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

import numpy as np
import cv2 as cv
import heapq
import math
import time
import matplotlib.pyplot as plt
from tqdm import tqdm


# Define colors globally
COLOR_BACKGROUND = (255, 255, 225)
COLOR_OBS = (0, 0, 0)
COLOR_CLEARANCE = (128, 128, 128)
COLOR_START = (255, 0, 0)
COLOR_GOAL = (0, 255, 0)
COLOR_EXPLORE = (0, 0, 255)
COLOR_ROBOT_RADIUS = (255, 0, 0)
COLOR_PATH = (0, 0, 128)


#############################################
# Map Environment with Obstacles
#############################################
class MapEnv:
    # The way the map is generated is not from scratch, but using the measurements from the previous map used for djikstra and BFS and then scaling each point
    # I found through testing that this is the most accurate way of scaling up the map, as putting a global scale factor of x = 3.33 and y = 5 leads to inaccur
    # half plane calculations, leading to the robot going through the obstacles or either ignoring the obstacles completely. There might be an easier way, but
    def __init__(
        self,
        map_width,
        map_height,
        final_canvas_width,
        final_canvas_height,
        robot_radius,
        clearance,
    ):
        """
        Initialize the Map Environment with obstacles.

        Parameters:
            map_width (int): Width of the raw map (e.g., 180).
            map_height (int): Height of the raw map (e.g., 50).
            final_canvas_width (int): Width of the final canvas (e.g., 600).
            final_canvas_height (int): Height of the final canvas (e.g., 250).
            robot_radius (float): The radius of the robot.
            clearance (float): Additional clearance required around obstacles.

        Process:
            1. Store map and robot parameters.
            2. Compute scale factors to convert raw map coordinates to canvas coordinates.
            3. Retrieve raw obstacle coordinates via pointCoords().
            4. Scale the raw obstacles to the canvas dimensions.
            5. Inflate the scaled obstacles by (clearance + robot_radius).
            6. Further scale the inflated obstacles for planning in a 60x25 grid.
        """

        # Set the raw dimensions and robot parameters.
        # Compute scaling factors from raw map dimensions to final canvas dimensions.
        # Retrieve raw obstacles using pointCoords().
        # Scale polygon obstacles (line_obs) and circle/arc obstacles (gol_obs).
        # Inflate obstacles using inflate_obstacles() with inflation = clearance + robot_radius.
        # Create a further scaled version (calc_scale) for planning (grid of 60x25).

        self.map_width = map_width  # 180
        self.map_height = map_height  # 50
        self.final_canvas_width = final_canvas_width  # 600
        self.final_canvas_height = final_canvas_height  # 250
        self.robot_radius = robot_radius
        self.clearance = clearance
        self.scalex = final_canvas_width / map_width  # 3.333
        self.scaley = final_canvas_height / map_height  # 5
        # Step 1: Raw obstacles in 180x50
        self.raw_line_obs, self.raw_gol_obs = self.pointCoords()
        # Step 2: Scale to 600x250 (uninflated)
        self.line_obs_uninflated = [
            [[x * self.scalex, y * self.scaley] for x, y in poly]
            for poly in self.raw_line_obs
        ]
        self.gol_obs_uninflated = []
        for group in self.raw_gol_obs:
            scaled_group = []
            for gol in group:
                x, y, r = gol[:3]
                scaled_gol = [x * self.scalex, y * self.scaley, r * self.scalex]
                if len(gol) == 5:
                    scaled_gol.extend(gol[3:])
                scaled_group.append(scaled_gol)
            self.gol_obs_uninflated.append(scaled_group)
        # Step 4: Inflate the scaled obstacles
        self.line_obs_inflated, self.gol_obs_inflated = self.inflate_obstacles(
            self.line_obs_uninflated, self.gol_obs_uninflated, clearance + robot_radius
        )
        # For planning (60x25 grid)
        calc_scale = 0.1
        self.scaled_line_obs = [
            [[x * calc_scale, y * calc_scale] for x, y in poly]
            for poly in self.line_obs_inflated
        ]
        self.scaled_gol_obs = []
        for group in self.gol_obs_inflated:
            scaled_group = []
            for gol in group:
                x, y, r = gol[:3]
                scaled_gol = [x * calc_scale, y * calc_scale, r * calc_scale]
                if len(gol) == 5:
                    scaled_gol.extend(gol[3:])
                scaled_group.append(scaled_gol)
            self.scaled_gol_obs.append(scaled_group)

    def pointCoords(self):
        """
        Define the raw coordinates for obstacles.

        Returns:
            tuple: (line_obs, gol_obs)
                - line_obs: List of polygon obstacles (each polygon is a list of [x, y] points).
                - gol_obs: List of circle/arc obstacles (each may have 3 or 5 parameters; arcs include extra angles).

        Comments:
            - The obstacles are defined in the raw map space (180x50).
            - For example, obstacles are defined to resemble the letters "E", "N", "P" and "M" and the numbers "6","6","1".
        """
        # Return the two lists: one for polygon obstacles and one for gol (Circular) obstacles.
        line_obs = [
            # 1
            [[158.5, 12.5], [158.5, 40.5], [163.5, 40.5], [163.5, 12.5]],
            # E
            [
                [15, 12.5],
                [15, 37.5],
                [28, 37.5],
                [28, 32.5],
                [20, 32.5],
                [20, 27.5],
                [28, 27.5],
                [28, 22.5],
                [20, 22.5],
                [20, 17.5],
                [28, 17.5],
                [28, 12.5],
            ],
            # N
            [
                [35.5, 12.5],
                [35.5, 37.5],
                [40.5, 37.5],
                [45.5, 22.5],
                [45.5, 37.5],
                [50.5, 37.5],
                [50.5, 12.5],
                [45.5, 12.5],
                [40.5, 27.5],
                [40.5, 12.5],
            ],
            # P Vert
            [
                [58, 12.5],
                [58, 37.5],
                [63, 37.5],
                [63, 12.5],
            ],
            # M
            [
                [76.5, 12.5],
                [76.5, 37.5],
                [81.5, 37.5],
                [86.5, 17.5],
                [91.5, 17.5],
                [96.5, 37.5],
                [101.5, 37.5],
                [101.5, 12.5],
                [96.5, 12.5],
                [96.5, 31.5],
                [92.5, 12.5],
                [85.5, 12.5],
                [81.5, 31.5],
                [81.5, 12.5],
            ],
        ]

        gol_obs = [
            # P circle (standalone)
            [
                [65.3, 33.6, 6],
            ],
            # first 6
            [
                [117.5, 18.5, 7],  # inner circle (hole)
                [119.5, 20.5, 12],  # outer circle (donut)
                [129.6, 24.2, 22.0, 120, 190],  # outer arc (donut)
                [132, 24.2, 15.0, 90, 270],  # inner arc (hole)
                [120.1, 34.9, 3],  # small circle (full obstacle)
            ],
            # second 6
            [
                [142, 18.5, 7],
                [145, 20.5, 12],
                [154.7, 24.2, 21.5, 120, 190],
                [160.5, 24.2, 14.5, 90, 270],
                [145.27, 34.5, 3],
            ],
        ]
        return line_obs, gol_obs

    def inflate_obstacles(self, obstacles, gols, inflation):
        """
        Inflate the obstacles by a specified inflation (Clearance) amount.

