Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Algoritmi

Seminarska naloga 2

Avtor:
Erik Kristian Janežič
Vpisna št.:
63160390

Contents

1 Uvod

1.1 Opredelitev problema

V drugi seminarski nalogi smo raziskali problem iskanja presečišč med segmenti sestavljenimi iz lomljenih daljic. Osnovni problem iskanja presečiš med daljicami in njegove izpeljanke z dodatnimi pogoji (kaj še štejemo kot presečišče in kaj ne) je možno trivialno rešiti tako, da za vsako daljico preverimo če se seka s katerokoli od ostalih daljic. Čepravj je tovrstni algoritem pravilen, je njegova časovna zahtevnost exponentna ($O(n^2)$. Naš cilj je implementacija algoritma s časovno zahtevnostjo $O(nlog(n)$. Pristop, ki smo se ga poslužili je implementacija Bentley–Ottmann oz. "Sweep line" algoritma za iskanje presečišč
Problem, ki smo ga reševali je zaradi dodatnih zahtev nekoliko zahtevnejši kot osnovni problem iskanja presečišč med daljicami. Natančen opis problema:

- **Vhodni parameter algoritma** - tekstovna datoteka, kjer je v vsaki vrstici zapisana koordinata ene točke ter njena barva. Zaporedne točke tvorijo daljice neke barve, zaporedne daljice iste barve pa tvorijo segmente te barve. Vse točke, ki pripadajo nekemu segmentu so navedene zaporedno brez vmesnih točk druge barve
- **Izhod algoritma** - Število presečiščč med segmenti različnih barv
- **Zahteve**
  - Daljici, različne barve, ki se stikata se štejeta za presečišče.
  - Daljici, ki se prekrivata, štejete kot eno presečišče.
  - Več daljic se lahko seka v eni točki.
  - Navpičnih daljic v testnih podatkih ni
  - Vse lomljene črte so enake dolžine in vsaka je različne barve.
  - Vse daljice bodo znotraj kvadrata [0,0] do [1000, 1000].

2 Algoritem

2.1 Osnove in podatkovne strukture

Algoritem je implementiran v jeziku Python in je odvisen od osnovnih knjižnic *sys* in *heapq*
Ključna ideja Bentley-Ottmann algoritma je opažanje, da tik preden se dve daljici sekata

morata biti med seboj sosedi, kar pomeni, da med njima ni nobene druge daljice. To pomeni da je potrebno za posamezno daljico voditi evidenco katara daljica je njena spodnja in zgornja soseda. V ta namen si predstavljamo navidezno navpično premico (SL), ki potuje v smeri naraščanja X koordinate. SL je implementirana kot prioritetna vrsta dogodkov, kjer je dogodek točka z X in Y koordinato in pripada enemu izmed treh tipov dogodkov:

1. **START** - V tej točki se začne daljica
2. **END** - V tej točki se neka daljica konča
3. **INTERSECT** - V tej točki se dve daljici sekata

Dogodki so v prioritetni vrsti razvrščeni po naraščajoči X koordinati, kar pomeni da je na začetku vrste zmeraj dogodek z najmanjšo X koordinato in to je prav dogodek na katerega bo v naslednjem trenutku naletela SL. Ključnega pomena za prioritetno vrsto je, da imajo operacije dodajanja dogodkov in odvzemanja dogodkov v in z vrste časovno zahtevnost največ $O(logn)$, ker med potekom algoritma v vrsto ves čas dodajamo in odvzemamo elemente. Druga ključna podatkovna struktura, ki omogoča delovanje algoritma v $O(nlogn)$ časovni zahtevnosti je AVL binarno iskalno drevo (TR), ki hrani informacijo o tem kako si daljice, ki jih v danem trenutku seka SL, sledijo glede na Y koordinato pri X kjer se trenutno nahaja SL. Lastnost AVL samo uravnoteževalnega iskalnega drevesa je da so operacije dodajanja, odvzemanja in iskanja elementov mogoče v $O(logn)$, kar je središčnega pomena za učinkovitost našega algoritma. V vozliščih iskalnega drevesa hranimo koeficient naraščanja in presečišče z Y osjo (k_i, m_i) daljice, ki se trenutno nahaja v tem vozlišču. k_i in m_i natančno definirata daljico in omogočata dinamično računanje Y koordinate v različnih pozicijah SL. Pri vsakem vstavljanju in iskanju po TR k_i in m_i v kombinaciji s trnutno X koordinato vodita iskanje v pravo smer.
Pri procesiranju dogodkov izvedeemo različen set operacij glede na to kateri tip dogodka procesiramo:

- **START** - Dodamo daljico v TR, najdemo spodnjeg ain zgornjega soseda (glede na Y koordinato) in preverimo, če ima daljica presečišče s katerim izmed njiju. Če presečišče najdemo dodamo nov dogodek, s točko presečišča v prioritetno vrsto.
- **END** - Poiščemo oba najbljižja soseda daljice, ki se v tej točki končuje. Preverimo če se soseda med seboj sekata, če se dodamo nov dogodek s točko presečišča v prioritetno vrsto. Izbrišemo daljico, ki se je v tej točki končala iz prioritetnega drevesa.
- **INTERSECT** - Preverimo, če smo že kdaj prej obdelali presečišče daljic $D1$ in $D2$. Če smo ga preskočimo ta korak in nadaljujemo z iteracijo. Daljici, ki se sekata sta druga drugi sosedi. Za vsako daljico poiščemo še drugega soseda (Soseda $D1$: $D2$ in $S1$; Soseda $D2$: $D1$ in $S2$). Preverimo če obstaja presečišče med $D2$ in $S1$ ter $D1$ in $S2$, če presečisšče najdemo ga dodamo v prioritetno vrsto. Presečišče $D1$ in $D2$ zaznačimo kot "sprocesirano" in zabeležimo presečišče v listo presečišč. V TR zamenjamo pozicije $D1$ in $D2$, tako da je $D1$ na mestu $D2$ in $D2$ na mestu $D1$.

