Computational Geometry (CS60064)

Assignment 3

Ashutosh Kumar Singh - 19CS30008

Vanshita Garg - 19CS10064

Question 1

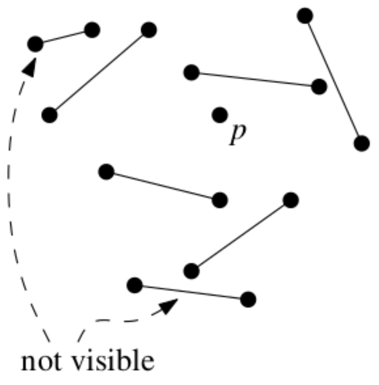

Figure 1

We are given a set S of n disjoint line segments in the plane and a point p which is not on any of the line segments of S. We need an $O(n\ log\ n)$ time algorithm to determine all the segments of S that are visible from p.
For this, we will use a radial sweep line algorithm as explained below.

Algorithm

1. Translate the coordinate system such that point p becomes the new origin and mark all the edges in the set S as not visible.

2. Sort the end points of all the line segments w.r.t. the angle $\theta \in [0, 2\pi)$ made by the line joining the end-point of the segment to the point p and the x-axis and store them in an event queue Q. There are two types of events, viz., occurrence of the first end-point of a segment and occurrence of the second end-point of a segment. Mark them differently in Q.

3. Sweep a ray R, 360°, originating from p radially, starting from the horizontal in the anti-clockwise direction and maintain a binary search tree that stores the sweep line status. The sweep line status will store the identifiers of the line segments of set S in the sorted order, sorted w.r.t. their distance from the point p.

4. Let the event that is currently at the front of Q be at an angle α. Keep extracting the events from the event queue until the angle is α and Q is non-empty. The two types of events are handled as:

   - If the extracted event is the first end-point of a segment, then insert the corresponding line segment identifier in the sweep line status using the distance of that end-point from p as key.
   - If the extracted event is the second end-point of a segment, then remove the corresponding line segment identifier from the sweep line status if it exists previously, else do nothing.

After handling all the events that are at an angle α, if the size of the sweep line status tree is ≥ 1, mark the line segment in the tree which is closes to p as visible.

5. Repeat step - 4 until the event queue Q is non-empty.

6. Output all those segments from S that are marked as visible at least once in step - 4.

Time Complexity Analysis

- Translating the coordinate system in step - 1 takes $O(n)$ time.
- The total number of end-points for the n line segments of set S is $2n$, hence the sorting in step - 2 takes $O(n\ log\ n)$ time.
- Ray R in step - 3 will only stop at events in the event queue Q. There are $2n$ such events in the event queue and handling each event in step - 4 is either a insertion or a deletion in the binary search tree which can take a maximum of $O(log\ n)$ time for each event. Hence, the total time for $2n$ such events will be bounded by $O(n\ log\ n)$.

Hence, the total time complexity of the algorithm is bounded by $O(n\ log\ n)$.

Question 2

We are given a subdivision S of complexity n represented using a DCEL data structure, and a set P of m query points. We need an $O((n+m)\ log\ (n+m))$ time algorithm that computes for every point in P in which face of P it is contained.
For this, we will use a sweep line algorithm as explained below.

Algorithm

1. Firstly, create a new edge list L and store an edge corresponding to each half edge and its twin edge in this edge list. Also, store both the half edges along with this edge information in the list.
   Now, for every query point $p\ (x_0, y_0) \in P$, we want to find an edge in L such that either point p lies on this edge or this edge intersects the line $x = x_0$ at some unique point (x_0, y) where $y < y_0$ and this y is maximum among all such edges (see Figure 2).

2. Iterate over x-coordinates of query points in P and end-points of edges in the edge list L in increasing order and for each x-coordinate we will add some events beforehand. These events can be of four types:

   (a) Add: For every edge that has distinct x-coordinates of end-points, we will have one *add* event for the minimum of x-coordinates of the endpoints.

   (b) Remove: For every edge that has distinct x-coordinates of end-points, we will add one *remove* event for the maximum of x-coordinates of the endpoints

   (c) Vertical: For each vertical edge (both endpoints have the same x-coordinate) we will add one *vertical* event for the corresponding x-coordinate.

   (d) Get: For each query point in P, we will add one *get* event for its x-coordinate.

3. For each x-coordinate, sort the events by their types in order (vertical, get, remove, add) (see Figure 3).

4. Now keep two sets during the sweep line process, set t for all non-vertical edges and set *vert* especially for the vertical edges. Clear the set *vert* at the beginning of processing each x-coordinate.