        Parameters:
            obstacles (list): List of scaled polygon obstacles.
            gols (list): List of scaled circle/arc obstacles.
            inflation (float): The inflation amount (clearance + robot_radius).

        Returns:
            tuple: (inflated, inflated_gols)
                - inflated: List of inflated polygon obstacles.
                - inflated_gols: List of inflated circle/arc obstacles.

        Comments:
            - For polygon obstacles, each vertex is offset outward based on the average normal of its adjacent edges.
            - For gol obstacles:
                * Standalone circles are simply inflated (radius increased).
                * For arc obstacles (where a pair of arcs is detected), the outer arc is expanded (r + inflation)
                  and the inner arc is contracted (r - inflation) to preserve the annular gap.
        """
        # For each polygon, loop through vertices and compute new positions by averaging the normals.
        # For each gol obstacle group, check for arc pairs (indices 2 and 3) and adjust radii accordingly.
        # Return the inflated obstacles for both polygons and gol obstacles.
        inflated = []
        for obstacle in obstacles:
            if not obstacle or len(obstacle) < 2:
                continue
            inflated_poly = []
            n = len(obstacle)
            for i in range(n):
                x, y = obstacle[i]
                prev_x, prev_y = obstacle[(i - 1) % n]
                next_x, next_y = obstacle[(i + 1) % n]
                dx1, dy1 = x - prev_x, y - prev_y
                nx1, ny1 = -dy1, dx1
                L1 = np.hypot(nx1, ny1)
                nx1, ny1 = (nx1 / L1, ny1 / L1) if L1 > 0 else (0, 0)
                dx2, dy2 = next_x - x, next_y - y
                nx2, ny2 = -dy2, dx2
                L2 = np.hypot(nx2, ny2)
                nx2, ny2 = (nx2 / L2, ny2 / L2) if L2 > 0 else (0, 0)
                nx, ny = (nx1 + nx2) / 2, (ny1 + ny2) / 2
                L = np.hypot(nx, ny)
                nx, ny = (nx / L, ny / L) if L > 0 else (0, 1)
                inflated_poly.append([x + inflation * nx, y + inflation * ny])
            if inflated_poly:
                inflated.append(inflated_poly)

        inflated_gols = []
        for gol_group in gols:
            scaled_group = []
            # Check if this group has an arc pair (assume indices 2 and 3 hold the arc pair)
            if (
                len(gol_group) >= 4
                and len(gol_group[2]) == 5
                and len(gol_group[3]) == 5
            ):
                # Process earlier items (if any) normally.
                for i in range(min(2, len(gol_group))):
                    gol = gol_group[i]
                    x, y, r = gol[:3]
                    new_r = r + inflation
                    if len(gol) == 3:
                        scaled_group.append([x, y, new_r])
                    elif len(gol) == 5:
                        scaled_group.append([x, y, new_r, gol[3], gol[4]])
                # For the arc pair:
                #   Outer arc: inflate outward (r + inflation)
                outer = gol_group[2]
                x_o, y_o, r_outer = outer[:3]
                new_r_outer = r_outer + inflation
                scaled_group.append([x_o, y_o, new_r_outer, outer[3], outer[4]])
                #   Inner arc: contract inward (r - inflation), ensuring non-negative radius.
                inner = gol_group[3]
                x_i, y_i, r_inner = inner[:3]
                new_r_inner = max(r_inner - inflation, 0)
                scaled_group.append([x_i, y_i, new_r_inner, inner[3], inner[4]])
                # Process any remaining items normally.
                for gol in gol_group[4:]:
                    x, y, r = gol[:3]
                    new_r = r + inflation
                    if len(gol) == 3:
                        scaled_group.append([x, y, new_r])
                    elif len(gol) == 5:
                        scaled_group.append([x, y, new_r, gol[3], gol[4]])
            else:
                # Process groups that are not in the arc pair format normally.
                for gol in gol_group:
                    x, y, r = gol[:3]
                    new_r = r + inflation
                    if len(gol) == 3:
                        scaled_group.append([x, y, new_r])
                    elif len(gol) == 5:
                        scaled_group.append([x, y, new_r, gol[3], gol[4]])
            if scaled_group:
                inflated_gols.append(scaled_group)

        return inflated, inflated_gols

    def createCanvas(self, width=600, height=250):
        """
        Create a blank white canvas for drawing obstacles.

        Parameters:
            width (int): The width of the canvas.
            height (int): The height of the canvas.

        Returns:
            numpy.ndarray: A blank white canvas image.

        Comments:
            - The canvas is created as an array filled with 255 (white).
            - Optionally, the canvas can be flipped vertically (the flip is currently commented out, this can be used for testing later, thus is still included
        """
        # Initialize and return a white canvas of the specified dimensions using NumPy.
        canvas = np.full((height, width, 3), COLOR_BACKGROUND, dtype=np.uint8)
        # return cv.flip(canvas, 0) - Was used for testing when the canvas was flipped by cv2 and did not have a fix in main
        return canvas

    def drawObs(self, canvas):
        """
        Draw the uninflated obstacles on the given canvas.

        Parameters:
            canvas (numpy.ndarray): The canvas on which to draw obstacles.

        Returns:
            numpy.ndarray: The canvas with obstacles drawn in black.