2.2 Potek

Grobi potek našega algoritma je sledeči:

1. Preberemo vhodno datoteko in ustvarimo objekte: točke, daljice in segmente

2. Iz liste točk ustvarimo prioritetno vrsto (SL) izvornih dogodkov, ki so začetki in konci daljic
3. Inicializiramo prazno AVL drevo (TR)
4. Iteracija sledečih korakov dokler prioritetna vrsta ni prazna:
   (a) Vzemi prvi dogodek iz prioritetne vrste
   (b) Izvedi zaporedje operacij glede na to katere vrste dogodek je *END*,*START* ali *INTERSECT*(opisano v prejšnjem poglavju)
5. Ko je vrsta prazna izstopimo iz zanke in filtriramo zbrana presečiča, zato da izločimo presečišča med daljicami istih barv

2.3 Analiza

2.3.1 Časovna analiza

Časovna zahtevnost algoritma je odvisna od števila vhodnih točk ter števila najdenih presečišč. Skupno časovno zahtevnost ocenimo po naslednji dedukciji:

- Branje vhodne datoteke: Sprehoditi se moramo preko vseh vrstic v datoteki, kar je enako $O(n)$
- Ustvarjanje prioritetne vrste: $O(nlogn)$ naredimo enkrat na začetku. n je število točk, ki definerajo daljice
- Dodajanje elementov v prioritetno vrsto: $O(logn)$ to storimo k-krat kjer je k število najdenih presečišč
- Odvzemanje elementov iz prioritetne vrste: $O(logn)$ kar storimo $n + k$-krat iz vrste moramo namreč do zaključka algoritma odstraniti vse elemente.
- Vse operacije (dodajanje, odvzemanje, zamenjava, iskanje) nad vozlišči v AVL drevesu imajo časovno zahtevnost $O(logn)$ ker moramo sprocesirati vse točke je skupna časovna zahtevnost operacij v AVL drevesu $O(nlogn)$

Končna časovna zahtevnost algoritma je tako $O((n + k)logn)$ kejr je n število točk, ki definirajo daljicec, k pa je število najdenih presečišč

2.3.2 Prostorska analiza

Prostorsko zahtevnost algoritma predstavlja število elementov shranjenih v AVL drevesu ter prioritetni vrsti. V AVL drevesu imamo shranjenih največ $n/2$ daljic, kjer je n število točk ki definirajo daljice. V Prioritetni vrsti pa je shranjenih največ $n + k$ elementov, kjer je k število najdenih presečišč. Prostorsko zahtevnost je tako ocenjena na $O(n + k)$, odvisna pa je tudi od velikosti objektov, ki jih shranjujemo.

