Project 3

ENPM661

Hamza Shah Khan | 119483152 hamzask@umd.edu
Vishnu Mandala | 119452608 | vishnum@umd.edu

Problem 1:

The robot is assumed to be a point robot (size/radius of the robot = 0)
The robot has a clearance of 5 mm
The workspace is an 8-connected space
The robot can move in 5 directions (60^0, 30^0, 0^0, -30^0, -60^0)
Step Size is the length of the vectors
Write 5 functions, one for each action. The output of each function is the state of a new node after taking the associated action.

Obstacles Coordinates -
Rectangle = (100,0)(100,100)(150,100)(150,0)
Rectangle = (100,250)(100,150)(150,100)(150,250)
Hexagon = (235.05,87.5)(235.05,162.5)(300,200)(364.95,162.5)(364.95,87.5)(300,50)
Triangle = (460,25)(460,225)(510,125)
The given map represents the space for clearance = 0 mm. For a clearance of 5 mm, the obstacles (including the walls) should be bloated by 5 mm distance on each side.
Use Half planes and semi-algebraic models to represent the obstacles in the map.
The equations must account for the 5 mm clearance of the robot.

Check the feasibility of all inputs/outputs
If the start and/or goal nodes are in the obstacle space, the user should be informed by a message and they should input the nodes again until valid values are entered.
The user input start and goal coordinates should be w.r.t. the origin
Implement A* Algorithm to find a path between the start and end point on a given map for a point robot (radius = 5; clearance = 5 mm).
Forward search using A* algorithm.
Consider Euclidean distance as a heuristic function.
Note:- Define a reasonable threshold value for the distance to the goal point. Due to the limited number of moves the robot cannot reach the exact goal location. So, use a threshold distance to check the goal.
Goal threshold(1.5 units radius)
Your code must output an animation of optimal path generation between start and goal point on the map. You need to show both the node exploration as well as the optimal path generated.
Using OpenCV/Matplotlib/Pygame/Tkinter (or any other graphical plotting library of your choice), create an empty canvas of size height=250 and width=600.
With the above equations, assign a different color for all the pixels within obstacles and walls. You may choose to represent the clearance pixels around the obstacle with another color.
To generate the graph consider the configuration space as a 3 dimensional space.
There are two methods to find the duplicate node:

1. In order to limit the number of the nodes, before adding the node make sure that the distance of the new node is greater than the threshold in x, y, and theta dimensions with respect to all existing nodes. This method is very slow. You should avoid using this method.
2. Use a matrix to store the information of the visited nodes. For threshold of 0.5 unit for x and y, and threshold of 30 degree for Theta in the given map, you should use a matrix V with 250/(threshold) x 600/(threshold) x 12 (i.e. 360/30) to store the visited regions information i.e. your matrix dimension will be (500x1200x12).
   Example:
   Set V[i][j][k]=0.
   If node1 = (3.2, 4.7, 0) visited → visited region: (3, 4.5, 0) V[6][9][0] =1
   If node2 = (10.2, 8.8, 30) visited → visited region: (10, 9, 30) V[20][18][1]=1
   If node3 = (10.1, 8.9, 30) visited → visited region: (10, 9, 30) V[20][18][1]=1
   (Here node2 and node 3 are duplicate nodes)

Before saving the nodes, check for the nodes that are within the obstacle space and ignore them.

To verify if a specific coordinate (node) is in the obstacle space, simply check its pixel color value in the created map
Once the goal node is popped, stop the search and backtrack to find the path.
Write a function that compares the current node with the goal node and return TRUE if they are equal. While generating each new node this function should be called
Write a function, when once the goal node is reached, using the child and parent relationship, backtracks from the goal node to start node and outputs all the intermediate nodes in the reversed order (start to goal).
Use the time library to print the runtime of your algorithm.

