]
        omega1 = rpm1  * (np.pi / 30)
        #print ("omega1:", omega1)
        omega2 = rpm2 * (np.pi / 30)
        #print("omega2:", omega2)

        v = R_over_2 * (omega1 + omega2)
        omega = R_over_L * (omega2 - omega1)
        t=0
        a,b=(int(point[1][0]),int(point[1][1]))
        x,y,theta =a,b,theta_draw*np.pi/180
        cost=0
        while True:
            if t>=1.0:
                break

            cos_theta_dt = np.cos(theta) * dt
            sin_theta_dt = np.sin(theta) * dt

            a=x
            b=y
            x = x + v * cos_theta_dt
            y = y + v * sin_theta_dt
            theta =theta+ (omega * dt)

            t+=dt
            cv2.line(obstacle_map, (int(a), int(b)), (int(x), int(y)), (255, 0, 0), 2)
            if path_interval%10==0:
                cv2.imshow("Shortest Path", obstacle_map)
                out.write(obstacle_map)
                cv2.waitKey(1)
            path_interval+=1

#
```

```
        theta_draw= np.degrees(theta)%360

velocity_with_position = []
theta_draw = 0
f=0
for point in shortest_path:
        R = 33  # Radius of the wheels
        L = 287  # Distance between the wheels (wheelbase)
        dt = 0.01  # Time step
        R_over_2 = R / 2
        R_over_L = R / L
        rpm1, rpm2 = point[1][2][0], point[1][2][1]
        omega1 = rpm1  * (np.pi / 30)
        omega2 = rpm2 * (np.pi / 30)

        v = R_over_2 * (omega1 + omega2)
        omega = R_over_L * (omega2 - omega1)
        t=0
        a,b=((0),(0))
        if f==0:
            x,y,theta =a,b,theta_draw*np.pi/180
            f=1
        cost=0
        while True:
            if t>=1.0:
                break

            cos_theta_dt = np.cos(theta) * dt
            sin_theta_dt = np.sin(theta) * dt


            x = x + v * cos_theta_dt
            y = y + v * sin_theta_dt
            theta =theta+ (omega * dt)

            t+=dt


        velocity_with_position.append(((round(x/1000, 2), round((y)/1000, 2), round(theta, 2)), (round(v/1000,
2), round(-omega, 2))))
        theta_draw= np.degrees(theta)%360
print()
print('velocity_with_position:' , velocity_with_position)
# Display the obstacle map with the shortest path
cv2.imshow("Shortest Path", obstacle_map)
out.write(obstacle_map)
out.release()

cv2.waitKey(0)
cv2.destroyAllWindows()
```