2.4 Programska koda

```
# -*- coding: utf-8 -*-
import sys
import heapq

DELETED = set()


#################### DEFINITION OF CLASSES
    #####################################
class Event:
    eventCounter = 0
    def __init__(self):
        self.start = False
        self.end = False
        self.intersect = []
        self.id = Event.eventCounter
        Event.eventCounter += 1


    def __repr__(self):
        s = "ID: "+str(self.id)+" Start: " + str(self.start) + "\n" + "
            End: " + str(self.end) + "\n" + "Intersect: " + str(self.
            intersect) + "\n"
        return s

class Point:
    pointCounter = 0
    def __init__(self,x,y,color=0):
        self.x = x
        self.y = y
        self.color = color
        self.id = Point.pointCounter
        Point.pointCounter += 1
        self.partOf = set()
        self.events = []

    def assignToLine(self,line):
        self.partOf.add(line)

    def __repr__(self):
        return "( "+ "ID: " + str(self.id) +" " + str(self.x) + ", " +
            str(self.y) + ", " + str(self. color) + " )"

    def __lt__(self,other):
        return self.x < other.x

class Line:
    def __init__(self,point1,point2):
        self.p1 = point1
        self.p2 = point2
```

```
        self.color = self._setColor()
        self.start = self._setStart()
        self.end = self._setEnd()
        self.lineEquation = self._setEquation()

    def _setColor(self):
        try:
            if self.p1.color == self.p2.color:
                return self.p1.color
            else:
                raise ValueError(" Point collors are not the same")
        except(ValueError,IndexError):
            exit("Could not produce right lines. Check code")

    def _setStart(self):
        if self.p1.x < self.p2.x:
            return self.p1
        else:
            return self.p2

    def _setEnd(self):
        if self.p1.x > self.p2.x:
            return self.p1
        else:
            return self.p2
    def _setEquation(self):
        x1 = self.start.x
        y1 = self.start.y
        x2 = self.end.x
        y2 = self.end.y

        k = (y2-y1)/(x2-x1)

        yInter = y1 -k*x1

        return k,yInter

    def __repr__(self):
        return "[" + str(self.start) + "-->" + str(self.end) + "]"

class Segment:
    def __init__(self,lineList, color):
        self.lines = lineList
        self.color = color

    def __repr__(self):
        return "{" + str(self.lines) + "}"

#----------------------------AVL TREE
    ------------------------------------------------------#
class node:
    def __init__(self,line=None):
```

```
        self.line = line
        self.value=self.line.lineEquation
        self.left_child=None
        self.right_child=None
        self.parent=None # pointer to parent node in tree
        self.height=1 # height of node in tree (max dist. to leaf) NEW
           FOR AVL
#      self.isChild = None


   def __repr__(self):
      return str(self.line)




class AVLTree:
   def __init__(self):
         self.root=None

   def __repr__(self):
         if self.root==None: return ''
         content='\n' # to hold final string
         cur_nodes=[self.root] # all nodes at current level
         cur_height=self.root.height # height of nodes at current level
         sep=' '*(2**(cur_height-1)) # variable sized separator between
             elements
         while True:
            cur_height+=-1 # decrement current height
            if len(cur_nodes)==0: break
            cur_row=' '
            next_row=''
            next_nodes=[]

            if all(n is None for n in cur_nodes):
               break

            for n in cur_nodes:

               if n==None:
                  cur_row+='   '+sep
                  next_row+='  '+sep
                  next_nodes.extend([None,None])
                  continue

               if n.value!=None:
                  buf=' '*int((5-len(str(n.line)))/2)
                  cur_row+='%s%s%s'%(buf,str(n.line),buf)+sep
               else:
                  cur_row+=' '*5+sep
```

```
            if n.left_child!=None:
               next_nodes.append(n.left_child)
               next_row+=' /'+sep
            else:
               next_row+=' '+sep
               next_nodes.append(None)

            if n.right_child!=None:
               next_nodes.append(n.right_child)
               next_row+='\ '+sep
            else:
               next_row+=' '+sep
               next_nodes.append(None)

         content+=(cur_height*' '+cur_row+'\n'+cur_height*' '+
            next_row+'\n')
         cur_nodes=next_nodes
         sep=' '*int(len(sep)/2) # cut separator size in half
      return content

def isHigherSamePoint(self,line1,line2):
   if line1.p1 == line2.p1:
      if line1.p2.y > line2.p2.y:
         return True
      else:
         return False
   elif line1.p1 == line2.p2:
      if line1.p2.y > line2.p1.y:
         return True
      else:
         return False
   elif line1.p2 == line2.p1:
      if line1.p1.y > line2.p2.y:
         return True
      else:
         return False
   elif line1.p2 == line2.p2:
      if line1.p1.y > line2.p1.y:
         return True
      else:
         return False
   else:
      #print("Colinear")
      return "Colinear"

def calcValue(self,line,cur_x):
   if cur_x == line.start.x:
      return line.start.y
   elif cur_x == line.end.x:
      return line.end.y
   else:
      y = line.lineEquation[0]*cur_x + line.lineEquation[1]
```

```
            return y

        def switchCrossedSegments(self,node1,node2,cur_x):
            assert isinstance(node1,node)
            assert isinstance(node2,node)
#           #print("agfasdg",node2.isChild)
            #print("ROOOT:",self.root)
            #print("__________BEFORE_______")
            #print("NODE1")
            #print(node1)
            #print("Parent",node1.parent)
            #print("L Child",node1.left_child)
            #print("R Child",node1.right_child)
            #print("Height",node1.height)
            #print("NODE2")
            #print(node2)
            #print("Parent",node2.parent)
            #print("L Child",node2.left_child)
            #print("R Child",node2.right_child)
            #print("Height",node2.height)
#           #print("Is Child",node1.isChild)
            if node1.parent == None:
                self.root = node2
            if node2.parent == None:
                self.root = node1

            if node1.parent == node2:
                if node2.left_child == node1:
                    node1.parent = node2.parent
                    node2.parent = node1
