```python
################################# GITHUB LINK ###################################
############## https://github.com/sorousheteMad/A_Star_PathPlanning.git #########
#################################################################################

# importing necessary libraries
import numpy as np
import math
import cv2
import queue
import time
from math import dist

# defining the search space(map) and other global variables
ROBOT_RADIUS = 220

map_width = 6000
map_height = 2000
threshold = 100
POINT_SIZE = 5
BUMPER_COLOR = (20, 20, 20)
OBSTACLE_COLOR = (10, 100, 255)

# Defining a class for nodes in the graph
class Node:
    def __init__(self, x, y, parent,theta,UL,UR, c2c, c2g, total_cost):
        self.x = x
        self.y = y
        self.parent = parent
        self.theta = theta
        self.UL = UL
        self.UR = UR
        self.c2c = c2c
        self.c2g = c2g
        self.total_cost = total_cost

    def __lt__(self,other):
        return self.total_cost < other.total_cost

# Define possible actions and associated cost increments
def cost_fn(Xi, Yi, Thetai, UL, UR, Nodes_list, Path_list, obstacle_frame):
    '''
    Xi, Yi,Thetai: Input point's coordinates
    Xs, Ys: Start point coordinates for plot function
    Xn, Yn, Thetan: End point coordintes
    '''

    # Constants for differential drive motion model
    t = 0
    r = 33   # Wheel radius (mm)
    L = 287  # Wheelbase (mm)
    dt = 0.1 # Time step (s)
    cost = 0
    Xn = Xi  # Initialize new point's x-coordinate
    Yn = Yi  # Initialize new point's y-coordinate
    Thetan = 3.14 * Thetai / 180  # Convert orientation from degrees to radians

    # Simulating motion over a small time interval (dt)
    while t < 1:
        t = t + dt
        Xs = Xn
        Ys = Yn
        Xn += r*0.5 * (UL + UR) * math.cos(Thetan) * dt
        Yn += r*0.5 * (UL + UR) * math.sin(Thetan) * dt
        Thetan += (r / L) * (UR - UL) * dt

        if Validity(Xn, Yn, obstacle_frame):
            c2g = dist((Xs, Ys), (Xn, Yn))
```

```
            cost = cost + c2g
            Nodes_list.append((Xn, Yn))  # Append new point to Nodes_list
            Path_list.append((Xs, Ys))   # Append previous point to Path_list
        else:
            return None

    Thetan = 180 * (Thetan) / 3.14
    return [Xn, Yn, Thetan, cost, Nodes_list, Path_list]

# Configuring the obstacle space and constructing the obstacles
def Configuration_space():
    # Initialize an empty grid for obstacle space
    frame = np.full([map_height, map_width, 3], (255, 255, 255)).astype(np.uint8)
    circle_center = (4200, 800)
    radius = 600

    # Adding a black border around the image
    frame[0:CLEARANCE, :] = BUMPER_COLOR  # Top border
    frame[:, 0:CLEARANCE] = BUMPER_COLOR  # Left border
    frame[-CLEARANCE:, :] = BUMPER_COLOR  # Bottom border
    frame[:, -CLEARANCE:] = BUMPER_COLOR  # Right border

    # Adding robot radius to the clearance around the
    frame[CLEARANCE:ROBOT_RADIUS, :] = (255,255,250) # Top border
    frame[:, CLEARANCE:ROBOT_RADIUS] = (255,255,250)  # Left border
