

```

import numpy as np
import cv2
import time
BLACK = (0, 0, 0)
GREY = (128, 128, 128)

# class to create canvas having obstacle and boundaries
class Canvas:
    def __init__(self, width, height, buffer=2, multiplier=1):
        self.multiplier = multiplier
        self.width = width
        self.height = height
        self.buffer = buffer
        # using 3D array for color visualization in opencv mat
        # WHITE canvas
        self.canvas = np.ones((self.height, self.width, 3), dtype=np.uint8) * 255

        print("Preparing Canvas")
        self._draw_borders()
        self._draw_obstacles()

    # Function to draw borders on canvas
    def _draw_borders(self):
        cv2.rectangle(
            img=self.canvas,
            pt1=(0, 0),
            pt2=(self.width, self.height),
            color=BLACK,
            thickness=self.buffer,
        )

    # Function to visualise the canvas for debugging
    def _visualize_canvas(self):
        resized = cv2.resize(
            self.canvas,
            (int(self.width * self.multiplier), int(self.height * self.multiplier)),
            interpolation=cv2.INTER_AREA,
        )
        cv2.imshow("img", resized)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    # Function calling each type of obstacle to be drawn
    def _draw_obstacles(self):
        for x in range(self.width):
            for y in range(self.height):
                if self.is_inside_obstacle(x, y):
                    self.canvas[y, x] = BLACK
                elif self.is_inside_buffer(x, y):
                    self.canvas[y, x] = GREY

    def is_inside_obstacle(self, x, y):
        pillar1 = (1000 <= x <= 1100) and (500 <= y <= 3000)
        pillar2 = (2100 <= x <= 2200) and (0 <= y <= 2500)
        pillar3 = (3200 <= x <= 3300) and (0 <= y <= 1250)
        pillar4 = (3200 <= x <= 3300) and (1750 <= y <= 3000)
        pillar5 = (4300 <= x <= 4400) and (500 <= y <= 3000)
        return any(
            [
                pillar1,
                pillar2,
                pillar3,
                pillar4,
                pillar5,
            ]
        )

```

```

def is_inside_buffer(self, x, y):
    pillar1 = (1000 - self.buffer <= x <= 1100 + self.buffer) and (
        500 - self.buffer <= y <= 3000
    )
    pillar2 = (2100 - self.buffer <= x <= 2200 + self.buffer) and (
        0 <= y <= 2500 + self.buffer
    )
    pillar3 = (3200 - self.buffer <= x <= 3300 + self.buffer) and (
        0 <= y <= 1250 + self.buffer
    )
    pillar4 = (3200 - self.buffer <= x <= 3300 + self.buffer) and (
        1750 - self.buffer <= y <= 3000
    )
    pillar5 = (4300 - self.buffer <= x <= 4400 + self.buffer) and (
        500 - self.buffer <= y <= 3000
    )
    return any(
        [
            pillar1,
            pillar2,
            pillar3,
            pillar4,
            pillar5,
        ]
    )

def is_colliding(self, x, y):
    return (
        self.canvas[y, x][0] == BLACK[0]
        and self.canvas[y, x][1] == BLACK[1]
        and self.canvas[y, x][2] == BLACK[2]
    ) or (
        self.canvas[y, x][0] == GREY[0]
        and self.canvas[y, x][1] == GREY[1]
        and self.canvas[y, x][2] == GREY[2]
    )

