```
import heapq
import cv2
import numpy as np
import time
import matplotlib.animation as animation
import matplotlib.pyplot as plt
from math import dist
from moviepy.editor import *

# defining a Node class to represent a node in the A* search algorithm.
class Node:

    def __init__(self, x, y, t, cost, parent_node_id, cost_to_go = 0):
        self.x = x
        self.y = y
        self.t = t
        self.cost = cost
        self.parent_node_id = parent_node_id
        self.cost_to_go = cost_to_go
# returns the x coordinate of the node.
    def get_x(self):
        return self.x
# returns the y coordinate of the node.
    def get_y(self):
        return self.y
# returns the orientation angle of the node in degrees.
    def get_t(self):
        return self.t
# returns the cost to reach the node
```

```
    def get_cost(self):
        return self.cost
# returns the ID of the node's parent
    def get_parent_node_id(self):
        return self.parent_node_id

    def get_cost_to_go(self):
        return self.cost_to_go

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

def half_plane_obstcles(width, height,radius,clearance):
    obstacle_space = np.zeros((height, width))
    c = radius + clearance

    #running a for loop for half plane functions to get all the as obstacles
    for y in range(height):
        for x in range(width):
            #bottom rectangle with clearance equations
            r11_c, r12_c, r13_c, r14_c = x+c-100, y-c-100, x-c-150, y-0
            #top rectangle with clearance equations
            r21_c, r22_c, r23_c, r24_c = x+c-100, y-250, x-c-150, y+c-150
            #equations for six lines of hexagon with clearance
            h1_c, h2_c, h3_c, h4_c, h5_c, h6_c = x+1.3*c-235.04, y-c-0.58*(x+c)-26.82, y-c+0.58*(x-c)-373.21,
(x-1.3*c)-364.95, y+c-(0.58*x-c)+123.21, y+c+0.58*(x+c)-223.18
            #equations for six lines of hexagon with clearance
            t1_c, t2_c, t3_c = x+c-460, y-c+2*(x-c)-1145, y+c-2*(x-c)+895
```

```
            #defining the boundary conditions for each line, to make an enclosed space, and then adding that to obstacle_space

            if (h6_c>0 and h5_c>0 and h4_c<0 and h3_c<0 and h2_c<0 and h1_c>0) or (r11_c>0 and r12_c<0 and r13_c<0) or (r21_c>0 and r23_c<0 and r24_c>0) or (t1_c>0 and t2_c<0 and t3_c>0):

                obstacle_space[y, x] = 2


    return obstacle_space


def map_space(width,height):

    obstacle_space = np.zeros((height, width))

    #running a for loop for half plane functions to get all the as obstacles

    for y in range(height):

        for x in range(width):

            #bottom rectangle equations // r14 is not required as it lies within the boundary

            r11, r12, r13 = x-100, y-100, x-150

            #top rectangle equations // r24 is not required as it lies within the boundary

            r21, r23, r24 = x-100, x-150, y-150

            #equations for six lines of hexagon

            h1, h2, h3, h4, h5, h6 = x-235.04, y-0.58*x-26.82, y+0.58*x-373.21, x-364.95, y-0.58*x+123.21, y+0.58*x-223.18

            #equations for triangle

            t1, t2, t3 = x-460, y+2*x-1145, y-2*x+895


            #defining the boundary conditions for each line, to make an enclosed space, and then adding that to obstacle_space

            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) or (r21>0 and r23<0 and r24>0) or (t1>0 and t2<0 and t3>0):

                obstacle_space[y, x] = 1

    return obstacle_space
```

```
def move_forward(x, y, t, step, cost):
   t += 0
   # Update the x and y values based on the current position, step size, and angle of movement in radians
   x += step * np.cos(np.radians(t))
   y += step * np.sin(np.radians(t))
   # Round off the updated x and y values to the nearest integer
   x, y = round(x), round(y)
   cost += 1
   return x, y, t, cost


def move_up(x, y, t, step, cost):
   t += 30
   # Update the x and y values based on the current position, step size, and angle of movement in radians
   x += step * np.cos(np.radians(t))
   y += step * np.sin(np.radians(t))
   # Round off the updated x and y values to the nearest integer
   x, y = round(x), round(y)
   cost += 1
   return x, y, t, cost


def move_down(x, y, t, step, cost):
   t -= 30
