```
#!/usr/env/bin python3


import sys
import os
import numpy as np
import matplotlib.pyplot as plt
from queue import PriorityQueue
import cv2
import argparse
import time
import math
import heapq

def standardLine(p1, p2):
    # ax+by+d=0
    assert(p1!=p2),"point1 equals to point2, cannot form line"

    tangent_vector = (p2[0]-p1[0], p2[1]-p1[1])
    if (tangent_vector[0]==0):
        normal_vector = (1,0)
    elif (tangent_vector[1]==0):
        normal_vector = (0,1)
    else:
        normal_vector = (1/(p2[0]-p1[0]), -1/(p2[1]-p1[1]))
    a, b = normal_vector
    norm = np.sqrt(pow(a, 2) + pow(b, 2))
    a, b = a / norm, b / norm
    d = -(a * p1[0] + b * p1[1])
    return a, b, d


class Map:
    #mm
    width = 600
    height = 250
    occGrid = np.zeros((height+1, width+1))
    robot_radius = 5 + 5

    def __init__(self, start, goal):
        self.start = start
        self.goal = goal

    @classmethod
    def generateOccGrid(self):

        # boundary
        boundaryLine_1 = standardLine((0,0), (0,250))
        boundaryLine_2 = standardLine((0,250), (600,250))
        boundaryLine_3 = standardLine((600,250), (600,0))
        boundaryLine_4 = standardLine((0,0), (600,0))

        #
        # triangle
        triangle_1 = standardLine((460, 225), (460, 25))
        triangle_2 = standardLine((460, 225), (510, 125))
        triangle_3 = standardLine((510, 125), (460, 25))


        # upper rectangle
        upperRectangleLine_1 = standardLine((100, 250), (100, 150))
        upperRectangleLine_2 = standardLine((150, 250), (150, 150))
```

```
        upperRectangleLine_3 = standardLine((100, 150), (150, 150))

        # upper rectangle
        lowerRectangleLine_1 = standardLine((100, 100), (100, 0))
        lowerRectangleLine_2 = standardLine((150, 100), (150, 0))
        lowerRectangleLine_3 = standardLine((100, 100), (150, 100))

        # hexagon
        edge =  75
        hexagonLine_1 = standardLine((235, 125 + edge/2), (235, 125 - edge/2))
        hexagonLine_2 = standardLine((235, 125 + edge/2), (300, 125 + edge))
        hexagonLine_3 = standardLine((300, 125 + edge), (365, 125 + edge/2))
        hexagonLine_4 = standardLine((365, 125 + edge/2), (365, 125 - edge/2))
        hexagonLine_5 = standardLine((300, 125 - edge), (365, 125 - edge/2))
        hexagonLine_6 = standardLine((235, 125 - edge/2), (300, 125 - edge))

        rows, cols = Map.occGrid.shape
        for i in range(0, rows):
            for j in range(0, cols):
                # transform from top-left (0,0) to bottom-left (0,0)
                x = j
                y = rows - 1 - i

                # boundary with clearance
                if ((boundaryLine_1[0] * x + boundaryLine_1[1] * y + boundaryLine_1[2]) <=
Map.robot_radius or \
                    (boundaryLine_2[0] * x + boundaryLine_2[1] * y + boundaryLine_2[2]) >= -
Map.robot_radius or \
                    (boundaryLine_3[0] * x + boundaryLine_3[1] * y + boundaryLine_3[2]) >= -
Map.robot_radius or \
                    (boundaryLine_4[0] * x + boundaryLine_4[1] * y + boundaryLine_4[2]) <=
Map.robot_radius ):
                    Map.occGrid[i, j]=2

                    # boundary
                    if ((boundaryLine_1[0] * x + boundaryLine_1[1] * y + boundaryLine_1[2])
<= 0 or \
                        (boundaryLine_2[0] * x + boundaryLine_2[1] * y + boundaryLine_2[2])
>= 0 or \
                        (boundaryLine_3[0] * x + boundaryLine_3[1] * y + boundaryLine_3[2])
>= 0 or \
                        (boundaryLine_4[0] * x + boundaryLine_4[1] * y + boundaryLine_4[2])
<= 0 ):
                        Map.occGrid[i, j]=1

                # triangle with clearance
                if ((triangle_1[0] * x + triangle_1[1] * y + triangle_1[2]) >= -
Map.robot_radius and \
                    (triangle_2[0] * x + triangle_2[1] * y + triangle_2[2]) <=
Map.robot_radius and \
                    (triangle_3[0] * x + triangle_3[1] * y + triangle_3[2]) >= -
Map.robot_radius):
                    Map.occGrid[i, j]=2

                    # triangle
                    if ((triangle_1[0] * x + triangle_1[1] * y + triangle_1[2]) >=0 and \
                        (triangle_2[0] * x + triangle_2[1] * y + triangle_2[2]) <=0 and \
                        (triangle_3[0] * x + triangle_3[1] * y + triangle_3[2]) >=0 ):
