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

# Import necessary libraries

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

USE_ROS =False

print('''
_____________________________________
    __          __
    / |       /    )
---/__|-------\------_/_----__---)__-
  /   |        \     /    /   ) /   )
_/____|____(____/___(_ __(___(_/_____

''')

# Get the mode from the user
while True:
    mode = input("Enter the mode (1 for Gazebo Mode, 2 for 2D Mode): ")
    if mode in ['1', '2']:
        break
    else:
        print("Invalid mode. Please enter a valid mode.")
if mode == '1':
    USE_ROS = True


if USE_ROS:
    try:  # Check if ROS2 libraries are installed
        import rclpy
        from rclpy.node import Node                  # Import the necessary ROS2 libraries
        from geometry_msgs.msg import Twist
        ROS_IMPORTED = True
    except ImportError:
        print("ROS2 libraries are not installed. Gazebo mode will not function.")
        mode = '2'
        ROS_IMPORTED = False
else:
    ROS_IMPORTED = False
    mode = '2'

WhR = 3.3             # Wheel radius in cm
K = (2*np.pi*WhR)/60 # Conversion factor from RPM to cm/s
L = 28.7              # Wheelbase in cm
canvas_height = 200
canvas_width = 600
```

Nitish Ravisankar Raveendran – 120385506
Pranav ANV - 1204886110

```
clearance_color = (127, 127, 127)
obstacle_color = (0, 0, 0)
free_space_color = (255, 255, 255)
threshold = 1.0

# Initiate the publisher node only if ROS2 is libaries are imported
if ROS_IMPORTED:
    class VelocityPublisher(Node):
        def __init__(self):
            super().__init__('velocity_publisher')
            self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 10)
            time.sleep(2)

        def publish_velocity(self, linear_velocity, angular_velocity):
            msg = Twist()
            msg.linear.x = linear_velocity
            msg.angular.z = angular_velocity
            self.publisher_.publish(msg)
            self.get_logger().info('Publishing: "%s"' % msg)

# Initialize a white canvas
canvas = np.ones((canvas_height, canvas_width, 3), dtype="uint8") * 255

# Function to check if the node is an obstacle
def obstacles(node):
    x, y = node
    Circ_center = (420, 120)
    R = 60
    Xc, Yc = Circ_center
    y = abs(y - canvas_height)
    obstacles = [
        (x >= 150 and x <= 175 and y <= 200 and y >= 100),
        (x >= 250 and x <= 275 and y <= 100 and y >= 0),
        (((x - Xc)**2 + (y - Yc)**2) <= R**2),
    ]
    return any(obstacles)

# Function to check if the node is within the clearance zone
def clearance(x, y, clearance):
    clearance = clearance + robo_radius
    Circ_center = (420, 120)
    R = 60 + clearance
    Xc, Yc = Circ_center
    y = abs(y - canvas_height)
    clearance_zones = [
        (x >= 150 - clearance and x <= 175 + clearance and y <= 200 + clearance  and y >= 
100 - clearance),
        (x >= 250 - clearance and x <= 275 + clearance and y <= 100 + clearance and y >= 0 
- clearance),
        (((x - Xc)**2 + (y - Yc)**2) <= R**2),
        (x <= clearance or x >= canvas_width - clearance or y <= clearance or y >=
canvas_height - clearance),
    ]
    return any(clearance_zones)


# Function to check if the node is free
def is_free(x, y):
    x = int(round(x))
    y = int(round(y))
    if x >= 0 and x < canvas_width and y >= 0 and y < canvas_height:
        if canvas_array[x, y] == 0:
            return True
    else:
        return False

