```python
import numpy as np
import cv2
import heapq
import copy
import time
import functools
# Timer decorator to measure execution time of functions
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()  # Start time
        result = func(*args, **kwargs)  # Execute the wrapped function
        end_time = time.perf_counter()  # End time
        run_time = end_time - start_time  # Calculate runtime
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return result  # Return the result of the wrapped function
    return wrapper

class Node:
    def __init__(self, Node_Cost, Node_Cost_est, Node_x, Node_y, Node_theta, Parent_Node_x, Parent_Node_y, Parent_Node_theta):
        self.Node_Cost = Node_Cost # The cost to reach this node.
        self.Node_Cost_est = Node_Cost_est # The estimated cost to the goal.
        self.Node_x = int(Node_x) # The node's x location.
        self.Node_y = int(Node_y) # The node's y location.
        self.Node_theta = int(Node_theta) # The node's angle.
        self.Parent_Node_x = int(Parent_Node_x) # The node's parent's x location.
        self.Parent_Node_y = int(Parent_Node_y) # The node's parent's y location.
        self.Parent_Node_theta = Parent_Node_theta # The node's parent's angle.

    # This method allows the heapq module to compare Node objects by their cost when sorting.
    # This ensures that the node with the smallest cost plus heuristic is popped first.
    # The heuristic used is Euclidean distance.
    def __lt__(self, other):
        return self.Node_Cost + self.Node_Cost_est < other.Node_Cost + other.Node_Cost_est

# Each of the five move functions takes in a node, copies its information
# to generate the basis of the new node as a result of movement,
# updates the cost of the new node to execute that movement from the
# parent node, and updates the position of the new node.

def move_major_left(given_Node, step_size, scale, goal_x, goal_y):
    return create_new_node(given_Node, step_size, 60, scale, goal_x, goal_y)

def move_minor_left(given_Node, step_size, scale, goal_x, goal_y):
    return create_new_node(given_Node, step_size, 30, scale, goal_x, goal_y)

def move_straight(given_Node, step_size, scale, goal_x, goal_y):
```

```
    return create_new_node(given_Node, step_size, 0, scale, goal_x, goal_y)

def move_minor_right(given_Node, step_size, scale, goal_x, goal_y):
    return create_new_node(given_Node, step_size, -30, scale, goal_x, goal_y)

def move_major_right(given_Node, step_size, scale, goal_x, goal_y):
    return create_new_node(given_Node, step_size, -60, scale, goal_x, goal_y)

# create_new_node is the main body of each of the move functions.
def create_new_node(given_Node, step_size, theta, scale, goal_x, goal_y):
    # Get the parent
    parent_x = given_Node.Node_x
    parent_y = given_Node.Node_y
    parent_theta = given_Node.Node_theta

    # Increment the cost from the movement.
    cost = given_Node.Node_Cost + step_size*scale

    # Calculate the new position from the movement.
    x_pos = round((given_Node.Node_x + step_size*np.cos(np.deg2rad(given_Node.Node_theta + theta))*scale)*2)/2
    y_pos = round((given_Node.Node_y + step_size*np.sin(np.deg2rad(given_Node.Node_theta + theta))*scale)*2)/2
    theta_pos = int(given_Node.Node_theta + theta)

    # Get the estimated cost to the goal.
    cost_est = np.sqrt((given_Node.Node_x - goal_x)**2 + (given_Node.Node_y - goal_y)**2)

    # Generate the new node.
    newNode = Node(cost,cost_est,x_pos,y_pos,theta_pos,parent_x,parent_y,parent_theta)

    return newNode

# Convert the angle given, in degrees, to an index from 0-11.
def angle_to_index(angle):
    # Normalize angle to the range [0, 360)
    angle = angle % 360

    # Compute the index by dividing the angle by 30
    return int(angle // 30)

# Check if the current node overlaps any of the closed nodes. This checks if the
# forward and backward searches have overlapped.
def overlap_check(closed_set, current_x, current_y, current_theta, sf):
    current_theta_index = angle_to_index(current_theta)

    # Wrap-around function for theta index
    def wrap_index(idx):
        return idx % 12
```

```python
    # Adjust index with wrap-around. Increment by six (180 degrees) because
    # the forward and backward search angles will be opposite to each other.
    current_theta_index = wrap_index(current_theta_index + 6)