                    node2.right_child,node1.right_child = node1.right_child,
                        node2.right_child
                    node2.left_child = node1.left_child
                    node1.left_child = node2
                    node1.height, node2.height = node2.height, node1.height
                else:#node2.right_child == node1:
                    node1.parent = node2.parent
                    node2.parent = node1
                    node2.left_child,node1.left_child = node1.left_child,node2
                        .left_child
                    node2.right_child = node1.right_child
                    node1.right_child = node2
                    node1.height, node2.height = node2.height, node1.height
            elif node2.parent == node1:
                if node1.left_child == node2:
                    node2.parent = node1.parent
                    node1.parent = node2
                    node2.right_child,node1.right_child = node1.right_child,
                        node2.right_child
                    node1.left_child = node2.left_child
                    node2.left_child = node1
                    node1.height, node2.height = node2.height, node1.height
```

```
            else:#node1.right_child == node2:
                node2.parent = node1.parent
                node1.parent = node2
                node2.left_child,node1.left_child = node1.left_child,node2
                    .left_child
                node1.right_child = node2.right_child
                node2.right_child = node1
                node1.height, node2.height = node2.height, node1.height
        else:
            node1.parent,node2.parent = node2.parent, node1.parent
            node1.left_child,node2.left_child = node2.left_child, node1.
                left_child
            node1.right_child,node2.right_child = node2.right_child,node1
                .right_child
            node1.height,node2.height = node2.height, node1.height
#        node1.isChild, node2.isChild = node2.isChild, node1.isChild
        #print("ROOOT:",self.root)
        #print("__________AFTER_______")
        #print("NODE1")
        #print(node1)
        #print("Parent",node1.parent)
        #print("L Child",node1.left_child)
        #print("R Child",node1.right_child)
        #print("Height",node1.height)
        #print("NODE2")
        #print(node2)
        #print("Parent",node2.parent)
        #print("L Child",node2.left_child)
        #print("R Child",node2.right_child)
        #print("Height",node2.height)
#        #print("Is Child",node1.isChild)

        # Reseting parents for all the childs
        if node1.left_child:
            node1.left_child.parent =node1
        if node1.right_child:
            node1.right_child.parent =node1
        if node2.left_child:
            node2.left_child.parent =node2
        if node2.right_child:
            node2.right_child.parent =node2

        #Reseting the childs for all the parents
        if node1.parent == node2.parent:
            if node1.parent.left_child == node1:
                node1.parent.left_child = node2
                node1.parent.right_child = node1
            else:
                node1.parent.right_child = node2
                node1.parent.left_child = node1

        elif node1.parent == node2:
```

```
            if node2.parent:
                if node2.parent.left_child == node1:
                    node2.parent.left_child = node2
                else:
                    node2.parent.right_child = node2
        elif node2.parent == node1:
            if node1.parent:
                if node1.parent.left_child == node2:
                    node1.parent.left_child = node1
                else:
                    node1.parent.right_child = node1

        else:
            if node1.parent:
                if node1.parent.left_child == node2:
                    node1.parent.left_child = node1
                else:
                    node1.parent.right_child = node1
            if node2.parent:
                if node2.parent.left_child == node1:
                    node2.parent.left_child = node2
                else:
                    node2.parent.right_child = node2



#        if node1.parent == node2:
#            if node2.parent:
#                if node2.parent.left_child == node1:
#                    node2.parent.left_child = node2
#                else:
#                    node2.parent.right_child = node2
#            if node2.left_child:
#                node2.left_child.parent = node2
#            if node2.right_child:
#                node2.right_child.parent = node2
#
#            if node1.left_child:
#                node1.left_child.parent = node1
#            if node1.right_child:
#                node1.right_child.parent = node1
#
#
#        elif node2.parent == node1:
#            if node1.parent:
#                if node1.parent.left_child == node2:
#                    node1.parent.left_child = node1
#                else:
#                    node1.parent.right_child = node1
#            if node1.left_child:
#                node1.left_child.parent = node1
#            if node1.right_child:
```

```
#             node1.right_child.parent = node1
#          if node2.left_child:
#             node2.left_child.parent = node2
#          if node2.right_child:
#             node2.right_child.parent = node2
#
#       else:
#          if node1.parent:
#             if node1.parent.left_child == node2:
#                node1.parent.left_child = node1
#             else:
#                node1.parent.right_child = node1
#          if node1.left_child:
#             node1.left_child.parent = node1
#          if node1.right_child:
#             node1.right_child.parent = node1
#
#          if node2.parent:
#             if node2.parent.left_child == node1:
#                node2.parent.left_child = node2
#             else:
#                node2.parent.right_child = node2
#          if node2.left_child:
#             node2.left_child.parent = node2
#          if node2.right_child:
#             node2.right_child.parent = node2

#       parent1 = node1.parent
#       lc1 = node1.left_child
#       rc1 = node1.right_child
#       height1 = node1.height
#       LR1 = node1.isChild
#
#       parent2 = node2.parent
#       lc2 = node2.left_child
#       rc2 = node2.right_child
#       height2 = node2.height
#       LR2 = node2.isChild




#       node1.height,node1.parent,node1.left_child,node1.right_child,
   node2.height,node2.parent,node2.left_child,node2.right_child =node2.
