```python
# github link: https://github.com/lekangtu123/ENPM661_PROJECT3PHASE1
import numpy as np
import cv2
import math
from queue import PriorityQueue

# Define the dimensions of the map
width = 1100
height = 300

# Check if a point (x,y) is within a rectangle defined by x_min, x_max, y_min, y
_max
def is_in_rectangle(x, y, x_min, x_max, y_min, y_max):
    return (x_min <= x <= x_max) and (y_min <= y <= y_max)

# Check if a point (x,y) is inside a polygon using the ray-casting algorithm
def is_in_polygon(x, y, polygon):
    inside = False
    n = len(polygon)
    for i in range(n):
        x1, y1 = polygon[i]
        x2, y2 = polygon[(i+1) % n]  # Wrap-around for last vertex
        if ((y1 > y) != (y2 > y)):
            x_intersect = x1 + (y - y1) * (x2 - x1) / (y2 - y1)
            if x_intersect > x:
                inside = not inside
    return inside

# Check if a point (x,y) lies inside a circle centered at (cx, cy) with radius r
def is_in_circle(x, y, cx, cy, r):
    return (x - cx)**2 + (y - cy)**2 <= r**2

# Check if a point (x,y) is within a filled elliptical arc
# The ellipse is rotated by angle_deg and the arc spans from start_deg to end_de
g
def is_in_ellipse_filled(x, y, cx, cy, rx, ry, angle_deg, start_deg, end_deg):
    dx = x - cx
    dy = y - cy
    theta = -math.radians(angle_deg)
    cosT = math.cos(theta)
    sinT = math.sin(theta)
    # Transform coordinates based on rotation
    x_ell = dx * cosT - dy * sinT
    y_ell = dx * sinT + dy * cosT
    # Check if point is inside the ellipse
    if (x_ell**2) / (rx * rx) + (y_ell**2) / (ry * ry) > 1.0:
        return False
    # Get the angle of the point in the ellipse’s coordinate system
    ang = math.degrees(math.atan2(y_ell, x_ell))
    if ang < 0:
        ang += 360
    s = start_deg % 360
    e = end_deg % 360
    if s <= e:
        return s <= ang <= e
    else:
        return (ang >= s) or (ang <= e)

# Define the obstacle region for the letter "E" using multiple rectangular secti
ons
def is_in_E(x, y):
    r1 = is_in_rectangle(x, y, 100, 120, 70, 250)
    r2 = is_in_rectangle(x, y, 120, 180, 230, 250)
    r3 = is_in_rectangle(x, y, 120, 160, 150, 170)
    r4 = is_in_rectangle(x, y, 120, 180, 70, 90)
    return r1 or r2 or r3 or r4

# Define the obstacle region for the letter "N"
def is_in_N(x, y):
```



```python
    r1 = is_in_rectangle(x, y, 250, 270, 70, 250)
    r2 = is_in_rectangle(x, y, 330, 350, 70, 250)
    # Define the diagonal section of the letter "N" as a polygon
    diag_poly = [(340, 70), (320, 70), (260, 250), (280, 250)]
    diag = is_in_polygon(x, y, diag_poly)
    return r1 or r2 or diag

# Define the obstacle region for the letter "P"
def is_in_P(x, y):
    rect_p = is_in_rectangle(x, y, 400, 420, 70, 250)
    # Define the curved part of "P" as an ellipse segment
    ell_p = is_in_ellipse_filled(x, y, 420, 210, 40, 40, 0, -90, 90)
    return rect_p or ell_p

# Define the obstacle region for the letter "M"
def is_in_M(x, y):
    r1 = is_in_rectangle(x, y, 510, 530, 70, 250)
    r2 = is_in_rectangle(x, y, 630, 650, 70, 250)
    # Define the left diagonal of "M" as a polygon
    left_diag = [(520, 250), (540, 250), (590, 70), (570, 70)]
    ld = is_in_polygon(x, y, left_diag)
    # Define the right diagonal of "M" as a polygon
    right_diag = [(590, 70), (570, 70), (620, 250), (640, 250)]
    rd = is_in_polygon(x, y, right_diag)
    return r1 or r2 or ld or rd

# Define the first part of the obstacle region for the digit "6"
def is_in_6_first(x, y):
    bottom_circle = is_in_circle(x, y, 745, 115, 50)
    radius_top_arc = 150
    center_top_arc = (845, 115)
    rect_size = 10
    top_arc_hit = False
    # Check if point is along the top arc by sampling points along the arc
    for deg in range(120, 180):
        rad = math.radians(deg)
        px = center_top_arc[0] + radius_top_arc * math.cos(rad)
        py = center_top_arc[1] + radius_top_arc * math.sin(rad)
        if (px - rect_size/2 <= x <= px + rect_size/2) and (py - rect_size/2 <=
y <= py + rect_size/2):
            top_arc_hit = True
            break
    return bottom_circle or top_arc_hit

# Define the second part of the obstacle region for the digit "6"
def is_in_6_second(x, y):
    bottom_circle = is_in_circle(x, y, 895, 115, 50)
    radius_top_arc = 150
    center_top_arc = (995, 115)
    rect_size = 10
    top_arc_hit = False
    # Check if point is along the top arc by sampling points along the arc
    for deg in range(120, 180):
        rad = math.radians(deg)
        px = center_top_arc[0] + radius_top_arc * math.cos(rad)
        py = center_top_arc[1] + radius_top_arc * math.sin(rad)
        if (px - rect_size/2 <= x <= px + rect_size/2) and (py - rect_size/2 <=
y <= py + rect_size/2):