    # Generate indices within +/-2 range with wrap-around. This allows for angles that
    # match within the robot's range of motion, +/- 60 degrees.
    theta_indices = [wrap_index(current_theta_index + i) for i in range(-2, 3)]

    # Check for any matches. Return True and the index found if match is found.
    for theta_idx in theta_indices:
        node_key = (int(current_y), int(current_x), theta_idx)
        if closed_set[node_key]:
            return True, theta_idx

    return False, None

# gen_obstacle_map generates the map and its obstacles using half planes
# and semi-algebraic models. Each obstacle is composed of a union of convex
# polygons that define it. It then constructs an in image in BGR and sets
# obstacle pixels as red in the image. Additionally, the entire obstacle map
# can be configured for a certain resolution by the given scale factor, sf.
# When sf = 1, each pixel represents 1 mm. sf = 10, each pixel represents .1 mm.
def gen_obstacle_map(sf=1):
    # Set the height and width of the image in pixels.
    height = 250*sf
    width = 600*sf
    # Create blank canvas.
    obstacle_map = np.zeros((height,width,3), dtype=np.uint8 )

    # Arbitrary increase in size of obstacles to fit new expanded map size. Map size was height = 50 and width = 180
    # in prior project. This makes the map more filled with obstacles by expanding their size.
    sf=sf*3.5

    # Define polygons for E obstacle.
    def E_obstacle1(x,y):
        return (10*sf <= x <= 15*sf) and (10*sf <= y <= 35*sf)

    def E_obstacle2(x,y):
        return (15*sf <= x <= 23*sf) and (10*sf <= y <= 15*sf)

    def E_obstacle3(x,y):
        return (15*sf <= x <= 23*sf) and (20*sf <= y <= 25*sf)

    def E_obstacle4(x,y):
        return (15*sf <= x <= 23*sf) and (30*sf <= y <= 35*sf)
```

```
    # Define polygons for N obstacle.
    def N_obstacle1(x,y):
        return (30*sf <= x <= 35*sf) and (10*sf <= y <= 35*sf)

    def N_obstacle2(x,y):
        return (40*sf <= x <= 45*sf) and (10*sf <= y <= 35*sf)

    def N_obstacle3(x,y):
        return (35*sf <= x <= 40*sf) and (-3*x+130*sf <= y <= -3*x+140*sf)

    # Define polygons for P obstacle.
    def P_obstacle1(x,y):
        return (53*sf <= x <= 58*sf) and (10*sf <= y <= 35*sf)

    def P_obstacle2(x,y):
        return (58*sf <= x <= 64*sf) and ((x-58*sf)**2 + (y-29*sf)**2 <= (6*sf)**2)

    # Define polygons for M obstacle.
    def M_obstacle1(x,y):
        return (70*sf <= x <= 75*sf) and (10*sf <= y <= 35*sf)

    def M_obstacle2(x,y):
        return (88*sf <= x <= 93*sf) and (10*sf <= y <= 35*sf)

    def M_obstacle3(x,y):
        return (79*sf <= x <= 84*sf) and (10*sf <= y <= 15*sf)

    def M_obstacle4(x,y):
        return (75*sf <= x <= 79*sf) and (-5*x+400*sf <= y <= -5*x+410*sf) and (10*sf <= y)

    def M_obstacle5(x,y):
        return (84*sf <= x <= 88*sf) and (5*x-415*sf <= y <= 5*x-405*sf) and (10*sf <= y )

    # Define polygons for first Six obstacle.
    def Six1_obstacle1(x,y):
        return ((x-109*sf)**2 + (y-19*sf)**2 <= (9*sf)**2)

    def Six1_obstacle2(x,y):
        return ((x-121.5*sf)**2 + (y-19*sf)**2 <= (21.50*sf)**2) and ((x-121.5*sf)**2 + (y-19*sf)**2 >= (16.50*sf)**2) and (19*sf <=
y <= -1.732*x+229.438*sf)

    def Six1_obstacle3(x,y):
        return ((x-112*sf)**2 + (y-35.454*sf)**2 <= (2.5*sf)**2)

    # Define polygons for second Six obstacle.
    def Six2_obstacle1(x,y):
        return ((x-132*sf)**2 + (y-19*sf)**2 <= (9*sf)**2)
```

```
    def Six2_obstacle2(x,y):
        return ((x-144.5*sf)**2 + (y-19*sf)**2 <= (21.50*sf)**2) and ((x-144.5*sf)**2 + (y-19*sf)**2 >= (16.50*sf)**2) and (19*sf <=
y <= -1.732*x+269.274*sf)

    def Six2_obstacle3(x,y):
        return ((x-135*sf)**2 + (y-35.454*sf)**2 <= (2.5*sf)**2)

    # Define polygon for One obstacle.
    def One_obstacle1(x,y):
        return (148*sf <= x <= 153*sf) and (10*sf <= y <= 38*sf)

