Project 3 Phase 2

ENPM661

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

Part 1:

Navigate a differential drive robot (TurtleBot3 Burger) in the given map environment from a given start point to a given goal point.

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 shown in the map.

Consider differential drive constraints while implementing the A* algorithm, with 8 set of action space.

Your code must take the following values from the user:

Start Point Coordinates (3-element vector): (Xs, Ys, Θs).

Θs - The orientation of the Robot at the start point.

Goal Point Coordinates (2-element vector): (Xg, Yg)

To simplify the path explored, final orientation input is not required.

Wheel RPMs (2-element vector) => (RPM1, RPM2)

RPM: Revolutions per Minute

2 possible values for the wheel RPMs

Clearance (in mm)

Your code must define the following parameters of the TurtleBot 3 Burger robot. These parameters are NOT user defined but rather are constants which can be defined in your Python code.

Robot Wheel Radius (R) - From the robot's datasheet/documentation

Robot Radius (r) - From the robot's datasheet/documentation (This needs to be added to the clearance value to create the obstacle map.)

Wheel Distance (L) - From the robot's datasheet/documentation

Let the two RPMs provided by the user be RPM1 and RPM2. Then the action space consisting of 8 possible actions for the A* algorithm is:

1. [0, RPM1]
2. [RPM1, 0]
3. [RPM1, RPM1]
4. [0, RPM2]
5. [RPM2, 0]
6. [RPM2, RPM2]
7. [RPM1, RPM2]
8. [RPM2, RPM1]

where, the 1st element in each vector above corresponds to the Left Wheel's RPM and the 2nd element in each vector above corresponds to the Right Wheel's RPM.

Example: actions = [[50,50], [50,0], [0,50], [50,100], [100,50], [100,100], [0,100], [100,0]]

For this project you consider the robot as a non-holonomic robot which means the robot cannot move in the y-direction independently.

You will have to define smooth moves for the robot by providing Left and Right wheel velocities. The time for each move will have to be fixed.

The equations for a Differential Drive robot are:

```
Delta_Xn = 0.5*r * (UL + UR) * math.cos(Thetan) * dt
Delta_Yn = 0.5*r * (UL + UR) * math.sin(Thetan) * dt
Thetan += (r / L) * (UR - UL) * dt
D=D+ math.sqrt(math.pow((0.5*r * (UL + UR) * math.cos(Thetan) * dt),2)+math.pow((0.5*r * (UL + UR) * math.sin(Thetan) * dt),2))
```

Step 1: Function for non-holonomic constraints

Write a function that will take 2 arguments (Rotational velocities of the Left wheel and Right wheel) and returns the new coordinate of the robot, i.e. (x, y, theta).

where x and y are the translational coordinates of the robot and theta shows the orientation of the robot with respect to the x-axis.

Step 2: Modify the Map to consider the geometry of the Rigid Robot

Dimensions of the robot are available in the Official Documentation which can be used to define the clearance value.

Step 3: Generate the tree using non-holonomic constraints

Consider the configuration space as a 3 dimensional space.
To check for duplicate nodes:
Euclidean distance threshold is 0.5 unit (for x, y)
Theta threshold is 30 degrees (for Θ)
Step 4: Display the tree in the configuration space
Use curves that address the non-holonomic constraints to connect the new node to previous nodes and display it on the Map.
Step 5: Implement A* search algorithm to search the tree and to find the optimal path
Consider Euclidean distance as a heuristic function.
Note:- Define a reasonable threshold value for the distance to the goal point. Do to the limited number of moves the robot cannot reach the exact goal location. So, use a threshold distance to check the goal
Step 6: Display the optimal path in the Map

Full Code:

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

scale = 50
R = 0.033 * scale  # Radius of the wheel in m
r = 0.105 * scale # Radius of the robot in m
L = 0.160 * scale # Distance between the wheels in m
map_width, map_height = 600, 250    # Map dimensions
threshold = 5 # Threshold for the goal node

