```python
import numpy as np
import cv2
import queue
import time
from math import dist

# Defining the search space(map) and threshold for finding the goal node.
map_width = 1200
map_height = 500
threshold = 1.5

# Defining a class for nodes in the graph
class Node:
    def __init__(self, x, y, theta, cost, parent_id, c2g=0):
        self.x = x
        self.y = y
        self.theta = theta
        self.cost = cost
        self.parent_id = parent_id
        self.c2g = c2g

    def __lt__(self, other):
        return self.cost + self.c2g < other.cost + other.c2g

# Configuring the obstacle space and constructing the obstacles
def Configuration_space():
    # Initialize an empty grid for obstacle space
    obs_space = np.full((map_height, map_width), 0)

    for y in range(map_height) :
        for x in range(map_width):

            # Rectangle 1 Obastacle
            r11_buffer = (x + clearance + robot_radius) - 100
            r12_buffer = (500 - y + clearance + robot_radius) - 100
            r13_buffer = (x - clearance - robot_radius) - 175
            r14_buffer = (500 - y - clearance - robot_radius) - 500

            # Rectangle 2 Obastacle
            r21_buffer = (x + clearance + robot_radius) - 275
            r22_buffer = (500 - y + clearance + robot_radius) - 0
            r23_buffer = (x - clearance - robot_radius) - 350
            r24_buffer = (500 - y - clearance - robot_radius) - 400

            # Hexagon Obstacle
            h6_buffer = (500 - y + clearance + robot_radius) +  0.58*(x + clearance +
robot_radius) - 475
            h5_buffer = (500 - y + clearance + robot_radius) - 0.58*(x - clearance -
robot_radius) + 275
            h4_buffer = (x - clearance - robot_radius) - 781
            h3_buffer = (500 - y - clearance - robot_radius) + 0.58*(x - clearance -
robot_radius) - 775
            h2_buffer = (500 - y - clearance - robot_radius) - 0.58*(x + clearance +
robot_radius) - 24
            h1_buffer = (x + clearance + robot_radius) - 519

            # Block Obstacle
            t1_buffer = (x + clearance + robot_radius) - 900
            t2_buffer = (x + clearance + robot_radius) - 1020
            t3_buffer = (x - clearance - robot_radius) - 1100
            t4_buffer = (500 - y + clearance + robot_radius) - 50
            t5_buffer = (500 - y - clearance - robot_radius) - 125
            t6_buffer = (500 - y + clearance + robot_radius) - 375
            t7_buffer = (500 - y - clearance - robot_radius) - 450

            # Conditions for setting the border around the frame of the map
            w1 = (map_height - y) - (clearance + robot_radius)
```

```
            w2 = (map_height - y) - (map_height - clearance - robot_radius)
            w3 = (x) - (clearance + robot_radius)
            w4 = (x) - (map_width - (clearance + robot_radius))

            # Setting the border around the block obstacle
            if (t1_buffer >= 0 and t2_buffer <= 0 and
                t4_buffer >= 0 and t5_buffer <= 0):
                obs_space[y, x] = 1

            if (t2_buffer >= 0 and t3_buffer <= 0 and
                t4_buffer >= 0 and t7_buffer <= 0):
                obs_space[y, x] = 1

            if (t6_buffer >= 0 and t7_buffer <= 0 and
                t1_buffer >= 0 and t2_buffer <= 0):
                obs_space[y, x] = 1

            # Setting the border around Rectangle 1 Obstacle
            if (r11_buffer >= 0 and r12_buffer >= 0 and
                r13_buffer <= 0 and r14_buffer <= 0):
                obs_space[y, x] = 1

            # Setting the border around Rectangle 2 Obstacle
            if (r21_buffer >= 0 and r23_buffer <= 0 and
                r24_buffer <= 0 and r22_buffer >= 0):
                obs_space[y, x] = 1

            # Setting the border around Hexagon Obstacle
            if (h6_buffer >= 0 and h5_buffer >= 0 and
                h4_buffer <= 0 and h3_buffer <= 0 and
                h2_buffer <= 0 and h1_buffer >= 0) or  (w1 < 0) or (w2 > 0) or (w3 < 0) or
(w4 > 0):

                obs_space[y, x] = 1

            # Rectangle 1 Obastacle
            r11 = (x) - 100
            r12 = (500 - y) - 100