        Comments:
            - Draw polygon obstacles using cv.fillPoly.
            - For each gol obstacle, if it is a circle use cv.circle; if it is an arc, use cv.ellipse.
            - Handles grouped obstacles (e.g., donut shapes) by processing each component separately.
            - This function is only used for drawing the obsctacles for visualization but not used for calcualations.
        """
        # Loop through each polygon obstacle, convert the points to integer format, and fill the polygon.
        # Loop through each gol obstacle group and draw each obstacle accordingly (using circle or ellipse).
        # Use masks for the accurate representation of arcs, cutting out the areas that are not required, resulting in accurate annular shapes.

        for obs in self.line_obs_uninflated:
            pts = np.array(obs, dtype=np.int32)
            pts = pts.reshape(-1, 1, 2)
            cv.fillPoly(canvas, [pts], COLOR_OBS)
        for group in self.gol_obs_uninflated:
            if len(group) == 1:
                gol = group[0]
                if len(gol) == 3:
                    center = (int(gol[0]), int(gol[1]))
                    radius = int(gol[2])
                    cv.circle(canvas, center, radius, (0, 0, 0), thickness=-1)
                elif len(gol) == 5:
                    center = (int(gol[0]), int(gol[1]))
                    radius = int(gol[2])
                    cv.ellipse(
                        canvas,
                        center,
                        (radius, radius),
                        0,
                        gol[3],
                        gol[4],
                        COLOR_OBS,
                        thickness=-1,
                    )
            else:
                for i in range(min(2, len(group))):
                    if len(group[i]) == 3:
                        center = (int(group[i][0]), int(group[i][1]))
                        radius = int(group[i][2])
                        cv.circle(canvas, center, radius, COLOR_OBS, thickness=-1)
                    elif len(group[i]) == 5:
                        center = (int(group[i][0]), int(group[i][1]))
                        radius = int(group[i][2])
                        cv.ellipse(
                            canvas,
                            center,
                            (radius, radius),
                            0,
                            group[i][3],
                            group[i][4],
                            (0, 0, 0),
                            thickness=-1,
                        )
                if len(group) >= 4 and len(group[2]) == 5 and len(group[3]) == 5:
                    outer = group[2]
                    inner = group[3]
                    center = (int(outer[0]), int(outer[1]))
                    outer_radius = int(outer[2])
                    inner_radius = int(inner[2])
                    start_angle = outer[3]
                    end_angle = outer[4]
                    mask = np.zeros(canvas.shape[:2], dtype=np.uint8)
                    cv.ellipse(
                        mask,
                        center,
                        (outer_radius, outer_radius),
                        0,
                        start_angle,
                        end_angle,
                        255,
                        thickness=-1,
                    )
                    cv.ellipse(
                        mask,
                        center,
                        (inner_radius, inner_radius),
                        0,
                        start_angle,
                        end_angle,
                        0,
                        thickness=-1,
                    )
                    canvas[mask == 255] = COLOR_OBS
                if len(group) > 4:
                    for item in group[4:]:
                        if len(item) == 3:
                            center = (int(item[0]), int(item[1]))
                            radius = int(item[2])
```

```python
                            cv.circle(canvas, center, radius, COLOR_OBS, thickness=-1)
                        elif len(item) == 5:
                            center = (int(item[0]), int(item[1]))
                            radius = int(item[2])
                            cv.ellipse(
                                canvas,
                                center,
                                (radius, radius),
                                0,
                                item[3],
                                item[4],
                                COLOR_OBS,
                                thickness=-1,
                            )
        return canvas

    def halfPlanes(self, x, y):
        """
        Check whether a point (x, y) is inside any obstacle using a half-plane representation.

        Parameters:
            x (float): The x-coordinate (in planning space).
            y (float): The y-coordinate (in planning space).

        Returns:
            bool: True if the point is inside an obstacle; False otherwise.

        Comments:
            - For polygon obstacles, uses the point_in_poly() method to test inclusion.
            - For circle obstacles, checks if the distance from the point to the center is less than the inflated radius.
            - For arc obstacles, also checks if the point's angle relative to the obstacle center falls within the arc's angular limits.
        """
        # Check each scaled polygon obstacle using point_in_poly().
        # For each gol obstacle, calculate the squared distance and, if necessary, the angle to decide collision.
        # Return True immediately if any collision is detected; otherwise, return False.
        # Has a check to see if the area of the arc falls strictly within the limits of the start and end angle to prevent the half planes function from treati
        # obstacle.
        for poly in self.scaled_line_obs:
            if self.point_in_poly(x, y, poly):
                return True
        for group in self.scaled_gol_obs:
            for gol in group:
                cx, cy, r = gol[:3]
                dist_sq = (x - cx) ** 2 + (y - cy) ** 2
                if len(gol) == 3:
                    if dist_sq <= r**2:
                        return True
                elif len(gol) == 5:
                    sa, ea = gol[3], gol[4]  # e.g., 180, 120
                    if dist_sq <= r**2:
                        angle = np.degrees(np.arctan2(y - cy, x - cx))
                        angle = (angle + 360) % 360
                        if sa > ea:  # 180 to 120
                            # Block only the arc from sa to ea counterclockwise
                            if angle >= sa or angle <= ea:
                                return True
                        else:
                            if sa <= angle <= ea:
                                return True
        return False

    def drawClearances(self, canvas):
        """
        Draw the inflated clearances (obstacle buffers) on the canvas.

        Parameters:
            canvas (numpy.ndarray): The canvas on which to draw the clearances.

        Returns:
            numpy.ndarray: The canvas with clearances drawn in gray.

        Comments:
            - Fill the inflated polygon obstacles with a gray color.
            - For gol obstacles, if a circle then draw a filled circle in gray.
            - For arc obstacles, use mask techniques: draw the outer arc, then subtract the inner arc to represent the clearance.
            - This function, like the draw obstacles function, is completely visual and is not considered when calculating half planes.
        """
        # For each inflated polygon obstacle, reshape and fill it with gray.
        # For each inflated gol obstacle, draw using circle or ellipse based on the obstacle type.
        # For arc obstacles, create masks for the outer and inner parts and subtract them to form the annular clearance.

        for poly in self.line_obs_inflated:
            if poly:
                pts = np.array(poly, dtype=np.int32)
                if pts.size == 0:
                    continue
                pts = pts.reshape(-1, 1, 2)
                cv.fillPoly(canvas, [pts], COLOR_CLEARANCE)
        for group in self.gol_obs_inflated:
            for gol in group:
                if len(gol) == 3:
                    center = (int(gol[0]), int(gol[1]))
                    radius = int(gol[2])
                    cv.circle(canvas, center, radius, COLOR_CLEARANCE, thickness=-1)
                elif len(gol) == 5:
                    center = (int(gol[0]), int(gol[1]))
                    print(center)
                    outer = group[2]
                    inner = group[3]
                    center = (int(outer[0]), int(outer[1]))
                    outer_radius = int(outer[2])
                    inner_radius = int(inner[2])
                    start_angle = outer[3]
                    end_angle = outer[4]