```

```

from math import sqrt

LIN_QUANT = 200
ANG_QUANT = 10

# Point container class
class Point:
    """
    Point class to represent a point in a 2D space. This data structure is also used in as key
    in dict so the thresholding happens here
    """

    def __init__(self, x, y, theta):
        self.x = x
        self.y = y
        self.theta = principal_theta(theta)

    def move(self, dx, dy, dtheta):
        self.x += dx
        self.y += dy
        self.theta += dtheta

    def __hash__(self):
        """Quantize x and y to a 1 mm grid"""
        quantized_x = round(self.x / LIN_QUANT) * LIN_QUANT
        quantized_y = round(self.y / LIN_QUANT) * LIN_QUANT
        # Quantize theta to bins of 30 degrees
        quantized_theta = round(self.theta / ANG_QUANT) * ANG_QUANT
        return hash((quantized_x, quantized_y, quantized_theta))

    def __eq__(self, other):
        """Two nodes are considered equal if they are within 0.5 units and their theta
        difference is ≤ 30 degrees."""
        if isinstance(other, Point):
            euclidean_distance = sqrt((self.x - other.x) ** 2 + (self.y - other.y) ** 2)
            theta_diff = abs(self.theta - other.theta)
            theta_diff = min(
                theta_diff, 360 - theta_diff
            ) # Normalize to range [0, 180]
            return euclidean_distance <= LIN_QUANT and theta_diff <= ANG_QUANT
        return False

    def __str__(self):
        """utility function to print Point directly"""
        return f"Point({self.x}, {self.y}, {self.theta})"

class WayPoint(Point):
    def __init__(self, x, y, theta, edgecost):
        super().__init__(x, y, theta)
        self.edgecost = edgecost

    def __str__(self):
        return f"WayPoint({self.x}, {self.y}, {self.theta}, {self.edgecost})"

# Node container class
def principal_theta(theta):
    """
    Function to find principal theta of a given theta i.e. theta should always be in (-180,180]
    Args:
        theta: theta to be converted

    Returns: normalised theta
    """

```

```
theta = (theta + 180) % 360 - 180 # Ensures theta is in (-180, 180]
return theta
```

```
class Node:
```

```
    """
    A node structure to be used in graph. It represents a valid configuration of robot in
    search graph
    """
```

```
def __init__(self, x, y, theta, c2c):
    self.x = x
    self.y = y
    self.theta = principal_theta(theta)
    self.c2c = c2c
    self.waypoints: list[WayPoint] = None
    self.total_cost = 0
    self.parent = None
    self.visited = False
```

```
# used for duplicate key finding in dict
```

```
def __hash__(self):
    """Quantize x and y to a 1 mm grid"""
    quantized_x = round(self.x / LIN_QUANT) * LIN_QUANT
    quantized_y = round(self.y / LIN_QUANT) * LIN_QUANT
    # Quantize theta to bins of 10 degrees
    quantized_theta = round(self.theta / ANG_QUANT) * ANG_QUANT
    return hash((quantized_x, quantized_y, quantized_theta))
```

```
def __eq__(self, other):
    """Two nodes are considered equal if they are within 0.5 units and their theta
    difference is ≤ 30 degrees."""
    if isinstance(other, Node):
        euclidean_distance = sqrt((self.x - other.x) ** 2 + (self.y - other.y) ** 2)
        theta_diff = abs(self.theta - other.theta)
        theta_diff = min(
            theta_diff, 360 - theta_diff
        ) # Normalize to range [0, 180]
        return euclidean_distance <= LIN_QUANT and theta_diff <= ANG_QUANT
    return False
```

```
def __lt__(self, other):
    if isinstance(other, Node):
        return self.total_cost < other.total_cost
    return False
```

```
def __str__(self):
    """Utility function to print Node directly"""
    return f"Node(x={self.x}, y={self.y}, θ={self.theta}, c2c={self.c2c}, totalcost={self.total_cost})"
```

```
# Container to store goal
```

```
class GoalPt:
```

```
    """
    Goal pt class to hold X Y Radius of goal point
    """
```

```
def __init__(self, x, y, radius):
    self.x = x
    self.y = y
    self.radius = radius
```

```
def __str__(self):
    return f"GoalPt(x={self.x}, y={self.y}, radius={self.radius})"
```

```

from helpers import *
from math import sin, cos, sqrt, pi as PI, radians, degrees
import numpy as np
from scipy.spatial.transform import Rotation as R

class Robot:
    def __init__(self, wheel_radius, robot_radius):
        self.dt = 0.1
        self.R = wheel_radius
        self.L = robot_radius
        self.r = self.L / 2
        self.RPM1 = 5
        self.RPM2 = 10
        self.valid_actions = [
            self.sharp_right_R1,
            self.sharp_left_R1,
            self.straight_R1,
            self.sharp_right_R2,
            self.sharp_left_R2,
            self.straight_R2,
            self.gradual_turn_R1R2,
            self.gradual_turn_R2R1,
        ]

    def transform_map_to_robot(self, x, y, yaw):
        # ---- Pose in Frame A ----
        position_A = np.array([x, y, 0.0])
        orientation_A = R.from_euler('xyz', [0, 0, yaw], degrees=True).as_matrix()

        # Homogeneous transformation matrix of pose in A
        T_pose_in_A = np.eye(4)
        T_pose_in_A[:3, :3] = orientation_A
        T_pose_in_A[:3, 3] = position_A

        # ---- Transform from Frame A to Frame B ----
        # Example: Frame B is rotated and translated w.r.t Frame A
        rotation_AB = R.from_euler('xyz', [-180, 0, 0], degrees=True).as_matrix()
        translation_AB = np.array([0, 0, 0])

        T_AB = np.eye(4)
        T_AB[:3, :3] = rotation_AB
        T_AB[:3, 3] = translation_AB

        # Now transform the pose from frame A to frame B
        T_pose_in_B = np.linalg.inv(T_AB) @ T_pose_in_A

        # Extract the new position and orientation
        position_B = T_pose_in_B[:3, 3]
        orientation_B = R.from_matrix(T_pose_in_B[:3, :3]).as_euler('xyz', degrees=True)

        return position_B[0], position_B[1], orientation_B[2]

    def transform_robot_to_map(self, x, y, yaw):
        # ---- Pose in Frame A ----
        position_A = np.array([x, y, 0.0])
        orientation_A = R.from_euler('xyz', [0, 0, yaw], degrees=True).as_matrix()

        # Homogeneous transformation matrix of pose in A
        T_pose_in_A = np.eye(4)
        T_pose_in_A[:3, :3] = orientation_A
        T_pose_in_A[:3, 3] = position_A

        # ---- Transform from Frame A to Frame B ----
        # Example: Frame B is rotated and translated w.r.t Frame A
        rotation_AB = R.from_euler('xyz', [180, 0, 0], degrees=True).as_matrix()
        translation_AB = np.array([0, 0, 0])