    # For every pixel in the image, check if it is within the bounds of any obstacle.
    # If it is, set it's color to red.
    for y in range(height):
        for x in range(width):
            if (E_obstacle1(x, y) or E_obstacle2(x,y) or E_obstacle3(x,y) or E_obstacle4(x,y)
                or N_obstacle1(x,y) or N_obstacle2(x,y) or N_obstacle3(x,y)
                or P_obstacle1(x,y) or P_obstacle2(x,y)
                or M_obstacle1(x,y) or M_obstacle2(x,y) or M_obstacle3(x,y) or M_obstacle4(x,y) or M_obstacle5(x,y)
                or Six1_obstacle1(x,y) or Six1_obstacle2(x,y) or Six1_obstacle3(x,y)
                or Six2_obstacle1(x,y) or Six2_obstacle2(x,y) or Six2_obstacle3(x,y)
                or One_obstacle1(x,y)):
                obstacle_map[y, x] = (0, 0, 255)


    # The math used assumed the origin was in the bottom left.
    # The image must be vertically flipped to satisy cv2 convention.
    return np.flipud(obstacle_map)

# expand_obstacles takes the obstacle map given by gen_obstacle_map as an image, along with
# the scale factor sf, and generates two images. The first output_image, is a BGR image
# to draw on used for visual display only. expanded_mask is a grayscale image with white
# pixels as either obstacles or clearance space around obstacles. This function will take
# the given obstacle image and apply a specified radius circular kernel to the image. This ensures
# an accurate clearance around every obstacle.
def expand_obstacles(image, scale_factor, radius):

    radius = scale_factor*radius

    # Convert image to HSV
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define color mask for red and create grayscale image.
    lower_red = np.array([0, 200, 200])
    upper_red = np.array([25, 255, 255])
    obstacle_mask = cv2.inRange(hsv, lower_red, upper_red)
```

```python
    # Create circular structuring element for expansion
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2 * radius + 1, 2 * radius + 1))
    # Apply kernel to get 2 mm dilation around all elements.
    expanded_mask = cv2.dilate(obstacle_mask, kernel, iterations=1)

    # Apply 2 mm dilation to all of the borders.
    h, w = expanded_mask.shape
    expanded_mask[:radius+1, :] = 255  # Top border
    expanded_mask[h-radius:, :] = 255  # Bottom border
    expanded_mask[:, :radius+1] = 255  # Left border
    expanded_mask[:, w-radius:] = 255  # Right border

    # Create the output image and apply color orange to all obstacle and clearance
    # pixels.
    output_image = image.copy()
    output_image[np.where(expanded_mask == 255)] = [0, 165, 255]  # Color orange

    # Restore original red pixels. This creates an image with red obstacles,
    # and orange clearance zones.
    output_image[np.where(obstacle_mask == 255)] = [0, 0, 255]

    return output_image, expanded_mask

# prompt the user for a point. prompt is text that specifies what
# type of point is be given. prompt is solely used for terminal text output.
# sf is the scale factor to ensure the user's input is scaled correctly for the map.
# image is passed to ensure the point is within the image bounds. obstacles is passed
# to ensure the user's point does not lie in an obstacle. The function returns the user's
# points as integers. It also prompts the user for an angle and ensures that the angle
# is a multiple of 30.
def get_point(prompt, sf, obstacles):
    valid_input = False

    while not valid_input:
        # Get x and y input and adjust by scale factor sf.
        try:
            x = int(input(f"Enter the x-coordinate for {prompt} (int): "))
            y = int(input(f"Enter the y-coordinate for {prompt} (int): "))
        except ValueError:
            print("Invalid input. Please enter a numerical value.")
            continue

        # Ensure theta meets constraints.
        while True:
            try:
                theta = int(input(f"Enter the theta-coordinate for {prompt} (must be 0-360 and a multiple of 30): "))
```

```
                if 0 <= theta <= 360 and theta % 30 == 0:
                    break
                else:
                    print("Invalid theta. It must be between 0 and 360 and a multiple of 30. Please try again.")
            except ValueError:
                print("Invalid input. Please enter an integer value for theta.")

