```python
# Import necessary libraries
import cv2
import numpy as np
from queue import PriorityQueue
import time


# Canvas dimensions
canvas_height = 501
canvas_width = 1201

# Define the colors
clearance_color = (127, 127, 127)
obstacle_color = (0, 0, 0)
free_space_color = (255, 255, 255)
threshold = 1.5

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

# Define obstacles using half plane model
def obstacles(node):
    x, y = node
    Hex_center = (650, 250)
    Xc, Yc = Hex_center
    y = abs(y - canvas_height)
    side_length = 150
    R = np.cos(np.pi / 6) * side_length
    obstacles = [
        (x >= 100 and x <= 175 and y >= 100 and y <= 500),
        (x >= 275 and x <= 350 and y >= 0 and y <= 400),
        (x >= 900 and x <= 1100 and y >= 50 and y <= 125),
        (x >= 900 and x <= 1100 and y >= 375 and y <= 450),
        (x >= 1020 and x <= 1100 and y >= 50 and y <= 450),
        (x >= Xc - R and x <= Xc + R and y <= ((np.pi/6)*(x-(Xc-R)))+325 and y <=
-((np.pi/6)*(x-(Xc+R)))+325 and y >= -((np.pi/6)*(x-(Xc-R)))+175 and y >=
((np.pi/6)*(x-(Xc+R)))+175),

    ]
    return any(obstacles)

# Define clearance zones
```

```
def clearance(x, y, clearance):
    clearance = clearance + robo_radius
    Hex_center = (650, 250)
    Xc, Yc = Hex_center
    y = abs(y - canvas_height)
    side_length = 150
    R = (np.cos(np.pi / 6) * side_length)  + clearance
    clearance_zones = [
        (x >= 100 - clearance and x <= 175 + clearance and y >= 100 - clearance
and y <= 500 + clearance),
        (x >= 275 - clearance and x <= 350 + clearance and y >= 0 - clearance and
y <= 400 + clearance),
        (x >= 900 - clearance and x <= 1100 + clearance and y >= 50 - clearance
and y <= 125 + clearance),
        (x >= 900 - clearance and x <= 1100 + clearance and y >= 375 - clearance
and y <= 450 + clearance),
        (x >= 1020 - clearance and x <= 1100 + clearance and y >= 50 - clearance
and y <= 450 + clearance),
        (x >= Xc - R and x <= Xc + R and y <= ((np.pi/6)*(x-(Xc-R)))+325 +
clearance and y <= -((np.pi/6)*(x-(Xc+R)))+325 + clearance and y >= -
((np.pi/6)*(x-(Xc-R)))+175 - clearance and y >= ((np.pi/6)*(x-(Xc+R)))+175 -
clearance),
        (x <= clearance or x >= canvas_width - clearance or y <= clearance or y
>= canvas_height - clearance), # Add clearance to the edges of the canvas
    ]
    return any(clearance_zones)

# Function to check if the node is free
def is_free(x, y, theta):
    theta_normalized = theta % 360
    theta_index = theta_normalized // 30
    if x >= 0 and x < canvas_width and y >= 0 and y < canvas_height:
        if canvas_array[x, y, theta_index] != np.inf:
            return True
    else:
        return False

# Function to get the neighbors of a node and generate the new nodes
def get_neighbors(node):
    x, y , theta = node
    neighbours = []
```

```
    action_set = [theta, theta +30, theta -30, theta +60, theta -60]   # Action
set for the robot
    for action in action_set:
        x_new = x + step_size*np.sin(np.deg2rad(action))
        y_new = y - step_size*np.cos(np.deg2rad(action))
        x_new = int(round(x_new))
        y_new = int(round(y_new))
        if is_free(x_new, y_new, action):
            cost = step_size
            neighbours.append(((x_new, y_new, action), cost))

    return neighbours

# Function to check if the goal is reached
def check_goal_reached(current_node, goal):
    distance = canvas_array[current_node[0], current_node[1], 0]
    return distance < threshold and current_node[2] == goal[2]

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

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

        for next_node, cost in get_neighbors(current_node[0]):    # Get the
neighbors of the current node
            theta_normalized = next_node[2] % 360
            theta_index = theta_normalized // 30
            cost_to_go = canvas_array[next_node[0], next_node[1], 0]
```

```
            new_cost = current_node[1] + cost + cost_to_go    # Calculate the new
cost
            nc = current_node[1] + cost
            if next_node not in cost_so_far or new_cost <
cost_so_far[next_node]:     # Check if the new cost is less than the cost so far
                cost_so_far[next_node] = nc    # Update the cost so far
                priority = new_cost        # Calculate the priority
                pq.put((priority, (next_node, nc)))   # Add the node to the
priority queue
                cv2.arrowedLine(canvas, (current_node[0][0], current_node[0][1]),
(next_node[0], next_node[1]), (255, 0, 0), 1) # Draw the path
                canvas_array[next_node[0], next_node[1], int(theta_index)] =
np.inf  # Update the visited nodes to eliminate revisiting
                came_from[next_node] = current_node[0]
                count += 1
                if count%3000 == 0:
                    out.write(canvas)
    return None, None, None   # Return None if no path is found

# Function to reconstruct the path
def reconstruct_path(came_from, start, goal):
    # Start with the goal node and work backwards to the start
    current = goal
    path = [current]
    while current != start:
        current = came_from[current]  # Move to the previous node in the path
        path.append(current)
    path.reverse()  # Reverse the path to go from start to goal
    return path

# Function to visualize the path
def visualize_path(path):
    count = 0
    for i in range(len(path)-1):
        x, y, t = path[i]
        xn, yn, tn = path[i+1]
        cv2.arrowedLine(canvas, (x, y), (xn,yn), (0, 0, 255), 2)
        count += 1
        if count%15 == 0:
            out.write(canvas)
    cv2.destroyAllWindows()
    for i in range(30):
```

```
        out.write(canvas)
    cv2.imshow('Path', canvas)
    out.release()


print('''
_____________________________________

    __          __
    / |       /    )
---/__|-------\------_/_----__---)__-
  /   |        \     /    /   ) /   )
_/____|____(____/___(_ __(___(_/_____

''')

# User input
while True:
    print("Step size should be between 1 and 10")
    step_size = input("Enter the step size: ")
    step_size = int(step_size)                                 # Get the step size from the user
    if step_size > 0 and step_size <= 10:
        break

while True:
    print("Clearance distance should be a positive number")
    clearance_distance = input("Enter the clearance distance: ")
    if clearance_distance.isdigit() and int(clearance_distance) >= 0:      # Get the clearance distance from the user
        clearance_distance = int(clearance_distance)
        break

while True:
    print("Robot radius should be a positive number")
    robo_radius = input("Enter the robot radius: ")
    if robo_radius.isdigit() and int(robo_radius) >= 0:                # Get the robot radius from the user
        robo_radius = int(robo_radius)
        break

print("\nGenerating the map...")
```