                    # start_angle = gol[3]
                    # end_angle = gol[4]
                    # Create a mask for the outer arc clearance
                    outer_mask = np.zeros(canvas.shape[:2], dtype=np.uint8)
                    cv.ellipse(
                        outer_mask,
                        center,
                        (outer_radius, outer_radius),
                        0,
                        start_angle,
                        end_angle,
                        255,
                        thickness=-1,
                    )

                    # Create a mask for the inner arc (area to subtract)
                    inner_mask = np.zeros(canvas.shape[:2], dtype=np.uint8)
                    cv.ellipse(
                        inner_mask,
                        center,
                        (inner_radius, inner_radius),
                        0,
                        start_angle,
                        end_angle,
                        255,
                        thickness=-1,
                    )

                    # Subtract the inner mask from the outer mask to get the clearance area
                    clearance_mask = cv.subtract(outer_mask, inner_mask)

                    # Apply the clearance mask to the canvas with the desired clearance color (gray)
                    canvas[clearance_mask == 255] = COLOR_CLEARANCE

        return canvas

    def point_in_poly(self, x, y, poly):
        """
        Determine if a point (x, y) lies within a polygon defined by 'poly'.

        Parameters:
            x (float): The x-coordinate of the point.
            y (float): The y-coordinate of the point.
            poly (list): A list of [x, y] pairs representing the polygon vertices.

        Returns:
            bool: True if the point is inside the polygon; False otherwise.

        Comments:
            - Implements the ray-casting algorithm to count edge crossings.
            - Iterates over each edge of the polygon and toggles an 'inside' flag when a crossing is detected.
            - Helper function for the half planes calculation, reducing time for iterating over each obstacle, by determining whether a point is in a polygon a
        """
        # Initialize an "inside" flag to False.
        # Loop through each edge of the polygon, checking if the horizontal ray from the point crosses the edge.
        # Toggle the flag accordingly and return the final status.
        inside = False
        n = len(poly)
        j = n - 1
        for i in range(n):
            xi, yi = poly[i]
            xj, yj = poly[j]
            if ((yi > y) != (yj > y)) and (x < (xj - xi) * (y - yi) / (yj - yi) + xi):
                inside = not inside
            j = i
        return inside


class Node:
    def __init__(self, state, parent, cost2come, cost2go):
        """
        Initialize a Node for the A* search algorithm.

        Parameters:
            state (tuple): The current state of the node, typically in the form (x, y, theta).
            parent (Node): The parent node from which this node was generated.
            cost2come (float): The cumulative cost from the start node to this node.
            cost2go (float): The heuristic estimate of the cost from this node to the goal.

        Comments:
            - The node stores its state along with the cost values used for A*.
            - The parent attribute allows for path reconstruction after the goal is reached.
        """
        self.state = state  # The current state - Position And Orientation.
        self.parent = parent  # Parent Node in the search tree (None for the start node)
        self.cost2come = cost2come  # Actual cost from the start node to this node
        self.cost2go = cost2go  # Heuristic cost estimate from this node to the goal

    def total_cost(self):
        """
        Compute the total estimated cost for this node.

        Returns:
            float: The sum of cost2come and cost2go.

        Comments:
            - This total cost is used as the priority in the A* search algorithm.
        """
        return (
            self.cost2come + self.cost2go
        )  # Total cost = actual cost + heuristic cost

    def __lt__(self, other):
        """
        Compare this node with another node based on total cost.

        Parameters:
            other (Node): The node to compare with.

        Returns:
            bool: True if this node's total cost is less than the other node's total cost.

        Comments:
            - This method is used by the priority queue (heapq) to order nodes.
        """
        return self.total_cost() < other.total_cost()

    def __hash__(self):
        """
        Compute the hash value of the node based on its state.

        Returns:
            int: The hash value computed from the node's state.

        Comments:
            - Hashing the node by its state allows using Node instances as keys in dictionaries or in sets.
        """
        return hash(self.state)


class PriorityQueue:
    def __init__(self):
        """
        Initialize the PriorityQueue for A* search.

        Attributes:
            heap (list): A list-based heap (using heapq) that stores nodes as (total_cost, node) tuples.
            open (dict): A dictionary storing nodes in the open set, keyed by a discretized state.
            closed (dict): A dictionary storing nodes that have been fully expanded.

        Comments:
            - The 'heap' is used for fast retrieval of the node with the lowest total cost.
            - The 'open' and 'closed' dictionaries help prevent re-expansion of nodes.
        """
        self.heap = []
        self.open = {}
        self.closed = {}

    def push(self, node, key):
        """
        Push a node onto the priority queue.

        Parameters:
            node (Node): The node to add.
            key (tuple): A discretized representation of the node's state used as the key.

        Comments:
            - The node is added to the heap with its total cost as the priority.
            - It is also stored in the 'open' dictionary so that we can quickly check if a node is already in the open set.
        """

        heapq.heappush(self.heap, (node.total_cost(), node))
        self.open[key] = node

    def pop(self):
        """
        Pop and return the node with the lowest total cost from the priority queue.

        Returns:
            Node: The node with the smallest total cost.

        Comments:
            - The node is removed from the heap.
            - This method does not remove the corresponding entry from the 'open' dictionary.
        """
        return heapq.heappop(self.heap)[1]

    def update(self, node, key):
        """
        Update the priority queue with a node if a lower-cost version is found.

        Parameters:
            node (Node): The new node to add or update.
            key (tuple): The key representing the node's discretized state.

        Comments:
            - If the node already exists in the 'open' set and the new node has a lower total cost,
              update the 'open' entry and push the new node onto the heap.
            - If the node does not exist in the open set, it is pushed onto the heap as a new entry.
            - This approach may leave outdated nodes in the heap, but they will be ignored once popped.
        """
        if key in self.open:
            if node.total_cost() < self.open[key].total_cost():
                self.open[key] = node
                heapq.heappush(self.heap, (node.total_cost(), node))
        else:
            self.push(node, key)


def discretize(state, pos_threshold, theta_threshold):
    """
    Convert a continuous state into a discrete representation.