5. Now, process the events for a fixed x-coordinate:

   (a) If it is a vertical event, insert the minimum y-coordinate of the corresponding edge's endpoints to $vert$.

   (b) If it is a remove or add event, remove the corresponding edge from t or add it to t respectively.

   (c) For each get event, check if the point lies on some vertical edge by performing a binary search in $vert$. If the point doesn't lie on any vertical edge, find the answer for this query in t. To do this, again make a binary search. In order to handle some degenerate cases, answer all get events again after all the events are processed for this x-coordinate and choose the best of two answers.

6. Now for every query point, we have the corresponding edge. Once we have found an edge corresponding to each query point in P, there can be three cases:

   (a) No edge is found in L. In this case, the query point p is contained in the outer face.

   (b) Point p belongs to the found edge. In this case, no unique face exists. So, we will output the incident faces of the half-edges corresponding this edge.

   (c) An edge is found in L such that this edge intersects the line $x = x_0$ at some unique point (x_0, y) where $y < y_0$ and this y is maximum among all such edges. In this case, the point p will be contained in the incident face of one of the half-edges that corresponds to this edge. To determine this incident face, we will check the orientation of this point w.r.t. both the half-edges and output the incident face of that half-edge for which this point lies to the left of the half-egde.

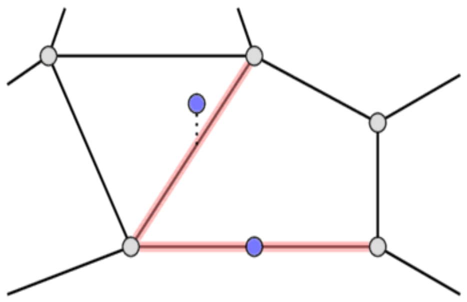

Figure 2

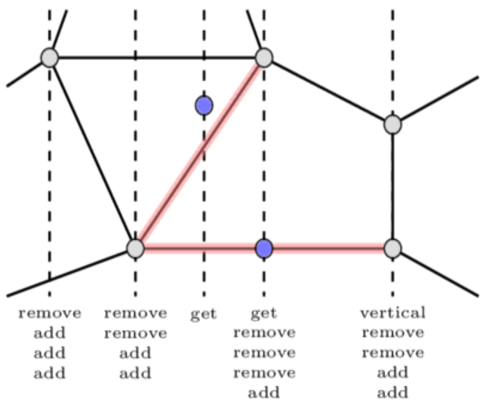

Figure 3

Time Complexity Analysis

- Creating edge list L in step - 1 takes $O(n)$ time since the complexity of subdivision is n.
- Creating event list in step - 2 and step - 3 requires sorting of m query points + $2 * n$ end-points of n edges. This will take $O((n+m)\ log\ (n+m))$ time.
- Set t and set $vert$ will be a balanced binary search tree and each of the add/remove/get event needs atmost $O(log\ (n+m))$ time. So for $(m+2*n)$ such events, it will take $O((n+m)\ log\ (n+m))$ time.
- Finding face in step - 6 will take constant time as it requires one access to DCEL per query point. So, total time for this step will be $O(n)$.

Hence, the total time complexity of the algorithm is bounded by $O((n+m)\ log\ (n+m))$.

Question 3

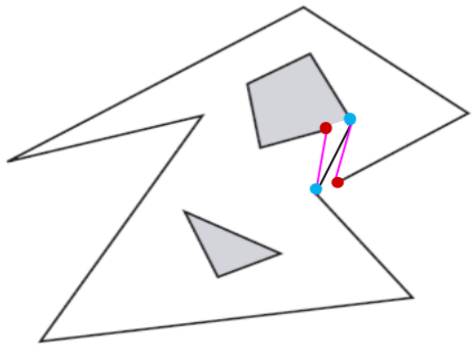

Figure 4: Triangulating a polygon with holes

We are given a polygon P with h holes and a total of n vertices (including the vertices of the holes). We need to prove that polygon P can always be guarded by $\lfloor \frac{(n+2h)}{3} \rfloor$ vertex guards.