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

    clearance = clearance + r
    y, x = np.meshgrid(np.arange(map_height), np.arange(map_width), indexing='ij')
    is_obstacle = np.zeros_like(x, dtype=bool)
    for eqn in eqns.values():   # Check if the point is an obstacle
        is_obstacle |= eqn(x, y)

    is_clearance = np.zeros_like(x, dtype=bool) # Check if the point is within the clearance
    for eqn in eqns.values():
        for c_x in np.arange(-clearance, clearance+0.5, 0.5):
            for c_y in np.arange(-clearance, clearance+0.5, 0.5):
                if (c_x**2 + c_y**2) <= clearance**2:
                    is_clearance |= eqn(x+c_x, y+c_y)

    pixels = np.full((map_height, map_width, 3), 255, dtype=np.uint8)
    pixels[is_obstacle] = [0, 0, 0]  # obstacle
    pixels[np.logical_not(is_obstacle) & np.logical_or.reduce((y < clearance, y >= map_height - clearance, x <
clearance, x >= map_width - clearance), axis=0)] = [192, 192, 192]  # boundary
    pixels[np.logical_not(is_obstacle) & np.logical_not(np.logical_or.reduce((y < clearance, y >= map_height -
clearance, x < clearance, x >= map_width - clearance), axis=0)) & is_clearance] = [192, 192, 192]  # clearance
    pixels[np.logical_not(is_obstacle) & np.logical_not(np.logical_or.reduce((y < clearance, y >= map_height -
clearance, x < clearance, x >= map_width - clearance), axis=0)) & np.logical_not(is_clearance)] = [255, 255, 255]
# free space

    return pixels

# Define a function to check if current node is in range
def is_in_range(node):
    if node is None:
        return False
    if len(node) == 3:
        x, y, _ = node
    else:
        x, y = 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()

def is_valid_node(node, visited):
    if node is None:
        return False
    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
    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, y + j * threshold, 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) <= 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)

def move_func(input_node, UL, UR, plot=False):
    t, dt = 0, 0.1 #Time step
    Xi, Yi, Thetai = input_node #Input point's coordinates
    Thetan = 3.14 * Thetai / 180 #Convert end point angle to radian
    Xn, Yn = Xi, Yi #End point coordinates
    Cost=0
    valid = True # Flag to indicate if all points in the curve are valid nodes
    while t<1:
        t += dt
        X_prev, Y_prev = Xn, Yn
        Dx = 0.5*R * (UL + UR) * math.cos(Thetan) * dt
        Dy = 0.5*R * (UL + UR) * math.sin(Thetan) * dt
        Xn += Dx
        Yn += Dy
        Thetan += (R / L) * (UR - UL) * dt
        if Thetan < 0:
            Thetan += 2*math.pi
        Cost += math.sqrt(math.pow(Dx,2)+math.pow(Dy,2))
        node = (Xn, Yn, Thetan)
        if (pixels[int(map_height - Yn - 1), int(Xn)] == [0, 0, 0]).all() or (pixels[int(map_height - Yn - 1), 
int(Xn)] == [192, 192, 192]).all():
            valid = False # Mark as invalid
            break
        if plot: cv2.arrowedLine(pixels, (int(X_prev), map_height - 1 - int(Y_prev)), (int(Xn), map_height - 1 - 
int(Yn)), (255, 0, 0), thickness=1)
```

```
    Thetan = (180 * (Thetan) / 3.14) % 360 #Convert back to degrees
    if valid:
        return (Xn, Yn, Thetan), Cost
    else:
        return None, float('inf')