    Parameters:
        state (tuple): The continuous state, represented as (x, y, theta).
        pos_threshold (float): The positional discretization factor. The x and y coordinates are divided by this value.
        theta_threshold (float): The angular discretization factor. The theta value is divided by this value.

    Returns:
        tuple: A discretized state (int_x, int_y, int_theta) where each component is an integer.

    Comments:
        - This function reduces the continuous state space by dividing each component by a threshold
```

```python
          and converting it to an integer. This helps in identifying and avoiding duplicate or
          nearly identical states during the A* search.
    """
    # Unpack the state into its individual components.
    x, y, theta = state
    # Convert the continuous state (x, y, theta) into a discrete representation.
    # This is done by dividing each component by its corresponding threshold and converting the result to an integer.
    #
    # Note:
    # Although the pos_threshold and theta_threshold are defined as constants in the aStar class,
    # this discretization function is applied dynamically to every state during the search.
    # This dynamic application is crucial because when using small step sizes (after scaling down for planning),
    # the resulting state values can be very small (e.g., around 1 or 2). In those cases, a much smaller
    # positional threshold (around 0.09-0.099) would be required to capture the fine resolution.
    #
    # However, using such a small threshold universally would be too coarse when larger step sizes are used.
    # Thus, by applying a fixed threshold dynamically to all states, we strike a balance that works
    # across the range of step sizes encountered in planning.

    return (
        int(x / pos_threshold),
        int(y / pos_threshold),
        int(theta / theta_threshold),
    )


# When calling discretize, set pos_threshold = scaled_step_size * some_factor

#############################################
# A* Search
#############################################


class aStar:
    def __init__(self, mapEnv, start, goal, step_size, visualizer=None):
        """
        Initialize the A* search algorithm with the given environment and parameters.

        Parameters:
            mapEnv (MapEnv): The map environment object containing obstacles and collision-check methods.
            start (tuple): The starting state as (x, y, theta).
            goal (tuple): The goal state as (x, y, theta).
            step_size (float): The step size used for each motion primitive.
            visualizer (Visualizer, optional): An optional visualizer object for displaying search progress.

        Comments:
            - Sets the thresholds for discretization (position and angle).
            - Precomputes sine and cosine for fixed offsets (0, ±30, ±60 degrees) for efficient neighbor computation.
        """
        self.mapEnv = mapEnv
        self.start = start
        self.goal = goal
        self.step_size = step_size
        self.pos_threshold = 0.5  # Discretization threshold for position
        self.theta_threshold = 30  # Discretization threshold for angle (Degrees)
        self.visualizer = visualizer
        # Precompute and store sine and cosine for the fixed offsets (in degrees)
        # Offsets needed: 0, 30, 60, -30, -60.
        self.offset_trig = {
            0: (math.cos(math.radians(0)), math.sin(math.radians(0))),
            30: (math.cos(math.radians(30)), math.sin(math.radians(30))),
            60: (math.cos(math.radians(60)), math.sin(math.radians(60))),
            -30: (math.cos(math.radians(-30)), math.sin(math.radians(-30))),
            -60: (math.cos(math.radians(-60)), math.sin(math.radians(-60))),
        }

    def _move(self, state, step, offset):
        """
        Compute a new state given a starting state, a step size, and an angular offset.

        Parameters:
            state (tuple): The current state (x, y, theta) where theta is in degrees.
            step (float): The distance to move from the current state.
            offset (int): The angle offset (in degrees) to apply to the current orientation.

        Returns:
            tuple: The new state as (new_x, new_y, new_theta).

        Comments:
            - The new orientation is computed by adding the offset to the current angle.
            - The current angle's cosine and sine are computed.
            - The fixed offset's cosine and sine are retrieved from precomputed values.
            - The angle-addition formulas are applied to compute the new x and y coordinates.
            - This function essentially uses all the precomputed angular values associated with all of the move functions (stored in self.offset_trig) in order
              speedup in the search algorithm, where a step size of 5 would take around 11 - 14 seconds to plan across the map, now it takes around 8 - 10 seco
        """
        x, y, theta = state
        # New orientation is current orientation plus offset.
        new_theta = (theta + offset) % 360

        # Compute sine and cosine of the current orientation.
        theta_rad = math.radians(theta)
        cos_theta = math.cos(theta_rad)
        sin_theta = math.sin(theta_rad)

        # Retrieve precomputed cosine and sine for the fixed offset.
        cos_offset, sin_offset = self.offset_trig[offset]

        # Apply the angle addition formulas:
        # cos(theta + offset) = cos(theta)*cos(offset) - sin(theta)*sin(offset)
        # sin(theta + offset) = sin(theta)*cos(offset) + cos(theta)*sin(offset)
        new_x = x + step * (cos_theta * cos_offset - sin_theta * sin_offset)
        new_y = y + step * (sin_theta * cos_offset + cos_theta * sin_offset)

        return (new_x, new_y, new_theta)
    # Moveset functions using the helper:

    def forward(self, state, step):
        """
        Compute the next state when moving forward with no change in orientation.

        Parameters:
            state (tuple): The current state (x, y, theta).
            step (float): The step size.

        Returns:
            tuple: The new state after moving forward.

        Comments:
            - Calls the generic _move with an offset of 0 degrees.
        """
        return self._move(state, step, 0)

    def forwardLeft30(self, state, step):
        """
        Compute the next state when moving forward while turning left by 30 degrees.

        Parameters:
            state (tuple): The current state (x, y, theta).
            step (float): The step size.

        Returns:
            tuple: The new state after the move.

        Comments:
            - Calls the generic _move with an offset of +30 degrees.
        """
        return self._move(state, step, 30)

    def forwardLeft60(self, state, step):
        """
        Compute the next state when moving forward while turning left by 60 degrees.

        Parameters:
            state (tuple): The current state (x, y, theta).
            step (float): The step size.

        Returns:
            tuple: The new state after the move.

        Comments:
            - Calls the generic _move with an offset of +60 degrees.
        """
        return self._move(state, step, 60)

    def forwardRight30(self, state, step):
        """
        Compute the next state when moving forward while turning right by 30 degrees.

        Parameters:
            state (tuple): The current state (x, y, theta).
            step (float): The step size.

        Returns:
            tuple: The new state after the move.

        Comments:
            - Calls the generic _move with an offset of -30 degrees.
        """
        return self._move(state, step, -30)

    def forwardRight60(self, state, step):
        """
        Compute the next state when moving forward while turning right by 60 degrees.

        Parameters:
            state (tuple): The current state (x, y, theta).
            step (float): The step size.

