Logo, company name

Description automatically generated

CS 478 HOMEWORK2  
  
ALP TUĞRUL AĞÇALI

21801799

1.a According to the book, between two Polygons there is at least 2 and at most 2min(x.y) supporting lines where x and y are amounts of vertices two polygons. Since we don’t know the vertice amounts and shapes of polygons we can’t say a specific number.   
1.b We know that, to find common supporting lines of P1 and P2 we first have to calculate Convex Hull of a union of P1 and P2. Then, we should find the consecutive vertices of CH(P1 U P2) and if one of this vertices originating From P1 and other one is originating from P2 this means that this line (edge of CH(P1 U P2) is common supporting line of these two polygons. Since it is wanted to use angles between polygon edges we can write an algorithm based on Jarvis' march algorithm. In this algoritm we need to find leftmost vertice in P2 with respect to a vertice in P1 and check if the line between this points is entirely on one side of P1 or P2.

Following algorithm will add supporting lines to lines[] array.

Procedure: findSupportingLines (P1, P2, lines):

for each vertex in P1.vertices:

leftmost = findLeftMostVertice (vertex, P2.vertices); //finds leftmost point using angle

//between x axis and point

line = constructLine (vertex, leftmost)

add = ture;

sideP1 = findSide(line, vertex.next) //returns “left” or “right” finds

sideP2 = findSide(line,leftmost.next)

for each V in P1.vertices:

if ( findSide( line, V ) != sideP1) //if all points of P1 is not at same side of line

add = false //do not add

for each V in P2.vertices:

if ( findSide( line, V ) != sideP2)

add = false

if add: //if all points are same side then add

lines.add(line)

2.

Procedure isTriangulation(dcel):

totalFace = 0 //counting face amount connected to the edges e.Leftface and E.rightFace

for each edge in dcel:

if( edge.leftFace != boundaryFace) //boundary face cannot be counted because it can be

//seen more than three.

totalFace ++

if( edge.RightFace != boundaryFace)  
  
 totalFace ++

if (totalFace / dcel.faces.getSize() != 3.0)

return false

return true

3.

Procedure weightBalance(G)

For each edge in G:

w(e) = 1

For iS = 2 to N-1

wIn(i) = sum of weights of incoming edges of vi

d1 = leftmost outgoing edge of vi

if( wIn(i) > wOut(i) ):

w(d1) = wIn(i) – wOut(i) +1

for i = N - 1 to 2

lines []

temp = leftmost outgoing edge of vi

line = temp

do :

degree = w(temp)

while ( degree > 0) :

lines.add(temp.x, temp.y);

degree - -

temp = temp.leftLine

if ( temp = incoming line to the vi):

temp = line

while ( temp != line )

for each e incoming edge to vi :

lineToConnect = lines.removeLast()

e.connect(LineToConnect)

wOut (i) = sum of weights of incoming edges of vi

d2 = leftmost incoming edge of vi

if( wOut (i) > ( wIn(i) ):

w(d2) = wOut (i) – wIn (i) + w(d2)

4.

We can use a general tree for the rectangles. We can construct it according to the x values of the rectangles. For example, if a rectangle R1 contains another rectangle R2 it will be parent of the R2rectangle. If it is not containing the other rectangle, the rectangle with smaller x1 value (minimum x value) will be at the left side of other one. Same, if the x2 value (maximum x value) of the rectangle bigger than other rectangle it will be at the right side of the other one. (Since, if the small x value is smaller and big x value is bigger than other it will be parent. Or, if the small x value is bigger and big x value is smaller it will be child rectangle.)  
With this algorithm we don’t need to check every rectangle. For example, when we checked all rectangles in the level one, let’s assume two of them containing the point P, we only need to check childs of them and so on. The Worst case algorithm is O(N) (when any rectangle does not contain another one, we need to check all of them but this data structure could be useful for finding the rectangles containing P.)  
To add new Rectangle, firstly, we need to check containers at level one. If the new one contains one of them or more add new one to level 1 and make the contained rectangle(s) it’s child. The place of the new rectangle will be position of the leftmost rectangle added. And if we added more than one rectangle,the empty spaces (The Rectangle becoming child lefts it’s position and causes it) will be filled with the rectangles on the right side of new added one (after insertion). If the rectangle is not contained by other rectangle and it is subset of another, it will be child of it’s and we should do same process with it at the level two (at any lower level). If these are not true then we should find the place of the rectangle at level 1 using it’s x values and put it at the right place.

The following is the structure of the rectangle.

struct Rectangle {

int minx, miny, maxx, maxy;

Rectangle\* right;

Rectangle\* left;

Rectangle\*\* childs;

int numChilds;

}

The following is adding new rectangle to tree.

procedure addRectangle(root, rectangle)

contained = []

parent = [];

for each r in root.childs:

if rectangle.minx < r.minx && rectangle.maxx > r.maxx: // if rectangle contains any of childs

contained.add(r)

else if rectangle.minx > r.minx && rectangle.maxx < r.maxx: //if any of child contains rectangle

contains.add(r)

if !contained.isEmpty(): //child rectangles contained by rectangle

contained[0].left.right = rectangle

rectangle.left = contained[0].left

rectangle.right = contained.getLast().right

contained.left = NULL

contained.getLast().right = NULL

rectangle.addChilds(contained)

else if !contains.isEmpty(): //child rectangles contains rectangle

for each r in contains:

addRectangle(r, rectangle)

else:

finded = false

temp = root.childs[0]

while temp != NULL:

if ( rectangle.minx < temp.minx): //check whether if new rectangle is at left side of the

//any of childs

rectangle.left = temp. left

rectangle.right = temp

temp.left = rectangle

finded = true

root.addChild(rectangle)

if !finded: //it it is not it is the rightest

temp = root.childs.getLast()

rectangle.left = temp

temp.right = rectangle

root.addChild(rectangle)

5.

Since points of a plygon given in sorted order we don’t need to sort it again ( O(NlogN)). We can use Scan Part of Graham’s scan algorithm. (O(N))

procedure findConvexHull(start):

o = calculateCenter(start)//calculates O in O(N) time

v = start

w = start.pred

f = false

while ( v.next != start || !f ):

if( v.next = w):

f = true

if( isLeft(v, v.next, v.next.next):

v = v.next

else :

delete v.next

v = v.pred