# Define the A* algorithm
def a_star(start_node, goal_node, display_animation=True):
    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('Path', 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 i in range(len(path) - 1):
                                    x1, y1, _ = path[i]
                                    x2, y2, _ = path[i+1]
                                    cv2.line(pixels, (int(x1), map_height - 1 - int(y1)), (int(x2), map_height - 1
- int(y2)), (0, 0, 255), thickness=2)
                cv2.imshow('Path', pixels)
                cv2.waitKey(0)
                out.write(pixels)

            print("Final Cost: ", cost[goal_node])
            out.release()
            cv2.destroyAllWindows()
            return path

        for action in actions:    # Iterate through all possible moves
            new_node, move_cost = move_func(current_node, action[0], action[1])
            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
                        V[i][j][k] = True
                        open_list.put((new_cost, new_node)) # Add to open list
                        visited.add(new_node)   # Add to visited list
                        _, _ = move_func(current_node, action[0], action[1], plot=display_animation) # Plot the 
path

        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: "))
    pixels = obstacles(clearance)
    start_node = tuple(map(int, input("Enter the start node (in the format 'x y o'): ").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'): ").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
    RPM1 = int(input("Enter RPM1: "))
    RPM2 = int(input("Enter RPM2: "))
    break

actions=[[RPM1,RPM1], [0,RPM1], [RPM1,0], [RPM2,RPM2], [0,RPM2], [RPM2,0], [RPM1,RPM2], [RPM2,RPM1]]  # List of 
actions