        # Correct the y value to account for OpenCV having origin in top left.
        obstacle_y = obstacles.shape[0] - y*2

        # Validate position against obstacles
        if valid_move(x*2, obstacle_y, obstacles.shape, obstacles):
            valid_input = True
        else:
            print("Invalid Input. Within Obstacle. Please try again.")

    return int(x*sf), int(y*sf), int(theta)

# valid_move checks if a given point lies within the map bounds and
# if it is located within an obstacle. If the point is in the image and NOT in an obstacle,
# it returns True, meaning the position is valid/Free/open space.
def valid_move(x, y, map_shape, obstacles):
    return 0 <= x < map_shape[1] and 0 <= y < map_shape[0] and obstacles[int(y), int(x)] == 0

# Valid line uses the brensenham line to analyze all points between two points.
# It then checks both the end points and everything in between with valid_move
# to ensure the entire movement does not lie in obstacle space.
# Returns true if the move is valid.
def valid_line(x1, y1, x2, y2, map_shape, obstacles):
    x1 = int(x1)
    x2 = int(x2)
    y1 = int(y1)
    y2 = int(y2)

    # Take in two points and generate all the points in between.
    def bresenham(x1, y1, x2, y2):
        points = []
        dx = abs(x2 - x1)
        dy = abs(y2 - y1)
        sx = 1 if x1 < x2 else -1
        sy = 1 if y1 < y2 else -1
        err = dx - dy

        while True:
            points.append((x1, y1))
            if x1 == x2 and y1 == y2:
                break
```

```
            e2 = 2 * err
            if e2 > -dy:
                err -= dy
                x1 += sx
            if e2 < dx:
                err += dx
                y1 += sy

        return points

    points = bresenham(x1, y1, x2, y2)
    for x, y in points:
        if not valid_move(x, y, map_shape, obstacles):
            return False
    return True

# Check if we are at the goal position.
def goal_check(x, y, theta, end, scale):
    end_x, end_y, end_theta = end
    dis = np.sqrt(((end_x - x))**2 + (end_y - y)**2)
    dif_theta = np.abs(end_theta - theta)
    # Check if position and angle is within thresholds.
    if dis < 1.5*scale:# and dif_theta < 15:
        return True
    else:
        return False

# A_star_search uses the A star method to explore a map and find the path from start to end.
# map is the image to draw on. Obstacles is a grayscale image of .5 mm resolution to path plan on.
# sf is the scale factor which increases the map size for visual aesthetic. step_size is how far the
# robot moves with each step. The planner will perform a bidirectional forward and backward search in tandem.
# This is done to ensure that the initial and goal orientations are met. It also ensures that the planner finds
# an exact solution. The planner will continue searching until it finds an overlap between the forward and
# backward search and ensures that overlap is an optimal solution. However, since solutions are exact, larger
# step sizes can result in paths that require jagged edges to meet the exact solutions.
@timer
def A_star_search(map, obstacles, start, end, sf, step_size):

    # Convert y coordinates from origin bottom left (user input) to origin top left (cv2 convention).
    height, width, _ = map.shape
    start_x, start_y, start_theta = start
    end_x, end_y, end_theta = end
    start_y = height - start_y
    end_y = height - end_y
    start = (start_x, start_y, start_theta)
    end = (end_x, end_y, end_theta)
```

```python
# # Open video file to write path planning images to.
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
video_filename = "A_star_Proj3_phase1_video.mp4"
fps = 60
video_out = cv2.VideoWriter(video_filename, fourcc, fps, (width, height))

# Create the start and end node.
start_node = Node(0, 0, start[0], start[1], start[2], start[0], start[1], start[2])
end_node = Node(0, 0, end[0], end[1], end[2], end[0], end[1], end[2])

open_set_forward = []  # Priority queue. Used to extract nodes with smallest cost from forward search.
heapq.heappush(open_set_forward, start_node)
open_set_backward = []  # Priority queue. Used to extract nodes with smallest cost from backward search.