# Function to get the neighbors of a node
def get_neighbors(node):
    dt = 0.1 # Time step
    neighbours = [] # List to store the neighbors
    initial_x, initial_y, initial_theta = node
    # Convert initial orientation to radians for calculation
    initial_theta_rad = np.deg2rad(initial_theta)
    action_set = [(0, RPM1), (RPM1, 0), (RPM1, RPM1), (0, RPM2), (RPM2, 0), (RPM2, RPM2),
(RPM1, RPM2), (RPM2, RPM1)] # Action set
    # action_weights = {(0, RPM1): 1.1, (RPM1, 0): 1.1, (RPM1, RPM1): 1, (0, RPM2): 1.2,
(RPM2, 0): 1.2, (RPM2, RPM2): 1.1, (RPM1, RPM2): 1.3, (RPM2, RPM1): 1.3}
    for action in action_set:
        X_new, Y_new, thetan = initial_x, initial_y, initial_theta_rad
        vel = []  # List to store the velocities
        Vl = action[0] * K # Calculate the left wheel velocity
        Vr = action[1] * K # Calculate the right wheel velocity
        t = 0  # Reset t for each action
        while t < 1:
            t += dt
            Xs = X_new
            Ys = Y_new
            Vn = 0.5 * (Vl + Vr) # Calculate the linear velocity in cm/s
            AVn = (Vr - Vl) / L  # Calculate the angular velocity in rad/s
            vel.append((Vn, AVn))
            X_new += Vn * np.cos(thetan) * dt
            Y_new += Vn * np.sin(thetan) * dt
            thetan += AVn* dt
            if is_free(X_new, Y_new):
                cv2.line(canvas, (int(round(Xs)), int(round(Ys))), (int(round(X_new)),
int(round(Y_new))), (255, 0, 0), 1)
        thetan_deg = np.rad2deg(thetan) % 360
        if is_free(X_new, Y_new):
            # weight = action_weights[action]
            cost = ((initial_x - X_new)**2 + (initial_y - Y_new)**2)**0.5  # Calculate the
cost
```

```
            # cost = cost * weight
            neighbours.append(((X_new,Y_new, thetan_deg), cost, vel))
    return neighbours

# Function to check if the goal is reached
def check_goal_reached(current_node, goal):  # orientation is not considered
    distance = ((current_node[0] - goal[0]) ** 2 + (current_node[1] - goal[1]) ** 2) **
0.5
    return distance < threshold

# A* algorithm
def a_star(start, goal):
    pq = PriorityQueue()
    cost_to_goal = ((goal[0] - start[0])**2 + (goal[1] - start[1])**2)**0.5 # Heuristic
cost
    pq.put((cost_to_goal, (start, 0)))
    came_from = {start: None}
    cost_so_far = {start: cost_to_goal}
    count = 0

    while not pq.empty():
        current_cost, current_node = pq.get()
        if check_goal_reached(current_node[0], goal):  # Check if the goal is reached
            print("Goal Reached")
            print("Cost to Goal: " , cost_so_far[current_node[0]])
            goal = current_node[0]
            return came_from, cost_so_far, goal  # Return the path

        for next_node, cost, action in get_neighbors(current_node[0]):  # Get the
neighbors of the current node
            cost_to_go = ((goal[0] - next_node[0])**2 + (goal[1] - next_node[1])**2)**0.5
            theta_normalized = next_node[2] % 360
            theta_index = theta_normalized // 30
            new_cost = current_node[1] + cost + cost_to_go
            nc = current_node[1] + cost
            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:   #
Check if the new cost is less than the cost so far
                cost_so_far[next_node] = nc  # Update the cost so far
                priority = new_cost   # Calculate the priority
                pq.put((priority, (next_node, nc)))   # Add the node to the priority queue
                canvas_array[int(round(next_node[0])), int(round(next_node[1]))] = np.inf
# Mark the node as visited
                came_from[next_node] = (current_node[0], action)
                count += 1
                if count%10 == 0:
                    canvas_resized = cv2.resize(canvas, (1200, 400),
interpolation=cv2.INTER_AREA)
                    out.write(canvas_resized)
    return None, None, None  # Return None if no path is found