# 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:

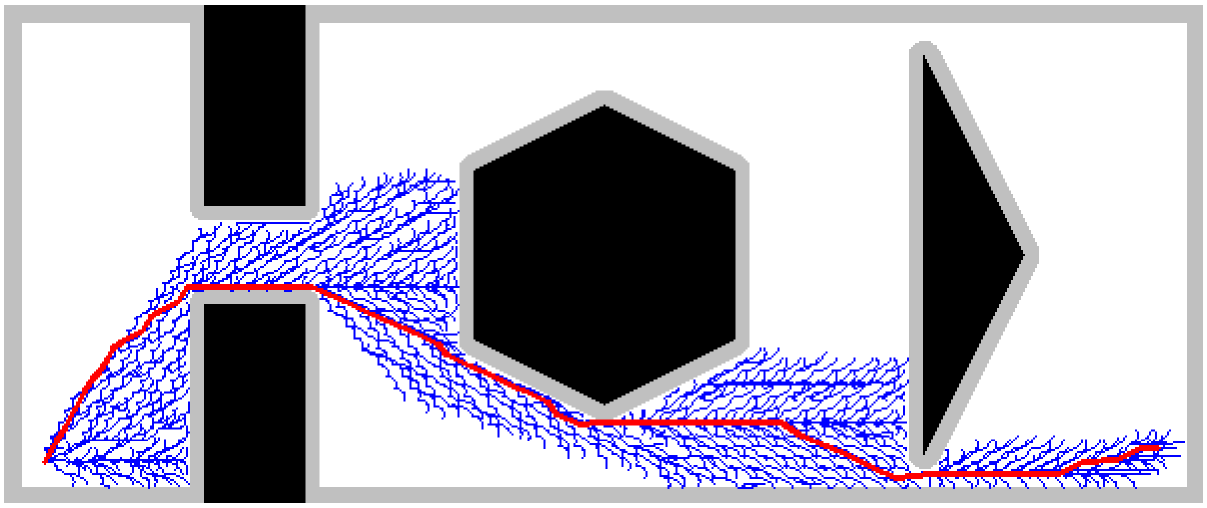

Figure 1 Optimal Path

Terminal Output:

```
PS C:\Users\manda> & "C:/Program Files/Python311/python.exe" "c:/Users/manda/OneDrive - University of
Maryland/Planning For Autonomous Robots/Projects/Project 3/Phase 2/Part 1/proj3p2_part1_hamza_vishnu.py"

Enter the clearance: 3
Enter the start node (in the format 'x y o'): 20 20 30
Enter the goal node (in the format 'x y'): 580 30
Enter RPM1: 5
Enter RPM2: 10
Final Cost:  664.3138513011415

Goal Node Reached!
Shortest Path:  [(20, 20, 30), (35.72076941345914, 29.070827362972288, 30.0), (42.26259834755264,
40.18903795081577, 95.02786624203819), (47.625138385156, 51.92161212802336, 30.0), (54.1669673192495,
63.03982271586683, 95.02786624203819), (59.52950735685286, 74.77239689307446, 30.0), (66.07133629094633,
85.89060748091796, 95.02786624203819), (71.43387632854966, 97.62318165812557, 30.0), (79.2942610352793,
102.15859533961168, 30.0), (95.01503044873844, 111.22942270258405, 30.0), (99.31490441136336, 111.26256822086647,
325.1547309848169), (103.59534663003146, 110.85289868027837, 30.182597226855364), (107.89509320054447,
110.89974042665413, 325.3373282116723), (120.74026947475298, 109.71164154058074, 30.36519445371073),
(125.03984502718045, 109.77217903978534, 325.51992543852765), (137.88874059020918, 108.62500194405989,
30.547791680566093), (150.78682332051963, 108.84769985748848, 325.702522665383), (158.26912261001243,
103.71244695792697, 325.702522665383), (171.12160709474105, 102.60620329200711, 30.730388907421457),
(184.01891503189748, 102.86998436171882, 325.8851198922383), (199.0161523547165, 92.64719742844855,
325.8851198922383), (214.0133896775355, 82.42441049517828, 325.8851198922383), (229.01062700035453,
72.20162356190801, 325.8851198922383), (244.00786432317355, 61.97883662863774, 325.8851198922383),
(259.0051016459926, 51.756049695367466, 325.8851198922383), (271.86104464888683, 50.69075068339304,
30.912986134276764), (276.15984541078075, 50.792371199935175, 326.0677171190937), (291.18956931286533,
40.61740685468509, 326.0677171190937), (304.04884039530003, 39.59306330536447, 31.095583361132128),
(316.9442061803752, 39.93900224172742, 326.2503143459491), (329.80667486995833, 38.955624548227696,
31.278180587987492), (342.70087331581135, 39.34263736137087, 326.4329115728044), (355.56640910770693,