    frame[-ROBOT_RADIUS:-CLEARANCE, :] = (255,255,250)  # Bottom border
    frame[:, -ROBOT_RADIUS:-CLEARANCE] = (255,255,250)  # Right border

    # Rectangle 1 obstacle with buffer
    rectangle1 = np.array([
        [(1500, 0), (1750, 0), (1750, 1000), (1500, 1000)]])
    rect1_clearance = np.array([
        [(1500 - CLEARANCE, 0), (1750 + CLEARANCE, 0), (1750 + CLEARANCE, 1000 + CLEARANCE),
(1500 - CLEARANCE, 1000 + CLEARANCE)]])
    rect1_robot = np.array([
        [(1500 - ROBOT_RADIUS, 0), (1750 + ROBOT_RADIUS, 0), (1750 + ROBOT_RADIUS, 1000 +
ROBOT_RADIUS), (1500 - ROBOT_RADIUS, 1000 + ROBOT_RADIUS)]])

    # Rectangle 2 obstacle with buffer
    rectangle2 = np.array([
        [(2500, 2000), (2750, 2000), (2750, 1000), (2500, 1000)]])
    rect2_clearance = np.array([
        [(2500 - CLEARANCE, 2000), (2750 + CLEARANCE, 2000), (2750+CLEARANCE, 1000-CLEARANCE),
(2500-CLEARANCE, 1000-CLEARANCE)]])
    rect2_robot = np.array([
        [(2500 - ROBOT_RADIUS, 2000), (2750 +ROBOT_RADIUS, 2000), (2750 + ROBOT_RADIUS, 1000 -
ROBOT_RADIUS), (2500 - ROBOT_RADIUS, 1000 - ROBOT_RADIUS)]])

    # Filling the Rectangle 1 obstacle for visualization
    cv2.fillPoly(frame, pts=rect1_robot, color=(255, 255, 250))  # robot radius
    cv2.fillPoly(frame, pts=rect1_clearance, color=(0, 0, 0))    # clearance
    cv2.fillPoly(frame, pts=rectangle1, color= OBSTACLE_COLOR)

    # Filling the Rectangle 2 obstacle for visualization
    cv2.fillPoly(frame, pts=rect2_robot, color=(255, 255, 250))  # robot radius
    cv2.fillPoly(frame, pts=rect2_clearance, color=(0, 0, 0))    # clearance
    cv2.fillPoly(frame, pts=rectangle2, color= OBSTACLE_COLOR)

    # Circle Obstacle
    cv2.circle(frame, circle_center, radius + CLEARANCE+ ROBOT_RADIUS, (255,255,250), -1)
    cv2.circle(frame, circle_center, radius + CLEARANCE, BUMPER_COLOR, -1)
    cv2.circle(frame, circle_center, radius, OBSTACLE_COLOR, -1)

    # Creating a copy of frame for obstacle_frame
    obstacle_frame = frame.copy()
    return obstacle_frame
```

```
# Check if orientation is valid (multiples of 30 degrees)
def Valid_Orient(theta):
    if theta % 30 == 0:
        return True
    elif theta == 0:
        return True
    else:
        return False

# Check if coordinates are within the boundaries of the obstacle space(considering the
clearance and buffer)
def Validity(x, y, obstacle_frame):
    # Check if coordinates are within the boundaries of the obstacle space
    if x < 0 or x >= map_width or y < 0 or y >= map_height or
np.array_equal(obstacle_frame[int(y), int(x)], OBSTACLE_COLOR) or
np.array_equal(obstacle_frame[int(y), int(x)], BUMPER_COLOR) or
np.array_equal(obstacle_frame[int(y), int(x)], (255,255,250)):
        return False
    # If the cell is not occupied by an obstacle or buffer, it's considered valid
    return True

# Check if the current node reaches the goal within a threshold distance
def Check_goal(present, goal):
    distance_to_goal = dist((present.x, present.y), (goal.x, goal.y))
    if distance_to_goal < threshold:
        return True