                        Map.occGrid[i, j]=1
                # Rectangle with clearance
                if ((upperRectangleLine_1[0] * x + upperRectangleLine_1[1] * y +
upperRectangleLine_1[2]) >= -Map.robot_radius and \
```

```
                    (upperRectangleLine_2[0] * x + upperRectangleLine_2[1] * y +
upperRectangleLine_2[2]) <=  Map.robot_radius and \
                    (upperRectangleLine_3[0] * x + upperRectangleLine_3[1] * y +
upperRectangleLine_3[2]) >= -Map.robot_radius):
                    Map.occGrid[i, j]=2
                    # Rectangle
                    if ((upperRectangleLine_1[0] * x + upperRectangleLine_1[1] * y +
upperRectangleLine_1[2]) >= 0 and \
                    (upperRectangleLine_2[0] * x + upperRectangleLine_2[1] * y +
upperRectangleLine_2[2]) <=  0 and \
                    (upperRectangleLine_3[0] * x + upperRectangleLine_3[1] * y +
upperRectangleLine_3[2]) >= 0):
                        Map.occGrid[i, j]=1
                # Rectangle with clearance
                if ((lowerRectangleLine_1[0] * x + lowerRectangleLine_1[1] * y +
lowerRectangleLine_1[2]) >= -Map.robot_radius and \
                    (lowerRectangleLine_2[0] * x + lowerRectangleLine_2[1] * y +
lowerRectangleLine_2[2]) <=  Map.robot_radius and \
                    (lowerRectangleLine_3[0] * x + lowerRectangleLine_3[1] * y +
lowerRectangleLine_3[2]) <= Map.robot_radius):
                        Map.occGrid[i, j]=2
                        # Rectangle
                        if ((lowerRectangleLine_1[0] * x + lowerRectangleLine_1[1] * y +
lowerRectangleLine_1[2]) >= 0 and \
                        (lowerRectangleLine_2[0] * x + lowerRectangleLine_2[1] * y +
lowerRectangleLine_2[2]) <=  0 and \
                        (lowerRectangleLine_3[0] * x + lowerRectangleLine_3[1] * y +
lowerRectangleLine_3[2]) <= 0):
                            Map.occGrid[i, j]=1

                # hexagon with clearance
                if ((hexagonLine_1[0] * x + hexagonLine_1[1] * y + hexagonLine_1[2]) >=  -
Map.robot_radius and \
                    (hexagonLine_2[0] * x + hexagonLine_2[1] * y + hexagonLine_2[2]) >=  -
Map.robot_radius and \
                    (hexagonLine_3[0] * x + hexagonLine_3[1] * y + hexagonLine_3[2]) <=
Map.robot_radius  and \
                    (hexagonLine_4[0] * x + hexagonLine_4[1] * y + hexagonLine_4[2]) <=
Map.robot_radius  and \
                    (hexagonLine_5[0] * x + hexagonLine_5[1] * y + hexagonLine_5[2]) <=
Map.robot_radius and \
                    (hexagonLine_6[0] * x + hexagonLine_6[1] * y + hexagonLine_6[2]) >=  -
Map.robot_radius ):
                    Map.occGrid[i, j]=2

                    # hexagon
                    if ((hexagonLine_1[0] * x + hexagonLine_1[1] * y + hexagonLine_1[2]) >= 0
and \
                        (hexagonLine_2[0] * x + hexagonLine_2[1] * y + hexagonLine_2[2]) >= 0
and \
                        (hexagonLine_3[0] * x + hexagonLine_3[1] * y + hexagonLine_3[2]) <= 0
and \
                        (hexagonLine_4[0] * x + hexagonLine_4[1] * y + hexagonLine_4[2]) <= 0
and \
                        (hexagonLine_5[0] * x + hexagonLine_5[1] * y + hexagonLine_5[2]) <= 0
and \
                        (hexagonLine_6[0] * x + hexagonLine_6[1] * y + hexagonLine_6[2]) >=
0 ):
                        Map.occGrid[i, j]=1

    @classmethod
    def isValid(self, pos):
```

```
        rows, cols = Map.occGrid.shape
        x, y, _ = pos
        j = x
        i = rows - 1 - y
        return Map.occGrid[i, j]==0



class Node:
    def __init__(self, pos=(0, 0, 0), cost2come = 0, cost2go = 0, parent=None):
        self.pos = pos
        self.cost2come = cost2come
        self.cost2go = cost2go
        self.parent = parent

    def __lt__(self, other):
        return self.cost2come + self.cost2go < other.cost2come + other.cost2go

    def __le__(self, other):
        return self.cost2come + self.cost2go <= other.cost2come + other.cost2go

def boundAngle(thetha, d_thetha = 0):
        result = thetha + d_thetha
        if result >= 360:
            result = result - 360
        elif result < 0:
            result = 360 + result

        return result

MAP_RESOLUTION_SCALE = 10
MAP_THRESOLD_REGION = int(0.5 * MAP_RESOLUTION_SCALE)
ANGLE_RESOLUTION = 30 # degree