38.40023550121251, 31.460777814842857), (368.4593093875668, 38.82831826443681, 326.6155087996598),
(381.32778174582006, 37.92690179939236, 31.64337504169822), (394.2192530460864, 38.396050169295954,
326.7981060265152), (407.0905314049469, 37.535628245293026, 31.825972268553585), (419.9804429265188,
38.045837461815246, 326.9807032533705), (427.575374350322, 33.078681165932196, 326.9807032533705),
(442.7652371979279, 23.144368574166137, 326.9807032533705), (457.95510004553375, 13.210055982400082,
326.9807032533705), (470.82905381078155, 12.390637329428579, 32.00856949540889), (478.5257329317975,
17.198626046981484, 32.00856949540889), (491.41395389139956, 17.749890933451205, 327.1633004802258),
(504.2904524416695, 16.97148386547474, 32.191166722264256), (517.1768520731787, 17.56379882866294,
327.3458977070812), (530.0557647612852, 16.826411243534505, 32.37376394911962), (537.7216580642385,
21.683334638826572, 32.37376394911962), (550.6061056200117, 22.316693669003023, 327.5284949339366),
(563.4873017742758, 21.620333048384794, 32.556361175974985), (578.7880690671005, 31.38296692715053,
32.55636117597499), (578.7880690671005, 31.38296692715053, 32.55636117597499)]

Runtime: 18.375009059906 seconds
```

Animation: https://drive.google.com/file/d/1d204RiwlYIIe02Jf-8912CeFpOY6TO14/view?usp=share_link

Github Repo: https://github.com/h4mz404/Path-Planning-A-Star-Differential-Drive-Bot

Part 2:

Gazebo Visualization

Simulate the path planning implementation on Gazebo with the TurtleBot 3 Burger robot.
The Gazebo environment has been provided for the map (map.world), which is different from that used in Part 1 2D Implementation.
To run the simulation in ROS, you should have everything wrapped in only one launch file.
You are NOT required to implement a controller to make the TurtleBot accurately follow the trajectory. This will be an open-loop controller and hence, it is okay if the robot does not follow the exact waypoints accurately.
Make note of the coordinate system in Gazebo, and the position of the origin. Input will be based on the coordinates in Gazebo for Part 2: Gazebo Visualization. Quantities in ROS are reported in meters, radians, meter/sec, and rad/sec. Each tile in the Gazebo world is 1m x 1m.
User input should be from the Terminal and not hardcoded into the code. Decide the clearance and wheel RPMs on your own.