heapq.heappush(open_set_backward, end_node)

# The path planning occurs on a positional resolution of .5 mms. This sets the dimensions for our
# sets to check if points are duplicates.
height = int(height/sf*2)
width = int(width/sf*2)


# The seen set is how I track if a node has already been given a cost. It is a boolean matrix that checks if
# every possible position/orientation combination has been seen. True = seen. seen nodes are in the open or closed set.
seen_forward = np.full((height, width, 12), False, dtype=bool)
start_angle_index = int(round(start_theta/30))
seen_forward[start_y, start_x, start_angle_index] = True
seen_backward = np.full((height, width, 12), False, dtype=bool)
end_angle_index = int(round(end_theta/30))
seen_backward[end_y, end_x, end_angle_index] = True

# This is my seen set, but as a dictionary to store all node information.
visited_forward = {}
visited_forward[(start_y, start_x, start_angle_index)] = start_node
visited_backward = {}
visited_backward[(end_y, end_x, end_angle_index)] = end_node

# This is the closed set. It stores nodes that have been fully explored.
closed_set_forward = np.full((height, width, 12), False, dtype=bool)
closed_set_backward = np.full((height, width, 12), False, dtype=bool)

# Create a list of functions of all the types of moves we can execute.
directions = [move_major_left, move_minor_left, move_straight, move_minor_right, move_major_right]

# Draw the start and end points as a magenta and cyan circle, respectively.
cv2.circle(map, (start[0],start[1]), sf, (255, 0, 255), -1)
cv2.circle(map, (end[0], end[1]), sf, (255, 255, 0), -1)
```

```python
    # Used to store only every 100th frame to the video. Otherwise the video is hours long.
    # Additionally, writing frames takes the most computation for every loop.
    video_frame_counter = 0

    # Used to switch between forward and backward search.
    forward_turn = True

    # Keeps track of how many times we have switched betwwen forward and backward search.
    times_switched = 0
    # We artificially limit the amount of switches by the turn_limit. The backward search will
    # occur as many times as the turn limit. This allows the backward search to create a mesh
    # for the forward search to latch onto and get the correct final orientation without
    # excessively double searching the free space.
    turn_limit = 100

    # This ensures that the first iteration expands only in one direction so the robot
    # appraoches the goal in the final orientation.
    first_backward_iter = True

    # Keep track of the cost and which nodes consist of the best overlap
    # found between the forward and backward search.
    best_cost = -1
    best_cost_node_forward = None
    best_cost_node_backward = None

    # Continue to search while the open_set is not empty.
    while open_set_forward and open_set_backward:
        # Continue if it is forward search.
        if forward_turn:
            # Get the node with the smallest cost from the open_set.
            current_node = heapq.heappop(open_set_forward)
            # Extract it's x and y position.
            current_x, current_y, current_theta = current_node.Node_x, current_node.Node_y, current_node.Node_theta

            # Verify that this position is not in the closed set.
            # Skip this iteration if it is in the closed_set as the position
            # has already been fully explored. This is required because
            # there is no efficient implementation to updating nodes within a heapq.
            # As such, a node may be added to the heapq, then added again to the heapq if
            # a better parent was found.
            if closed_set_forward[int(current_y/sf/.5), int(current_x/sf/.5), angle_to_index(current_theta)] == True:
                continue

            # Add the current node to the closed set.
            closed_set_forward[int(current_y/sf/.5), int(current_x/sf/.5), angle_to_index(current_theta)] = True

            # If the node currently popped has a greater cost + cost_est than the cost of the best path found,
```

```python
        # then we can say definitively that we have found the optimal path. Alert user and get that path.
        if current_node.Node_Cost + current_node.Node_Cost_est > best_cost and best_cost != -1:
            print("Goal Reached!")
            print("Overlap Path Found!")
            # Generate final path.
            path = get_bi_directional_path(visited_forward, visited_backward, best_cost_node_forward, best_cost_node_backward, sf)

            print(path)
            print(f"end_x: {end_x}")
            print(f"end_y: {end_y}")
            print(f"best_forward_x: {best_cost_node_forward.Node_x}")
            print(f"best_forward_y: {best_cost_node_forward.Node_y}")
            print(f"best_backward_x: {best_cost_node_backward.Node_x}")
            print(f"best_backward_y: {best_cost_node_backward.Node_y}")

            # For each pixel in the path, draw a magenta line
            for i in range(len(path) - 1):
                x1, y1 = path[i]
                x2, y2 = path[i + 1]
                cv2.line(map, (int(x1), int(y1)), (int(x2), int(y2)), (255, 0, 255), 2)
                video_out.write(map)