Full Code:

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

# Define the move functions
def move_p60(point, L):
    x = np.ceil(point[0] - L * math.cos(math.radians(60 + point[2])))
    y = np.ceil(point[1] - L * math.sin(math.radians(60 + point[2])))
    o = (point[2] - 60) % 360
    return (x, y, o), 1

def move_p30(point, L):
    x = np.ceil(point[0] - L * math.cos(math.radians(30 + point[2])))
    y = np.ceil(point[1] - L * math.sin(math.radians(30 + point[2])))
    o = (point[2] - 30) % 360
    return (x, y, o), 1

def move_forward(point, L):
    x = np.ceil(point[0] + L * math.cos(math.radians(point[2])))
    y = np.ceil(point[1] + L * math.sin(math.radians(point[2])))
    o = point[2]
    return (x, y, o), 1

def move_m30(point, L):
    x = np.ceil(point[0] + L * math.cos(math.radians(-30 + point[2])))
    y = np.ceil(point[1] + L * math.sin(math.radians(-30 + point[2])))
    o = (point[2] + 30) % 360
    return (x, y, o), 1

def move_m60(point, L):
    x = np.ceil(point[0] + L * math.cos(math.radians(-60 + point[2])))
    y = np.ceil(point[1] + L * math.sin(math.radians(-60 + point[2])))
    o = (point[2] + 60) % 360
    return (x, y, o), 1

map_width, map_height = 600, 250

def obstacles(clearance, radius):
    #Define the Obstacle Equations and Map Parameters
    eqns = {
        "Rectangle1": lambda x, y: 0 <= y <= 100 and 100 <= x <= 150,
        "Rectangle2": lambda x, y: 150 <= y <= 250 and 100 <= x <= 150,
        "Hexagon": lambda x, y: (75/2) * abs(x-300)/75 + 50 <= y <= 250 - (75/2) * abs(x-300)/75 - 50 and 235
<= x <= 365,
        "Triangle": lambda x, y: (200/100) * (x-460) + 25 <= y <= (-200/100) * (x-460) + 225 and 460 <= x <=
510
    }

    clearance = clearance + radius
    pixels = np.full((map_height, map_width, 3), 255, dtype=np.uint8)

    for i in range(map_height):
        for j in range(map_width):
            is_obstacle = any(eqn(j, i) for eqn in eqns.values())
```

```
            if is_obstacle:
                pixels[i, j] = [0, 0, 0]  # obstacle
            else:
                is_clearance = any(
                    eqn(x, y)
                    for eqn in eqns.values()
                    for y in range(i - clearance, i + clearance + 1)
                    for x in range(j - clearance, j + clearance + 1)
                    if (x - j)**2 + (y - i)**2 <= clearance**2
                )
                if i < clearance or i >= map_height - clearance or j < clearance or j >= map_width -
clearance:
                    pixels[i, j] = [192, 192, 192]  # boundary
                elif is_clearance:
                    pixels[i, j] = [192, 192, 192]  # clearance
                else:
                    pixels[i, j] = [255, 255, 255]  # free space
    return pixels

# Define a function to check if current node is in range
def is_in_range(node):
    x, y, o = node
    y = map_height - y - 1
    return 0 <= x < map_width and 0 <= y < map_height and (pixels[int(y), int(x)] == [255, 255, 255]).all()
and o%30 == 0

def is_valid_node(node, visited):
    if not is_in_range(node):
        return False  # out of range
    x, y, _ = node
    y = map_height - y - 1
    if not (pixels[int(y), int(x)] == [255, 255, 255]).all():
        return False  # in obstacle space
    # Check if the node is within threshold distance from any visited nodes
    threshold_x = 0.5
    threshold_y = 0.5
    threshold_theta = math.radians(30)
    for i in range(-1, 2):
        for j in range(-1, 2):
            for k in range(-1, 2):
                neighbor_node = (x + i * threshold_x, y + j * threshold_y, k * threshold_theta)
                if neighbor_node in visited:
                    return False  # too close to a visited node
    return True

# Define a function to check if current node is the goal node
def is_goal(current_node, goal_node):
    return np.sqrt((goal_node[0]-current_node[0])**2 + (goal_node[1]-current_node[1])**2) <= 1.5

# Define a function to find the optimal path
def backtrack_path(parents, start_node, goal_node):
    path, current_node = [goal_node], goal_node
    while current_node != start_node:
        path.append(current_node)
        current_node = parents[current_node]
    path.append(start_node)
    return path[::-1]

# Define a function to calculate the euclidean distance
def euclidean_distance(node1, node2):
    x1, y1, _ = node1
    x2, y2, _ = node2
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# Define the A* algorithm
def a_star(start_node, goal_node, display_animation=True):
    threshold = 0.5 # threshold distance
    rows = int(map_height / threshold)  # number of rows
    cols = int(map_width / threshold)   # number of columns
    angles = int(360 / 30)           # number of angles
```