Full Code:

```
#!/usr/bin/env python3

import numpy as np
import time
import cv2
import math
from queue import PriorityQueue
from geometry_msgs.msg import Twist, Pose
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
from nav_msgs.msg import Odometry
import rospy

scale = 100 # Scale of the map to convert from m to pixels
R = 0.033 * scale  # Radius of the wheel in m
r = 0.105 * scale  # Radius of the robot in m
L = 0.160 * scale # Distance between the wheels in m
map_width, map_height = (6*scale), (2*scale)    # Map dimensions
threshold = 0.05 * scale # Threshold for the goal node

def obstacles(clearance):
    # Define the Obstacle Equations and Map Parameters
    eqns = {
        "Rectangle1": lambda x, y: np.logical_and(0.75*scale <= map_height - y - 1, map_height - y - 1 <= 2*scale) 
& np.logical_and(1.5*scale <= x, x <= 1.65*scale),
        "Rectangle2": lambda x, y: np.logical_and(0*scale <= map_height - y - 1, map_height - y - 1 <= 1.25*scale) 
& np.logical_and(2.50*scale <= x, x <= 2.65*scale),
        "Circle": lambda x, y: (x - (4*scale))**2 + (map_height - y - 1 - (1.1*scale))**2 <= (0.5*scale)**2
    }
    clearance = clearance + r
    y, x = np.meshgrid(np.arange(map_height), np.arange(map_width), indexing='ij')
    is_obstacle = np.zeros_like(x, dtype=bool)
    for eqn in eqns.values():   # Check if the point is an obstacle
        is_obstacle |= eqn(x, y)

    is_clearance = np.zeros_like(x, dtype=bool) # Check if the point is within the clearance
    for eqn in eqns.values():
        for c_x in np.arange(-clearance, clearance+0.5, 0.5):
            for c_y in np.arange(-clearance, clearance+0.5, 0.5):
                if (c_x**2 + c_y**2) <= clearance**2:
                    is_clearance |= eqn(x+c_x, y+c_y)

    pixels = np.full((map_height, map_width, 3), 255, dtype=np.uint8)
    pixels[is_obstacle] = [0, 0, 0]  # obstacle
    pixels[np.logical_not(is_obstacle) & np.logical_or.reduce((y < clearance, y >= map_height - clearance, x < 
clearance, x >= map_width - clearance), axis=0)] = [192, 192, 192]  # boundary
    pixels[np.logical_not(is_obstacle) & np.logical_not(np.logical_or.reduce((y < clearance, y >= map_height - 
clearance, x < clearance, x >= map_width - clearance), axis=0)) & is_clearance] = [192, 192, 192]  # clearance
```

```
    pixels[np.logical_not(is_obstacle) & np.logical_not(np.logical_or.reduce((y < clearance, y >= map_height - clearance, x < clearance, x >= map_width - clearance), axis=0)) & np.logical_not(is_clearance)] = [255, 255, 255]  # free space

    return pixels

# Define a function to check if current node is in range
def is_in_range(node):
    if node is None:
        return False
    if len(node) == 3:
        x, y, _ = node
    else:
        x, y = 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()

def is_valid_node(node, visited):
    if node is None:
        return False
    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
    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, y + j * threshold, 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) <= threshold

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

def move_func(input_node, UL, UR, plot=False):
    t, dt = 0, 0.1 #Time step
    Xi, Yi, Thetai = input_node #Input point's coordinates
    Thetan = math.pi * Thetai / 180 #Convert end point angle to radian
    Xn, Yn = Xi, Yi #End point coordinates
    Cost=0
    valid = True # Flag to indicate if all points in the curve are valid nodes
    while t<1:
        t += dt
        X_prev, Y_prev = Xn, Yn
        linearx = 0.5*R * (UL + UR) * math.cos(Thetan) * dt
        lineary = 0.5*R * (UL + UR) * math.sin(Thetan) * dt
        lineartheta = (R / L) * (UR - UL) * dt
        Xn += linearx
        Yn += lineary
        Thetan += lineartheta
        if Thetan < 0:
            Thetan += 2*math.pi
        Cost += math.sqrt(math.pow(linearx,2)+math.pow(lineary,2))
        if (pixels[int(map_height - Yn - 1), int(Xn)] == [0, 0, 0]).all() or (pixels[int(map_height - Yn - 1),
int(Xn)] == [192, 192, 192]).all():
            valid = False # Mark as invalid
            break
        if plot: cv2.arrowedLine(pixels, (int(X_prev), map_height - 1 - int(Y_prev)), (int(Xn), map_height - 1 -
int(Yn)), (255, 0, 0), thickness=1)
    Thetan = (180 * (Thetan) / math.pi) % 360 #Convert back to degrees
    if valid:
        return (Xn,Yn, Thetan), Cost
    else:
        return None, float('inf')

# Define the A* algorithm
def a_star(start_node, goal_node, display_animation=True):
    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,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('Path', 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, final_actions = backtrack_path(parent, start_node, approx_goal_node) # Backtrack the path
            if display_animation:
                for i in range(len(path) - 1):
                                    x1, y1, _ = path[i]
