CS 164 - Computational Geometry **Winter 2016**

HW 2 Solutions

January 15, 2016 *Timothy Johnson*

1. (Problem 2.1) Let S be a set of n disjoint line segments whose upper endpoints lie on the line $y = 1$ and show lower endpoints lie on the line $y = 0$. These segments partition the horizontal strip $[-\infty : \infty] \times [0 : 1]$ into $n + 1$ regions. Give an $O(n \log n)$ time algorithm to build a binary search tree on the segments in S such that the region containing a query point can be determined in $O(\log n)$ time. Also describe the query algorithm in detail.

   We first build a balanced binary search tree on our line segments. To compare two line segments, we can either compare their upper or lower endpoints (it doesn't matter which).

   When we compare a point to a line segment, we use the orientation test to determine if it is on the left or right side. So we can search down through the binary tree in $O(\log n)$ time. We will eventually reach a leaf node, and compare our point to that leaf node.

   If the point is on the left side of the leaf segment, then it will be in the region between that line and the previous one in the tree. If it is on the right side of the leaf segment, then it will be in the region between that line and the next one in the tree.

2. (Problem 2.2) The intersection detection problem for a set S of n line segments is to determine whether there exists a pair of segments in S that intersect. Give a plane sweep algorithm that solves the intersection detection problem in $O(n \log n)$ time.

   We run the plane sweep algorithm given in the text for line segment intersection, but we stop if we discover an intersection. This takes $O(n \log n)$ time, because in the worst case we will have to process the events for the endpoints of each edge, each of which takes $O(\log n)$ time.

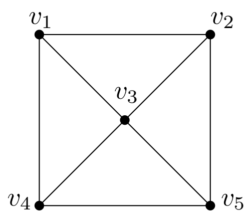
3. (Problem 2.5) Which of the following equalities are always true?

$$\begin{aligned} Twin(Twin(\overrightarrow{e})) &= \overrightarrow{e} \\ Next(Prev(\overrightarrow{e})) &= \overrightarrow{e} \\ Twin(Prev(Twin(\overrightarrow{e})) &= Next(\overrightarrow{e}) \\ IncidentFace(\overrightarrow{e}) &= IncidentFace(Next(\overrightarrow{e})) \end{aligned}$$

   The first equality will always be true, because edges are arranged in twin pairs. Thus the twin of the twin of an edge will always return the same edge.

   Similarly, the second equality is true, because Next and Prev are opposite of one another. One walks clockwise around a face, and the other walks counterclockwise.

The third equality is not necessarily true, as can be seen in the following diagram. Take the from v_1 to v_3. It's twin is e_{31}, whose previous edge is e_{43}, whose twin is e_{34}. By comparison, the next edge is e_{32}.



The fourth equality will be true, because following the Next pointer keeps us on an edge for the same face.

4. (Problem 2.6) Give an example of a doubly-connected edge list where for an edge e the faces $IncidentFace(\overrightarrow{e})$ and $IncidentFace(Twin(\overrightarrow{e}))$ are the same.

   Suppose that our map consists of two vertices with a single edge between them. Then there is a single face, whose inside contains the two half-edges. Each of these half-edges is thus incident to the same face.

5. (Problem 2.9) Suppose that a doubly-connected edge list of a connected subdivision is given. Give pseudocode for an algorithm that lists all faces with vertices that appear on the outer boundary.

   We first find the leftmost vertex. Then we follow the edge with the largest slope, since it will have to be on the outer boundary, and continue following the Next pointers from that edge until we return to the start. At each vertex that we visit, we find all of the faces that it touches.

   Pseudocode:

```
Input: vertices, edges, and faces

//Step 1: Loop over all vertices to find leftmost vertex
leftVertex = vertices[0]
for(vertex in vertices)
    if(vertex.x < leftVertex.x)
        leftVertex = vertex;

//Step 2: Find edge with largest slope
startEdge = leftVertex.incidentEdge
currEdge = startEdge

//Assuming general position, so the denominator is nonzero
maxVertex = currEdge.origin
```

```
maxSlope = (maxVertex.y − leftVertex.y)/(maxVertex.x − leftVertex.x)
maxEdge = currEdge

while(currEdge != startEdge)
    //Walk around our vertex
    currEdge = (currEdge.next).twin
    currVertex = currEdge.origin
    currSlope = (currVertex.y − leftVertex.y)/(currVertex.x − leftVertex.x)
    if(currSlope > maxSlope)
        maxEdge = currEdge
        maxVertex = currVertex
        maxSlope = currSlope

//Step 3: Now that we've found one edge on the outer face,
//we can walk around the boundary and add every face
faces = [] //Initialize empty list
startEdge = maxEdge
faces += getFaces(startEdge)

currEdge = startEdge.next
while(currEdge != startEdge)
    faces += getFaces(currEdge)

return faces


//Walk around a vertex starting at the origin of the given edge
//returning the list of all faces seen
def getFaces(startEdge):
    currFaces = [startEdge.incidentFace]
    currEdge = startEdge.twin.next

    while(currEdge != startEdge)
        currFaces.append(currEdge.incidentFace)
        currEdge = currEdge.twin.next

    return currFaces
```