```python
# Part1 code
import time
from collections import deque
import matplotlib.patches as patches
import numpy as np
import matplotlib.pyplot as plt
import math
import os

plt.ion()

print("Hi!! \n")
print("As the entire map is in meters there are only few starting and ending points which will not be in the obstacle space \n")
print("The preferable points are given along with the input command \n")

start_time = time.time()

clearance = int(input("Enter the clearance to be maintained around the obstacles (preferrably 50) \n"))
print("______________________________________________________________________")

robot_radius = 105

total_bloat = clearance + robot_radius

total_bloat = total_bloat/1000

def map():

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.set_xlim([-0.5, 5.5])
    ax.set_ylim([-1, 1])

    circle = patches.Circle((3.5, 0.1), radius=0.5, fill=True)
    rectangle_1 = patches.Rectangle((1, -0.25), width=0.15, height=1.25, color='blue')
    rectangle_2 = patches.Rectangle((2, -1), width=0.15, height=1.25, color='blue')

    ax.add_artist(circle)
    ax.add_patch(rectangle_1)
    ax.add_patch(rectangle_2)

    ax.set_aspect('equal')
    plt.show()

def circle(input):
    x = input[0]
    y = input[1]
    total_bloat = input[2]
    if  (((x-3.5)**2)+((y-0.1)**2)-((0.5+total_bloat)**2)) <= 0:
        return True
    else:
        return False

def rectangle_1(input):
    x = input[0]
    y = input[1]
    total_bloat = input[2]
    if x-(1-total_bloat) >= 0 and x-(1.15+total_bloat) <= 0 and y-(-0.25-total_bloat) >= 0 and y-(1+total_bloat) <= 0:
        return True
    else:
        return False

def rectangle_2(input):
```

```
    x = input[0]
    y = input[1]
    total_bloat = input[2]
    if x-(2-total_bloat) >= 0 and x-(2.15+total_bloat) <= 0 and y-(-1-total_bloat) >= 0 and y-(0.25+total_bloat) <= 0:
        return True
    else:
        return False

def wall(input):
    x = input[0]
    y = input[1]
    total_bloat = input[2]
    if (x - total_bloat <= -0.5) or (x + total_bloat >= 5.5) or (y - total_bloat <= -1) or (y + total_bloat >= 1):
        return True
    else:
        return False

def if_obstacle(input):
    if (wall(input) or circle(input) or rectangle_1(input) or rectangle_2(input)) == True:
        return True
    else:
        return False
#Taking start coordinates
print("Choose your start point such that 0.2 <= X <= 0.5 and 0.2 <= Y <= 1.8 <\n")
start_point_x = input("Enter the x-coordinate of the start point \n")
start_point_y = input("Enter the y-coordinate of the start point \n")
start_point_theta = input("Enter the start orientation \n")
# start = (int(start_point_x), int(start_point_y), int(start_point_theta))
start = (float(start_point_x)-0.5, float(start_point_y)-1, float(start_point_theta))
# start = (1, 1, 0)
while if_obstacle((start[0], start[1], total_bloat)):
    print("These coordinates lie inside the obstacle space. Please enter new values such that 0.2 <= X <= 0.5 and 0.2 <= Y <= 1.8 \n")
    start_point_x = input("Enter the x-coordinate of the start point \n")
    start_point_y = input("Enter the y-coordinate of the start point \n")
    start_point_theta = input("Enter the start orientation \n")
    # start = (int(start_point_x), int(start_point_y), int(start_point_theta))
    start = (float(start_point_x)-0.5, float(start_point_y)-1, float(start_point_theta))

#Taking goal coordinates
print("_____________________________________________________________________________")
print("Choose your goal point such that 4.7 <= X <= 5.8 and 0.2 <= Y <= 1.8 \n")
goal_point_x = input("Enter the x-coordinate of the goal point \n")