                                    x2, y2, _ = path[i+1]
                                    cv2.line(pixels, (int(x1), map_height - 1 - int(y1)), (int(x2), map_height - 1
- int(y2)), (0, 0, 255), thickness=2)
                cv2.imshow('Path', pixels)
                cv2.waitKey(2000)
                out.write(pixels)
            print("Final Cost: ", cost[goal_node])
            out.release()
            return path, final_actions

        for action in actions:    # Iterate through all possible moves
            new_node, move_cost = move_func(current_node, action[0], action[1])
            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, action)  # Update parent
                        V[i][j][k] = True
                        open_list.put((new_cost, new_node)) # Add to open list
                        visited.add(new_node)   # Add to visited list
                        _, _ = move_func(current_node, action[0], action[1], plot=display_animation) # Plot the 
path

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

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

time.sleep(5)
# Get valid start and goal nodes from user input
while True:
    clearance = float(input("\nEnter the clearance (in m): "))
    print("Generating 2D map...")
    pixels = obstacles(clearance*scale)
    print("Map generated.")
    start_node = tuple(map(float, input("Enter the start node (in the format 'x y o')(in m): ").split()))
    gazebostart_node = start_node
    start_node = ((start_node[0]+0.5)*scale, (start_node[1]+1)*scale, start_node[2]%360)
    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(float, input("Enter the goal node (in the format 'x y')(in m): ").split()))
    goal_node = ((goal_node[0]+0.5)*scale, (goal_node[1]+1)*scale)
    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
    RPM1, RPM2 = map(int, input("Enter RPM1, RPM2 (in the format 'R1 R2'): ").split())
    break

actions=[[RPM1,RPM1], [0,RPM1], [RPM1,0], [RPM2,RPM2], [0,RPM2], [RPM2,0], [RPM1,RPM2], [RPM2,RPM1]]  # List of 
actions

# Run A* algorithm
start_time = time.time()
path, act = a_star(start_node, goal_node)
for i in range(len(path)):
    x, y, theta = path[i]
    path[i] = (x/scale - 0.5, y/scale - 1, theta)
if path is None:
    print("\nError: No path found.")
else:
    print("\nGoal Node Reached!\n\nShortest Path: ", path, "\n\nActions: ", act, "\n")
end_time = time.time()
print("Runtime:", end_time - start_time, "seconds\n\nStarting ROS simulation...")

#ROS Simulation
def get_orientation(msg, current):
    current['x'], current['y'] = msg.pose.pose.position.x, msg.pose.pose.position.y
    _, _, current['yaw'] = euler_from_quaternion((msg.pose.pose.orientation.x,
                                                  msg.pose.pose.orientation.y,
                                                  msg.pose.pose.orientation.z,
                                                  msg.pose.pose.orientation.w))


rospy.init_node('algorithm')
rospy.wait_for_service('/gazebo/set_model_state')   # Wait for the service to be available
set_model_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
state = ModelState(model_name='burger', reference_frame='world')
state.pose.position.x, state.pose.position.y, state.pose.position.z = gazebostart_node[0], gazebostart_node[1], 0 
# Set the initial position of the robot
```

```
q = quaternion_from_euler(0, 0, gazebostart_node[2] * math.pi / 180)    # Set the initial orientation of the robot
state.pose.orientation.x, state.pose.orientation.y, state.pose.orientation.z, state.pose.orientation.w = q
set_model_state(state)
rospy.sleep(2)

current = {'x': 0, 'y': 0, 'yaw': 0}    # Dictionary to store the current position of the robot
odom_sub = rospy.Subscriber('/odom', Odometry, get_orientation, callback_args=current)  # Subscribe to the odom
topic to get the current position of the robot

for point in path:
    goal_x, goal_y, _ = point
    pub, rate = rospy.Publisher('/cmd_vel', Twist, queue_size=10), rospy.Rate(10)   # Create a publisher and rate
object
    reached_goal, twist = False, Twist()

    while not reached_goal and not rospy.is_shutdown():
        Deltax, Deltay = goal_x - current['x'], goal_y - current['y']   # Calculate delta x and delta y
        goal_angle, distance_to_goal = math.atan2(Deltay, Deltax), math.sqrt(Deltax**2 + Deltay**2) # Calculate
goal angle and distance to goal

        if abs(goal_angle - current['yaw']) > 0.1:  # If the robot is not facing the goal, rotate
            twist.linear.x, twist.angular.z = 0.0, goal_angle - current['yaw']  # Rotate towards the goal
        else:
            twist.linear.x, twist.angular.z = min(0.2, distance_to_goal), 0.0   # Move towards the goal
            reached_goal = distance_to_goal < 0.05  # Check if the robot has reached the goal

        pub.publish(twist)
        rate.sleep()

print("Simulation complete. Reached Goal Node.\n\n")
```