# A* search algorithm implementation for path planning.
def a_star(start, goal, rpm1, rpm2, obstacle_frame):
    # Start time for measuring execution time
    start_time = time.time()

    if Check_goal(start, goal):
        return None, 1  # If start node is already the goal node, return goal found

    # Initialize start and goal nodes
    goal_node = goal
    start_node = start

    # Define possible moves (combinations of wheel RPMs)
    moves = [[rpm1, 0],
             [0, rpm1],
             [rpm1, rpm1],
             [0, rpm2],
             [rpm2, 0],
             [rpm2, rpm2],
             [rpm1, rpm2],
             [rpm2, rpm1]]

    # Initialize dictionaries and priority queue for storing nodes
    unexplored_nodes = {}  # Dictionary to store all open nodes
    unexplored_nodes[(start_node.x, start_node.y)] = start_node

    explored_nodes = {}  # Dictionary to store all explored nodes
    priority_queue = queue.PriorityQueue()  # Priority queue to store nodes based on their
priority
    priority_queue.put((start_node.total_cost, start_node))  # Put the start node into the
priority queue

    Nodes_list = [] # List to store all nodes traversed during the search
    Path_list = []  # List to store all points in the path


    while not priority_queue.empty():
        present_node = priority_queue.get()[1]  # Get the node with the lowest cost from the
priority queue
```

```
        current_id = (present_node.x, present_node.y)
        # Check if the goal node is reached
        if Check_goal(present_node, goal_node):
            # Update goal node attributes
            goal_node.parent = present_node.parent
            goal_node.total_cost = present_node.total_cost
            print("Goal Node found")
            end_time = time.time()  # End time for measuring time taken
            time_taken = end_time - start_time
            print("Time taken to explore:", time_taken, "seconds")
            return 1, Nodes_list, Path_list

        # Check if current node has been explored
        if current_id in explored_nodes:
            continue
        else:
            explored_nodes[current_id] = present_node

        del unexplored_nodes[current_id]  # Remove current node from unexplored nodes

        for move in moves:
            # Calculate cost and new node attributes
            X1 = cost_fn(present_node.x, present_node.y, present_node.theta, move[0], move[1],
                         Nodes_list, Path_list, obstacle_frame)

            if X1 is not None:
                angle = X1[2]
                x = (round(X1[0] * 10) / 10)
                y = (round(X1[1] * 10) / 10)
                th = (round(angle / 15) * 15)
                c2g = dist((x, y), (goal.x, goal.y))

                new_node = Node(x, y, present_node, th, move[0], move[1], present_node.c2c +
X1[3], c2g, present_node.c2c + X1[3] + c2g)
                new_node_id = (new_node.x, new_node.y)

                # Check validity of new node
                if not Validity(new_node.x, new_node.y, obstacle_frame):
                    continue
                elif new_node_id in explored_nodes:
                    continue

                if new_node_id in unexplored_nodes:
                    # Update cost and parent of existing node if new cost is lower
                    if new_node.total_cost < unexplored_nodes[new_node_id].total_cost:
                        unexplored_nodes[new_node_id].total_cost = new_node.total_cost
                        unexplored_nodes[new_node_id].parent = new_node.parent
                else:
                    unexplored_nodes[new_node_id] = new_node

                priority_queue.put((new_node.total_cost, new_node))  # Put the new node into
the priority queue

            # Explore nodes within a radius of 10 units from the current node
            for coord in unexplored_nodes.copy():
                if dist(coord, current_id) <= 100:
                    explored_nodes[coord] = True
                    del unexplored_nodes[coord]
    return 0, Nodes_list, Path_list

def Backtrack(goal_node):
    x_path = []
    y_path = []
    x_path.append(goal_node.x)
    y_path.append(goal_node.y)

    total_cost = goal_node.total_cost
```

```python
    parent_node = goal_node.parent
    while parent_node != -1:
        x_path.append(parent_node.x)
        y_path.append(parent_node.y)
        parent_node = parent_node.parent

    x_path.reverse()
    y_path.reverse()

    return x_path, y_path , total_cost # Return optimal path and total cost

# Define video properties
output_video_path = "Astar_path2_planning_video.mp4"
fps = 60
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
video_out = cv2.VideoWriter(output_video_path, fourcc, fps, (map_width, map_height))

# Frame counter
frame_counter = 0
frame_interval = 50  # Write every 50th frame

if __name__ == '__main__':
    # Get clearance of the obstacle
    while True:
        CLEARANCE = input("Assign Clearance to the Obstacles: ")
        try:
            CLEARANCE = int(CLEARANCE)
            break
        except ValueError:
            print("Invalid input format. Please enter an integer.")

    #obstacle space(map)
    obstacle_frame = Configuration_space()

    # Taking start node coordinates as input from user
    while True:
        start_coordinates = input("Enter coordinates for Start Node (x y): ")
        try:
            s_x, s_y = map(int, start_coordinates.split())
            if not Validity(s_x, s_y, obstacle_frame):
                print("Start node is out of bounds or within the obstacle. Please enter valid 
coordinates.")
                continue
            break
        except ValueError:
            print("Invalid input format. Please enter two integers separated by space.")

