```python
#Imports
#Libraries
import numpy as np
import cv2
from tqdm import tqdm
#Inbuilt modules
from queue import PriorityQueue as pq
from ordered_set import OrderedSet
import copy
import typing
import time

## Constants
# y direction is a row # x direction is a column. Operations are y,x or row,column
print("\r\nGENERATING OBSTACLE MAP")
YBOUND = range(5,245,1) # Padding of 5mm on each dimension
XBOUND = range(5,595,1) # Padding of 5mm on each dimension
ACTIONS = {"U":(+1,0),"D":(-1,0),"L":(0,-1),"R":(0,+1),"UL":(+1,-1),"UR":(+1,+1),"DR":(-1,+1),"DL":(-1,-1)}
DIAGONAL_COST = 1.4
SIDEWAY_COST = 1.0
COSTFORACTION = {"U":SIDEWAY_COST,"D":SIDEWAY_COST,"L":SIDEWAY_COST,"R":SIDEWAY_COST,"UL":DIAGONAL_COST,"UR":DIAGONAL_COST,"DR":DIAGONAL_COST,"DL":DIAGONAL_COST}
OBSTACLE_COLOR = [255,0,0]
obstacle_image = np.full((YBOUND[-1]+5+1,XBOUND[-1]+5+1,3),125,dtype=np.uint8)
# Define obstacles
# Define Rectangles
def rectangle1(pixelCoordinate):
    if pixelCoordinate[0] > (100-5) and pixelCoordinate[0]<(150+5) and pixelCoordinate[1]<(100+5):
        return False
    else:
        return True

def rectangle2(pixelCoordinates):
    if pixelCoordinates[0] > (100-5) and pixelCoordinates[0]<(150+5) and pixelCoordinates[1]>(150-5):
        return False
    else:
        return True

# Define Hexagon
hex_actual_vertex = [[235,162.5],[300,200],[365,162.5],[365,87.5],[300,50],[235,87.5]]
Hex_Padded     = [[230.7,165],[300,205],[369.3,165],[369.3,85],[300,45],[230.7,85]]
line1 = np.polyfit([Hex_Padded[0][0],Hex_Padded[1][0]] , [Hex_Padded[0][1],Hex_Padded[1][1]] , 1)
line2 = np.polyfit([Hex_Padded[1][0],Hex_Padded[2][0]] , [Hex_Padded[1][1],Hex_Padded[2][1]] , 1)
line3 = np.polyfit([Hex_Padded[3][0],Hex_Padded[4][0]] , [Hex_Padded[3][1],Hex_Padded[4][1]] , 1)
line4 = np.polyfit([Hex_Padded[4][0],Hex_Padded[5][0]] , [Hex_Padded[4][1],Hex_Padded[5][1]] , 1)

def hexagon(pixelCoordinate):
    line1Cond = pixelCoordinate[1] - line1[0]*pixelCoordinate[0] - line1[1] <0
    line2Cond = pixelCoordinate[1] - line2[0]*pixelCoordinate[0] - line2[1] <0
    line3Cond = pixelCoordinate[1] - line3[0]*pixelCoordinate[0] - line3[1] >0
    line4Cond = pixelCoordinate[1] - line4[0]*pixelCoordinate[0] - line4[1] >0
    if line1Cond and line2Cond and line3Cond and line4Cond and pixelCoordinate[0]>230.7 and pixelCoordinate[0]<369.3:
        return False
    else:
        return True

# Define Triangle
triangle_actual = [[460,225],[510,125],[460,25]]
triangle_padded     = [[455,238],[517,125],[455,10]]
side1 = np.polyfit([triangle_padded[0][0],triangle_padded[1][0]] , [triangle_padded[0][1],triangle_padded[1][1]] , 1)
side2 = np.polyfit([triangle_padded[1][0],triangle_padded[2][0]] , [triangle_padded[1][1],triangle_padded[2][1]] , 1)

def triangle(pixelCoordinates):
    side1Cond = pixelCoordinates[1] - side1[0]*pixelCoordinates[0] - side1[1] <0
    side2Cond = pixelCoordinates[1] - side2[0]*pixelCoordinates[0] - side2[1] >0
    if side1Cond and side2Cond and pixelCoordinates[0]>455:
        return False
    else:
        return True

def npObstacleMap(image):
    for y in range(image.shape[0]):
        for x in range(image.shape[1]):
            if (not triangle((x,y))) or (not rectangle1((x,y))) or (not rectangle2((x,y))) or (not hexagon((x,y))):
                image[y,x] = OBSTACLE_COLOR
    return image

OBSTACLE_MAP = npObstacleMap(obstacle_image)
print("\r\nFINISHED GENERATING OBSTACLE MAP")
#Node data structure
class GraphNode:

    #Constructor Data:Tuple of(y,x) or (row,column) data
    def __init__(self, data,parent,id:int,cost=0,level=0):
        self.DATA = (data)
        self.children = []
        self.parent = parent
        self.ID = id
        self.cost = cost
        self.LEVEL = level

    #Getter for this node's parent
    def get_parent(self):
        return self.parent

    #Generate children according to pre-defined actions
    def generate_children(self):
        curr_y,curr_x= self.DATA
        #For each action mentioned in actions, check if a action is valid, and if it is, insert it in the children's list
        newId = int(self.ID)
        newLevel = self.LEVEL+1
        for [key,value] in ACTIONS.items():
            dy,dx = value
            newy,newx = curr_y+dy,curr_x+dx
            #TODO(ADD a check for obstacle intersection
            if((newy in YBOUND) and ( newx in XBOUND)) and (OBSTACLE_MAP[newy][newx][0]!=OBSTACLE_COLOR[0]) and (OBSTACLE_MAP[newy][newx][1]!=OBSTACLE_COLOR[1]) and (OBSTACLE_MAP[newy][newx][2]!=OBSTACL
                newId+=1
                newCost = self.cost+COSTFORACTION[key]
                self.children.append(GraphNode((newy,newx),self,newId,newCost,newLevel))

    #Getter for children
    def get_children(self):
        return self.children

    #Setter for children
    def set_children(self,children):
        self.children = copy.deepcopy(children)
        for child in self.children:
            child.parent = self
        return

    #Override for < operator
    def __lt__(self, other):
        return self.cost < other.cost

    #Override for == operator
    def __eq__(self, other):
        if other is None:
            return False
        return self.DATA==other.DATA

    def getSelfCost(self):
```

```
    def getSelfCost(self):
        return self.cost

    def setCost(self,cost):
        self.cost = cost

    #Override for hashing this type
    def __hash__(self):
        b,a =  self.DATA
        return hash((a << 32) + b)

#utility linear search function , looks for specific node in the queue
def checkForChildInQueue(child,queue)->GraphNode:
    for elem in queue.queue:
        if elem == child:
            return elem

#Main Dikstra Function
def dikstra(startGoal,endGoal):
    #Ensure state and end goal are uint8s
    startGoal = (startGoal)
    endGoal = (endGoal)
    #Make an ordered set to save visited Nodes. This is to be used in the BFS algorithm
    visited = OrderedSet()
    #Make a PRIORITY Queue to save nodes that is to be visited next.
