Project 2

ENPM661

Vishnu Mandala
119452608

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 UP, DOWN, LEFT, RIGHT [Cost: 1.0]
It can also move diagonally between UP-LEFT, UP-RIGHT, DOWN-LEFT, and DOWN-RIGHT [Cost: 1.4]
Actions Set = {(1,0), (-1,0), (0,1), (0,-1), (1,1), (-1,1), (1,-1), (-1,-1)}
Write 8 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 Dijkstra's Algorithm to find a path between the start and end point on a given map for a point robot (radius = 0; clearance = 5 mm).
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.
Generate the graph using the action set for an 8-connected space
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
from queue import PriorityQueue

# Define the move functions
move_up = lambda node: ((node[0] - 1, node[1]), 1)
move_down = lambda node: ((node[0] + 1, node[1]), 1)
move_left = lambda node: ((node[0], node[1] - 1), 1)
move_right = lambda node: ((node[0], node[1] + 1), 1)
move_up_left = lambda node: ((node[0] - 1, node[1] - 1), np.sqrt(2))
move_up_right = lambda node: ((node[0] - 1, node[1] + 1), np.sqrt(2))
```

```
move_down_left = lambda node: ((node[0] + 1, node[1] - 1), np.sqrt(2))
move_down_right = lambda node: ((node[0] + 1, node[1] + 1), np.sqrt(2))

#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 225 <= x <=
375,
    "Triangle": lambda x, y: (200/100) * (x-460) + 25 <= y <= (-200/100) * (x-460) + 225 and 460 <= x <= 510
}

map_width, map_height, clearance = 600, 250, 5
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(max(i-clearance, 0), min(i+clearance+1, map_height))
                for x in range(max(j-clearance, 0), min(j+clearance+1, map_width))
            )
            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

# Define the start and goal nodes
def is_valid_node(node):
    x, y = node
    y = map_height - y - 1
    return 0 <= x < map_width and 0 <= y < map_height and (pixels[y, x] == [255, 255, 255]).all()

# Define a function to check if current node is the goal node
def is_goal(current_node, goal_node):
    return current_node == goal_node

# Define a function to find the optimal path
def backtrack_path(parents, start_node, goal_node, pixels):
    path, current_node = [goal_node], goal_node
    while current_node != start_node:
        path.append(current_node)
        current_node = parents[current_node]
        pixels[map_height-1-current_node[1], current_node[0]] = (0, 255, 0)  # Mark path (in green)
    path.append(start_node)
    return path[::-1]

# Define the Dijkstra algorithm
def dijkstra(start_node, goal_node, display_animation=True):
    open_list = PriorityQueue()
    closed_list = set()
    cost_to_come = {start_node: 0}
    cost = {start_node: 0}
    parent = {start_node: None}
    open_list.put((0, start_node))
    visited = set([start_node])
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter('animation.mp4', fourcc, 15.0, (map_width, map_height))
    # Loop until the open list is empty
    while not open_list.empty():
        _, current_node = open_list.get()
        closed_list.add(current_node)
        pixels[map_height -1- current_node[1], current_node[0]] = (255, 0, 0)  # Mark current node as visited (in
blue)
```

```
        out.write(pixels)
        if display_animation:
            cv2.imshow('Explored', pixels)
            cv2.waitKey(1)
        # Check if the current node is the goal node
        if is_goal(current_node, goal_node):
            path = backtrack_path(parent, start_node, goal_node, pixels)
            if display_animation:
                cv2.imshow('Optimal Path', pixels)
                cv2.waitKey(0)
            print("Final Cost: ", cost[goal_node])
            out.release()
            cv2.destroyAllWindows()
            return path
        # Generate the children of the current node
        for move_func in [move_up, move_down, move_left, move_right, move_up_left, move_up_right, move_down_left,
move_down_right]:
            new_node, move_cost = move_func(current_node)
            # Check if the new node is valid and not already visited
            if is_valid_node(new_node) and new_node not in closed_list:
                new_cost_to_come = cost_to_come[current_node] + move_cost
                # Check if the new node is already in the open list
                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
                    cost[new_node] = new_cost_to_come
                    parent[new_node] = current_node
                    open_list.put((new_cost_to_come, new_node))
                    visited.add(new_node)
        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:
    start_node = tuple(map(int, input("\nEnter the start node (in the format 'x y'): ").split()))
    if not is_valid_node(start_node):
        print("Error: Start node is in the obstacle space, clearance area or out of bounds. Please input a valid
node.")
        continue
    goal_node = tuple(map(int, input("Enter the goal node (in the format 'x y'): ").split()))
    if not is_valid_node(goal_node):
        print("Error: Goal node is in the obstacle space, clearance area or out of bounds. Please input a valid
node.")
        continue
    break

