Assignment 3

Computational Geometry (CS60064)

Suhas Jain | 19CS30048 | suhasjain142@gmail.com
Sarthak Johnson Prasad | 18CS10049 | sarthakjohnsonprasad@gmail.com

Question 1

Let S be a set of n disjoint line segments in the plane, and let p be a point not on any of the line segments of S. We wish to determine all line segments of S that p can see, that is, all line segments of S that contain some point q so that the open segment pq does not intersect any line segment of S. Give an $O(n \log n)$ time algorithm for solving this problem. See the example shown on the right.

Answer: For this we will use a sweep line algorithm where a line originating from the point p moves in a circle and finds all the line segments that are visible from that point.

Algorithm:

1. Make a list of all the endpoints of the line segments and represent them in polar coordinate system with respect to point p as origin, this is the set S. Also mark all of the points of S as unvisitied.
2. Sort all the points in S according to their polar angles. Make an event queue with all the points from S stored in the sorted order. Also mark starting and ending points of each individual line segment differently as two different events will occur at both of these points.
3. For the sweep line algorithm originate a ray from p, let us call it R, and rotate it starting from the horizontal increasing the angle from 0 to 2π.
4. Sweep status will store all the current line segments in S which are intersecting with R at any particular angle sorted in order of their distance from the point p. Let us call ray R at an angle α as $R(\alpha)$. We can use a suitable data structure like a binary search tree for this purpose.
5. At a particular angle α we extract all the events from the event queue which have the angle α. As mentioned earlier, two type of events can occur:
    - **Start of a segment:** We insert a new element in the in the sweep line status with value as the distance of this start point from point p.
    - **End of a segment:** We remove the element which was inserted at the start of this segment from the event queue.

6. We can observe that in a particular direction if there are multiple line one behind the other then the line which is closest to us will be visible and all the other lines will not be visible At a particular angle α after we have processed all the points with polar angle as α we check if there are one or more than elements in event queue. We mark the segment which is closest (element with the lowest value) as visible.

7. We keep extracting events from the event queue and repeat the previous step till the event queue becomes empty. All the line segments which we marked as visible throughout the process are our answer so we return those as output.

Time Complexity:

1. Calculating polar angles for endpoints of n line segments take $O(n)$ time.

2. Sorting all the points (total $2n$ points) and making a event queue takes $O(n \log n)$ time.

3. There can be at max $2n$ events at which R has to stop, at each event insertion or deletion of an element from the sweep status takes place. As deletion and insertion in a binary search tree is a $O(\log n)$ operation so for atmost $2n$ events it takes a $O(n \log n)$ time.

4. At the end checking for all the segments which were marked as visible also takes $O(n)$ time.

Therefore the whole algorithm is bounded by O(n log n) time complexity.

Question 2

Let S be a subdivision of complexity n, represented using DCEL data structure, and let P be a set of m query points. Give an $O((n+m)\log(n+m))$ time algorithm that computes for every point in P in which face of S it is contained.

Answer:

Algorithm:

1. For each query point $P(x_0, y_0)$ we want to find such an edge that if the point belongs to any edge, the point lies on the edge we found, otherwise this edge must intersect the line $x = x_0$ at some unique point $P(x_0, y)$ where $y < y_0$ and this y is maximum among all such edges. The following image shows both cases.

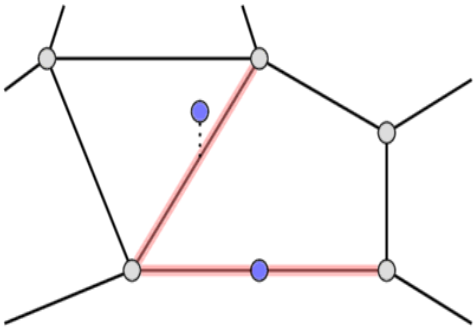

Figure 1: Edge we want to find for different query points

2. We will solve this problem offline using the sweep line algorithm. Let's iterate over x-coordinates of query points and edges' endpoints in increasing order and keep a set of edges s. For each x-coordinate we will add some events beforehand.
3. The events we talked about in the previous point will be of four types: ***add, remove, vertical, get***:
    - For each vertical edge (both endpoints have the same x-coordinate) we will add one ***vertical*** event for the corresponding x-coordinate.
    - For every other edge we will add one ***add*** event for the minimum of x-coordinates of the endpoints and one ***remove*** event for the maximum of x-coordinates of the endpoints.
    - Finally, for each query point we will add one ***get*** event for its x-coordinate.
4. For each x-coordinate we will sort the events by their types in order (***vertical, get, remove, add***). The following image shows all events in sorted order for each x-coordinate.

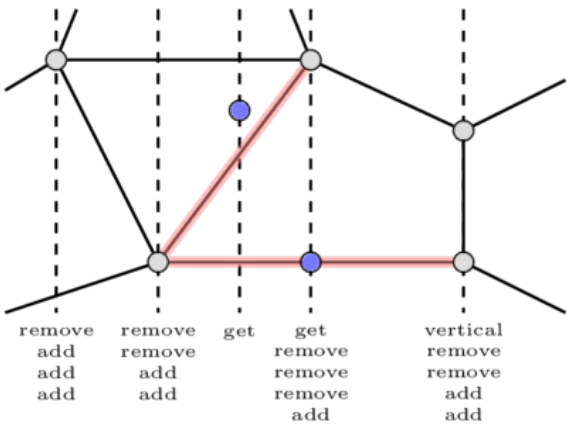

Figure 2: Sorted order of events by x-coordinate

5. We will keep two sets during the sweep-line process. A set ***s*** for all non-vertical edges, and one set ***t*** especially for the vertical ones. We will clear the set at the beginning of processing each x-coordinate.