   height,node2.parent,node2.left_child,node2.right_child,node1.height,
   node1.parent,node1.left_child,node1.right_child

#       a = node1
#       b = node2 #self.find(line2,cur_x)
#
```

```
#       auxLine = a.line
#       auxVal = a.value
#
#       a.value = b.value
#       a.line = b.line
#
#       b.value = auxVal
#       b.line = auxLine


    def insert(self,line,cur_x):
          if self.root==None:
             self.root=node(line)
          else:
             self._insert(line,cur_x,self.root)

    def _insert(self,line,cur_x,cur_node):
          if self.calcValue(line,cur_x) < self.calcValue(cur_node.line,
             cur_x):
             #print("Insert via <:","X:",cur_x,"inserting:",self.
                calcValue(line,cur_x),line,"comp:",self.calcValue(
                cur_node.line,cur_x),cur_node.line)
             if cur_node.left_child==None:
                cur_node.left_child=node(line)
                cur_node.left_child.parent=cur_node # set parent
#                cur_node.left_child.isChild = "LEFT"
                self._inspect_insertion(cur_node.left_child)
             else:
                self._insert(line,cur_x,cur_node.left_child)
          elif self.calcValue(line,cur_x) > self.calcValue(cur_node.line
             ,cur_x):
             #print("Insert via >:","X:",cur_x,"inserting:",self.
                calcValue(line,cur_x),line,"comp:",self.calcValue(
                cur_node.line,cur_x),cur_node.line)
             if cur_node.right_child==None:
                cur_node.right_child=node(line)
                cur_node.right_child.parent=cur_node # set parent
#                cur_node.right_child.isChild = "RIGHT"
                self._inspect_insertion(cur_node.right_child)
             else:
                self._insert(line,cur_x,cur_node.right_child)
          elif self.calcValue(line,cur_x) == self.calcValue(cur_node.
             line,cur_x):
             #print("Inserting EQ:",line.lineEquation[0],"Current EQ: ",
                cur_node.line.lineEquation[0])
             if self.isHigherSamePoint(line,cur_node.line):# line.
                lineEquation[0] > cur_node.line.lineEquation[0]:

                if cur_node.right_child==None:
                   cur_node.right_child=node(line)
                   cur_node.right_child.parent=cur_node # set parent
#                   cur_node.right_child.isChild = "RIGHT"
```

```
                    self._inspect_insertion(cur_node.right_child)
                else:
                    self._insert(line,cur_x,cur_node.right_child)
            elif not self.isHigherSamePoint(line,cur_node.line): #line.
                lineEquation[0] < cur_node.line.lineEquation[0]:
                if cur_node.left_child==None:
                    cur_node.left_child=node(line)
                    cur_node.left_child.parent=cur_node # set parent
#                    cur_node.left_child.isChild = "LEFT"
                    self._inspect_insertion(cur_node.left_child)
                else:
                    self._insert(line,cur_x,cur_node.left_child)
            else:
                print("Lines are colinear")

        else:
            print("Value already in tree!")

    def print_tree(self):
        if self.root!=None:
            self._#print_tree(self.root)

    def _print_tree(self,cur_node):
        if cur_node!=None:
            self._#print_tree(cur_node.left_child)
            #print ('%s, h=%d'%(str(cur_node.value),cur_node.height))
            self._#print_tree(cur_node.right_child)

    def height(self):
        if self.root!=None:
            return self._height(self.root,0)
        else:
            return 0

    def _height(self,cur_node,cur_height):
        if cur_node==None: return cur_height
        left_height=self._height(cur_node.left_child,cur_height+1)
        right_height=self._height(cur_node.right_child,cur_height+1)
        return max(left_height,right_height)

    def find(self,line,cur_x):
        if self.root != None:
#            #print("Stepped in with:",line,cur_x,self.root)
            return self._find(line,cur_x,self.root)
        else:
            #print("None is from here")
            return None

    def _find(self,line,cur_x,cur_node):
        #print("Searching: ",line)
        #print("Cur node: ",cur_node.line)
        #print("X:",cur_x)
```

```
        if line == cur_node.line:
#           #print(line)
#           #print(cur_x)
#           #print(cur_node)
#           #print("!!!!!!!!!!!!!!!!!!!!!!!!")
            return cur_node
        elif self.calcValue(line,cur_x) < self.calcValue(cur_node.line
            ,cur_x) and cur_node.left_child != None:
#           #print("2222222222222222222222222")
            #print("Via <","Search:",self.calcValue(line,cur_x),"Node
                :",self.calcValue(cur_node.line,cur_x))
            return self._find(line,cur_x,cur_node.left_child)
        elif self.calcValue(line,cur_x) > self.calcValue(cur_node.line
            ,cur_x) and cur_node.right_child != None:
#           #print("3333333333333333333333333")
            #print("Via >","Search:",self.calcValue(line,cur_x),"Node
                :",self.calcValue(cur_node.line,cur_x))
            return self._find(line,cur_x,cur_node.right_child)

        elif self.calcValue(line,cur_x) == self.calcValue(cur_node.
            line,cur_x):
            #print("Via ==","Search:",self.calcValue(line,cur_x),"Node
                :",self.calcValue(cur_node.line,cur_x))
            if self.isHigherSamePoint(line,cur_node.line) and cur_node.
                right_child != None: #line.lineEquation[0] > cur_node.
                line.lineEquation[0]
                return self._find(line,cur_x,cur_node.right_child)
            elif not self.isHigherSamePoint(line,cur_node.line) and
                cur_node.left_child != None:
                return self._find(line,cur_x,cur_node.left_child)
#           if cur_node.left_child != None and line == cur_node.
    left_child.line:
#               return cur_node.left_child
#           elif cur_node.right_child != None and line == cur_node.
    right_child.line:
#               return cur_node.right_child
            else:
                #print("wazuup")
                return None
        else:
            return None

    def delete_value(self,line,cur_x):
        return self.delete_node(self.find(line,cur_x))

    def delete_node(self,node):

        ## -----
        # Improvements since prior lesson
```

```
# Protect against deleting a node not found in the tree
if node == None: #or self.find(node.line,cur_x) == None:
   #print("Node to be deleted not found in the tree!")
   return None
## -----

# returns the node with min value in tree rooted at input node
def min_value_node(n):
   current=n
   while current.left_child != None:
      current = current.left_child
   return current

# returns the number of children for the specified node
def num_children(n):
   num_children = 0
   if n.left_child != None: num_children += 1
   if n.right_child != None: num_children += 1
   return num_children

# get the parent of the node to be deleted
node_parent = node.parent

# get the number of children of the node to be deleted
node_children = num_children(node)

# break operation into different cases based on the
# structure of the tree & node to be deleted

# CASE 1 (node has no children)
if node_children == 0:

   if node_parent != None:
      # remove reference to the node from the parent
      if node_parent.left_child == node:
         node_parent.left_child = None
      else:
         node_parent.right_child = None
   else:
      self.root = None

# CASE 2 (node has a single child)
if node_children == 1:

   # get the single child node
   if node.left_child != None:
      child = node.left_child
   else:
      child = node.right_child

   if node_parent != None:
      # replace the node to be deleted with its child
```

```
            if node_parent.left_child == node:
               node_parent.left_child = child
            else:
               node_parent.right_child = child
         else:
            self.root = child

         # correct the parent pointer in node
         child.parent=node_parent

      # CASE 3 (node has two children)
      if node_children == 2:

         # get the inorder successor of the deleted node
         successor = min_value_node(node.right_child)

         # copy the inorder successor's value to the node formerly
         # holding the value we wished to delete
         node.line = successor.line
         node.value = successor.value

         # delete the inorder successor now that it's value was
         # copied into the other node
         self.delete_node(successor)

         # exit function so we don't call the _inspect_deletion
            twice
         return

      if node_parent != None:
         # fix the height of the parent of current node
         node_parent.height = 1+max(self.get_height(node_parent.
            left_child),self.get_height(node_parent.right_child))

         # begin to traverse back up the tree checking if there are
         # any sections which now invalidate the AVL balance rules
         self._inspect_deletion(node_parent)

def search(self,value):
      if self.root!=None:
         return self._search(value,self.root)
      else:
         return False

def _search(self,value,cur_node):
      if value==cur_node.value:
         return cur_node
      elif value<cur_node.value and cur_node.left_child!=None:
         return self._search(value,cur_node.left_child)
      elif value>cur_node.value and cur_node.right_child!=None:
         return self._search(value,cur_node.right_child)
      return False
```

```
   # Functions added for AVL...

def _inspect_insertion(self,cur_node,path=[]):
      if cur_node.parent==None: return
      path=[cur_node]+path

      left_height =self.get_height(cur_node.parent.left_child)
      right_height=self.get_height(cur_node.parent.right_child)

      if abs(left_height-right_height)>1:
         path=[cur_node.parent]+path
         self._rebalance_node(path[0],path[1],path[2])
         return

      new_height=1+cur_node.height
      if new_height>cur_node.parent.height:
         cur_node.parent.height=new_height

      self._inspect_insertion(cur_node.parent,path)

def _inspect_deletion(self,cur_node):
      if cur_node==None: return

      left_height =self.get_height(cur_node.left_child)
      right_height=self.get_height(cur_node.right_child)

      if abs(left_height-right_height)>1:
         y=self.taller_child(cur_node)
         x=self.taller_child(y)
         self._rebalance_node(cur_node,y,x)

      self._inspect_deletion(cur_node.parent)

def _rebalance_node(self,z,y,x):
      if y==z.left_child and x==y.left_child:
         self._right_rotate(z)
      elif y==z.left_child and x==y.right_child:
         self._left_rotate(y)
         self._right_rotate(z)
      elif y==z.right_child and x==y.right_child:
         self._left_rotate(z)
      elif y==z.right_child and x==y.left_child:
         self._right_rotate(y)
         self._left_rotate(z)
      else:
         raise Exception('_rebalance_node: z,y,x node configuration
            not recognized!')

def _right_rotate(self,z):
      #print("RR")
```

```
        sub_root=z.parent
        y=z.left_child
        t3=y.right_child
        y.right_child=z
        z.parent=y
        z.left_child=t3
        if t3!=None: t3.parent=z
        y.parent=sub_root
        if y.parent==None:
              self.root=y
        else:
           if y.parent.left_child==z:
              y.parent.left_child=y
           else:
              y.parent.right_child=y
        z.height=1+max(self.get_height(z.left_child),
           self.get_height(z.right_child))
        y.height=1+max(self.get_height(y.left_child),
           self.get_height(y.right_child))

  def _left_rotate(self,z):
        #print("LR")
        sub_root=z.parent
        y=z.right_child
        t2=y.left_child
        y.left_child=z
        z.parent=y
        z.right_child=t2
        if t2!=None: t2.parent=z
        y.parent=sub_root
        if y.parent==None:
           self.root=y
        else:
           if y.parent.left_child==z:
              y.parent.left_child=y
           else:
              y.parent.right_child=y
        z.height=1+max(self.get_height(z.left_child),
           self.get_height(z.right_child))
        y.height=1+max(self.get_height(y.left_child),
           self.get_height(y.right_child))

  def get_height(self,cur_node):
        if cur_node==None: return 0
        return cur_node.height

  def taller_child(self,cur_node):
        left=self.get_height(cur_node.left_child)
        right=self.get_height(cur_node.right_child)
        return cur_node.left_child if left>=right else cur_node.
           right_child
```

```
   def findNeighbours(self,cur_node):

      def min_value_node(n):
         current = n
         while current.left_child != None:
            current = current.left_child
         return current

      def max_value_node(n):
         current = n
         while current.right_child != None:
            current = current.right_child