class A_star:
    def __init__(self, startPos, goalPos, stepSize ):
        self.openList = []
        self.closedList = set()
        self.closedListNodes = []

        self.forVisualization = []

        self.closedNodeMap = np.zeros((251 * MAP_RESOLUTION_SCALE,
                                       601 * MAP_RESOLUTION_SCALE,
                                       360 // ANGLE_RESOLUTION), np.uint8)

        self.stepSize = stepSize
        self.startPos = startPos
        self.goalPos = goalPos


    def addNode(self, node):
        if node != None:
            isNodeSafe = Map.isValid(node.pos)

            if isNodeSafe:
                if not self.isNodeClosed(node):
                    heapq.heappush(self.openList, node)
                    self.forVisualization.append(node)

    def isNodeClosed(self, node):
        # Transform x, y cart coord to w, h image coord
```

```
        rows, cols = Map.occGrid.shape
        x, y, _ = node.pos
        j = x
        i = rows - 1 - y
        return self.closedNodeMap[i * MAP_RESOLUTION_SCALE,
                                  j * MAP_RESOLUTION_SCALE,
                                  boundAngle(node.pos[2]) // ANGLE_RESOLUTION] != 0

    def generateChildNodes(self, node):

        action_sets = [-60, -30, 0, 30, 60]

        branches = []
        for action in action_sets:
            child = self.generateChild(node, action)
            if(child != None):
                branches.append(child)

        return branches

    def generateChild(self, node, action):

        objNode = None
        x, y, originalThetha = node.pos

        thetha = boundAngle(originalThetha, action)
        newX = round(x + self.stepSize * math.cos(math.radians(thetha)))
        newY = round(y + self.stepSize * math.sin(math.radians(thetha)))

        res = (newX, newY, thetha)

        if Map.isValid(res):
            # if res != self.coord:
                #calculating the cost2come by adding the parent and action cost2come
                cost2come = round(self.stepSize + node.cost2come, 3)
                objNode = Node(pos = res,cost2come = cost2come,cost2go =
self.calculateCost2GO(res),parent=node)

        return objNode


    def generatePath(self, node):
        path = []

        while(node.parent != None):
            path.append(node.pos)
            node = node.parent
        path.append(self.startPos)
        path.reverse()

        print("Searched nodes: ", len(self.closedList))
        print("Solution steps: ", len(path))
        return self.forVisualization, path

    def search(self):

        self.addNode(Node(pos = self.startPos,cost2come = 0,cost2go =
self.calculateCost2GO(self.startPos)))

        while(self.openList):

            currNode = heapq.heappop(self.openList)
```

```
            if self.isNodeClosed(currNode):
                continue


            self.closedList.add(currNode.pos)
            self.AddtoClosedNodeMap(currNode)

            if(self.isThisGoalNode(currNode.pos)):
                print("Goal Reached")
                return self.generatePath(currNode)

            branches = self.generateChildNodes(currNode)
            for child in branches:
                self.addNode(child)
        else:
            print("Search failed")
            sys.exit(1)

    def calculateCost2GO(self, pos):
        x,y, _ = pos
        x1,y1, _ = self.goalPos
        return round(math.sqrt((x1 - x)**2 + (y1 - y)**2))

    def AddtoClosedNodeMap(self, node):
        rows, cols = Map.occGrid.shape
        x, y, _ = node.pos
        j = x
        i = rows - 1 - y
        matrix_x = int(i * MAP_RESOLUTION_SCALE - MAP_THRESOLD_REGION)
        matrix_y = int(j * MAP_RESOLUTION_SCALE - MAP_THRESOLD_REGION)
        matrix_degree = boundAngle(node.pos[2]) // ANGLE_RESOLUTION
        self.closedNodeMap[matrix_x, matrix_y, matrix_degree] = 1

        for counter_x in range(1, 11):
            for counter_y in range(1, 11):
                self.closedNodeMap[matrix_x + counter_x ,
                                   matrix_y + counter_y, matrix_degree] = 1

    def isThisGoalNode(self, nodeToCheck):
        xcentre, ycentre, end_theta = self.goalPos
        x, y, node_theta = nodeToCheck
        in_goal = (x - xcentre)**2 + (y -ycentre)**2 - (1.5)**2 < 0