6. Now let's process the events for a fixed x-coordinate.

    - If we got a vertical event, we will simply insert the minimum y-coordinate of the corresponding edge's endpoints to ***t***.
    - If we got a remove or add event, we will remove the corresponding edge from ***s*** or add it to ***s***.
    - For each get event we must check if the point lies on some vertical edge by performing a binary search in ***t***. If the point doesn't lie on any vertical edge, we must find the answer for this query in ***s***. To do this, we again make a binary search.

7. Now for each query point we have the edge that intersects the line $x = x_0$ at some unique point $P(x_0, y)$. In this case we know that our query point lies on one of the incidence faces of the edge-pair corresponding to this edge. We will always choose the face which lies in the upper part of this edge as we know that $y < y_0$. It can be determined by a simple orientation test.

8. We also have some edge cases we have to handle separately if no edge is found then we can directly deduce that the point lies in the outer face. Also if a point lies on an edge then there will be two faces that point will be a part of, we will output both of them.

REFERENCE: https://github.com/e-maxx-eng/e-maxx-eng/tree/master/src/geometry

Time Complexity:

1. There are in total m query points along with $2n$ endpoints of n edges, but two edges share an endpoint so there are total n endpoints. Step 2-4 will require sorting these according to their x-coordinates and their types. This process takes $O((n + m)\ log\ (n + m))$ time.

2. Insertion and deletion of elements in ***s*** and ***t*** takes $O(log\ (n + m))$ time as we have to maintain the sorted order while deletion or insertion.

3. At maximum we have $m + n$ such events so it takes $O((n + m)\ log\ (n + m))$ time.

4. Step 6-7 for each query point takes $O(1)$ time as we have to access the DCEL and find out the incidence face for certain half-edges and handle the edge cases if any. So, in total this takes $O(m)$ time.

Therefore the whole algorithm is bounded by $O((n + m)\ log\ (n + m))$ time complexity.

Question 3

Write a formal proof for the following claim: Any polygon with h holes and a total of n vertices (including those defining the polygon and holes), can always be guarded by $\lfloor (n+2h)/3 \rfloor$ vertex guards. Note that a hole may be surrounded by other holes, and thus it may not be always visible from the boundary of the polygon.

Answer: First we find a triangulation of the polygon P, triangulate it into t triangles, let the triangulation be T. The plan of the proof is to "cut" the polygon along diagonals of the triangulation in order to remove each hole by connecting it to the exterior of P. It is clear that every hole must have diagonals in T from some of its vertices to either other holes or the outer boundary of P. Cutting along any such diagonal either merges the hole with another, or connects it to the outside. In either case, each cut reduces the number of holes by one. We make h such cuts, one cut on any of the diagonal originating from h holes. However, we need to choose the cuts so that the result is a single polygon as it may be that a choice of cuts results in several disconnected pieces. We can ensure that by making a dual of this triangulation and removing edges from it shared with the exterior face, in this manner guarantees that a single connected graph.
Let P' be the polygon that results after all holes are cut in the above manner, so there are no remaining holes in P'. Then P' has $n+2h$ vertices, since two vertices are introduced per cut, but because cuts do not create new triangles, it still has t triangles. Now, according to Chvatal's Art Gallery Theorem a simple polygon with n vertices can always be guarded by $\lfloor \frac{n}{3} \rfloor$ vertex guards. So, when we apply the same principle to P' with $n+2h$ vertices we can say that it can be guarded by $\lfloor \frac{n+2h}{3} \rfloor$ vertex guards.

Question 4

Consider an implementation of Hertel-Melhorn (HM) Algorithm for convex-partitioning of a simple polygon P with n vertices. Assume that a triangulation of P is given. Suggest a data structure and the required procedure so that HM-Algorithm can be implemented in $O(n)$-time.

Answer: We use DCEL (doubly connected edge list) for implementation of Hertel-Melhorn (HM) Algorithm in $O(n)$ time. We will consider all the edges of P along with the diagonals from the triangulation as a part of a planar graph and store it in a DCEL. For each pair of half-edges corresponding to a edge we also store if that edge is a diagonal or not in DCEL data structure. We also maintain a list of all the diagonals which are essential, these will be the diagonals which will do the convex partitioning. Let us call the list of essential diagonals as E and initialise it to empty.

Algorithm:

1. Iterate on all the half-edges which belong to a diagonal and check if the diagonal is essential or not (shown in the next step how to check). If the diagonal is essential insert it into E else remove that half-edge along with its pair from the DCEL.
2. **Checking essentiality:** We calculate two angles using next and previous half-edges of the pair of half-edges (let us call them e_1 and e_2 we are checking the essentiality for. If any of these angles is not convex then this diagonal is essential otherwise it is not.
    - First angle is between next edge of e_1 and previous edge of e_2.
    - Second angle is between previous edge of e_1 and next edge of e_2.
3. After doing this for pairs the half-edges corresponding to all the diagonals we get a list of essential diagonals which will divide the polygon into smaller convex polygons.

Time Complexity:

1. Storing which edge is part of a diagonal for all the half-edges takes $O(n)$ time.
2. Iterating over all the half-edges and checking if the diagonal they belong to is essential also takes $O(n)$ time.
3. For each half-edge calculating the two angels and checking if they are convex or not takes $O(1)$ time.
4. Also note that deleting a pair of half-edges from DCEL also takes $O(1)$ time.

Therefore the whole algorithm is bounded by O(n) time complexity.