            r13 = (x) - 175
            r14 = (500 - y) - 500

            # Rectangle 2 Obastacle
            r21 = (x) - 275
            r22 = (500 - y) - 0
            r24 = (x) - 350
            r23 = (500 - y) - 400

            # Hexagon Obstacle
            h6 = (500 - y) +  0.58*(x) - 475.098
            h5 = (500 - y) - 0.58*(x) + 275.002
            h4 = (x) - 779.9
            h3 = (500 - y) + 0.58*(x) - 775.002
            h2 = (500 - y) - 0.58*(x) - 24.92
            h1 = (x) - 520.1

            # Block Obstacle
            t1 = (x) - 900
            t2 = (x) - 1020
            t3 = (x) - 1100
            t4 = (500 - y) - 50
            t5 = (500 - y) - 125
            t6 = (500 - y) - 375
            t7 = (500 - y) - 450

            # Setting the line constraint to obtain the obstacle space with buffer
            if ((h6 > 0 and h5 > 0 and h4 < 0 and h3 < 0 and h2 < 0 and h1 > 0) or
                (r11 > 0 and r12 > 0 and r13 < 0 and r14 < 0) or
                (r21 > 0  and r23 < 0 and r24 < 0 and r22 > 0) or
```

```
                (t1 > 0 and t2 < 0 and t4 > 0 and t5 < 0) or
                (t2 > 0 and t3 < 0 and t4 > 0 and t7 < 0) or
                (t6 > 0 and t7 < 0 and t1 > 0 and t2 < 0)):
                obs_space[y, x] = 2

    for i in range(map_width):
        for j in range(clearance+robot_radius):
            obs_space[j][i] = 1
            obs_space[map_height-1-j][i] = 1

    for i in range(map_height):
        for j in range(clearance+robot_radius):
            obs_space[i][j] = 1
            obs_space[i][map_width-1-j] = 1

    return obs_space

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

# Check if coordinates are within the boundaries of the obstacle space and if the cell is
occupied by an obstacle (value 1 or 2)
def Validity(x, y, obs_space):
    if x < 0 or x >= map_width or y < 0 or y >= map_height or obs_space[y][x] == 1 or
obs_space[y][x] == 2:
        return False

    return obs_space[y, x] == 0

# Heuristic function to find euclidean distance between start and goal node.
def heuristic(current_node, goal_node):
    return np.sqrt((current_node.x - goal_node.x)**2 + (current_node.y - goal_node.y)**2)

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

# Actions and cost calculation

# Action of moving up, by positive angle 60 degrees
def UP_60(x, y, theta, robot_step_size, cost):
    theta = theta + 60
    x += round(robot_step_size * np.cos(np.radians(theta)))
    y += round(robot_step_size * np.sin(np.radians(theta)))
    cost = robot_step_size + cost  # Update cost with step size
    return x, y, theta, cost

# Action of moving up, by positive angle 30 degrees
def UP_30(x, y, theta, robot_step_size, cost):
    theta = theta + 30
    x += round(robot_step_size * np.cos(np.radians(theta)))
    y += round(robot_step_size * np.sin(np.radians(theta)))
    cost = robot_step_size + cost  # Update cost with step size
    return x, y, theta, cost

# Action of moving straight, with angle of 0 degrees
def STRAIGHT_0(x, y, theta, robot_step_size, cost):
    theta = theta + 0
    x += round(robot_step_size * np.cos(np.radians(theta)))
```

```
    y += round(robot_step_size * np.sin(np.radians(theta)))
    cost = robot_step_size + cost  # Update cost with step size
    return x, y, theta, cost

# Action of moving down, by positive angle 30 degrees
def DOWN_30(x, y, theta, robot_step_size, cost):
    theta = theta - 30
    x += round(robot_step_size * np.cos(np.radians(theta)))
    y += round(robot_step_size * np.sin(np.radians(theta)))
    cost = robot_step_size + cost  # Update cost with step size
    return x, y, theta, cost

# Action of moving down, by positive angle 60 degrees
def DOWN_60(x, y, theta, robot_step_size, cost):
    theta = theta - 60
    x += round(robot_step_size * np.cos(np.radians(theta)))
    y += round(robot_step_size * np.sin(np.radians(theta)))
    cost = robot_step_size + cost  # Update cost with step size
    return x, y, theta, cost

# A* search algorithm implementation for path planning.
def a_star(start, goal, obs_space, robot_step_size):