        Returns:
            tuple: The new state after the move.

        Comments:
            - Calls the generic _move with an offset of -60 degrees.
        """
        return self._move(state, step, -60)

    def heuristic(self, state):
        """
        Compute the heuristic cost (Euclidean distance) from the current state to the goal.

        Parameters:
            state (tuple): The current state (x, y, theta).

        Returns:
            float: The Euclidean distance between the current state and the goal.

        Comments:
            - Only the positional difference (x and y) is considered in this heuristic.
        """
        x, y, _ = state
        gx, gy, _ = self.goal
        return math.hypot(x - gx, y - gy)

    def getNeighbors(self, state):
        """
        Generate the neighbor states for a given state using available motion primitives.

        Parameters:
            state (tuple): The current state (x, y, theta).
        Returns:
            list: A list of neighboring states that are within bounds and not in collision.

        Comments:
            - Applies each of the five moveset functions (forward, left/right turns) to generate neighbors.
            - Checks that the resulting state is within the planning grid and not in collision using mapEnv.halfPlanes().
        """
        actions = [
            self.forward,
            self.forwardLeft30,
            self.forwardLeft60,
            self.forwardRight30,
            self.forwardRight60,
        ]
        neighbors = []
        for action in actions:
            newState = action(state, self.step_size)
            x, y, _ = newState
            if (
                0 <= x < 60
                and 0 <= y < 25
                and not self.mapEnv.halfPlanes(newState[0], newState[1])
            ):
                neighbors.append(newState)
        return neighbors

    def aStarSearch(self):
        """
        Perform the A* search algorithm to find a path from the start state to the goal state.

        Returns:
            tuple: A pair (exploredPoints, path) where:
                - exploredPoints is a list of all states that were expanded.
                - path is the list of states representing the final path from start to goal.
                  If no path is found, path is None.

        Comments:
            - Initializes the search with a priority queue (open set) and a closed set.
            - Uses a progress bar (tqdm) to provide real-time feedback on the number of nodes expanded.
            - For each node, checks if the goal condition is met (both positional and angular thresholds).
            - Updates neighbor nodes in the priority queue using the update method.
            - Reconstructs the path by backtracking from the goal node to the start node.
        """

        pq = PriorityQueue()
        startNode = Node(self.start, None, 0, self.heuristic(self.start))
        startKey = discretize(self.start, self.step_size, self.theta_threshold)
        pq.push(startNode, startKey)
        exploredPoints = []
        iterations = 0
        search_start_time = time.time()

        # Create a tqdm progress bar; total unknown so it updates dynamically.
        pbar = tqdm(desc="Nodes expanded", unit="node", dynamic_ncols=True)

        while pq.heap:
            currentNode = pq.pop()
            currentKey = discretize(
                currentNode.state, self.step_size, self.theta_threshold
            )
            if currentKey in pq.closed:
                continue  # Skip already expanded nodes
            pq.closed[currentKey] = currentNode
            exploredPoints.append(currentNode.state)
            iterations += 1

            # Update the progress bar with the current node's total cost
            pbar.update(1)
            pbar.set_postfix({"f(n)": f"{currentNode.total_cost():.2f}"})

            # Prints status every 1000 iterations for debugging and performance checking
            if iterations % 1000 == 0:
                print(
                    f"Nodes expanded: {iterations} | Cost2Go: {currentNode.cost2go:.2f}"
                )

            # Check if the current node satisfies the goal conditions (position and orientation).
            # The heuristic has to be a bit coarse to accomodate for a step size of 10.
            # TODO : This can probably be tightened using a helper function or a loop to check for a larger step size and ajusting it dynamically.
            if (
                self.heuristic(currentNode.state) <= (self.step_size + 0.065)
                and abs((currentNode.state[2] - self.goal[2]) % 360) <= 10
            ):

                pbar.close()
                total_time = time.time() - search_start_time
                total_cost = currentNode.total_cost()  # Total cost from start to goal.
                path = self.reconstructPath(currentNode)
                path.append(self.goal)
                print(
                    f"Total time: {total_time:.2f} seconds, Total cost: {total_cost:.2f}"
                )
                return exploredPoints, self.reconstructPath(currentNode)

            # Expand neighbors and update the priority queue.
            for neighbor in self.getNeighbors(currentNode.state):
                neighborKey = discretize(neighbor, self.step_size, self.theta_threshold)
                if neighborKey in pq.closed:
                    continue
                cost2come = currentNode.cost2come + self.step_size  # Incremental Cost
                neighborNode = Node(
                    neighbor, currentNode, cost2come, self.heuristic(neighbor)
                )
                pq.update(neighborNode, neighborKey)

        pbar.close()
        total_time = time.time() - search_start_time
        print(f"Total time: {total_time:.2f} seconds")
        return exploredPoints, None

        # pbar.close()
        # total_time = time.time() - search_start_time
        # print(f"Total time: {total_time:.2f} seconds")
        # return exploredPoints, None

    def reconstructPath(self, node):
        """
        Reconstruct the path from the start state to the given node by following parent links.

        Parameters:
            node (Node): The goal node from which to start backtracking.

        Returns:
            list: The path as a list of states, starting from the start state and ending at the goal state.

        Comments:
            - Backtracks from the given node to the start node using the parent attribute.
            - The resulting path is reversed to present it from start to goal.
        """
        path = []
        while node is not None:
            path.append(node.state)
            node = node.parent
        path.reverse()
        return path


############################################
# Visualizer
############################################
class Visualizer:
    def __init__(self, canvas, start, goal, scale_factor=1.0):
        """
        Initialize the Visualizer.

        Parameters:
            canvas (numpy.ndarray): The initial canvas image.
            start (tuple): The start state as (x, y, theta).
            goal (tuple): The goal state as (x, y, theta).
            scale_factor (float): The factor used to scale planning coordinates to canvas coordinates.

        Comments:
            - A copy of the canvas is stored to avoid modifying the original.
            - The start and goal states are stored for later visualization.
        """
        self.canvas = canvas.copy()  # Store a copy of the initial canvas
        self.start = start  # Starting state
        self.goal = goal  # Goal state
        self.scale_factor = scale_factor  # Scaling factor to convert planning coordinates back to canvas coordinates

    def drawPoints(self):
        """
        Draw the start and goal points on the canvas.