Proof: In order to prove that polygom P can always be guarded by $\lfloor \frac{(n+2h)}{3} \rfloor$ vertex guards, we modify polygon P to P' such that modified polygon P' is devoid of any hole.
To modify polygon P to P', we first triangulate polygon P. Let the triangulation be T. Now, we know that every hole in P must have a diagonal in T from some of its vertices that connects this hole to either other holes or the outer boundary of P. So, for every hole in P, we take one such diagonal from T and make a cut along that diagonal. This cut will either merge two holes or connect it to the outer boundary of the polygon P. In both the cases, this cut reduces the number of holes by one and increases the number of vertices by two. Since there are h holes, we will make h such cuts and since each cut merges two holes, so at the end of h such cuts we will obtain a new simple polygon P' that has no hole. Now, since every such cut introduces two new vertices, so after h such cuts, we will get $2h$ such new vertices. Hence, this modified polygon P' will have $n+2h$ vertices.
Also, according to Chvatal's Art Gallery Theorem, we know that a simple polygon with N vertices can always be guarded by $\lfloor \frac{N}{3} \rfloor$ vertex guards. So according to this theorem, we can say that the modified simple polygon P' with $n+2h$ vertices can be guarded by $\lfloor \frac{(n+2h)}{3} \rfloor$ vertex guards.

Question 4

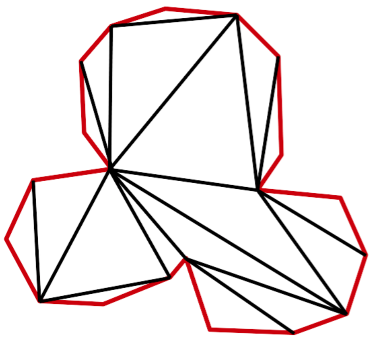
Figure 5: Before removing the inessential diagonals

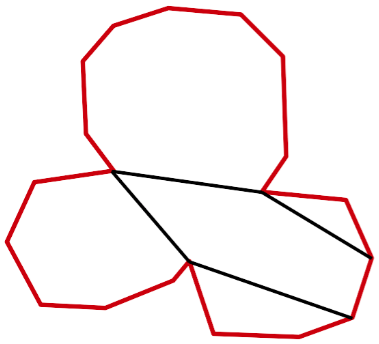
Figure 6: After removing the inessential diagonals

We will use a DCEL (doubly connected edge list) data structure for the implementation of the Hertel-Mehlhorn (HM) algorithm and we will store the edges of P and diagonals in the triangulation of P in a DCEL. In addition to this, we will maintain two lists, an *essential* list to store all the essential diagonals after the algorithm terminates and a list L to store all the diagonals in the triangulation of P along with pointers to both the half-edges in the DCEL corresponding to this diagonal.

Algorithm

- Initialize list *essential* to empty.
- Iterate on list L and check for the essentiality of the current diagonal. If it is found essential, add it to the *essential* list, else, discard it.
- To check for the essentiality of the diagonal, we find the *next* and *prev* half-edges of both the half-edges corresponding to this diagonal. Now we compute:
    1. The angle between the *next* edge of the first half-edge and the *prev* edge of the second half-edge, and,
    2. The angle between the *prev* edge of the first half-edge and the *next* edge of the second half-edge

  If both these angles are convex, then this diagonal is inessential, else it is essential.
- To remove an inessential diagonal, we need to update the DCEL and delete both the half-edges corresponding to this diagonal and also change the *prev* and *next* pointers and incidence faces of the half-edges associated with the removed half-edges (see Figure 7 and 8).
- Output all the diagonals that are present in the *essential* list when the end of the list L is reached.

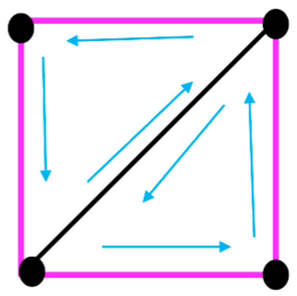
Figure 7: Before removing a diagonal in the DCEL

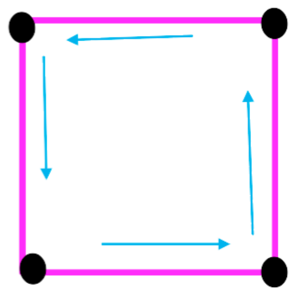
Figure 8: After removing a diagonal in the DCEL

Time Complexity Analysis

- Creating list L takes $O(n)$ time as the number of diagonals in the triangulation is linear.
- Iterating on list L takes linear time and each iteration requires checking essentiality of one diagonal and removal of constant number of half edges from the DCEL which can be done in constant time, hence, the total time complexity of n such iterations takes $O(n)$ time.

Hence, the total time complexity of this algorithm is bounded by $O(n)$.