# Run Dijkstra's algorithm
start_time = time.time()
path = dijkstra(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:

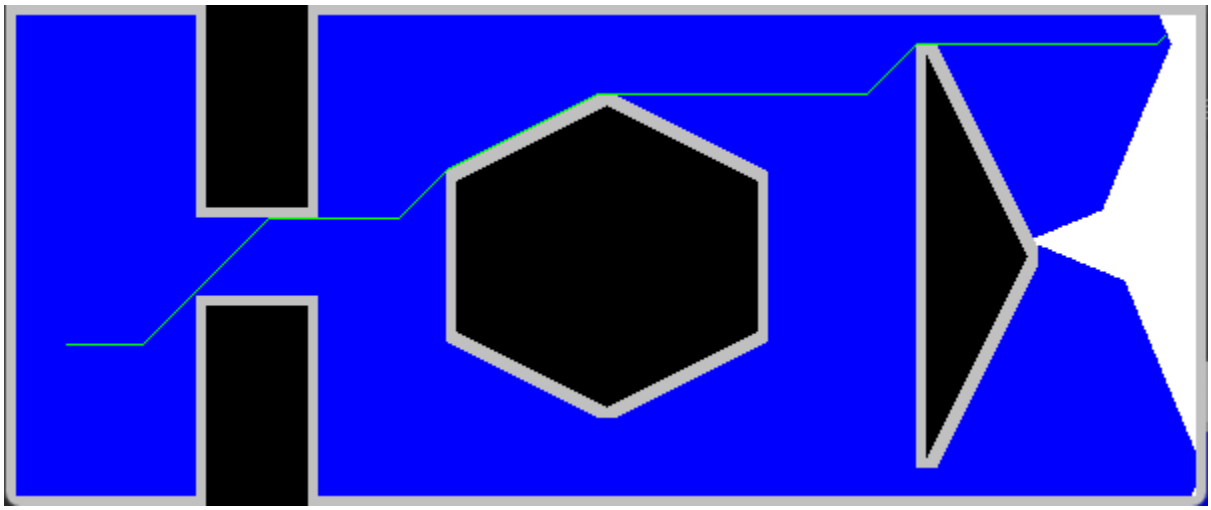

Figure 1 Optimal Path

Terminal Output:

```
PS C:\Users\manda\OneDrive - University of Maryland\Planning For Autonomous Robots\Projects\Project 2> &
"C:/Program Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of Maryland/Planning For Autonomous
Robots/Projects/Project 2/test.py"

Enter the start node (in the format 'x y'): 30 80
Enter the goal node (in the format 'x y'): 580 235
Final Cost:  614.2031021678306

Goal Node Reached!
Shortest Path: [(30, 80), (31, 80), (32, 80), (33, 80), (34, 80), (35, 80), (36, 80), (37, 80), (38, 80), (39, 80),
(40, 80), (41, 80), (42, 80), (43, 80), (44, 80), (45, 80), (46, 80), (47, 80), (48, 80), (49, 80), (50, 80), (51,
80), (52, 80), (53, 80), (54, 80), (55, 80), (56, 80), (57, 80), (58, 80), (59, 80), (60, 80), (61, 80), (62, 80),
(63, 80), (64, 80), (65, 80), (66, 80), (67, 80), (68, 80), (69, 81), (70, 82), (71, 83), (72, 84), (73, 85), (74,
86), (75, 87), (76, 88), (77, 89), (78, 90), (79, 91), (80, 92), (81, 93), (82, 94), (83, 95), (84, 96), (85, 97),
(86, 98), (87, 99), (88, 100), (89, 101), (90, 102), (91, 103), (92, 104), (93, 105), (94, 106), (95, 107), (96,
108), (97, 109), (98, 110), (99, 111), (100, 112), (101, 113), (102, 114), (103, 115), (104, 116), (105, 117),
(106, 118), (107, 119), (108, 120), (109, 121), (110, 122), (111, 123), (112, 124), (113, 125), (114, 126), (115,
127), (116, 128), (117, 129), (118, 130), (119, 131), (120, 132), (121, 133), (122, 134), (123, 135), (124, 136),
(125, 137), (126, 138), (127, 139), (128, 140), (129, 141), (130, 142), (131, 143), (132, 143), (133, 143), (134,
143), (135, 143), (136, 143), (137, 143), (138, 143), (139, 143), (140, 143), (141, 143), (142, 143), (143, 143),
(144, 143), (145, 143), (146, 143), (147, 143), (148, 143), (149, 143), (150, 143), (151, 143), (152, 143), (153,
143), (154, 143), (155, 143), (156, 143), (157, 143), (158, 143), (159, 143), (160, 143), (161, 143), (162, 143),
(163, 143), (164, 143), (165, 143), (166, 143), (167, 143), (168, 143), (169, 143), (170, 143), (171, 143), (172,
143), (173, 143), (174, 143), (175, 143), (176, 143), (177, 143), (178, 143), (179, 143), (180, 143), (181, 143),
(182, 143), (183, 143), (184, 143), (185, 143), (186, 143), (187, 143), (188, 143), (189, 143), (190, 143), (191,