        Comments:
            - The start point is drawn in blue and the goal point in green.
            - The y-coordinate is adjusted using final_canvas_height (assumed to be defined globally)
              to account for the coordinate system (flipping y-axis if needed).
        """
        # Convert start and goal coordinates to canvas coordinates.
        startpt = (int(self.start[0]), int(self.start[1]))
        endpt = (int(self.goal[0]), int(self.goal[1]))
        # Draw the start point (blue circle).
        cv.circle(self.canvas, startpt, 2, COLOR_START, -1)
        # Draw the goal point (green circle).
        cv.circle(self.canvas, endpt, 2, COLOR_GOAL, -1)

    def updateExploration(self, state):
        """
        Update the canvas with a new explored state.

        Parameters:
            state (tuple): The state (x, y, theta) to be drawn.

        Comments:
            - A small red dot is drawn at the given state location.
            - The canvas is then displayed using OpenCV.
        """
        x, y, _ = state
        # Draw a small red circle at the explored state's location.
        cv.circle(self.canvas, (int(x), int(y)), 1, COLOR_EXPLORE, -1)
        cv.imshow("Exploration", self.canvas)
        cv.waitKey(1)

    def animateExplore(self, exploredPoints, delay=0, robot_radius=5):
        """
        Animate the exploration process by drawing all explored points and the robot's footprint.

        Parameters:
            exploredPoints (list): A list of states (x, y, theta) that were expanded.
            delay (int): Delay (in milliseconds) between updates for display.
            robot_radius (float): The radius of the robot (used to draw the footprint).

        Comments:
            - The canvas is updated periodically (every 100 points or at the end) to improve performance.
            - Each explored state is drawn with its center (blue dot) and the corresponding robot footprint (red circle).
            - The footprint is scaled using the scale_factor to match the canvas dimensions.
        """
        animateCanvas = self.canvas.copy()  # Work on a copy of canvas for animation
        # Use the canvas height from the image shape instead of a global variable.
        canvas_height = animateCanvas.shape[0]
        for i, state in enumerate(exploredPoints):
            x, y, _ = state
            # Draw a blue dot for the explored node (center)
            cv.circle(
                animateCanvas,
                (int(x), int(y)),
                1,
                COLOR_EXPLORE,
                -1,
            )
            # Draw the robot's footprint around this node.
            # Scale the robot radius to the canvas using self.scale_factor.
            scaled_robot_radius = int(robot_radius * self.scale_factor)
            cv.circle(
                animateCanvas,
                (int(x), int(y)),
                scaled_robot_radius,
                COLOR_ROBOT_RADIUS,
                1,
            )
            # Update the display every 100 nodes or at the end of the animation.
            if i % 100 == 0 or i == len(exploredPoints) - 1:
                flipped_canvas = cv.flip(animateCanvas, 0)
                cv.imshow("Exploration", flipped_canvas)
                cv.waitKey(1)
        self.canvas = animateCanvas  # Save the updated canvas

    def animatePath(self, path, delay=10):
        """
        Animate the final path by drawing lines between consecutive states.

        Parameters:
            path (list): The list of states (x, y, theta) that form the final path.
            delay (int): Delay (in milliseconds) between drawing each line segment.

        Comments:
            - Instead of drawing single points, consecutive states are connected with lines.
            - This provides a clearer, continuous visual representation of the path.
            - The line thickness and color can be adjusted as needed.
        """
        animateCanvas = self.canvas.copy()  # Copy the current canvas for path animation
        canvas_height = animateCanvas.shape[0]
        # Loop through consecutive pairs of points in the path
        for i in range(len(path) - 1):
            # Convert both points to canvas coordinates.
            pt1 = (int(path[i][0]), int(path[i][1]))
            pt2 = (int(path[i + 1][0]), int(path[i + 1][1]))
            # Draw a line segment connecting pt1 and pt2.
            cv.line(animateCanvas, pt1, pt2, COLOR_PATH, thickness=2)
            flipped_canvas = cv.flip(animateCanvas, 0)
            cv.imshow("Path", flipped_canvas)
            cv.waitKey(delay)
        self.canvas = animateCanvas


############################################
# Inputs
############################################
class Inputs:
    def __init__(self, final_canvas_height, final_canvas_width):
        """
        Initialize the Inputs object for gathering user-provided parameters.

        Parameters:
            final_canvas_height (int): Height of the final canvas (used for input validation).
            final_canvas_width (int): Width of the final canvas (used for input validation).

        Comments:
            - This object will store the start and goal states, step size, and optionally the robot radius.
            - Initially, these values are set to None.
        """
        self.start = None  # Start state (x, y, theta)
        self.goal = None  # Goal State (x, y, theta)
        self.step_size = None  # User defined step size
        self.final_canvas_height = (
            final_canvas_height  # Canvas height for validation of y coordinates
        )
        self.final_canvas_width = (
            final_canvas_width  # Canvas width for validation of x coordinates
        )
        self.robot_radius = None  # Robot radius if needed. Hardcoded as 5 here. Change call in main to use input_handler for using user defined robot radius

    def get_start_goal(self, final_canvas_width, final_canvas_height, env):
        """
        Prompt the user to enter start and goal coordinates until valid inputs are provided.

        Parameters:
            final_canvas_width (int): Maximum x-coordinate allowed.
            final_canvas_height (int): Maximum y-coordinate allowed.
            env (MapEnv): The map environment object used for collision checking.

        Returns:
            tuple: A tuple containing:
                - start (tuple): The start state as (x, y, theta) with theta normalized to [0, 360).
                - goal (tuple): The goal state as (x, y, theta) with theta normalized to [0, 360).

        Comments:
            - In addition to verifying the coordinates are within bounds, this method checks that
            the start and goal are not inside any obstacles using the env.halfPlanes() method.
        """
        valid = False
        while not valid:
            try:
                start_input = input(
                    f"Enter start coordinates (x,y,theta) with x in [0, {final_canvas_width}), y in [0, {final_canvas_height}): "
                )
                goal_input = input(
                    f"Enter goal coordinates (x,y,theta) with x in [0, {final_canvas_width}), y in [0, {final_canvas_height}): "
                )
                sx, sy, stheta = map(float, start_input.split(","))
                gx, gy, gtheta = map(float, goal_input.split(","))
                # Check if coordinates are within bounds.
                if (
                    0 <= sx < final_canvas_width
                    and 0 <= sy < final_canvas_height
                    and 0 <= gx < final_canvas_width
                    and 0 <= gy < final_canvas_height
                ):
                    # Check if start or goal is inside an obstacle.
                    if env.halfPlanes(sx * 0.1, sy * 0.1):
                        print(
                            "Start coordinates are inside an obstacle. Please try again."
                        )
                    elif env.halfPlanes(gx * 0.1, gy * 0.1):
                        print(
                            "Goal coordinates are inside an obstacle. Please try again."
                        )
                    else:
                        self.start = (sx, sy, stheta % 360)
                        self.goal = (gx, gy, gtheta % 360)
                        valid = True
                else:
                    print("Coordinates out of bounds. Please try again.")
            except Exception as e:
                print("Invalid format. Please use the format: x,y,theta")
        return self.start, self.goal

    def get_parameters(self):
        """
        Prompt the user to input the step size until a valid value is provided.