   # Update the x and y values based on the current position, step size, and angle of movement in radians
   x += step * np.cos(np.radians(t))
   y += step * np.sin(np.radians(t))
   # Round off the updated x and y values to the nearest integer
   x, y = round(x), round(y)
```

```
    cost += 1
    return x, y, t, cost

def move_up2(x, y, t, step, cost):
    t += 60
    # Update the x and y values based on the current position, step size, and angle of movement in radians
    x += step * np.cos(np.radians(t))
    y += step * np.sin(np.radians(t))
    # Round off the updated x and y values to the nearest integer
    x, y = round(x), round(y)
    cost += 1
    return x, y, t, cost

def move_down2(x, y, t, step, cost):
    t -= 60
    x += step * np.cos(np.radians(t))
    y += step * np.sin(np.radians(t))
    x, y = round(x), round(y)
    cost += 1
    return x, y, t, cost

# defining a dictionary called ACTION_SET that maps action names to corresponding functions that perform the above actions
ACTION_SET = {
    '2up': move_up2,
    'up': move_up,
    'forward': move_forward,
    'down': move_down,
```

```
    '2down': move_down2
}

# Define a function to execute actions from the action set
def execute_action(action, x, y, t, step, cost):
    if action in ACTION_SET:
        return ACTION_SET[action](x, y, t, step, cost)
    else:
        return None



# Checking for validity of the move
def valid_move(x, y, obstacle_space):
    shape = obstacle_space.shape
    return (x >= shape[1] or x < 0 or y >= shape[0] or y < 0
            or obstacle_space[y, x] in {1, 2})



# Checking for validity of orientation
def valid_orientation(t):
    if t%30 == 0:
        return t
    else:
        return False

# Checking if goal node is current node
def  goal_checker(c, g):
    dis = dist((c.x, c.y), ( g.x, g.y))
    if dis<1.5:
```

```
        return True
    else:
        return False

# Generating key
def key(node):
    key = 200*node.x + 100*node.y
    return key

# defining the astar algorithm
def a_star(start, goal, obstacle_space, step):

    if goal_checker(start, goal):
        return None,1
    g_node = goal
    s_node = start

    moves = ['2up','up', 'forward', 'down', '2down']
    unexplored = {}

    # Generating a unique key for identifying the node
    start_key = key(s_node)
    unexplored[(start_key)] = s_node

    explored = {}
    priority_list = []
    heapq.heappush(priority_list, [s_node.cost, s_node])

    nodes = []
```

```
# keep looping until priority_list is empty
 while (len(priority_list) != 0):
    # pop the node with the lowest cost from priority_list
    c_node = (heapq.heappop(priority_list))[1]
  # add the current node to the nodes list
    nodes.append([c_node.x, c_node.y, c_node.t])
    # get the unique identifier for the current node
    c_id = key(c_node)

    # check if the current node is the goal node
    if goal_checker(c_node, g_node):
       # update the goal node's parent to be the current node's parent
       g_node.parent_node_id = c_node.parent_node_id
        # update the goal node's cost to be the current node's cost
       g_node.cost = c_node.cost
       print("A* search complete, printing results:-")
       return nodes,1
     # check if the current node has already been explored
    if c_id in explored:
       continue
    else:
       # add the current node to the explored dictionary
       explored[c_id] = c_node
     # remove the current node from the unexplored dictionary
    del unexplored[c_id]

    for move in moves:
```

```
        x,y,t,cost = execute_action(move,c_node.x,c_node.y,c_node.t, step_size, c_node.cost)
        # calculate the new x, y, t, and cost values for the new node
       # calculate the cost-to-go for the new node
        cost_to_go = dist((x, y), (goal.x, goal.y))
      # create a new node object with the new values
        new_node = Node(x,y,t, cost,c_node, cost_to_go)


        new_node_id = key(new_node)
         # check if the new node is in collision with any obstacles
        if valid_move(new_node.x, new_node.y, obstacle_space):
           continue
         # check if the new node has already been explored
        elif new_node_id in explored:
           continue
        if new_node_id in unexplored:
           #  check if the new node has a lower cost than the existing node
           if new_node.cost < unexplored[new_node_id].cost:
               #  update the existing node's cost to the new node's cost
              unexplored[new_node_id].cost = new_node.cost
               # update the existing node's parent to the new node's parent
              unexplored[new_node_id].parent_node_id = new_node.parent_node_id
        else:
            #  add if the new node is not in the unexplored dictionary
           unexplored[new_node_id] = new_node
        # add the new node to the priority list
        heapq.heappush(priority_list, [(new_node.cost + new_node.cost_to_go), new_node])


return nodes,0
```

```
#given size of the map
width = 600
height = 250

# backtrackering the path to generate final path
def backtracker(g_node):
    x_p = []
    y_p = []
    x_p.append(g_node.x)
    y_p.append(g_node.y)

    parent_node = g_node.parent_node_id
    while parent_node != -1:
        x_p.append(parent_node.x)
        y_p.append(parent_node.y)
        parent_node = parent_node.parent_node_id

    x_p.reverse()
    y_p.reverse()

    x = np.asarray(x_p)
    y = np.asanyarray(y_p)

    return x,y


# Creating the video
import matplotlib.pyplot as plt
```

```
def animate(s_node, g_node, x_p, y_p, nodes, obstacle_space, map_space, interval=50):
    fig = plt.figure()
    im = plt.imshow(obstacle_space, "GnBu", alpha=0.9) # add alpha parameter to adjust transparency
    m = plt.imshow(map_space, "Reds", alpha=0.5)
    plt.plot(s_node.x, s_node.y, "Dr")
    plt.plot(g_node.x, g_node.y, "Dg")
    ax = plt.gca()
    ax.invert_yaxis()


    def u(i):
        if i >= len(nodes):
            return
        node = nodes[i]
        plt.plot(node[0], node[1], "2g-")
        if i == len(nodes) - 1:
            plt.plot(x_p, y_p, ':r')
        return [im]

    ani = animation.FuncAnimation(fig, u, frames=len(nodes), interval=interval, blit=True)
    ani.save('manav19_ishaanp.gif')
    clip = (VideoFileClip("manav19_ishaanp.gif")
        .resize(height=480)
        .write_videofile("manav19_ishaanp.mp4", fps=24))
    plt.show()



# Main Body
```

```
def check_valid(x,y,t):

    if  valid_move(x, y, obstacle_space):
        print("Node is out of bounds, try different values in range")
        return False

    if not valid_orientation(t):
        print("Orientation has to be a multiple of 30")
        return False
    else:
        return True

# Main Body

check = False
while(check == False):
    # taking the clearance
    #taking the radius
    rad = input("Enter robot radius ")
    rad = int(rad)
    cl = input("Enter the clearance for the obstacles ")
    cl = int(cl)
    robot_step_size = input("Enter step size of the Robot: ")
    step_size = int(robot_step_size)
    obstacle_space = half_plane_obstcles(width,height,rad,cl)
    map_space = map_space(width,height)
    cost_to_go = 0
```

```
    # Take the start node from user
    start_coordinates = input("Enter Start node x coordinte, y coordinate and orientation seperated by spaces ")
    s_x, s_y,s_t = start_coordinates.split()
    s_x = int(s_x)
    s_y = int(s_y)
    s_t= int(s_t)

    # Take the goal node from user
    goal_coordinates = input("Enter Goal node x coordinte, y coordinate and orientation seperated by spaces ")
    g_x, g_y,g_t = goal_coordinates.split()
    g_x = int(g_x)
    g_y = int(g_y)
    g_t = int(g_t)

    if(check_valid(s_x,s_y,s_t)):
        if(check_valid(g_x,g_y,g_t)):
            timer_start = time.time()

            # Creating s_node and g_node objects
            s_node = Node(s_x, s_y,s_t, 0.0, -1,cost_to_go)
            g_node = Node(g_x, g_y,g_t, 0.0, -1, cost_to_go)
            nodes, flag = a_star(s_node, g_node, obstacle_space, robot_step_size)
            if (flag)==1:

                x_p,y_p = backtracker(g_node)
```

```
        else:
            print("No Path found ")
            x_p = 0
            y_p = 0

        animate(s_node, g_node, x_p, y_p, nodes, obstacle_space,map_space, interval=50)
        timer_stop = time.time()

        C_time = timer_stop - timer_start
        print("Time to complete the search was  ", C_time, "seconds")
        check = True
    else:
        print("Try different Values")
else:
    print("Try different Values")
```