            top_arc_hit = True
            break
    return bottom_circle or top_arc_hit

# Define the obstacle region for the digit "1" as a simple rectangle
def is_in_1(x, y):
    return is_in_rectangle(x, y, 1000, 1020, 70, 250)

# Combine all individual obstacle definitions to check if a point is within any
obstacle
def is_obstacle_enpm661(x, y):
```

```
    return (is_in_E(x, y) or is_in_N(x, y) or is_in_P(x, y) or is_in_M(x, y) or
            is_in_6_first(x, y) or is_in_6_second(x, y) or is_in_1(x, y))

# Generate a binary mask of obstacles for the defined map dimensions
def generate_enpm661_mask():
    mask = np.zeros((height, width), dtype=np.uint8)
    for py in range(height):
        for px in range(width):
            # Convert from pixel coordinate to model coordinate system (flip y)
            x_model = px
            y_model = height - 1 - py
            if is_obstacle_enpm661(x_model, y_model):
                mask[py, px] = 1
    return mask

# Check if the neighbor point is valid (within bounds and not colliding)
def is_valid_neighbor(point, collision_mask):
    x, y, _, _, _ = point
    c = int(round(x))
    r = int(round(y))
    if c < 0 or c >= width or r < 0 or r >= height:
        return False
    if collision_mask[r, c] == 1:
        return False
    return True

# Check if a point for user input is valid (within bounds and not colliding)
def is_valid_point_for_input(x, y, collision_mask):
    c = int(round(x))
    r = int(round(y))
    if c < 0 or c >= width or r < 0 or r >= height:
        return False
    if collision_mask[r, c] == 1:
        return False
    return True

# Map function for coordinate transformation; currently a placeholder
def map_to_bottom_left(point):
    return point

# Compute the Euclidean distance between two points (ignoring extra parameters)
def euclidean_distance(p1, p2):
    return math.dist(p1[:2], p2[:2])

# Get valid neighboring nodes based on robot kinematics using specified RPM acti
ons
def get_neighbors(point, collision_mask, drawing_map, Robot_radius, clearance, r
1, r2, out, print_interval):
    neighbors_l = []
    R = 33 / 5  # Wheel radius scaled to map
    L = 287 / 5 # Distance between wheels scaled to map
    dt = 0.01   # Time increment for simulation
    Xi, Yi, Thetai = point
    # Define 8 possible actions as combinations of RPM values
    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
        # Convert RPM to angular velocities (rad/s)
        omega1 = rpm1 * (math.pi / 30)
        omega2 = rpm2 * (math.pi / 30)
        # Compute linear velocity and angular velocity for the robot
        v = (R / 2) * (omega1 + omega2)
        omega = (R / L) * (omega2 - omega1)
        t = 0
        a = Xi
        b = Yi
        x, y, theta = Xi, Yi, Thetai * math.pi / 180  # Convert theta to radians
        cost = 0
```



```
        flag = 0
        i_n = []   # Store new points for drawing
        i_n2 = []  # Store previous points for drawing
        while True:
            if t >= 1.0:
                break
            theta += (omega * dt)
            cos_theta_dt = math.cos(theta) * dt
            sin_theta_dt = math.sin(theta) * dt
            a = x
            b = y
            x += v * cos_theta_dt
            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 the new point is not valid, break out of simulation
            if not is_valid_neighbor((x, y, 0, 0, 0), collision_mask):
                flag = 1
                break
            t += dt
            cost += math.dist((a, b), (x, y))
        # If path was clear, draw the line segment on the drawing map
        if flag == 0:
            for i in range(len(i_n)):
                cv2.line(drawing_map, i_n2[i], i_n[i], (0, 0, 0), 1)
        # Store neighbor node with (x,y,theta), cost, and action details
        neighbor = (round(x, 3), round(y, 3), round(math.degrees(theta) % 360, 3
), round(cost, 3), (action[0], action[1]))
        if not flag and is_valid_neighbor(neighbor, collision_mask):
            if print_interval % 1 == 0:
                cv2.imshow("Shortest Path", drawing_map)
                out.write(drawing_map)
                cv2.waitKey(1)
            neighbors_l.append(neighbor)
    return neighbors_l

# Request start point input from the user with a default value and validate it
def ask_for_start_point(message, default, collision_mask):
    while True:
        user_input = input(f"{message} (default: {default[0]},{default[1]},{default[2]}): ")
        if user_input.strip() == "":
            x, y, theta = default
        else:
            x, y, theta = map(int, user_input.split(','))
        if is_valid_point_for_input(x, y, collision_mask) 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.")