goal_point_y = input("Enter the y-coordinate of the goal point \n")
goal_point_orien = input("Enter the goal orientation \n")
# goal = (int(goal_point_x), int(goal_point_y), int(goal_point_orien))
goal = (float(goal_point_x)-0.5, float(goal_point_y)-1, float(goal_point_orien))

# goal = (5, 1, 0)
while if_obstacle((goal[0], goal[1], total_bloat)):
    print("These coordinates lie inside the obstacle space. Please enter new values such that 4.7 <= X <= 5.8 and 0.2 <= Y <= 1.8 \n")
    goal_point_x = input("Enter the x-coordinate of the goal point \n")
    goal_point_y = input("Enter the y-coordinate of the goal point \n")
    goal_point_orien = input("Enter the goal orientation \n")
    # goal = (int(goal_point_x), int(goal_point_y), int(goal_point_orien))
    goal = (float(goal_point_x)-0.5, float(goal_point_y)-1, float(goal_point_orien))
print("_____________________________________________________________________________")


ul = int(input("Enter the velocity of left wheel (preferrable 60 to 70 rpm) \n"))
ur = int(input("Enter the velocity of right wheel (preferrably 120 to 140 rpm) \n"))
ul = ul*0.1047
ur = ur*0.1047
start = (float(start_point_x)-0.5, float(start_point_y)-1, float(start_point_theta), ul, ur)
goal = (float(goal_point_x)-0.5, float(goal_point_y)-1, float(goal_point_orien), ul, ur)
angle = 30
```

```
angle = 20

def rounding_value(x, y, thetas, th=20):
    return round(x, 1), round(y, 1), round(thetas/th) * th

def cost(Xi, Yi, Thetai, u_left, u_right, UL, UR, total_bloat):
    Thetai = Thetai % 360
    t = 0
    r = 0.033
    L = 0.160
    dt = 0.1
    Xn = Xi
    Yn = Yi
    Thetan = 3.14 * Thetai / 180

    D = 0

    while t < 1:
        t = t + dt
        Xs = Xn
        Ys = Yn

        input = (Xn, Yn, total_bloat)
        if if_obstacle(input):
            break

        Xn += 0.5 * r * (UL + UR) * math.cos(Thetan) * dt
        Yn += 0.5 * r * (UL + UR) * math.sin(Thetan) * dt
        Thetan += (r / L) * (UR - UL) * dt
        # Xn += 0.5 * r * (UL*0.1047 + UR*0.1047) * math.cos(Thetan) * dt
        # Yn += 0.5 * r * (UL*0.1047 + UR*0.1047) * math.sin(Thetan) * dt
        # Thetan += (r / L) * (UR*0.1047 - UL*0.1047) * dt

        plt.plot([Xs, Xn], [Ys, Yn], color="blue")

        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))

    Thetan = 180 * (Thetan) / 3.14
    cost = (*rounding_value(Xn, Yn, Thetan, angle), D, UL, UR)
    return cost

def correct_children(current_node, ul, ur, total_bloat):
    children = []
    # ul = current_node[3]
    # ur = current_node[4]
    actions = [[0, ul], [ul,0], [ul, ul], [0, ur], [ur, 0], [ur, ur], [ul, ur], [ur, ul]]

    for action in actions:
        c_x, c_y, c_theta, c_cost_, c_UL, c_UR = cost(*current_node, *action, total_bloat)
        # c_x, c_y, c_theta, c_cost_, c_UL, c_UR = cost(*current_node, total_bloat)

        input = (c_x, c_y, total_bloat)
        if if_obstacle(input):
            continue

        plt.pause(0.01)

        child = (c_x, c_y, c_theta, c_UL, c_UR, c_cost_)
        children.append(child)

    return children

# def A_star(start_node, goal_node, total_bloat, left_RPM, right_RPM):
def A_star(start_node, goal_node, total_bloat, left_RPM, right_RPM):
```

```
    open_list = deque()
    visited_close_list = {}

    initial_cost_to_go = float('inf')
    initial_cost_to_come = 0

    open_list.append((start_node, initial_cost_to_go, initial_cost_to_come))

    generated_path = {}
    while len(open_list) != 0:

        current_node, dist, cost_to_come = open_list.popleft()
        visited_close_list[(current_node[0], current_node[1])] = 1

        if dist <= 0.6:
            print("Goal has been reached!!!!")