    toBeVisited = pq()
    #Initiate Root node from startGoal
    node1 = GraphNode(startGoal,None,0,0)
    #Initial Q with root node
    toBeVisited.put(node1)
    #Initiate visitedNodes Counter
    visitedNodesCount = 1
    #djikstra logic, Run this loop until we have an empty node
    while not toBeVisited.empty():
        #Pop node to be visited out of the Queue
        node = toBeVisited.get()
        # Add the node in the visited set, if it already exists, a new node is not added in a set
        visited.add(node)
        #Check if goal is met, if it is, return the node and VisitedNodesList
        if node.DATA==endGoal:
            print('\r\nVISITED NODE COUNTS:{}'.format(visitedNodesCount))
            return node,visited
        #If the goal is not met, generate children of the node
        node.generate_children()
        #Iterate over childen
        for child in node.get_children():
            # If the child is not visited, add it in the toBeVisited Queue
            #if X not in closed list (generate children already handles actions which are invalid)
            if child not in visited:
                if child not in (toBeVisited.queue):
                    #Cost calculation is already handled inside generate_children
                    toBeVisited.put(child)
                    # Find current length of set
                    setLength = len(visited)
                    # Add this child in the visited set, as it will be visited in the next iteration of while loop
                    visited.add(child)
                    # If the visited set length has changed, that means we have a unique member which will be visited next
                    if len(visited) != setLength:
                        # Raise the visitedNodesCount
                        visitedNodesCount+=1
                else:
                    queueItem = checkForChildInQueue(child,toBeVisited)
                    if queueItem.cost > child.cost:
                        toBeVisited.queue.remove(queueItem)
                    toBeVisited.put(child)
                    pass
    #Return None in case of no solution found
    return None,None

#This is basic linked list traversal algorithm
#for every node, store that node in a list, and replace node by node.parent
def backTrack(inputNode:GraphNode):
    if(inputNode is None):
        return []
    path = []
    thisNode = inputNode
    while True:
        if thisNode != None:
            path.append(thisNode)
        if thisNode.get_parent() is None:
            break
        thisNode = thisNode.get_parent()
    path.reverse()
    print('parent COST:{} ,end COST:{}'.format(path[0].cost,path[-1].cost))
    return path

#Execute Djikstra with debug prints, and save files
def dikPrintReversePath(start,end,printPath:bool):
    start2 = copy.deepcopy(start)
    end2 = copy.deepcopy(end)
    start2=(249-start2[0],start2[1])
    end2=(249-end2[0],end2[1])
    print('START:\r\n{}'.format(start2))
    print('Expected END:\r\n{}'.format(end2))
    result,visitedNodes = dikstra(start,end)
    if result is None:
        print("Unable to find result")
        return
    print('\r\nFOUND A SOLUTION \r\n')
    res = copy.deepcopy(result.DATA)
    res= (249-res[0],res[1])
    print('Result:\r\n{}'.format(res))
    back = backTrack(result)
    print('\r\nSTEPS:\r\n{}'.format(len(back)-1))
    if printPath:
        print("PATH :")
        for i in back:
            print(i.DATA)
    return back,visitedNodes

#find visitedNotesAtEachInstanceOfSolutionPath
def findVisitedNotesPerFrame(path,visited:OrderedSet):
    visitedNodesPerFrame = []
    for point in path:
        visited_array=[]
        for node in visited:
            if (node.ID <= point.ID):
                visited_array.append(node.DATA)
        visitedNodesPerFrame.append(visited_array)
    return visitedNodesPerFrame

# Visualize Path and obstacles
def vizPath(empty_images,path):
    obstacle_color = (255,0,0)