143), (192, 143), (193, 143), (194, 143), (195, 143), (196, 143), (197, 144), (198, 145), (199, 146), (200, 147),
(201, 148), (202, 149), (203, 150), (204, 151), (205, 152), (206, 153), (207, 154), (208, 155), (209, 156), (210,
157), (211, 158), (212, 159), (213, 160), (214, 161), (215, 162), (216, 163), (217, 164), (218, 165), (219, 166),
(220, 167), (221, 168), (222, 168), (223, 169), (224, 169), (225, 170), (226, 170), (227, 171), (228, 171), (229,
172), (230, 172), (231, 173), (232, 173), (233, 174), (234, 174), (235, 175), (236, 175), (237, 176), (238, 176),
(239, 177), (240, 177), (241, 178), (242, 178), (243, 179), (244, 179), (245, 180), (246, 180), (247, 181), (248,
181), (249, 182), (250, 182), (251, 183), (252, 183), (253, 184), (254, 184), (255, 185), (256, 185), (257, 186),
(258, 186), (259, 187), (260, 187), (261, 188), (262, 188), (263, 189), (264, 189), (265, 190), (266, 190), (267,
191), (268, 191), (269, 192), (270, 192), (271, 193), (272, 193), (273, 194), (274, 194), (275, 195), (276, 195),
(277, 196), (278, 196), (279, 197), (280, 197), (281, 198), (282, 198), (283, 199), (284, 199), (285, 200), (286,
200), (287, 201), (288, 201), (289, 202), (290, 202), (291, 203), (292, 203), (293, 204), (294, 204), (295, 205),
(296, 205), (297, 205), (298, 205), (299, 205), (300, 205), (301, 205), (302, 205), (303, 205), (304, 205), (305,
205), (306, 205), (307, 205), (308, 205), (309, 205), (310, 205), (311, 205), (312, 205), (313, 205), (314, 205),
(315, 205), (316, 205), (317, 205), (318, 205), (319, 205), (320, 205), (321, 205), (322, 205), (323, 205), (324,
205), (325, 205), (326, 205), (327, 205), (328, 205), (329, 205), (330, 205), (331, 205), (332, 205), (333, 205),
(334, 205), (335, 205), (336, 205), (337, 205), (338, 205), (339, 205), (340, 205), (341, 205), (342, 205), (343,
205), (344, 205), (345, 205), (346, 205), (347, 205), (348, 205), (349, 205), (350, 205), (351, 205), (352, 205),
(353, 205), (354, 205), (355, 205), (356, 205), (357, 205), (358, 205), (359, 205), (360, 205), (361, 205), (362,
205), (363, 205), (364, 205), (365, 205), (366, 205), (367, 205), (368, 205), (369, 205), (370, 205), (371, 205),
```

```
(372, 205), (373, 205), (374, 205), (375, 205), (376, 205), (377, 205), (378, 205), (379, 205), (380, 205), (381,
205), (382, 205), (383, 205), (384, 205), (385, 205), (386, 205), (387, 205), (388, 205), (389, 205), (390, 205),