# Request goal point input from the user with a default value and validate it
def ask_for_goal_point(message, default, collision_mask):
    while True:
        user_input = input(f"{message} (default: {default[0]},{default[1]},{default[2]}): ")
        if user_input.strip() == "":
            x, y, theta = default
        else:
            x, y, theta = map(int, user_input.split(','))
        if is_valid_point_for_input(x, y, collision_mask) 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.")

# Request RPM values from the user with a default and ensure they are positive
def ask_for_rpm(message, default):
    while True:
```

```
        user_input = input(f"{message} (default: {default[0]},{default[1]}): ")
        if user_input.strip() == "":
            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.")

# Request clearance from the user with a default value and ensure it is positive
def ask_clearance(message, default):
    print("Click ENTER for entering default value ")
    while True:
        user_input = input(f"{message} (default: {default}): ")
        if not user_input:
            return default
        try:
            c = int(user_input)
            if c > 0:
                return c
            else:
                print("Enter a positive value for clearance")
        except ValueError:
            print("Invalid input. Please enter a number or press Enter for default.")

# A* search algorithm to find the shortest path from start to goal
def a_star(start, goal, collision_mask, drawing_map, threshold, rpm1, rpm2, Robo
t_radius, clearance, out):
    frontier = PriorityQueue()
    frontier.put((0, start))  # Start with the initial node
    cost_so_far = {(start[0], start[1]): 0}
    came_from = {(start[0], start[1]): None}
    print_interval = 0
    current_node = start
    # Process nodes until the frontier is empty
    while not frontier.empty():
        current_cost, current_node = frontier.get()
        # Check if the goal threshold is reached (goal vicinity)
        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: " + f"({current_node[0]},{width - current_node[1]
},{360 - current_node[2]})")
            break
        # Get all neighbors based on possible actions
        for nxt in get_neighbors(current_node, collision_mask, drawing_map, Robo
t_radius, clearance, rpm1, rpm2, out, print_interval):
            next_node = nxt[:3]
            current_node_int = (int(current_node[0]), int(current_node[1]))
            new_cost = cost_so_far[current_node_int] + nxt[3]
            new_cost_check = new_cost + 10 * euclidean_distance(next_node, goal)
            next_node_int = (int(next_node[0]), int(next_node[1]))
            # Check if this neighbor has not been visited or a cheaper path is f
ound
            if (next_node_int not in cost_so_far) or (new_cost_check < cost_so_f
ar[next_node_int]):
                cost_so_far[next_node_int] = new_cost
                priority = round(new_cost + 10 * euclidean_distance(next_node, g
oal), 3)
                frontier.put((priority, next_node))
                came_from[(int(math.floor(next_node[0])), int(math.floor(next_no
de[1])))] = (int(current_node[0]), int(current_node[1]), nxt[4])
        print_interval += 1
    # Backtrack from goal to start to construct the path
    path = []
    start_int = (int(start[0]), int(start[1]))
    print("Start:", start_int)
```



```
    current_node_int = (int(current_node[0]), int(current_node[1]))
    print("Current Node Int:", current_node_int)
    while True:
        if current_node_int == start_int:
            break
        path.append(((current_node_int[0], current_node_int[1]), came_from[curre
nt_node_int]))
        current_node_int = (came_from[current_node_int][0], came_from[current_no
de_int][1])
    path.reverse()
    print("Path:", path)
    return path

# Main section of the code
if __name__ == "__main__":
    # Generate original collision mask based on obstacles