# Function to reconstruct the path
```

```
def reconstruct_path(came_from, start, goal):
    current = goal
    path_with_velocities = [(current, came_from[current][1] if current in came_from else
(0, 0))] # Initialize the path with velocities
    while current != start:
        # Retrieve the current node and the velocities used to reach it.
        node_info = came_from[current]
        prev_node, velocities = node_info
        current = prev_node
        path_with_velocities.append((current, velocities))
    # Reverse the path to get the correct order from start to goal.
    path_with_velocities.reverse()

    return path_with_velocities

# Function to visualize the path
def visualize_path(path):
    V = [] # List to store the velocities
    for i in range(len(path) - 1):
        current_node, velocities = path[i]
        next_node, _ = path[i + 1]
        x, y, t = current_node
        xn, yn, _ = next_node
        x_new = x
        y_new = y
        for linear_vel, angular_vel in velocities:
            V.append((linear_vel/100, -angular_vel)) # Extract the linear and angular
velocities and convert to m/s
            tn = t
            xa = x_new
            ya = y_new
            x_new += linear_vel * np.cos(np.deg2rad(tn)) * 0.1
            y_new += linear_vel * np.sin(np.deg2rad(tn)) * 0.1
            tn += angular_vel * 0.1
            cv2.line(canvas, (int(round(xa)), int(round(ya))), (int(round(x_new)),
int(round(y_new)),), (0, 0, 255), 1) # Draw the path on the canvas
            canvas_resized = cv2.resize(canvas, (1200, 400), interpolation=cv2.INTER_AREA)
            out.write(canvas_resized)
        cv2.line(canvas, (int(round(x_new)), int(round(y_new))), (int(round(xn)),
int(round(yn)),), (0, 0, 255), 1)

    for i in range(30):
        out.write(canvas_resized)
    canvas_resized = cv2.resize(canvas, (1200, 400), interpolation=cv2.INTER_AREA)
    cv2.imshow('Path', canvas_resized)
    return V

# Function to get the clearance distance and robot radius
def get_clearance():
    while True:
        print("Clearance distance should be a positive number")
        clearance_distance = input("Enter the clearance distance: ")           # User
input for clearance distance
        if clearance_distance.isdigit() and int(clearance_distance) >= 0:
            clearance_distance = int(clearance_distance)
            break

    while True:
        print("Robot radius should be a positive number")
        robo_radius = input("Enter the robot radius: ")                        # User input
for robot radius
        if robo_radius.isdigit() and int(robo_radius) >= 0:
            robo_radius = int(robo_radius)
            break
    return clearance_distance, robo_radius

# Function to generate the map
def generate_map():
    print("\nGenerating the map...\n")

    # Generate the map
    for x in range(canvas_width):
        for y in range(canvas_height):
            if clearance(x, y, clearance_distance):
                canvas[y, x] = clearance_color
            if obstacles((x, y)):
                canvas[y, x] = obstacle_color

    canvas_array = np.zeros((canvas_width, canvas_height))
    for x in range(canvas_width):
        for y in range(canvas_height):
            if all(canvas[y, x] != free_space_color):
                canvas_array[x, y] = np.inf
            else:
                canvas_array[x, y] = 0
    return canvas, canvas_array

# Function to get the start node
def get_start():
    C = clearance_distance + robo_radius + 1
    Xc = canvas_width - C
    Yc = canvas_height - C
    while True:
        print(f"\nThe start node and goal node should be within the canvas dimensions
({C}-{Xc}, {C}-{Yc}) and not inside an obstacle.\n")
        Xi = input("Enter the start node X: ")
        Yi = input("Enter the start node Y: ")
        Ti = input("Enter the start node Angle: ")
        Xi = int(Xi)
        Yi = int(Yi)
        Ti = int(Ti)
        if not (Xi < 0 or Xi >= canvas_width or Yi < 0 or Yi >= canvas_height):    # Check
if the start node is within the canvas dimensions
            if is_free(Xi, Yi):
                break
            else:
                print("Start node is inside an obstacle")
        else:
            print("Start node is out of bounds.")
    return Xi, Yi, Ti