         return current

      def findFirstBigger(n):
         parentN = n.parent
         if parentN == None:
            #print("Root fails")
            return None
         #print("BIGGER Node:",n,"Parent:",parentN)
         if parentN != None and parentN.left_child == n:
            return parentN
         else:
            return findFirstBigger(parentN)

      def findFirstSmaller(n):
         parentN = n.parent
         if parentN == None:
            return None
         #print("SMALLER Node:",n,"Parent:",parentN)
         if parentN != None and parentN.right_child == n:
            return parentN
         else:
            return findFirstSmaller(parentN)

#       #print("Type", type(cur_node))
      assert isinstance(cur_node,node)

      if cur_node.left_child and cur_node.right_child:
         belowNeighbour = max_value_node(cur_node.left_child)
         aboveNeighbour = min_value_node(cur_node.right_child)
         #print("Found neighbour:",cur_node,"Below: ",belowNeighbour,"
            Above: ",aboveNeighbour)
         return belowNeighbour,aboveNeighbour
      elif cur_node.left_child and not cur_node.right_child:
         belowNeighbour = max_value_node(cur_node.left_child)
         aboveNeighbour = findFirstBigger(cur_node)
         #print("Found neighbour:",cur_node,"Below: ",belowNeighbour,"
            Above: ",aboveNeighbour)
         return belowNeighbour,aboveNeighbour
      elif cur_node.right_child and not cur_node.left_child:
         belowNeighbour = findFirstSmaller(cur_node)
```

```
            aboveNeighbour = min_value_node(cur_node.right_child)
            #print("Found neighbour:",cur_node,"Below: ",belowNeighbour,"
               Above: ",aboveNeighbour)
            return belowNeighbour,aboveNeighbour
        elif not cur_node.left_child and not cur_node.right_child:
            belowNeighbour = findFirstSmaller(cur_node)
            aboveNeighbour = findFirstBigger(cur_node)
            #print("Found neighbour:",cur_node,"Below: ",belowNeighbour,"
               Above: ",aboveNeighbour)
            return belowNeighbour,aboveNeighbour
        else:
            #print("NO NEIGHBOUR")
            return None




#---------------------------AVL TREE
   --------------------------------------------------#


#################### DEFINITION OF CLASSES
   #####################################

#################### DEFINITION OF FUNCTIONS
   #####################################
def createSegments(rawEntries):
   pointList = []
   lineList = []
   segmentList = []

   for point in rawEntries:
      newPoint = Point(point[0],point[1],point[2])
      pointList.append(newPoint)

   i = 0
   while(i<len(pointList)-1):
      p1 = pointList[i]
      p2 = pointList[i+1]
      if p1.color == p2.color:
         newLine = Line(p1,p2)
         lineList.append(newLine)
         p1.assignToLine(newLine)
         p2.assignToLine(newLine)

         eStart = Event()
         eStart.start = newLine
         eEnd = Event()
         eEnd.end = newLine

         if p1 == newLine.start:
#            #print("Was here" )
```

```
            p1.events.append(eStart)
            p2.events.append(eEnd)
         else:
#            #print("Was here 1")
            p2.events.append(eStart)
            p1.events.append(eEnd)


      i += 1

   currentSegment = []
   currentColor = None
   for line in lineList:
      if currentColor == None:
         currentColor = line.color
      if currentColor == line.color:
         currentSegment.append(line)
      else:
         newSegment = Segment(currentSegment,currentColor)
         segmentList.append(newSegment)
         currentColor = line.color
         currentSegment = []
   newSegment = Segment(currentSegment,currentColor)
   segmentList.append(newSegment)


   return pointList, lineList, segmentList

#def intersect2(line1,line2):
#   assert isinstance(line1,Line)
#   assert isinstance(line2,Line)


def lineItersection(node1,node2,inf = False):
   assert isinstance(node1,node)
   assert isinstance(node2,node)

   line1 = node1.line
   line2 = node2.line

   def equalCoordinates(p1,p2):
      assert isinstance(p1,Point)
      assert isinstance(p2,Point)
      if p1.x == p2.x and p1.y == p2.y:
         #print("$$$$$$$$$$$$$$$$",p1,p2,"$$$$$$$$$$$$$$$$")
         return True
      else:
         #print("################",p1,p2,"################")
         return False

   if equalCoordinates(line1.p1,line2.p1) or equalCoordinates(line1.p2,
      line2.p1) or equalCoordinates(line1.p2,line2.p2) or
```

```
        equalCoordinates(line1.p1,line2.p2) :
        return None



    A1 = line1.p2.y - line1.p1.y
    B1 = line1.p1.x - line1.p2.x
    C1 = A1 * line1.p1.x + B1 * line1.p1.y

    A2 = line2.p2.y - line2.p1.y
    B2 = line2.p1.x - line2.p2.x
    C2 = A2 * line2.p1.x + B2 * line2.p1.y

    det = A1 * B2 - A2* B1

#    #print(A1,A2,"----",B1,B2,"-----",C1,C2)
#    #print("C1: ", C1, "Racun: ",A1 * line2.p1.x + B1 * line2.p1.y )
#    #print("C1: ", C1, "Racun: ",A1 * line2.p2.x + B1 * line2.p2.y )
    if C1 == A1 * line2.p1.x + B1 * line2.p1.y and C1 == A1 * line2.p2.x
         + B1 * line2.p2.y:
        return "Colinear"
    if det == 0:
        return None
    else:
        xp = (B2 * C1 - B1 * C2)/det
        yp = (A1 * C2 - A2 * C1)/det
        xpBetweenL1 = xp <= max([line1.p1.x,line1.p2.x]) and xp >= min([
            line1.p1.x,line1.p2.x])
        ypBetweenL1 = yp <= max([line1.p1.y,line1.p2.y]) and yp >= min([
            line1.p1.y,line1.p2.y])
        xpBetweenL2 = xp <= max([line2.p1.x,line2.p2.x]) and xp >= min([
            line2.p1.x,line2.p2.x])
        ypBetweenL2 = yp <= max([line2.p1.y,line2.p2.y]) and yp >= min([
            line2.p1.y,line2.p2.y])

        if inf:
            return Point(xp,yp)
        if xpBetweenL1 and ypBetweenL1 and xpBetweenL2 and ypBetweenL2:
            intPoint = Point(xp,yp)
            event = Event()
            event.intersect.append(node1)
            event.intersect.append(node2)
            intPoint.events.append(event)
            #print("Found intersect: ",intPoint,line1,line2)
            return intPoint
        else:
            return None

#################### DEFINITION OF FUNCTIONS
    #####################################
```

```
#################### Sweep line
   ###################################################

def SweepLine(queue,sweepTree):
   assert isinstance(queue,list)
   assert isinstance(sweepTree,AVLTree)

   intersects = []
   originalInter = []
   allreadyChecked = set()


   while(queue):
      #print