```

```

T_AB = np.eye(4)
T_AB[:3, :3] = rotation_AB
T_AB[:3, 3] = translation_AB

# Now transform the pose from frame A to frame B
T_pose_in_B = np.linalg.inv(T_AB) @ T_pose_in_A

# Extract the new position and orientation
position_B = T_pose_in_B[:3, 3]
orientation_B = R.from_matrix(T_pose_in_B[:3, :3]).as_euler('xyz', degrees=True)

return position_B[0], position_B[1], orientation_B[2]

# Function to get the angular velocity from RPM
def rpm_to_rad_ps(self, RPM):
    return (2 * PI * RPM) / 60

# Defining set of actions based on wheel RPMs
def action(self, node, RPM_RW, RPM_LW):
    # Convert the units from RPM to radians/second

    rx, ry, ryaw = self.transform_map_to_robot(node.x, node.y, node.theta)
    u_l = self.rpm_to_rad_ps(RPM_LW)
    u_r = self.rpm_to_rad_ps(RPM_RW)
    edgcost = 0
    x_i = rx
    y_i = ry
    theta_i = radians(ryaw)

    waypoints: list[WayPoint] = []
    for _ in np.arange(0, 1, 0.1):
        dx = 0.5 * self.R * (u_l + u_r) * cos(theta_i) * self.dt
        dy = 0.5 * self.R * (u_l + u_r) * sin(theta_i) * self.dt
        dtheta = (self.R / self.L) * (u_r - u_l) * self.dt
        dcost = sqrt(dx**2 + dy**2)
        edgcost += dcost
        x_i += dx
        y_i += dy
        theta_i = radians(principal_theta(degrees(theta_i) + degrees(dtheta)))
        tx, ty, tyaw = self.transform_robot_to_map(x_i, y_i, degrees(theta_i))

        waypoints.append(WayPoint(tx, ty, tyaw, edgcost))
    return waypoints, edgcost

def sharp_right_R1(self, node):
    return self.action(
        node, self.RPM1, 0
    ) # Pivoting right by applying RPM1 on the left wheel and 0 on the right wheel

def sharp_left_R1(self, node):
    return self.action(
        node, 0, self.RPM1
    ) # Pivoting left by applying 0 on the left wheel and 0 on the right wheel

def straight_R1(self, node):
    return self.action(
        node, self.RPM1, self.RPM1
    ) # Move straight by applying RPM1 on both wheels

def sharp_right_R2(self, node):
    return self.action(
        node, self.RPM2, 0
    ) # Pivoting right by applying RPM2 on the left wheel and 0 on the right wheel

def sharp_left_R2(self, node):