```
    V = [[[False for _ in range(angles)] for _ in range(cols)] for _ in range(rows)]    # visited nodes
matrix

    open_list = PriorityQueue()
    closed_list = set()
    cost_to_come = {start_node: 0}
    cost_to_go = {start_node: euclidean_distance(start_node, goal_node)}
    cost = {start_node: cost_to_come[start_node] + cost_to_go[start_node]}
    parent = {start_node: None}
    open_list.put((cost[start_node], start_node))
    visited = set([start_node])
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter('animation.mp4', fourcc, 15.0, (map_width, map_height))

    while not open_list.empty():
        _, current_node = open_list.get()
        closed_list.add(current_node)
        x, y, theta = current_node
        V[int(y / threshold)][int(x / threshold)][int(theta / 30)] = True   # Mark current node as visited
        out.write(pixels)
        if display_animation:
            cv2.imshow('Explored', pixels)
            cv2.waitKey(1)
        # Check if current node is the goal node
        if is_goal(current_node, goal_node):
            approx_goal_node = current_node # Approximate goal node (within threshold distance)
            cost[goal_node] = cost [current_node]   # Cost of goal node
            path = backtrack_path(parent, start_node, approx_goal_node) # Backtrack the path
            if display_animation:
                for node in path:
                    x, y, _ = node
                    cv2.circle(pixels, (int(x), map_height - 1 - int(y)), 1, (0, 0, 255), thickness=-1)
                out.write(pixels)
                cv2.waitKey(0)
            print("Final Cost: ", cost[goal_node])
            out.release()
            cv2.destroyAllWindows()
            return path

        for move_func in [move_m30, move_m60, move_forward, move_p30, move_p60]:    # Iterate through all
possible moves
            new_node, move_cost = move_func(current_node, L)
            if is_valid_node(new_node, visited):    # Check if the node is valid
                i, j, k = int(new_node[1] / threshold), int(new_node[0] / threshold), int(new_node[2] / 30) #
Get the index of the node in the 3D array
                if not V[i][j][k]:  # Check if the node is in closed list
                    new_cost_to_come = cost_to_come[current_node] + move_cost
                    new_cost_to_go = euclidean_distance(new_node, goal_node)
                    new_cost = new_cost_to_come + new_cost_to_go    # Update cost
                    if new_node not in cost_to_come or new_cost_to_come < cost_to_come[new_node]:
                        cost_to_come[new_node] = new_cost_to_come   # Update cost to come
                        cost_to_go[new_node] = new_cost_to_go    # Update cost to go
                        cost[new_node] = new_cost   # Update cost
                        parent[new_node] = current_node  # Update parent
                        open_list.put((new_cost, new_node)) # Add to open list
                        visited.add(new_node)   # Add to visited list
                        # Draw vector from current_node to new_node
                        cv2.line(pixels, (int(current_node[0]), map_height - 1 - int(current_node[1])),
(int(new_node[0]), map_height - 1 - int(new_node[1])), (255, 0, 0), thickness=1)

        if cv2.waitKey(1) == ord('q'):
            cv2.destroyAllWindows()
            break

    out.release()
    cv2.destroyAllWindows()
    return None

# Get valid start and goal nodes from user input
while True:
    clearance = int(input("\nEnter the clearance: "))
```

```
    radius = int(input("Enter the radius: "))
    pixels = obstacles(clearance, radius)
    start_node = tuple(map(int, input("Enter the start node (in the format 'x y o'). O should be given in
degrees and as a multiple of 30 i.e. {.., -60, -30, 0, 30, 60, ..}: ").split()))
    if not is_in_range(start_node):
        print("Error: Start node is in the obstacle space, clearance area, out of bounds or orientation was
not given in the required format. Please input a valid node.")
        continue
    goal_node = tuple(map(int, input("Enter the goal node (in the format 'x y o'). O should be given in
degrees and as a multiple of 30 i.e. {.., -60, -30, 0, 30, 60, ..}: ").split()))
    if not is_in_range(goal_node):
        print("Error: Goal node is in the obstacle space, clearance area, out of bounds or orientation was
not given in the required format. Please input a valid node.")
        continue
    L = int(input("Enter the step size of the robot in the range 1-10: "))
    if L < 1 or L > 10:
        print("Error: Step size should be in the range 1-10. Please input a valid step size.")
        continue
    break