    # starting the timer to calculate time taken to run the algorithm.
    start_time = time.time()
    if Check_goal(start, goal):
        return None, 1
    goal_node = goal
    start_node = start

    # possible moves (action set)
    moves = [UP_60, UP_30, STRAIGHT_0, DOWN_30, DOWN_60]
    # Dictionary to store all open nodes
    unexplored_nodes = {}
    unexplored_nodes[(start_node.x, start_node.y)] = start_node  # Add start node to
unexplored nodes

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

    all_nodes = []  # List to store all nodes that have been traversed, for visualization

    while not priority_queue.empty():
        present_node = priority_queue.get()[1]  # Get the node with the lowest cost from the
priority queue
        all_nodes.append([present_node.x, present_node.y, present_node.theta])

        current_id = (present_node.x, present_node.y)
        if Check_goal(present_node, goal_node):
            goal_node.parent_id = present_node.parent_id
            goal_node.cost = present_node.cost
            print("Goal Node found")
            end_time = time.time()  # End time for measuring time taken
            time_taken = end_time - start_time
            print("Time taken to find the goal node:", time_taken , "seconds")
            return all_nodes, time_taken, 1

        if current_id in explored_nodes:
            continue
        else:
            explored_nodes[current_id] = present_node

        del unexplored_nodes[current_id]
```

```
        for move in moves:
            x, y, theta, cost = move(present_node.x, present_node.y, present_node.theta, 
robot_step_size,
                                     present_node.cost)
            c2g = heuristic(Node(x, y, theta, 0, -1), goal_node)  # Calculate heuristic cost-
to-goal

            new_node = Node(x, y, theta, cost, present_node, c2g)
            new_node_id = (new_node.x, new_node.y)

            if not Validity(new_node.x, new_node.y, obs_space):
                continue
            elif new_node_id in explored_nodes:
                continue

            if new_node_id in unexplored_nodes:
                if new_node.cost < unexplored_nodes[new_node_id].cost:
                    unexplored_nodes[new_node_id].cost = new_node.cost
                    unexplored_nodes[new_node_id].parent_id = new_node.parent_id
            else:
                unexplored_nodes[new_node_id] = new_node

            priority_queue.put((new_node.cost + new_node.c2g, new_node))  # Put the new node 
into the priority queue

    return all_nodes, 0 # If goal not found, return all nodes traversed and failure flag

# Backtrack to find the path from goal to start node
def Backtrack(goal_node):
    x_path = []
    y_path = []
    x_path.append(goal_node.x)
    y_path.append(goal_node.y)

    # Initialize total cost
    total_cost = goal_node.cost
    parent_node = goal_node.parent_id
    while parent_node != -1:
        x_path.append(parent_node.x)
        y_path.append(parent_node.y)
        parent_node = parent_node.parent_id

    x_path.reverse()
    y_path.reverse()

    return x_path, y_path, total_cost

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

# Frame counter
frame_counter = 0
frame_interval = 200  # Write every 200th frame

if __name__ == '__main__':
    # Visualize the map and path
    image = np.ones((map_height, map_width, 3), dtype=np.uint8) * 255
    # Get clearance of the obstacle
    while True:
        clearance = input("Assign Clearance to the Obstacles: ")
        try:
            clearance = int(clearance)
            break
        except ValueError:
```

```
            print("Invalid input format. Please enter an integer.")

    # Get radius of the robot
    while True:
        robot_radius = input("Enter the Radius of the Robot: ")
        try:
            robot_radius = int(robot_radius)
            break
        except ValueError:
            print("Invalid input format. Please enter an integer.")

    obs_space = Configuration_space()
    # OBSTACLE OUTLINE
    image[obs_space == 1] = (0, 0, 0)
    # OBSTACLE FILL (orange)
    image[obs_space == 2] = (0, 169, 255)

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

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

    # Get step size of the robot
    while True:
        robot_step_size = input("Enter Step size of the Robot: ")
        try:
            robot_step_size = int(robot_step_size)
            if 1 <= robot_step_size <= 10:
                break
            else:
                print("The robot step size must be between 1 and 10")
        except ValueError:
            print("Invalid input format. Please enter an integer.")

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

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

    print("Processing......")

    start_node = Node(s_x,map_height- s_y,start_theta, 0, -1)  # Start node with cost 0 and no
parent
    goal_node = Node(e_x,map_height- e_y,end_theta, 0, -1)  # You can adjust the goal node
coordinates as needed
    all_nodes, found_goal, time_taken = a_star(start_node, goal_node, obs_space,
robot_step_size)

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

    # Visualize the map and path
    image_with_path = np.copy(image)
    # Draw obstacle boundary in black
    image_with_path[obs_space == 1] = [0, 0, 0]

    if found_goal:
        for idx, node in enumerate(all_nodes):
            cv2.circle(image_with_path, (node[0], node[1]), 1, (0, 255, 0), -1)
            frame_counter += 1

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

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

        # Draw start and end points
        cv2.circle(image_with_path, (s_x,map_height- s_y), 3, (0, 255, 255), -1)  # yellow
circle for start point
        cv2.circle(image_with_path, (e_x,map_height- e_y), 3, (0, 0, 255), -1)  # Red circle
for end point

        # Draw shortest path
        for i in range(len(x_path) - 1):
            cv2.line(image_with_path, (x_path[i], y_path[i]), (x_path[i + 1], y_path[i + 1]),
(255, 0, 0), 2)
            video_out.write(image_with_path)
            cv2.imshow("Map with Path", image_with_path)
            cv2.waitKey(1)
        video_out.write(image_with_path)
        cv2.imshow("Map with path", image_with_path)
        cv2.waitKey(1)

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