```

```

    return self.action(
        node, 0, self.RPM2
    ) # Pivoting left by applying 0 on the left wheel and RPM2 on the right wheel

def straight_R2(self, node):
    return self.action(
        node, self.RPM2, self.RPM2
    ) # Move straight by applying RPM2 on both wheels

def gradual_turn_R1R2(self, node):
    return self.action(
        node, self.RPM1, self.RPM2
    ) # Turn by applying RPM1 on right wheel and RPM2 on left wheel

def gradual_turn_R2R1(self, node):
    return self.action(
        node, self.RPM2, self.RPM1
    ) # Turn by applying RPM2 on right wheel and RPM1 on left wheel

```

```

#
#
# r = Robot(33,287)
# ll, dd = r.action(Node(0,0,90,0),5,5)
#
# for node in ll:
#     print(node)

```

```
# Created by arthavnuc
```

```
import numpy as np
import heapq
import time
import cv2
from collections import deque
from canvas import Canvas
from helpers import *
import threading
from robot import Robot
from math import hypot
```

```
DEBUG = False
# Canvas dimensions
WIDTH = 5400
HEIGHT = 3000
```

```
# Globally defined multiplier
MULTIPLIER = 0.25
```

```
# A container to hold multiple time values in key: value pair to be used for analysis at the
termination of program
time_dict = {}
```

```
# Defining some colors for visualization in BGR
# obstacle space
# WHITE denotes free space
WHITE = (255, 255, 255)
# final path tracing
RED = (0, 0, 255)
# explored nodes
GREEN = (0, 255, 0)
NAVY_BLUE = (42, 27, 13)

BLUE = (255, 0, 0)
```

```
if DEBUG:
    file = open("logs.txt", "w")
```

```
class DataQueue:
    def __init__(self, max_size=100):
        self.queue = deque()
        self.max_size = max_size
        self.lock = threading.Lock()

    def put(self, item):
        with self.lock:
            if len(self.queue) == self.max_size:
                self.queue.popleft()
            self.queue.append(item)

    def get(self):
        with self.lock:
            if self.queue:
                return self.queue.popleft()
            return None

    def get_all(self):
        with self.lock:
            items = list(self.queue)
            self.queue.clear()
            return items
```



```

class Search:
    """
    Search class encapsulating the A* algorithm.
    """

    def __init__(self, robot, canvas):
        # A container to store nodes in dictionary. The dict to store only unique nodes
        self.goal_reached = False
        self.nodes_dict = {}
        self.path: list[Node] = []
        # A container to store the nodes
        self.queue = []
        self.robot: Robot = robot
        self.canvas: Canvas = canvas
        self.search_start = self.get_start()
        self.search_goal = self.get_goal()
        self.search_last_node = None
        self.robot.RPM1, self.robot.RPM2 = self.get_wheel_rpms()
        # self.search_start = Point(500, HEIGHT - 300, -90)
        # self.search_goal = GoalPt(5000, HEIGHT - 3000, 150)

    def get_start(self):
        print("Enter Start Coordinates (x y):")
        while True:
            try:
                x_s, y_s, theta_s = map(
                    int,
                    input(
                        f"Start (x y  $\theta$ ) [Note: ( $0 \leq x \leq \{self.canvas.width - 1\}$ ), ( $0 \leq y \leq \{self.canvas.height - 1\}$ ), ( $-180 \leq \theta < 180$ ): "]
                    ).split(),
                )

                if not (0 <= x_s < self.canvas.width and 0 <= y_s < self.canvas.height):
                    print("Coordinates are out of bounds... Try again")
                    continue

                if self.canvas.is_colliding(x_s, y_s):
                    print("Start position is colliding... Try again")
                    continue

                if not (-180 <= theta_s < 180):
                    print("Orientation is out of bounds... Try again")

                print(
                    f"Start point validated: Start (x y  $\theta$ ) = ({x_s}, {y_s}, {theta_s})"
                )
                return Point(x_s, self.canvas.height - y_s, theta_s)

            except ValueError:
                print("Error: Enter three numbers separated by a space")

    def get_goal(self):
        print("Enter Goal Coordinates (x y R):")
        while True:
            try:
                x_g, y_g, radius = map(
                    int,
                    input(
                        f"Goal (x y R) [Note: ( $0 \leq x \leq \{self.canvas.width - 1\}$ ), ( $0 \leq y \leq \{self.canvas.height - 1\}$ ): "]
                    ).split(),
                )

                if not (0 <= x_g < self.canvas.width and 0 <= y_g < self.canvas.height):
                    print("Coordinates are out of bounds... Try again")
                    continue