# Function to get the goal node
def get_goal():
    C = clearance_distance + robo_radius + 1
    Xc = canvas_width - C
    Yc = canvas_height - C
    while True:
        if mode == '1':
            print(f"\nThe goal node should be within the canvas dimensions ({C}-{Xc}, {C}-
{Yc}) and not inside an obstacle.\n")
        Xg = input("Enter the goal node X: ")
        Yg = input("Enter the goal node Y: ")
        To = input("Enter the goal node Angle: ")
        Xg = int(Xg)
        Yg = int(Yg)
        To = int(To)
        if not (Xg < 0 or Xg >= canvas_width or Yg < 0 or Yg >= canvas_height):    # Check
if the goal node is within the canvas dimensions
            if is_free(Xg, Yg):
                break
            else:
                print("Goal node is inside an obstacle")
        else:
            print("Goal node is inside an obstacle or out of bounds.")
    return Xg, Yg, To

# Function to get the RPM values
def get_rpm():
    while True:
        print("Enter the RPM values for the left and right wheels eg. 10, 20")
        RPM1 = input("Enter the RPM 1: ")
        RPM2 = input("Enter the RPM 2: ")
        if RPM1.isdigit() and RPM2.isdigit():
            RPM1 = int(RPM1)
            RPM2 = int(RPM2)
            break
    return RPM1, RPM2

# check the mode and set the parameters accordingly
if mode == '1':
    print("\nGazebo mode selected.\n")
    clearance_distance = 2
    RPM1 = 10
    RPM2 = 20
    robo_radius = 22
    canvas, canvas_array = generate_map()
    Xi = 50
    Yi = 94
    Ti = 0
    Xg, Yg, To = get_goal()

else:
    print("\n2D mode selected.\n")
    clearance_distance, robo_radius = get_clearance()
    canvas, canvas_array = generate_map()
    RPM1, RPM2 = get_rpm()
    Xi, Yi, Ti = get_start()
    Xg, Yg, To = get_goal()


out = cv2.VideoWriter('A_star.mp4', cv2.VideoWriter_fourcc(*'mp4v'), 30, (canvas_width*2,
canvas_height*2))

Ti = Ti % 360
Ti = round(Ti/30)*30                                      # Round the angle to the
nearest multiple of 30
To = To % 360
To = round(To/30)*30                                     # Round the angle to the
nearest multiple of 30

print("Start Node: ", (int(Xi), int(Yi), int(Ti)))
print("Goal Node: ", (int(Xg), int(Yg), int(To)))
print("\nCalculating the path...")
Yi = abs(canvas_height - int(Yi))
start_node = (int(Xi), int(Yi), int(Ti))

Yg = abs(canvas_height - int(Yg))
goal_node = (int(Xg), int(Yg), int(To))


cv2.circle(canvas, (Xi, Yi), 2, (0, 0, 255), -1)   # Draw the start node
cv2.circle(canvas, (Xg, Yg), 2, (0, 255, 0), -1)  # Draw the goal node
canvas_resized = cv2.resize(canvas, (1200, 400), interpolation=cv2.INTER_AREA)
for j in range(30):
    out.write(canvas_resized)


start_time = time.time()  # Start the timer
came_from, cost_so_far, goal = a_star(start_node, goal_node)  # Run the A* algorithm
if came_from is None:
    print("No path found")
    end_time = time.time()
    execution_time = end_time - start_time         # Calculate the execution time
    print("Execution time: %.4f seconds" % execution_time)
    exit()
path = reconstruct_path(came_from, start_node, goal) # Reconstruct the path
velocities = visualize_path(path) # Visualize the path

end_time = time.time() # Stop the timer
execution_time = end_time - start_time

canvas_resized = cv2.resize(canvas, (1200, 400), interpolation=cv2.INTER_AREA)
for i in range(30):
    out.write(canvas_resized)

out.release()
print("Execution time: %.4f seconds" % execution_time)
cv2.waitKey(0)

# Publish the velocities to the robot
def main(velocities):
    rclpy.init()
    velocity_publisher = VelocityPublisher()
    try:
        for velocity in velocities:
            linear_vel, angular_vel = velocity
            velocity_publisher.publish_velocity(linear_vel, angular_vel)
            time.sleep(0.1)
        velocity_publisher.publish_velocity(0.0, 0.0)  # Stop the robot
    except KeyboardInterrupt:
        pass
    finally:
        velocity_publisher.destroy_node()
        rclpy.shutdown()
if __name__ == '__main__':
    if mode == '1':
        main(velocities)
cv2.destroyAllWindows()
```