    # Taking start node orientation as input from user
    while True:
        start_theta = input("Enter Orientation of the robot at start node (multiple of 30): ")
        try:
            start_theta = int(start_theta)
            if not Valid_Orient(start_theta):
                print("Start orientation has to be a multiple of 30")
                continue
            break
        except ValueError:
            print("Invalid input format. Please enter an integer.")

    # Taking Goal Node coordinates as input from user
    while True:
        goal_coordinates = input("Enter coordinates for Goal Node (x y): ")
        try:
            e_x, e_y = map(int, goal_coordinates.split())
            if not Validity(e_x, e_y, obstacle_frame):
                print("Goal node is out of bounds or within the obstacle. Please enter valid 
coordinates.")
```

```
                continue
            break
        except ValueError:
            print("Invalid input format. Please enter two integers separated by space.")

    # Taking rpms from the user
    while True:
        rpms = input("Enter rpms of the wheels(rpm1 rpm2): ")
        try:
            rpm1,rpm2 = map(int, rpms.split())
            if rpm1 < 0 or rpm2 < 0 or rpm1 > 100 or rpm2 > 100:
                print("Please Enter valid rpms(greater than 0 and less than 100)")
                continue
            break
        except ValueError:
            print("Invalid input format. Please enter an integer.")


    c2g = math.dist((s_x,map_height - s_y), (e_x, map_height - e_y))
    total_cost =  c2g

    # Create start and goal nodes
    start_node = Node(s_x, map_height - s_y,-1,start_theta,0,0,0,c2g,total_cost)
    goal_node = Node(e_x, map_height - e_y, -1,0,0,0,c2g,0,total_cost)

    # Run A* algorithm to find the shortest path
    found_goal,Nodes_list,Path_list = a_star(start_node, goal_node, rpm1, rpm2,
obstacle_frame)

    if found_goal:
        # Generate shortest path
        x_path, y_path , total_cost= Backtrack(goal_node)
        print("total cost:", total_cost)
    else:
        print("Goal not found.")

    # Visualize the map and path
    image_with_path = np.copy(obstacle_frame)


    if found_goal:
        # Draw explored nodes and paths
        for i in range(len(Path_list)):
                    # Get the coordinates of the parent node and the present node
                    parent_node = Path_list[i]
                    present_node = Nodes_list[i]

                    # Draw a line segment from the parent node to the present node
                    cv2.line(image_with_path, (int(parent_node[0]), int(parent_node[1])),
(int(present_node[0]), int(present_node[1])), (0, 255, 0), 1)
                    # Draw circles at the present node
                    cv2.circle(image_with_path, (int(present_node[0]), int(present_node[1])),
POINT_SIZE, (0, 255, 0), -1)
                    frame_counter+=1

                    if frame_counter == frame_interval:
                    # Write the frame to video
                        video_out.write(image_with_path)
                        frame_counter = 0

                    # Display the frame
                    cv2.imshow("Map with Path", image_with_path)
                    cv2.waitKey(1)

        # Draw start and end points
        cv2.circle(image_with_path, (s_x, map_height - s_y),20, (0, 255, 255), -1)  # Green
circle for start point
```

```
        cv2.circle(image_with_path, (e_x, map_height - e_y), 20, (0, 0, 255), -1)  # Red
circle for end point

        # Draw the shortest path
        for i in range(len(x_path) - 1):
            # Convert coordinates to integers
            start_point = (int(x_path[i]), int(y_path[i]))
            end_point = (int(x_path[i + 1]), int(y_path[i + 1]))

            # Draw line segment
            cv2.line(image_with_path, start_point, end_point, (255, 0, 0), 7)

            # write it to the video
            video_out.write(image_with_path)

            cv2.imshow("Map with Path", image_with_path)
            cv2.waitKey(1)

        video_out.write(image_with_path)
        cv2.imshow("Map with path", image_with_path)
        cv2.waitKey(1)

    video_out.release()
    cv2.destroyAllWindows()
```