# Run A* algorithm
start_time = time.time()
path = a_star(start_node, goal_node)
if path is None:
    print("\nError: No path found.")
else:
    print("\nGoal Node Reached!\nShortest Path: ", path, "\n")
end_time = time.time()
print("Runtime:", end_time - start_time, "seconds\n")
```

Final Output:

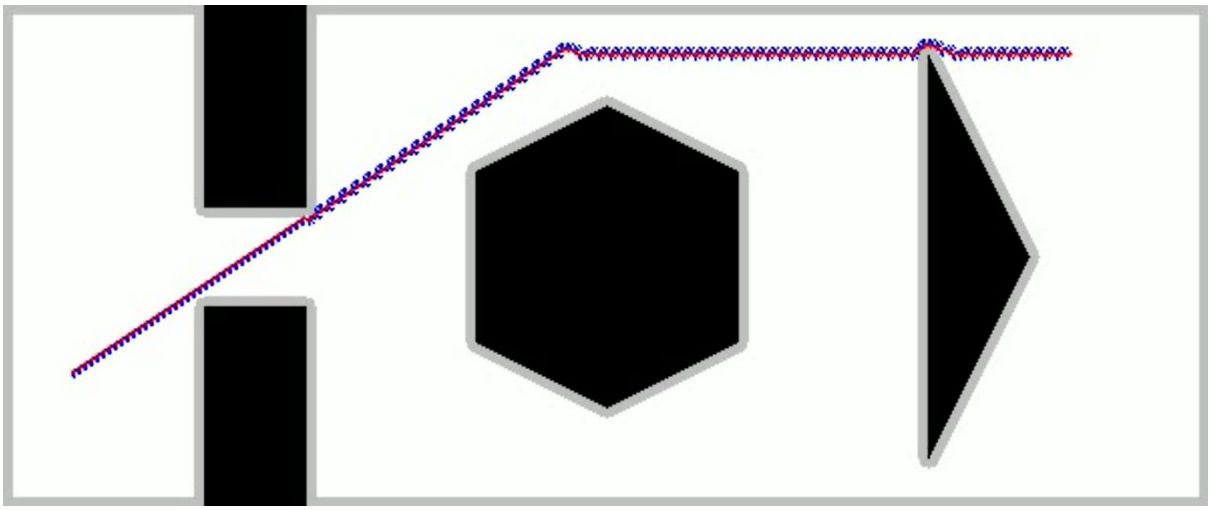

Figure 1 Optimal Path

Terminal Output:

```
PS C:\Users\manda\OneDrive - University of Maryland\Planning For Autonomous Robots\Projects\Project 3> &
"C:/Program Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Planning For Autonomous
Robots/Projects/Project 3/proj3_phase1.py"

Enter the clearance: 3
Enter the radius: 2
Enter the start node (in the format 'x y o'). O should be given in degrees and as a multiple of 30 i.e. {.., -60, -
30, 0, 30, 60, ..}: 35 67 30
Enter the goal node (in the format 'x y o'). O should be given in degrees and as a multiple of 30 i.e. {.., -60, -
30, 0, 30, 60, ..}: 530 226 60
Enter the step size of the robot in the range 1-10: 3
```