(391, 205), (392, 205), (393, 205), (394, 205), (395, 205), (396, 205), (397, 205), (398, 205), (399, 205), (400,
205), (401, 205), (402, 205), (403, 205), (404, 205), (405, 205), (406, 205), (407, 205), (408, 205), (409, 205),
(410, 205), (411, 205), (412, 205), (413, 205), (414, 205), (415, 205), (416, 205), (417, 205), (418, 205), (419,
205), (420, 205), (421, 205), (422, 205), (423, 205), (424, 205), (425, 205), (426, 205), (427, 205), (428, 205),
(429, 205), (430, 205), (431, 206), (432, 207), (433, 208), (434, 209), (435, 210), (436, 211), (437, 212), (438,
213), (439, 214), (440, 215), (441, 216), (442, 217), (443, 218), (444, 219), (445, 220), (446, 221), (447, 222),
(448, 223), (449, 224), (450, 225), (451, 226), (452, 227), (453, 228), (454, 229), (455, 230), (456, 230), (457,
230), (458, 230), (459, 230), (460, 230), (461, 230), (462, 230), (463, 230), (464, 230), (465, 230), (466, 230),
(467, 230), (468, 230), (469, 230), (470, 230), (471, 230), (472, 230), (473, 230), (474, 230), (475, 230), (476,
230), (477, 230), (478, 230), (479, 230), (480, 230), (481, 230), (482, 230), (483, 230), (484, 230), (485, 230),
(486, 230), (487, 230), (488, 230), (489, 230), (490, 230), (491, 230), (492, 230), (493, 230), (494, 230), (495,
230), (496, 230), (497, 230), (498, 230), (499, 230), (500, 230), (501, 230), (502, 230), (503, 230), (504, 230),
(505, 230), (506, 230), (507, 230), (508, 230), (509, 230), (510, 230), (511, 230), (512, 230), (513, 230), (514,
230), (515, 230), (516, 230), (517, 230), (518, 230), (519, 230), (520, 230), (521, 230), (522, 230), (523, 230),
(524, 230), (525, 230), (526, 230), (527, 230), (528, 230), (529, 230), (530, 230), (531, 230), (532, 230), (533,
230), (534, 230), (535, 230), (536, 230), (537, 230), (538, 230), (539, 230), (540, 230), (541, 230), (542, 230),
(543, 230), (544, 230), (545, 230), (546, 230), (547, 230), (548, 230), (549, 230), (550, 230), (551, 230), (552,
230), (553, 230), (554, 230), (555, 230), (556, 230), (557, 230), (558, 230), (559, 230), (560, 230), (561, 230),
(562, 230), (563, 230), (564, 230), (565, 230), (566, 230), (567, 230), (568, 230), (569, 230), (570, 230), (571,
230), (572, 230), (573, 230), (574, 230), (575, 230), (576, 231), (577, 232), (578, 233), (579, 234), (580, 235),
(580, 235)]

Runtime: 3170.4209847450256 seconds
```