            # Release the video file.
            video_out.release()
            # Terminate search and return the final map with the path and area explored.
            return map

        # Increment the video_frame_counter and save a frame if it is the 100th frame.
        video_frame_counter += 1
        if video_frame_counter == 100:
            # Redraw start and end circles.
            cv2.circle(map, (start_x, start_y), sf, (255, 0, 255), -1)
            cv2.circle(map, (end_x, end_y), sf, (255, 255, 0), -1)
            # Save current map state as a frame in the final video.
            video_out.write(map)
            # Reset the frame counter.
            video_frame_counter = 0

        # Check if the current node overlaps/connects to a node in the closed set. If it does, check if
        # that overlap is the best path found.
        overlap, overlap_theta_idx = overlap_check(closed_set_backward, current_x, current_y, current_theta, sf)
        if overlap and current_x != end_x and current_y != end_y:

            # Generate the key for the matching node and get it.
            node_key_matching = (int(current_y /sf/ 0.5), int(current_x /sf/ 0.5), overlap_theta_idx)
            matching_node = visited_backward[node_key_matching]
```

```
                # Get the cost of the path and update the best cost if this is better.
                cost_of_path = current_node.Node_Cost + matching_node.Node_Cost
                if cost_of_path < best_cost or best_cost == -1:
                    best_cost = cost_of_path
                    best_cost_node_forward = current_node
                    best_cost_node_backward = matching_node

            # For the current node, apply each of the five move functions and examine
            # the newNode generated from moving in each direction.
            for move in directions:
                # Get newNode from current move.
                newNode = move(current_node, step_size, sf, end_x, end_y)

                if valid_line(current_x, current_y, newNode.Node_x, newNode.Node_y, map.shape, obstacles): # Check that it isn't in an obstacle.
                    node_key = (int(newNode.Node_y), int(newNode.Node_x), angle_to_index(newNode.Node_theta))

                    if closed_set_forward[node_key] == False: # Check that it is not in the closed set.
                        if seen_forward[node_key] == False: # Check that it isn't in the open nor closed lists.

                            # Add it to the seen set.
                            seen_forward[node_key] = True

                            # Add it to the visited set.
                            visited_forward[node_key] = newNode
                            heapq.heappush(open_set_forward, newNode)

                        # If the node is in the open list AND the new cost is cheaper than the old cost to this node, rewrite it
                        # within visited and add the newNode to the open_set. The old version will be safely skipped.
                        elif seen_forward[node_key] == True:
                            if visited_forward[node_key].Node_Cost > newNode.Node_Cost:
                                visited_forward[node_key] = newNode
                                heapq.heappush(open_set_forward, newNode)

                    # Draw each of the movement directions.
                    cv2.line(map, (int(newNode.Node_x), int(newNode.Node_y)),(int(current_node.Node_x), int(current_node.Node_y)), (155,155,155),1)
            if times_switched < turn_limit:
                # Switch back to backward search.
                forward_turn = False
                times_switched+=1

        else: ### !!!!! ### !!!!! ### !!!!! ### BEGIN BACKWARD SEARCH ### !!!!! ### !!!!! ### !!!!! ###

            # Get the node with the smallest cost from the open_set.
            current_node = heapq.heappop(open_set_backward)
            # Extract it's x and y position.
```

```
        current_x, current_y, current_theta = current_node.Node_x, current_node.Node_y, current_node.Node_theta

        # Verify that this position is not in the closed set.
        # Skip this iteration if it is in the closed_set as the position
        # has already been fully explored. This is required because
        # there is no efficient implementation to updating nodes within a heapq.
        # As such, a node may be added to the heapq, then added again to the heapq if
        # a better parent was found.
        if closed_set_backward[int(current_y/sf/.5), int(current_x/sf/.5), angle_to_index(current_theta)] == True:
            continue
        else:
            # Add the current node to the closed set.
            closed_set_backward[int(current_y/sf/.5), int(current_x/sf/.5), angle_to_index(current_theta)] = True

        # If the node currently popped has a greater cost + cost_est than the cost of the best path found,
        # then we can say definitively that we have found the optimal path. Alert user and get that path.
        if current_node.Node_Cost + current_node.Node_Cost_est > best_cost and best_cost != -1:
            print("Goal Reached!")