```
Final Cost:  166.0

Goal Node Reached!
Shortest Path:  [(35, 67, 30), (38.0, 69.0, 30), (41.0, 71.0, 30), (44.0, 73.0, 30), (47.0, 75.0, 30), (50.0, 77.0,
30), (53.0, 79.0, 30), (56.0, 81.0, 30), (59.0, 83.0, 30), (62.0, 85.0, 30), (65.0, 87.0, 30), (68.0, 89.0, 30),
(71.0, 91.0, 30), (74.0, 93.0, 30), (77.0, 95.0, 30), (80.0, 97.0, 30), (83.0, 99.0, 30), (86.0, 101.0, 30), (89.0,
103.0, 30), (92.0, 105.0, 30), (95.0, 107.0, 30), (98.0, 109.0, 30), (101.0, 111.0, 30), (104.0, 113.0, 30),
(107.0, 115.0, 30), (110.0, 117.0, 30), (113.0, 119.0, 30), (116.0, 121.0, 30), (119.0, 123.0, 30), (122.0, 125.0,
30), (125.0, 127.0, 30), (128.0, 129.0, 30), (131.0, 131.0, 30), (134.0, 133.0, 30), (137.0, 135.0, 30), (140.0,
137.0, 30), (143.0, 139.0, 30), (146.0, 141.0, 30), (149.0, 143.0, 30), (152.0, 143.0, 60), (155.0, 145.0, 90),
(158.0, 147.0, 150), (161.0, 149.0, 90), (164.0, 151.0, 150), (167.0, 153.0, 90), (170.0, 155.0, 150), (173.0,
157.0, 90), (176.0, 159.0, 150), (179.0, 161.0, 90), (182.0, 163.0, 150), (185.0, 165.0, 90), (188.0, 167.0, 150),
(191.0, 169.0, 90), (194.0, 171.0, 150), (197.0, 173.0, 90), (200.0, 175.0, 150), (203.0, 177.0, 90), (206.0,
179.0, 150), (209.0, 181.0, 90), (212.0, 183.0, 150), (215.0, 185.0, 90), (218.0, 187.0, 150), (221.0, 189.0, 90),
(224.0, 191.0, 150), (227.0, 193.0, 90), (230.0, 195.0, 150), (233.0, 197.0, 90), (236.0, 199.0, 150), (239.0,
201.0, 90), (242.0, 203.0, 150), (245.0, 205.0, 90), (248.0, 207.0, 150), (251.0, 209.0, 90), (254.0, 211.0, 150),
(257.0, 213.0, 90), (260.0, 215.0, 150), (263.0, 217.0, 90), (266.0, 219.0, 150), (269.0, 221.0, 90), (272.0,
223.0, 150), (275.0, 225.0, 90), (278.0, 227.0, 150), (281.0, 227.0, 120), (284.0, 226.0, 90), (287.0, 225.0, 30),
(290.0, 225.0, 60), (293.0, 225.0, 120), (296.0, 225.0, 60), (299.0, 225.0, 120), (302.0, 225.0, 60), (305.0,
225.0, 120), (308.0, 225.0, 60), (311.0, 225.0, 120), (314.0, 225.0, 60), (317.0, 225.0, 120), (320.0, 225.0, 60),
(323.0, 225.0, 120), (326.0, 225.0, 60), (329.0, 225.0, 120), (332.0, 225.0, 60), (335.0, 225.0, 120), (338.0,
225.0, 60), (341.0, 225.0, 120), (344.0, 225.0, 60), (347.0, 225.0, 120), (350.0, 225.0, 60), (353.0, 225.0, 120),
(356.0, 225.0, 60), (359.0, 225.0, 120), (362.0, 225.0, 60), (365.0, 225.0, 120), (368.0, 225.0, 60), (371.0,
225.0, 120), (374.0, 225.0, 60), (377.0, 225.0, 120), (380.0, 225.0, 60), (383.0, 225.0, 120), (386.0, 225.0, 60),
(389.0, 225.0, 120), (392.0, 225.0, 60), (395.0, 225.0, 120), (398.0, 225.0, 60), (401.0, 225.0, 120), (404.0,
225.0, 60), (407.0, 225.0, 120), (410.0, 225.0, 60), (413.0, 225.0, 120), (416.0, 225.0, 60), (419.0, 225.0, 120),
(422.0, 225.0, 60), (425.0, 225.0, 120), (428.0, 225.0, 60), (431.0, 225.0, 120), (434.0, 225.0, 60), (437.0,
225.0, 120), (440.0, 225.0, 60), (443.0, 225.0, 120), (446.0, 225.0, 60), (449.0, 225.0, 120), (452.0, 225.0, 60),
(455.0, 227.0, 90), (458.0, 229.0, 150), (461.0, 229.0, 120), (464.0, 228.0, 90), (467.0, 227.0, 30), (470.0,
226.0, 90), (473.0, 225.0, 30), (476.0, 225.0, 60), (479.0, 225.0, 120), (482.0, 225.0, 60), (485.0, 225.0, 120),
(488.0, 225.0, 60), (491.0, 225.0, 120), (494.0, 225.0, 60), (497.0, 225.0, 120), (500.0, 225.0, 60), (503.0,
225.0, 120), (506.0, 225.0, 60), (509.0, 225.0, 120), (512.0, 225.0, 60), (515.0, 225.0, 120), (518.0, 225.0, 60),
(521.0, 225.0, 120), (524.0, 225.0, 60), (527.0, 225.0, 120), (530.0, 225.0, 60), (530.0, 225.0, 60)]

Runtime: 4.806797742843628 seconds
```