        is_goal = False
        if in_goal:
            if (boundAngle(end_theta) == boundAngle(node_theta)):
                is_goal = True

        return is_goal


def showOccGrid(occGrid):
    rows, cols = occGrid.shape
    color_map = np.zeros((rows, cols, 3))

    color_map[np.where(occGrid == 0)] = np.array([255, 255, 255])
    color_map[np.where(occGrid == 1)] = np.array([0, 0, 0])
    color_map[np.where(occGrid == 2)] = np.array([0, 0, 255])
    color_map[np.where(occGrid == 3)] = np.array([255, 0, 0])
    color_map[np.where(occGrid == 4)] = np.array([0, 255, 0])

    return color_map
```

```
def generateVideo(process , path , goal, occGrid):

    rows, cols = occGrid.shape
    fourcc = cv2.VideoWriter_fourcc('F','M','P','4')
    video = cv2.VideoWriter('TestCase_3.avi', fourcc, float(2000), (601, 251))

    c_x, c_y, _ = goal
    for x in range(c_x-3, c_x+3):
        for y in range(c_y-3, c_y+3):
            if(pow((x-c_x), 2) + pow((y-c_y), 2)) <= pow(3, 2):
                j = x
                i = rows - 1 - y
                occGrid[i, j]=4

    visualizationGrid = occGrid.copy()
    frame = showOccGrid(visualizationGrid)
    initialized = False
    for node in process:
            if node.parent != None:
            # x, y to row col system


                x, y, _ = node.parent.pos
                j = x
                i = rows - 1 - y

                x, y, _ = node.pos
                j1 = x
                i1 = rows - 1 - y

                start = (j,i)
                end = (j1,i1)

                cv2.arrowedLine(frame,start,end,(255,0,0),1)
                video.write(np.uint8(frame))

    initialized = False
    # cv2.imshow("Map", frame)
    path.pop(0)
    # cv2.waitKey(0)
    for pos in path:
        if not initialized:
            x, y, _ = pos
            j = x
            i = rows - 1 - y
            initialized = True
            continue
        x, y, _ = pos
        j1 = x
        i1 = rows - 1 - y

        start = (j,i)
        end = (j1,i1)

        cv2.arrowedLine(frame,start,end,(0,255,0),3)

        j = j1
        i = i1

        video.write(np.uint8(frame))
    cv2.imwrite("TestCase_3.jpg", frame)
```

```
    cv2.imshow("Map", frame)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    video.release()

def getInputs():
    success = False
    while (not success):
        # read input
        print("\nEnter a start and goal point in the map \nNote: map width: 600, map height:
250")

        start_x = int(input("Start x:"))
        start_y = int(input("Start y:"))
        start_theta = int(input("Start theta (in multiples of 30 degrees):"))
        start = (start_x, start_y, start_theta)

        goal_x = int(input("Goal x:"))
        goal_y = int(input("Goal y:"))
        goal_theta = int(input("Goal theta (in multiples of 30 degrees):"))
        goal = (goal_x, goal_y,goal_theta )

        stepSize = int(input("Enter step isze between 1 - 10:"))

        print("Start pos: ({}, {})".format(start_x, start_y))
        print("Goal pos: ({}, {})".format(goal_x, goal_y))




        success = True

        if (not Map.isValid(start)):
            print("This start point is not valid.")
            success = False
        if ((start_theta % 30) != 0):
            print("This start orientation is not valid.")
            success = False
        if (not Map.isValid(goal)):
            print("This goal point is not valid.")
            success = False
        if ((goal_theta % 30) != 0):
            print("This goal orientation is not valid.")
            success = False
        if stepSize > 11 or stepSize < 1:
            print("This step size is not valid.")
            success = False
        if (success == False):
              print("Please re-enter targets")


    return start, goal , stepSize

startTime = time.time()
Map.generateOccGrid()
# cv2.imshow("Map", showOccGrid(Map.occGrid))
# cv2.waitKey(0)
# cv2.destroyAllWindows()
start , goal , stepSize = getInputs()
graph = A_star(start, goal, stepSize)
process, path = graph.search()
intermediateTime = time.time()
```

```
print("Algorithm Execution time:", intermediateTime - startTime, "seconds")
generateVideo(process , path , goal, Map.occGrid)
endTime = time.time()
print("Rendering time:",endTime -  intermediateTime, "seconds")
```