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

out = cv2.VideoWriter('A_star.mp4', cv2.VideoWriter_fourcc(*'mp4v'), 30,
(canvas_width, canvas_height))  # Create a video writer object

C = clearance_distance + robo_radius + 1
Xc = canvas_width - C
Yc = canvas_height - C

# Get the start Node from the user
while True:
    print(f"\nThe start node and goal node should be within the canvas dimensions
({C}-{Xc}, {C}-{Yc}) and not inside an obstacle.\n")
    Xi = input("Enter the start node X: ")
    Yi = input("Enter the start node Y: ")
    Ti = input("Enter the start node Angle: ")
    Xi = int(Xi)
    Yi = int(Yi)
    Ti = int(Ti)

    if not (Xi < 0 or Xi >= canvas_width or Yi < 0 or Yi >= canvas_height):      #
Check if the start node is within the canvas dimensions
        if all(canvas[Yi, Xi] == free_space_color):                              #
Check if the start node is not inside an obstacle
            break
        else:
            print("Start node is inside an obstacle")
    else:
        print("Start node is out of bounds.")

# Get the goal node from the user
while True:
    Xg = input("Enter the goal node X: ")
    Yg = input("Enter the goal node Y: ")
    To = input("Enter the goal node Angle: ")
    Xg = int(Xg)
```

```
    Yg = int(Yg)
    To = int(To)

    if not (Xg < 0 or Xg >= canvas_width or Yg < 0 or Yg >=
canvas_height):         # Check if the goal node is within the canvas dimensions
        if all(canvas[Yg, Xg] ==
free_space_color):                           # Check if the goal node is
not inside an obstacle
            break
        else:
            print("Goal node is inside an obstacle")
    else:
        print("Goal node is inside an obstacle or out of bounds.")

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

print("Start Node: ", (int(Xi), int(Yi), int(Ti)))
print("Goal Node: ", (int(Xg), int(Yg), int(To)))

Yi = abs(500 - int(Yi))  # Flip the Y coordinate
start_node = (int(Xi), int(Yi), int(Ti))

Yg = abs(500 - int(Yg))  # Flip the Y coordinate
goal_node = (int(Xg), int(Yg), int(To))

print("\nRunning A* algorithm...")
start_time = time.time()   # Start the timer

# Initialize the canvas array
canvas_array = np.zeros((canvas_width, canvas_height, 12))
for x in range(canvas_width):
    for y in range(canvas_height):
        if all(canvas[y, x] != free_space_color):
            canvas_array[x, y] = np.inf
        else:
            canvas_array[x, y] = ((Xg-x)**2 + (Yg-y)**2)**0.5  # Calculate the
heuristic cost for every node
```

```
cv2.circle(canvas, (Xi, Yi), 2, (0, 0, 255), -1)
cv2.circle(canvas, (Xg, Yg), 2, (0, 255, 0), -1)

for j in range(30):
    out.write(canvas)

came_from, cost_so_far, goal = a_star(start_node, goal_node) # Run the A*
algorithm

if came_from is None:
    print("No path found")
    end_time = time.time()
    execution_time = end_time - start_time          # Calculate the execution time
    print("Execution time: %.4f seconds" % execution_time)
    exit()
path = reconstruct_path(came_from, start_node, goal)
visualize_path(path)
for i in range(30):
    out.write(canvas)

end_time = time.time()
execution_time = end_time - start_time              # Calculate the execution
time

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