```

```

        if self.canvas.is_colliding(x_g, y_g):
            print("Goal position is colliding... Try again")
            continue
        if radius <= 0:
            print("Goal can't be negative or zero ... Try again")
            continue
        print(f"Goal point validated: Goal (x y R) = ({x_g}, {y_g}, {radius})")
        return GoalPt(x_g, self.canvas.height - y_g, radius)

    except ValueError:
        print("Error: Enter three numbers separated by a space")

def get_wheel_rpms(self):
    while True:
        try:
            RPM1, RPM2 = map(
                float,
                input(f"Enter Wheel RPMs (RPM1 RPM2) (must be > 0): ").split(),
            )

            if RPM1 <= 0 or RPM2 <= 0:
                print("RPM values cannot be non-positive")
                continue

            print(f"Wheel RPMs validated: (RPM1 RPM2) = ({RPM1}, {RPM2})")
            return RPM1, RPM2

        except ValueError:
            print("Invalid input. Please enter numeric values for RPMs.")

def reached_goal(self, x, y, goal: GoalPt):
    """
    Termination condition if robot reached the goal region, In this Project ignoring the
    goal theta so basically robot can be in any orientation in goal region
    :param x: Current X coordinate
    :param y: Current Y coordinate
    :param goal: Goal region (X,Y,Radius)
    :return: Boolean
    """
    if (x - goal.x) ** 2 + (y - goal.y) ** 2 < goal.radius**2:
        return True
    else:
        return False

def plotter(self, dataq: DataQueue, stop_event: threading.Event):
    canvas_copy = self.canvas.canvas.copy()
    while not stop_event.is_set():
        new_data = dataq.get_all()
        if new_data:
            for x, y in new_data:
                cv2.circle(
                    canvas_copy,
                    (int(x), int(y)),
                    5,
                    RED,
                    5,
                )
            scaled_canvas = cv2.resize(
                canvas_copy,
                (
                    int(self.canvas.width * self.canvas.multiplier),
                    int(self.canvas.height * self.canvas.multiplier),
                ),
                interpolation=cv2.INTER_AREA,
            )
            cv2.imshow(f"Map Scaled {self.canvas.multiplier} times", scaled_canvas)