        Returns:
            float: The valid step size entered by the user.

        Comments:
            - The step size must be a numeric value between 1 and 10.
            - If the input is not a valid number or falls outside the allowed range,
              the user is prompted again.
            - (Optionally) The robot radius input can also be included by uncommenting the relevant lines.
        """
        valid = False  # Check if valid step size is obtained
        while not valid:
            try:
                # Prompt user for step size with proper bounds.
                step = float(input("Enter step size (1 <= step <= 10): "))
                # Optional: Uncomment for using user defined robot radius
                # radius = float(input("Enter robot radius (positive number): "))
                if 1 <= step <= 10:
                    self.step_size = step
                    # Uncomment below to use user defined robot radius
                    # self.robot_radius = radius
                    valid = True
                else:
                    print(
                        "Invalid values. Ensure step size is between 1 and 10 and radius is positive."
                    )
            except Exception as e:
                print("Invalid input. Please enter numeric values.")
        return self.step_size


#############################################
# MAIN FUNCTION
#############################################
if __name__ == "__main__":
    """
    Main entry point for the path planning program.

    Process:
      1. Define raw map dimensions and final canvas size.
      2. Initialize an Inputs object to collect user parameters.
      3. Set robot parameters (radius and clearance).
      4. Create the MapEnv environment (setup obstacles and inflate them).
      5. Create a full-scale canvas, draw clearances and obstacles, then flip the canvas if needed.
      6. Display the initial map with clearances using matplotlib (with axes), then wait for a button press.
      7. Obtain the step size and start/goal states from the user.
      8. Scale the start, goal, and step size for planning on a reduced grid.
      9. Initialize the Visualizer with the full-scale canvas, start, goal, and scaling factor, and draw start/goal points.
     10. Display the start/goal confirmation using matplotlib (with axes), then wait for a button press.
     11. Run the A* search to get explored points and the planned path.
     12. Animate the exploration process and final path using OpenCV.
     13. Display the final canvas with the final path using matplotlib (with axes), then wait for a button press.
    """

    # 1. Define raw map dimensions and final canvas size.
    map_width = 180
    map_height = 50
    final_canvas_width = 600
    final_canvas_height = 250

    # 2. Initialize the input handler.
    input_handler = Inputs(final_canvas_height, final_canvas_width)

    # 3. Set robot parameters.
    robot_radius = 5
    clearance = 5

    # 4. Create the map environment.
    env = MapEnv(
        map_width,
        map_height,
        final_canvas_width,
        final_canvas_height,
        robot_radius,
        clearance,
    )

    # # 5. Create a blank white canvas and draw obstacles/clearances.
    full_canvas = env.createCanvas()
    full_canvas = env.drawClearances(full_canvas)
    full_canvas = env.drawObs(full_canvas)
    # cv.imshow("test canvas", full_canvas)
    # # full_canvas = cv.flip(full_canvas, 0)  # Flip if needed.

    # 6. Display the initial map with clearances using matplotlib.
    # Convert from BGR (OpenCV) to RGB (matplotlib).
    full_canvas_rgb = cv.cvtColor(full_canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 6))
    plt.imshow(full_canvas_rgb, origin="lower")
    plt.title("Initial Map with Clearances")
    plt.xlabel("X coordinate")
    plt.ylabel("Y coordinate")
    plt.waitforbuttonpress()  # Wait for a key/mouse press.
    plt.close()

    # 7. Get user inputs: step size, start and goal coordinates.
    step_size = input_handler.get_parameters()
    start, goal = input_handler.get_start_goal(
        final_canvas_width, final_canvas_height, env
    )

    # 8. Define scale factor and scale the states for planning.
    calc_scale = 0.1
    scaled_start = (start[0] * calc_scale, start[1] * calc_scale, start[2])
    scaled_goal = (goal[0] * calc_scale, goal[1] * calc_scale, goal[2])
    scaled_step_size = step_size * calc_scale

    # 9. Initialize the Visualizer and draw start/goal points.
    vis = Visualizer(full_canvas, start, goal, scale_factor=calc_scale)
    vis.drawPoints()

    # 10. Display start/goal confirmation using matplotlib.
    vis_canvas_rgb = cv.cvtColor(vis.canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 6))
    plt.imshow(vis_canvas_rgb, origin="lower")
    plt.title("Start/Goal for Confirmation")
    plt.xlabel("X coordinate")
    plt.ylabel("Y coordinate")
    plt.waitforbuttonpress()  # Wait for confirmation.
    plt.close()

    # 11. Create the A* search planner and run the search.
    astar_planner = aStar(
        env, scaled_start, scaled_goal, scaled_step_size, visualizer=vis
    )
    explored_points, path_scaled = astar_planner.aStarSearch()

    if path_scaled is None:
        print("No path found.")
    else:
        print("Path found:", path_scaled)
        # Scale the explored points and path back to full-canvas coordinates.
        explored_points_full = [
            (x / calc_scale, y / calc_scale, theta) for (x, y, theta) in explored_points
        ]
        path_full = [
            (state[0] / calc_scale, state[1] / calc_scale, state[2])
            for state in path_scaled
        ]
        # 12. Animate exploration and final path using OpenCV.
        vis.animateExplore(explored_points_full, delay=0)
        vis.animatePath(path_full, delay=10)

    # 13. Display the final canvas with the final path using matplotlib.
    final_canvas_rgb = cv.cvtColor(vis.canvas, cv.COLOR_BGR2RGB)
    plt.figure(figsize=(8, 6))
    plt.imshow(final_canvas_rgb, origin="lower")
    plt.title("Final Path")
    plt.xlabel("X coordinate")
    plt.ylabel("Y coordinate")
    plt.waitforbuttonpress()  # Wait for final confirmation.
    plt.close()

    cv.destroyAllWindows()
```