    empty_images2 = np.full((len(empty_images)+5,250,600,3),125,dtype=np.uint8)
    #make the background common
    for idx,image in enumerate(empty_images2):
        empty_images2[idx] = empty_images[-1]
    #draw the path
    #for this, find path
    path_pts = []
    #find path
    #find path
    for idx,node in enumerate(path):
        path_pts.append(node.DATA)
    # For image in empty_images2 , draw path
    # Marks path
    for image in empty_images2:
        for data in path_pts:
            y,x = data
            image =cv2.circle(image, (x,y), 1, (0,0,255),1)
    empty_images = np.concatenate((empty_images,empty_images2),axis=0)
    for idx,image in enumerate(empty_images):
        #Rectangle 1:
        empty_images[idx] = cv2.rectangle(empty_images[idx], (99,0) , (149,99), obstacle_color ,  -1)
        #Rectangle 2:
        empty_images[idx] = cv2.rectangle(empty_images[idx], (99,149) , (149,249), obstacle_color ,  -1)
        #Triangle 1:
        triangle_corners = [(460-1, int(25-1)), (460-1, int(225-1)), (int(510-1), 125-1)]
        empty_images[idx] = cv2.fillPoly(empty_images[idx], np.array([triangle_corners]), obstacle_color)
        #Hexagon 1:
        hex_corners = [(235-1, 163-1),(300-1,200-1),(365-1,163-1),(365-1,88-1),(300-1,50-1),(235-1,88-1)]
        empty_images[idx] = cv2.fillPoly(empty_images[idx], np.array([hex_corners]), obstacle_color)
    for idx,node in enumerate(path):
        y,x = node.DATA
        #Mark Node position by a circle
        empty_images[idx+5+len(path)] = cv2.circle(empty_images[idx+5+len(path)], (x,y), 4, (0,0,255),-1)
    return empty_images

#Explored color == GREEN
def vizExplore(visitedNodesPerFrame,path):
    empty_images = np.full((len(path),250,600,3),125,dtype=np.uint8)
    for frame,nodes in zip(empty_images,visitedNodesPerFrame):
        for node in nodes:
            y,x = node
            frame[y][x] = [0,255,0]
    for idx,frame in enumerate(empty_images,1):
        color = np.array([0, 255, 0])
        indices = np.where(np.all(empty_images[idx-1] == color, axis=-1))
        frame[y,x] = [0,255,0]
    return empty_images

## Runs everything, saves a video
def djikstraViz(start,end,input_num=0):
    if start[0] not in YBOUND or start[1] not in XBOUND:
        print("START point outside of bounds")
        return
    if end[0] not in YBOUND or end[1] not in XBOUND:
        print("END point outside of bounds")
        return
    if np.array_equal(OBSTACLE_MAP[start[0],start[1]],np.array(OBSTACLE_COLOR)) or np.array_equal(OBSTACLE_MAP[end[0],end[1]],np.array(OBSTACLE_COLOR)):
        print("START OR GOAL POINT INSIDE OBSTACLE SPACE")
        return
    before  = time.time()
    path,visitedNodes = dikPrintReversePath(start,end,False)
    print("\r\nTIME FOR DJIKSTRA SOLN:{}".format(time.time()-before))
    if path is None or visitedNodes is None:
        print("\r\n NO OUTPUT GENERATED \r\n")
        return
    print("\r\n STARTED VISUALIZATION \r\n")
    before  = time.time()
    visitedNodesPerFrame = findVisitedNotesPerFrame(path,visitedNodes)
    assert len(visitedNodesPerFrame) == len(path)
    viz = vizExplore(visitedNodesPerFrame,path)
    pathViz = vizPath(viz,path)

    size = (pathViz[0].shape[1],pathViz[0].shape[0])
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    voObj = cv2.VideoWriter('./viz/PathViz'+str(input_num)+'.mp4',fourcc, 15,size)

    for frame in tqdm(pathViz):
        image = frame
        voObj.write(image)
    voObj.release()
    print("\r\nTIME FOR VISUALIZATION OUTPUT SOLN:{}".format(time.time()-before))
    print("\r\nFINISHED GENERATING OUTPUT VIDEO at ./viz/ \r\n")
    return

while True:
    startY = int(input('Start Point Y(Row) coordinate:'))
    startX = int(input('Start Point X(Column) coordinate:'))
    endY = int(input('End Point Y(Row) coordinate:'))
    endX = int(input('End Point X(Column) coordinate:'))
    #Invert axis
    startY = 249 - startY
    endY = 249 - endY
    break
djikstraViz((startY,startX),(endY,endX))
```