```

```
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
def heuristic(self, node: Point, goal):
```

```
    """
```

```
    Heuristic function for A*. Since C-space is  $R^2$  using Euclidean distance
    Args:
```

```
        node: point 1
```

```
        goal: point 2
```

```
    Returns: Euclidean distance between node and goal
```

```
    """
```

```
    return hypot(goal.x - node.x, goal.y - node.y)
```

```
def a_star(self):
```

```
    """
```

```
    Main function for A* algorithm. All the valid actions are defined here. The robot is
    non-holonomic and consists of 8 actions given as an differential RPM pair:
```

```
    This includes (RPM1, 0), (0, RPM1), (RPM2, 0), (0, RPM2), (RPM1, RPM2), (RPM2, RPM1),
    (RPM1, RPM1), (RPM2, RPM2)
```

```
    Args:
```

```
        start: start point (X,Y,Theta) in graph
```

```
        goal: goal region (X,Y,Radius) in graph
```

```
    Returns: Boolean: True is an optimal path is found, False otherwise
```

```
    """
```

```
    # start_time = time.perf_counter()
```

```
    # Create grid of x, y coordinates
```

```
    # stop_event = threading.Event()
```

```
    # data_queue = DataQueue()
```

```
    # self.plotter_thread = threading.Thread(
```

```
    #     target=self.plotter,
```

```
    #     args=(data_queue, stop_event),
```

```
    # )
```

```
    # self.plotter_thread.start()
```

```
self.nodes_dict.clear()
```

```
start_node = Node(
```

```
    self.search_start.x, self.search_start.y, self.search_start.theta, 0
```

```
)
```

```
start_node.total_cost = self.heuristic(self.search_start, self.search_goal)
```

```
valid_actions = self.robot.valid_actions
```

```
print(f"Start:{start_node}")
```

```
print(f"Goal:{self.search_goal}")
```

```
heapq.heappush(self.queue, start_node)
```

```
self.goal_reached = False
```

```
self.nodes_dict = {
```

```
    Point(start_node.x, start_node.y, start_node.theta): start_node
```

```
}
```

```
early_exit = False
```

```
try:
```

```
    while self.queue:
```

```
        if early_exit:
```

```
            break
```

```
        node: Node = heapq.heappop(self.queue)
```

```
        # TODO do we need this ??
```

```
        if self.reached_goal(node.x, node.y, self.search_goal):
```

```
            self.nodes_dict["last"] = node
```

```
            self.search_last_node = node
```

```

        self.goal_reached = True
        print(f"Goal found at {node}")
        break

if DEBUG:
    print(f"Exploring node: {node}")
    file.write(f"Exploring node: {node}\n")

for act_idx, action in enumerate(valid_actions):
    waypoints, cost = action(node)
    # Check waypoints and endpoint for collisions with the obstacles or the
    wall

    is_colliding = False
    is_wp_near_goal = False
    wp_goal_idx: int = None
    for wp_idx, wp in enumerate(waypoints):
        # data_queue.put((wp.x, wp.y))
        if self.canvas.is_colliding(round(wp.x), round(wp.y)) or not (
            0 <= wp.x < self.canvas.width
            and 0 <= wp.y < self.canvas.height
        ):
            is_colliding = True
            break

        # Not colliding, but check for the cornercase where the waypoint
        itself is the near the Goal
        if self.reached_goal(wp.x, wp.y, self.search_goal):
            is_wp_near_goal = True
            wp_goal_idx = wp_idx
            break

    if is_colliding:
        if DEBUG:
            print(
                f"Collision detected while exploring {node}, discarding action
{act_idx}"
            )
            file.write(
                f"Collision detected while exploring {node}, discarding action
{act_idx}\n"
            )
        continue # At least one waypoint is colliding, so exclude this action
        and proceed with the next one

    if is_wp_near_goal:
        # The waypoint is near the goal, so add it as a node and proceed
        wp_goal: WayPoint = waypoints[wp_goal_idx]
        wp_goal_node: Node = Node(
            wp_goal.x,
            wp_goal.y,
            wp_goal.theta,
            node.c2c + wp_goal.edgecost,
        )
        wp_goal_node.total_cost = wp_goal_node.c2c + self.heuristic(
            Point(wp_goal_node.x, wp_goal_node.y, wp_goal_node.theta),
            self.search_goal,
        )
        wp_goal_node.visited = True
        wp_goal_node.parent = node
        wp_goal_node.waypoints = waypoints[:wp_goal_idx]
        self.nodes_dict["last"] = wp_goal_node
        self.nodes_dict[
            Point(wp_goal_node.x, wp_goal_node.y, wp_goal_node.theta)
        ] = wp_goal_node
        self.search_last_node = wp_goal_node
        print(f"Goal found near waypoint at {wp_goal_node}")
        if DEBUG:

```

```

        file.write(f"Goal found near waypoint at {wp_goal_node}\n")
        early_exit = True
        self.goal_reached = True
        break

    # Check in dictionary if the node exists at the point, else create a new
    # node at the last waypoint
    next_node = self.nodes_dict.get(
        Point(
            waypoints[-1].x,
            waypoints[-1].y,
            waypoints[-1].theta,
        ),
        Node(
            waypoints[-1].x,
            waypoints[-1].y,
            waypoints[-1].theta,
            0,
        ),
    )
    if (next_node.c2c > node.c2c + cost) or not next_node.visited:
        # Updating total cost to ctoc with heuristic, total cost and parent
        # and marking it as visited
        next_node.c2c = node.c2c + cost
        next_node.total_cost = next_node.c2c + self.heuristic(
            Point(next_node.x, next_node.y, 0), self.search_goal
        )
        next_node.visited = True
        next_node.parent = node
        heapq.heappush(self.queue, next_node)
        self.nodes_dict[
            Point(next_node.x, next_node.y, next_node.theta)
        ] = next_node
        next_node.waypoints = waypoints
        if DEBUG:
            print(f"Added node: {next_node}")
            file.write(f"Added node: {next_node}\n")
        elif DEBUG:
            print(
                f"Discarded node: {next_node} because {next_node.visited == False}
and {next_node.c2c > node.c2c + cost}"
            )
            file.write(
                f"Discarded node: {next_node} because {next_node.visited == False}
and {next_node.c2c > node.c2c + cost}\n"
            )

    except KeyboardInterrupt:
        pass

    finally:
        # stop_event.set()
        # self.plotter_thread.join()
        cv2.destroyAllWindows()
        print("Program terminated.")

    if not self.goal_reached:
        print(f"No path found! Queue: {self.queue}")
        return False
    return True

def backtrack_path(self):
    """
    Backtracks from the goal to the start using the dict
    """
    start_time = time.perf_counter()
    path = []

```

```

# Backtracking using the parent in node
g = self.nodes_dict.pop("last")
for _ in iter(int, 1):
    path.append(g)
    if g.parent.x == self.search_start.x and g.parent.y == self.search_start.y:
        break
    g = g.parent
path.append(g)
path.reverse()
end_time = time.perf_counter()
time_dict["Backtracking"] = end_time - start_time
self.path: list[Node] = path

def animate_search(self):
    """
    Visualizes the explored nodes of graph.
    """
    if not self.goal_reached:
        return

    canvas_copy = self.canvas.canvas.copy()
    cv2.circle(
        canvas_copy,
        (round(self.search_start.x), round(self.search_start.y)),
        15,
        GREEN,
        15,
    )
    cv2.circle(
        canvas_copy,
        (round(self.search_goal.x), round(self.search_goal.y)),
        self.search_goal.radius,
        BLUE,
        5,
    )

    for point, node in self.nodes_dict.items():
        for wp in node.waypoints:
            cv2.circle(canvas_copy, (int(wp.x), int(wp.y)), 5, RED, 5)

    scaled_canvas = cv2.resize(
        canvas_copy,
        (
            round(self.canvas.width * self.canvas.multiplier),
            round(self.canvas.height * self.canvas.multiplier),
        ),
        interpolation=cv2.INTER_AREA,
    )
    cv2.imshow(f"Map Scaled {self.canvas.multiplier} times", scaled_canvas)
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

    for node in self.path:
        for wp in node.waypoints:
            cv2.circle(canvas_copy, (round(wp.x), round(wp.y)), 5, NAVY_BLUE, 5)

    scaled_canvas = cv2.resize(
        canvas_copy,
        (
            round(self.canvas.width * self.canvas.multiplier),
            round(self.canvas.height * self.canvas.multiplier),
        ),
        interpolation=cv2.INTER_AREA,
    )
    cv2.imshow(f"Map Scaled {self.canvas.multiplier} times", scaled_canvas)
    time.sleep(0.1)
    if cv2.waitKey(1) & 0xFF == ord("q"):

```

break

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
if __name__ == "__main__":
    robot = Robot(33, 287)
    start_time = time.perf_counter()
    canvas = Canvas(WIDTH, HEIGHT, round(2 + robot.r), MULTIPLIER)
    end_time = time.perf_counter()
    time_dict["Map generated in "] = end_time - start_time
    search = Search(robot, canvas)
    start_time = time.perf_counter()
    success = search.a_star()
    end_time = time.perf_counter()
    time_dict["ASTAR"] = end_time - start_time
    if success:
        search.backtrack_path()
        search.animate_search()
        print("Search success, path found")

    for time in time_dict:
        print(time, time_dict[time], "seconds")
```