            print("Overlap Path Found!")
            path = get_bi_directional_path(visited_forward, visited_backward, best_cost_node_forward, best_cost_node_backward, sf)

            print(path)

            # For each pixel in the path, draw a magenta line
            for i in range(len(path) - 1):
                x1, y1 = path[i]
                x2, y2 = path[i + 1]
                cv2.line(map, (int(x1), int(y1)), (int(x2), int(y2)), (255, 0, 255), 2)
                video_out.write(map)

            # Release the video file.
            video_out.release()
            # Terminate search and return the final map with the path and area explored.
            return map

        # Increment the video_frame_counter and save a frame if it is the 100th frame.
        video_frame_counter += 1
        if video_frame_counter == 100:
            # Redraw start and end circles.
            cv2.circle(map, (start_x, start_y), sf, (255, 0, 255), -1)
            cv2.circle(map, (end_x, end_y), sf, (255, 255, 0), -1)
            # Save current map state as a frame in the final video.
            video_out.write(map)
            # Reset the frame counter.
            video_frame_counter = 0

        # Check if the current node overlaps/connects to a node in the closed set. If it does, check if
```

```
            # that overlap is the best path found.
            overlap, overlap_theta_idx = overlap_check(closed_set_forward, current_x, current_y, current_theta, sf)
            if overlap:

                # Generate the key for the matching node and get it.
                node_key_matching = (int(current_y /sf/ 0.5), int(current_x /sf/ 0.5), overlap_theta_idx)
                matching_node = visited_forward[node_key_matching]

                # Get the cost of the path and update the best cost if this is better.
                cost_of_path = current_node.Node_Cost + matching_node.Node_Cost
                if cost_of_path < best_cost or best_cost == -1:
                    best_cost = cost_of_path
                    best_cost_node_forward = matching_node
                    best_cost_node_backward = current_node

            # For the current node, apply each of the five move functions and examine
            # the newNode generated from moving in each direction. Check if it is the first iteration.
            # If it is the first iteration, only expand forward. This ensures we reach the end goal at the
            # correct orientation.
            if first_backward_iter:
                move = move_straight

                # Get newNode from current move.
                newNode = move(current_node, step_size, sf, start_x, start_y)

                if valid_line(current_x, current_y, newNode.Node_x, newNode.Node_y, map.shape, obstacles): # Check that it isn't in an
obstacle.
                    node_key = (int(newNode.Node_y), int(newNode.Node_x), angle_to_index(newNode.Node_theta))

                    if closed_set_backward[node_key] == False: # Check that it is not in the closed set.
                        if seen_backward[node_key] == False: # Check that it isn't in the open nor closed lists.

                            seen_backward[node_key] = True

                            visited_backward[node_key] = newNode
                            heapq.heappush(open_set_backward, newNode)

                        # If the node is in the open list AND the new cost is cheaper than the old cost to this node, rewrite it
                        # within visited and add the newNode to the open_set. The old version will be safely skipped.
                        elif seen_backward[node_key] == True:
                            if visited_backward[node_key].Node_Cost > newNode.Node_Cost:
                                visited_backward[node_key] = newNode
                                heapq.heappush(open_set_backward, newNode)
                    # Draw a line for each movement.
                    cv2.line(map, (int(newNode.Node_x), int(newNode.Node_y)),(int(current_node.Node_x), int(current_node.Node_y)),
(155,155,155),1)
                    first_backward_iter = False
```

```python
            else:
                for move in directions:
                    # Get newNode from current move.
                    newNode = move(current_node, step_size, sf, start_x, start_y)

                    if valid_line(current_x, current_y, newNode.Node_x, newNode.Node_y, map.shape, obstacles): # Check that it isn't in
an obstacle.
                        node_key = (int(newNode.Node_y /sf/ 0.5), int(newNode.Node_x /sf/ 0.5), angle_to_index(newNode.Node_theta))

                        if closed_set_backward[node_key] == False: # Check that it is not in the closed set.
                            if seen_backward[node_key] == False: # Check that it isn't in the open nor closed lists.