Final Output:

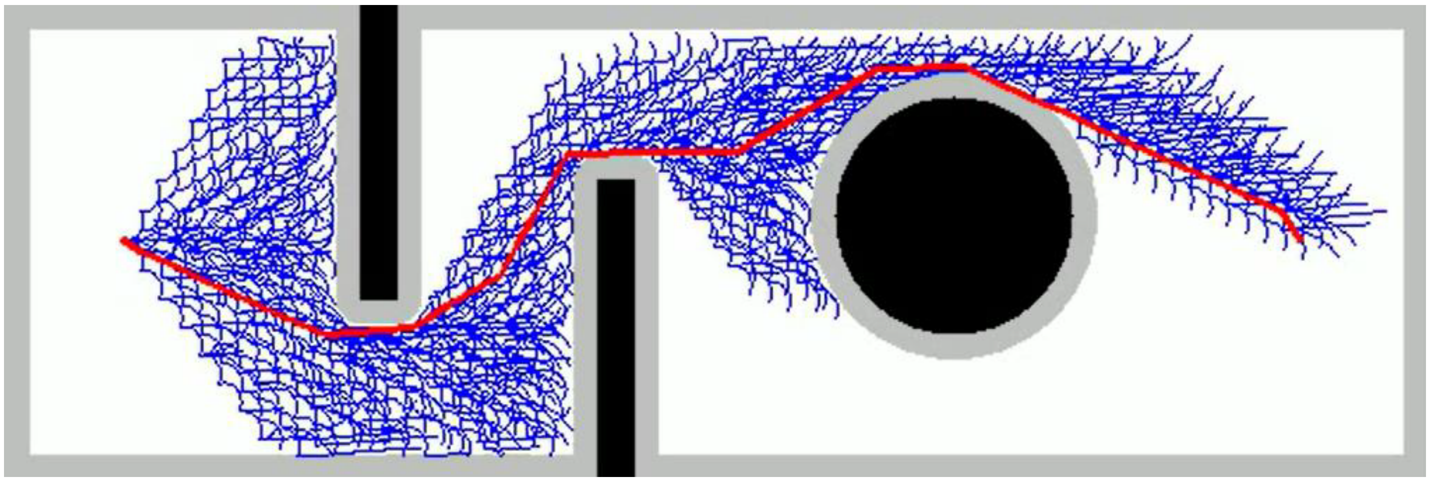
Figure 2 A Output*

Terminal Output:

```
~$ rosrun proj3p2part2 algorithm.py

Enter the clearance (in m): 0.01
Generating 2D map...
Map generated.
Enter the start node (in the format 'x y o')(in m): 0 0 30
Enter the goal node (in the format 'x y')(in m): 5 0
Enter RPM1, RPM2 (in the format 'R1 R2'): 5 10
Final Cost:  602.5932457259098

Goal Node Reached!
```

```
Shortest Path:  [(0.0, 0.0, 30.0), (0.2579919007751933, 0.0020572131231002544, 325.00510011484715),
(0.5553626250804788, -0.2061245640000413, 325.00510011484715), (0.8527333493857643, -0.414306341123183,
325.00510011484715), (1.109566398074471, -0.4388183400766602, 30.000000000000114), (1.2667500088613468, -
0.34806834007666, 30.00000000000011), (1.5811172304350989, -0.16656834007665933, 30.00000000000011),
(1.7118947796482993, 0.05583059340374197, 94.99489988515283), (1.8190832902014042, 0.2905105375763066,
30.000000000000103), (2.133450511775156, 0.47201053757630707, 30.000000000000103), (2.219447812033555,
0.4726962752840078, 325.0051001148472), (2.476280860722262, 0.4481842763305308, 30.00000000000017),
(2.790648082296014, 0.6296842763305315, 30.00000000000017), (3.1050153038697657, 0.8111842763305321,
30.00000000000017), (3.3630072046449597, 0.8132414894536333, 325.00510011484727), (3.619840253333667,
0.7887294905001565, 30.00000000000017), (3.7058375535920653, 0.789415228207857, 325.00510011484727),
(3.8545229157447087, 0.6853243396462878, 325.0051001148472), (4.151893640049994, 0.477142562523146,
325.0051001148472), (4.44926436435528, 0.26896078540000445, 325.0051001148472), (4.7466350886605655,
0.06077900827686289, 325.0051001148472), (5.003468137349273, 0.03626700932338611, 30.00000000000017)]

Actions:  [None, [10, 10], [10, 10], [5, 10], [5, 5], [10, 10], [5, 10], [10, 5], [10, 10], [5, 0], [5, 10], [10,
10], [10, 10], [10, 5], [5, 10], [5, 0], [5, 5], [10, 10], [10, 10], [10, 10], [5, 10], [5, 10]]

Runtime: 21.750512838363647 seconds

Starting ROS simulation...
Simulation complete. Reached Goal Node.
```

Animation: https://drive.google.com/file/d/1fZ7YNa50fs31JxDoAsgtxc9kvh9STm9Q/view?usp=share_link

Gazebo Simulation: https://drive.google.com/file/d/1GjYFWi2oQGLZSQk5o0g2MLNK1YDFE3it/view?usp=share_link