            print("_____________________________________________________________________________")

            goal_node = current_node

            path = [goal_node]
            while current_node[0]!=start_node[0] or current_node[1]!=start_node[1]:
                current_node = generated_path[current_node]
                path.append(current_node)
            return path[::-1]

        # children = set(correct_children(current_node, total_bloat, left_RPM, right_RPM))
        children = set(correct_children(current_node, left_RPM, right_RPM, total_bloat))
        for modi_x, modi_y, modi_theta , modi_ul, modi_ur, modi_cost in children:
            dist = math.dist((modi_x, modi_y), goal_node[:2])
            if visited_close_list.get((modi_x, modi_y)) == 1:
                continue
            new_cost = cost_to_come + modi_cost
            new_cost1 = dist*2.5
            for i, node in enumerate(open_list):
                if node[1] + node[2] > new_cost + dist*2.5:
                    open_list.insert(i,((modi_x, modi_y, modi_theta, modi_ul, modi_ur), new_cost1, new_cost))
                    break
            else:
                open_list.append(((modi_x, modi_y, modi_theta, modi_ul, modi_ur), new_cost1, new_cost))

            generated_path[(modi_x, modi_y, modi_theta, modi_ul, modi_ur)] = current_node

def shortest_path(path):

    start_node = path[0]
    goal_node = path[-1]

    plt.plot(start_node[0], start_node[1], marker="o", markersize=10, color="red")
    plt.plot(goal_node[0], goal_node[1], marker="o", markersize=10, color="red")

    for i, (x, y, theta, ul, ur) in enumerate(path[:-1]):
        n_x, n_y, theta, u_l, u_r = path[i+1]
        plt.plot([x, n_x], [y, n_y], color="green", linewidth=3)

    plt.show(block=False)
    plt.pause(5)
    plt.close()


map()
generated_path = A_star(start, goal, total_bloat, ul, ur)
print(generated_path)
shortest_path(generated_path)
```

```
path_array = np.array(generated_path)
np.savetxt('generated_path.txt', path_array, delimiter='\t')

# Part 2 code
#!/usr/bin/env python3
import rospy
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry

import math

# Definining the X and Y coordinates to be travelled by the robot based on the generated shortest path

# pose_list = [(0.0, 0.0, 0.0, 7.329, 14.658), (0.5, 0.0, 0, 14.658, 14.658),
#               (0.8, -0.2, -100, 14.658, 7.329), (1.0, -0.5, 360, 7.329, 14.658),
#               (1.5, -0.5, 0, 14.658, 14.658), (1.8, -0.3, 100, 7.329, 14.658),
#               (1.8, -0.0, 100, 7.329, 7.329), (1.8, 0.3, 100, 7.329, 7.329),
#               (2.0, 0.6, 0, 14.658, 7.329), (2.5, 0.6, 0, 14.658, 14.658),
#               (3.0, 0.6, 0, 14.658, 14.658), (3.1, 0.7, 100, 0, 7.329),
#               (3.2, 0.8, 0, 7.329, 0), (3.7, 0.8, 0, 14.658, 14.658),
#               (4.0, 0.6, -100, 14.658, 7.329), (4.1, 0.5, 360, 0, 7.329),
#               (4.4, 0.3, -100, 14.658, 7.329), (4.6, 0.0, 360, 7.329, 14.658),
#               (4.9, 0.0, 0, 7.329, 7.329)]

# def read_path_list(input_array):
#     coordinates = []
#     for item in input_array:
#         coordinates.append([item[0], item[1]])
#     return coordinates


#Reading file from file path. Make sure to change the file path based on where you download the file.
def read_file_path(filename='/home/sarin/Documents/661/Project3_phase1/proj3_p2_sarin_aditi/generated_path.txt'):
    coordinates = []
    with open(filename, 'r') as file_name:
        lines = file_name.readlines()
    for line in lines:
        x, y, th, left_rpm, right_rpm = line.strip().split('\t') #This line takes the points from the path separated by spaces
        x = round(float(x), 2)
        y = round(float(y), 2)
        # coordinates.append([float(x), float(y)])
        coordinates.append([x, y])
    return coordinates

#Defining the initial coordinates of the robot as defined in the launch file
x_coord = 0.0
y_coord = 0.0
theta = 0.0

def orientation_between_nodes(vel_msg):
    global x_coord, y_coord, theta
    x_coord = vel_msg.pose.pose.position.x
    y_coord = vel_msg.pose.pose.position.y
    quaternion = (
        vel_msg.pose.pose.orientation.x,
        vel_msg.pose.pose.orientation.y,
        vel_msg.pose.pose.orientation.z,
        vel_msg.pose.pose.orientation.w
    )
    _, _, theta = euler_from_quaternion(quaternion)
```

```python
def turtlebot_motion(goal_x_coord, goal_y_coord):
    publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
    rate = rospy.Rate(10)
    vel_msg = Twist()
    goal_reached = False

    while not goal_reached and not rospy.is_shutdown():

        delta_y = goal_y_coord - y_coord
        delta_x = goal_x_coord - x_coord

        rotation = math.atan2(delta_y, delta_x)
        dist = math.sqrt(delta_x**2 + delta_y**2)

        if abs(rotation - theta) > 0.3:
            vel_msg.linear.x = 0.0
            vel_msg.angular.z = rotation - theta

        else:
            vel_msg.linear.x = min(0.5, dist)
            vel_msg.angular.z = 0.0

            if dist < 0.01:
                goal_reached = True
                vel_msg.linear.x = 0.0
                vel_msg.angular.z = 0.0

        publisher.publish(vel_msg)
        rate.sleep()

if __name__ == "__main__":
    # rospy.init_node(read_path'turtlebot_move')
    rospy.init_node('move_turtlebot')
    odom_sub = rospy.Subscriber('/odom', Odometry, orientation_between_nodes)
    # poses = read_path_list(pose_list)
    poses = read_file_path()
    # poses = read_path_list(pose_list)
    for pose in poses:
        temp_goal_x, temp_goal_y = pose
        turtlebot_motion(temp_goal_x, temp_goal_y)
```