                                seen_backward[node_key] = True

                                visited_backward[node_key] = newNode
                                heapq.heappush(open_set_backward, newNode)

                            # If the node is in the open list AND the new cost is cheaper than the old cost to this node, rewrite it
                            # within visited and add the newNode to the open_set. The old version will be safely skipped.
                            elif seen_backward[node_key] == True:
                                if visited_backward[node_key].Node_Cost > newNode.Node_Cost:
                                    visited_backward[node_key] = newNode
                                    heapq.heappush(open_set_backward, newNode)
                        # Draw a line for each movement.
                        cv2.line(map, (int(newNode.Node_x), int(newNode.Node_y)),(int(current_node.Node_x), int(current_node.Node_y)),
(155,155,155),1)
            # Switch back to forward search.
            forward_turn = True

    # Release video and alert the user that no path was found.
    video_out.release()
    print("Path not found!")
    return map

# get_final_path backtracks the position to find the path.
def get_final_path(visited, end_node, sf):
    # create a list of x and y positions.
    path_xys = []
    current_x, current_y, current_theta = end_node.Node_x, end_node.Node_y, end_node.Node_theta

    # Node key is used for indexing to get nodes.
    node_key = (int(current_y /sf/ 0.5), int(current_x /sf/ 0.5), angle_to_index(current_theta))

    while node_key in visited:  # Ensure the node exists in visited.
        path_xys.append((current_x, current_y)) # Add the current x and y.
        # Get the current parent's positon.
        parent_x = visited[node_key].Parent_Node_x
```

```python
        parent_y = visited[node_key].Parent_Node_y
        parent_theta = visited[node_key].Parent_Node_theta

        # Stop when we reach the starting node.
        if (current_x, current_y, current_theta) == (parent_x, parent_y, parent_theta):
            break

        # Update for the next iteration.
        current_x, current_y, current_theta = parent_x, parent_y, parent_theta
        node_key = (int(current_y /sf/ 0.5), int(current_x /sf/ 0.5), angle_to_index(current_theta))

    path_xys.reverse()  # Reverse to get the correct order.
    return path_xys

# Function to stitch forward and backward paths together.
def get_bi_directional_path(visited_forward, visited_backward, forward_node, backward_node, sf):
    path_forward = get_final_path(visited_forward, forward_node, sf)
    path_backward = get_final_path(visited_backward, backward_node, sf)

    path_backward.reverse()  # Reverse backward path to align with forward path.
    return path_forward + path_backward[1:]  # Merge paths, removing duplicate meeting node.

def main():
    print("Program Start")
    print("Please enter the start and end coordinates.")
    print("Coordinates should be given as integers in units of mm from the bottom left origin.")
    print("Image Width is 600 mm. Image Height is 250 mm.")

    # The scale factor is the resolution of the image for pathing. A scale factor of 2
    # makes the image .5 mm in resolution. DO NOT MODIFY.
    sf = 2

    # Generate and expand the obstacle map.
    obstacle_map = gen_obstacle_map(sf=sf)

    # Prompt the user for robot radius, clearance, and step size.
    robot_radius = int(input(f"Enter the robot's radius (int): "))
    clearance = int(input(f"Enter the clearance radius (int): "))
    step_size = int(input(f"Enter the step size (int): "))

    # Expand the obstacle space for the robot radius then for the clearance.
    expanded_obstacle_map, obs_map_gray = expand_obstacles(obstacle_map, sf, robot_radius)
    expanded_obstacle_map2, obs_map_gray = expand_obstacles(expanded_obstacle_map, sf, clearance)

    # Prompt the user for the start and end points for planning.
    start_x, start_y, start_theta = get_point(prompt="start", sf=sf, obstacles=obs_map_gray)
    end_x, end_y, end_theta = get_point(prompt="end", sf=sf, obstacles=obs_map_gray)
```

```python
    # Correct the angles to align with OpenCV's top left origin system.
    start_theta = -start_theta % 360
    end_theta = (-end_theta + 180 )% 360

    print("Planning Path...")

    # Start timer to get computation time.
    # start_time = time.time()
    # Apply A* search
    final_path_image = A_star_search(map=expanded_obstacle_map2, obstacles=obs_map_gray, start=(start_x, start_y, start_theta),
end=(end_x, end_y, end_theta), sf=sf, step_size=step_size)
    # total_time = time.time() - start_time
    print("Program Finished.")
    # print(f"Time to compute path: {total_time} seconds")
    print("Click image and press keyboard to close program and final image.")

    # Show the solution.
    cv2.imshow("Map", final_path_image)
    cv2.waitKey(0)
    return

if __name__ == "__main__":
    main()
```