    collision_mask_orig = generate_enpm661_mask()
    # Get clearance from user (in mm) and convert to pixels
    clearance_mm = ask_clearance("Enter clearance in mm:", 25)
    clearance_px = int(clearance_mm / 5)
    Robot_radius = 100 / 5  # Scale robot radius to pixel dimensions
    # Create a structuring element for dilation to account for clearance
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2 * clearance_px + 1,
 2 * clearance_px + 1))
    collision_mask = cv2.dilate(collision_mask_orig, kernel)
    # Create a white drawing map for visualization
    drawing_map = np.ones((height, width, 3), dtype=np.uint8) * 255
    drawing_map[collision_mask_orig == 1] = (0, 0, 0)  # Draw obstacles in black
    # Mark the clearance region in gray (areas where clearance is applied)
    clearance_region = (collision_mask == 1) & (collision_mask_orig == 0)
    drawing_map[clearance_region] = (128, 128, 128)
    # Get start and goal points from user input
    start = ask_for_start_point("Enter start point (x,y,theta):", [50, 168, 0], collision_m
ask)
    goal = ask_for_goal_point("Enter goal point (x,y,theta):", [820, 90, 0], collision_mas
k)
    rpm1, rpm2 = ask_for_rpm("Enter RPM1,RPM2:", (25, 50))
    # Mark the goal on the drawing map
    cv2.circle(drawing_map, (int(goal[0]), int(goal[1])), 3, (0, 0, 255), -1)
    # Set up video writer to record the shortest path visualization
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    out = cv2.VideoWriter("Shortest_Path.mp4", fourcc, 60.0, (width, height))
    threshold = 12
    cv2.circle(drawing_map, (int(goal[0]), int(goal[1])), threshold, (0, 0, 255)
, 1)
    # Run the A* algorithm to compute the shortest path
    shortest_path = a_star(start, goal, collision_mask, drawing_map, threshold,
rpm1, rpm2, Robot_radius, clearance_px, out)
    path_interval = 0
    theta_draw = 0
    # Animate the robot's movement along the shortest path
    for pt in shortest_path:
        R = 33 / 5
        L = 287 / 5
        dt = 0.01
        rpm1_, rpm2_ = pt[1][2]
        omega1 = rpm1_ * (math.pi / 30)
        omega2 = rpm2_ * (math.pi / 30)
        v = (R / 2) * (omega1 + omega2)
        omega = (R / L) * (omega2 - omega1)
        t = 0
        a, b = (pt[1][0], pt[1][1])
        x, y, theta = a, b, theta_draw * math.pi / 180
        while True:
            if t >= 1.0:
                break
            cos_theta_dt = math.cos(theta) * dt
            sin_theta_dt = math.sin(theta) * dt
            a = x
```

```
            b = y
            x += v * cos_theta_dt
            y += v * sin_theta_dt
            theta += (omega * dt)
            t += dt
            cv2.line(drawing_map, (int(a), int(b)), (int(x), int(y)), (255, 0, 0
), 2)
            if path_interval % 10 == 0:
                cv2.imshow("Shortest Path", drawing_map)
                out.write(drawing_map)
                cv2.waitKey(1)
            path_interval += 1
        theta_draw = math.degrees(theta) % 360
    # Calculate and print the velocity and position information along the path
    velocity_with_position = []
    theta_draw = 0
    f = 0
    for p in shortest_path:
        R = 33
        L = 287
        dt = 0.01
        rpm1_, rpm2_ = p[1][2]
        omega1 = rpm1_ * (math.pi / 30)
        omega2 = rpm2_ * (math.pi / 30)
        v = (R / 2) * (omega1 + omega2)
        omega = (R / L) * (omega2 - omega1)
        t = 0
        if f == 0:
            x, y, theta = 0, 0, theta_draw * math.pi / 180
            f = 1
        while True:
            if t >= 1.0:
                break
            x += v * math.cos(theta) * dt
            y += v * math.sin(theta) * dt
            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 = math.degrees(theta) % 360
    print()
    print("velocity_with_position:", velocity_with_position)
    # Final visualization of the shortest path
    cv2.imshow("Shortest Path", drawing_map)
    out.write(drawing_map)
    out.release()
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```