         ("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
#      if len(originalInter)> 3:
         #print(originalInter[2])
      #print
         ("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
      xEv = heapq.heappop(queue)
      assert isinstance(xEv,Point)
      #print("Point : ", xEv)
      startEvents = []
      endEvents = []
      intersectEvents = []

      for e in xEv.events:
         if e.start:
            startEvents.append(e)
         if e.end:
            endEvents.append(e)
         if e.intersect != []:
            intersectEvents.append(e)

#      for e in xEv.events:
#         assert isinstance(e,Event)

      for e in endEvents:
         #print("----------------Processing event: \n", e,"\n
            ----------------------------------------")
         if e.end:
            line = e.end
            cur_x = xEv.x
            assert isinstance(line,Line)
            cur_n = sweepTree.find(line,cur_x)
#            #print("CN end",type(cur_n))
            neighbours = sweepTree.findNeighbours(cur_n)
            belowNeighbour = neighbours[0]
            aboveNeighbour = neighbours[1]

            if belowNeighbour and aboveNeighbour:
```

```
                intersect = lineItersection(belowNeighbour,
                    aboveNeighbour)
                if intersect:
                    heapq.heappush(queue,intersect)
            DELETED.add(cur_n.line)
            sweepTree.delete_node(cur_n)
        #print("Tree")
        #print(tree)
        #print("Queue: ",queue)
    for e in startEvents:
        #print("----------------Processing event: \n", e,"\n
            ------------------------------------------")
        if e.start:
            line = e.start
            cur_x = xEv.x
            assert isinstance(line,Line)
            sweepTree.insert(line,cur_x)
            cur_n = sweepTree.find(line,cur_x)
#            #print("CN start",type(cur_n))
            neighbours = sweepTree.findNeighbours(cur_n)
            belowNeighbour = neighbours[0]
            aboveNeighbour = neighbours[1]

            if(belowNeighbour):
                intersect = lineItersection(cur_n,belowNeighbour)
                if intersect:
                    heapq.heappush(queue,intersect)
            if(aboveNeighbour):
                intersect = lineItersection(cur_n, aboveNeighbour)
                if intersect:
                    heapq.heappush(queue,intersect)
        #print("Tree")
        #print(tree)
        #print("Queue: ",queue)
    for e in intersectEvents:
        #print("----------------Processing event: \n", e,"\n
            ------------------------------------------")
        if e.intersect != []:
            cur_n1 = e.intersect[0]
            cur_n2 = e.intersect[1]

            assert isinstance(cur_n1,node)
            assert isinstance(cur_n2,node)

#            #print("Line one: ",line1)
#            #print("Line two: ",line2)
            cur_x = xEv.x
#            cur_n1 = sweepTree.find(line1,cur_x)
#            cur_n2 = sweepTree.find(line2,cur_x)
#            #print("CN1 inter",type(cur_n1))
#            #print("CN2 inter",type(cur_n2))
```

```
        if not ((cur_n1.line,cur_n2.line) in allreadyChecked or (
           cur_n2.line,cur_n1.line) in allreadyChecked):
           sweepTree.switchCrossedSegments(cur_n1,cur_n2,cur_x)

           neigh1 = sweepTree.findNeighbours(cur_n1)
           neigh2 = sweepTree.findNeighbours(cur_n2)

           belNeigh1 = neigh1[0]
           aboveNeigh1 = neigh1[1]
           belNeigh2 = neigh2[0]
           aboveNeigh2 = neigh2[1]


           if belNeigh1 and belNeigh1 != cur_n2:
              intersect1 = lineItersection(cur_n1, belNeigh1)
              if intersect1:
                 heapq.heappush(queue,intersect1)

           if aboveNeigh1 and aboveNeigh1 != cur_n2:
              intersect1 = lineItersection(cur_n1, aboveNeigh1)
              if intersect1:
                 heapq.heappush(queue,intersect1)

           if belNeigh2 and belNeigh2 != cur_n1:
              intersect2 = lineItersection(cur_n2, belNeigh2)
              if intersect2:
                 heapq.heappush(queue,intersect2)

           if aboveNeigh2 and aboveNeigh2 != cur_n1:
              intersect2 = lineItersection(cur_n2, aboveNeigh2)
              if intersect2:
                 heapq.heappush(queue,intersect2)

#          if belNeigh1 == cur_n2:
#             if belNeigh2:
#                intersect1 = lineItersection(cur_n1, belNeigh2)
#                if intersect1:
#                   heapq.heappush(queue,intersect1)
#             if aboveNeigh1:
#                intersect2 = lineItersection(cur_n2, aboveNeigh1)
#                if intersect2:
#                   heapq.heappush(queue,intersect2)
#
#          elif aboveNeigh1 == cur_n2:
#             if aboveNeigh2:
#                intersect1 = lineItersection(cur_n1, aboveNeigh2)
#                if intersect1:
#                   heapq.heappush(queue,intersect1)
```

```
   #                if belNeigh1:
   #                   intersect2 = lineItersection(cur_n2, belNeigh1)
   #                   if intersect2:
   #                      heapq.heappush(queue,intersect2)
                intersects.append((cur_n1.line,cur_n2.line))
                originalInter.append(e)
                allreadyChecked.add((cur_n1.line,cur_n2.line))

#            else:
                #print("**************************************")
                #print("(cur_n1,cur_n2): ", (cur_n1,cur_n2) in
                   allreadyChecked)
                #print("(cur_n2,cur_n1): ", (cur_n2,cur_n1) in
                   allreadyChecked)
                #print("**************************************")
          #print("Tree")
          #print(tree)
          #print("Queue: ",queue)
   filteredIntersects = []
   for i in intersects:
      l1 = i[0]
      l2 = i[1]
      if l1.color != l2.color:
         filteredIntersects.append(i)

   return filteredIntersects



################################### EXECUTION OF THE PROGRAM
   ###########################
INPUT_FILE = sys.argv[1]
#print("Input file: ", INPUT_FILE)

with open(INPUT_FILE) as file:
   l = file.readlines()
   for line in enumerate(l):
      splitLine = line[1].split(",")
      for ele in enumerate(splitLine):
         newEle = ele[1].strip()
         newEle = float(newEle)
         splitLine[ele[0]] = newEle
      l[line[0]] = splitLine


points, lines, segments = createSegments(l)
heapq.heapify(points)
tree = AVLTree()

i = SweepLine(points,tree)
print(len(i))
```

```
#sys.stdout.write(len(i))
```