COMPUTATIONAL GEOMETRY

Lec 6

SWEEP LINE

Before diving into the concept of sweep line, let us look at some applications which will help us understand the need of this concept in a better manner.

An important application is Geographical Information System (GIS). This is basically related to a region & various features of that region that we want to represent.

For ex: Let the region be India & the features that we want to represent are forest cover, road network, rail network, flight connectivity, lakes & water bodies.

So much information is available for any region which makes it difficult to represent all of them in one go. Of course, all information of that region might not be relevant for any specific application. Ex:- travelling by road only requires road network information. River network, rail network are irrelevant for this case.

For efficiency perspective, each feature is stored in a different layer (a data structure/database) by somehow dividing them into logical entities & storing them separately.

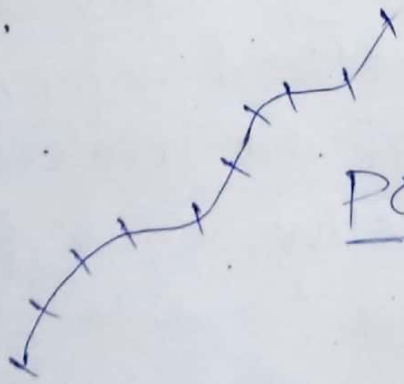
Depending on the application, whatever layer is necessary to be used, that information is used & result is displayed on the screen. Whenever we are dealing with 2 types of network (for eg:- road network, river network), whenever they cross then there is some important structure or information at the intersection point.

Forex:- A bridge is present at the intersection point/region of a road network & river network.

Similarly, there may be information related to road network & forest area. For eg:- if road passes through a dense forest or a normal city area.

finding intersection b/w diff. subjects in diff. layers is comman problem in all types of application in Geographical Information Systems.

The most simplest of them all is finding intersection b/w line segments (Networks for eg. road network can be approximated using line segments).



POLYLINE

Similarly, region boundaries can also be approximated using line segments. So whenever we are overlaying one layer over another layer, we need to find intersection b/w these objects (layers, boundaries, lines)

Simplest approach: Solve intersection of line segments by breaking all objects into line segments & use the result to compute intersection b/w objects.

Hence, our problem reduces to

Given two class of line segments, determine if there is an intersection b/w them. If yes, find intersection points.

$$S = \{S_1, S_2, \ldots, S_n\}$$

(n line segments, find intersection b/w these line segments)

LINE SEGMENTS INTERSECTION PROBLEM

Here, $S_i = (p_i, q_i)$

S_i: line segment; represented by two endpoints.

$p_i = (x_i, y_i)$ Each point is represented using two coordinates

$q_i = (x_i', y_i')$ x-coordinate & y-coordinate.

Simplest solution is

Take one segment & see if it intersects with any other segment or not.

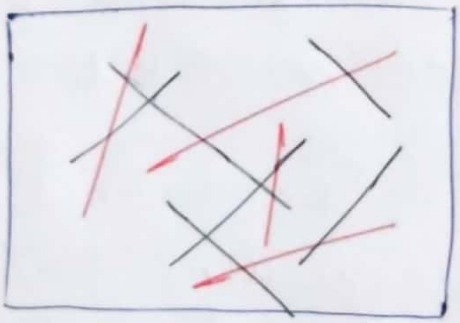
```
for i=1 to n-1 do
    for j=i+1 to n do
        if (si & sj intersect)
            report intersection
```

TIME COMPLEXITY = $O(n-1) + O(n-2) + \ldots$

$= O(n(n-1)/2)$

$= O(n^2)$

If we look at this real application problem, we have 2 layers

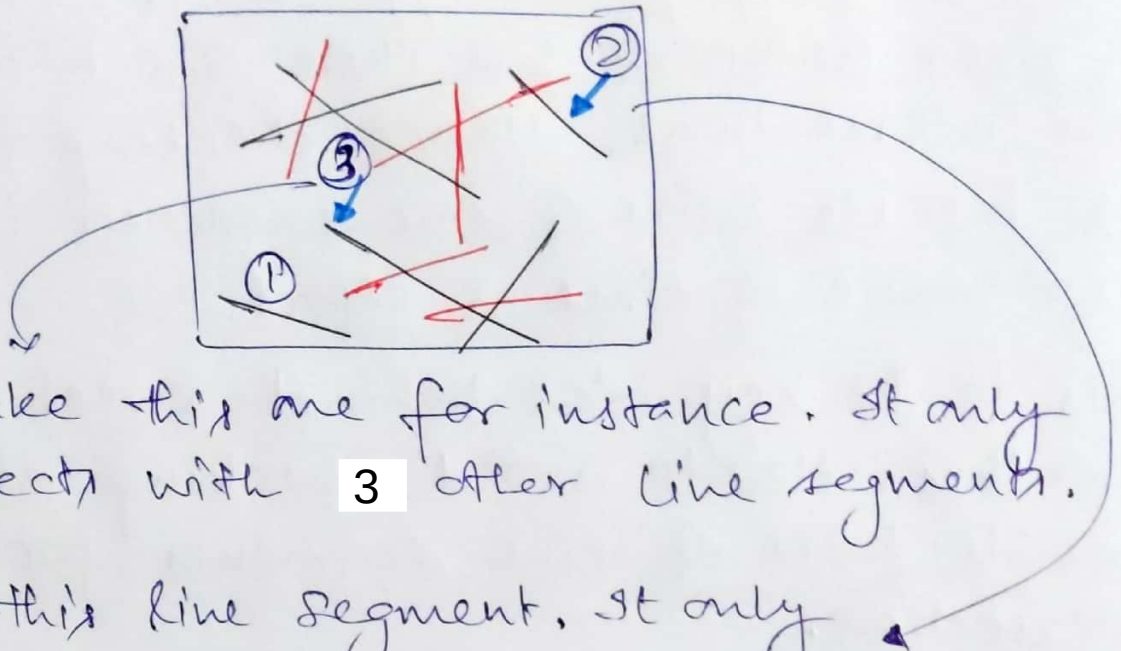


FIND INTERSECTION B/W RED & BLACK LINE SEGMENTS

In the brute force method, we are taking one segment & taking intersection with all other segments. This is too much work. Here, we have no information as to which segment is red & which is black. For us, all are alike & we are finding intersection b/w any two segments.

We came up with a brute force solution but that takes $O(n^2)$ time. But, as we can see

In the figure, many line segments do not intersect with many other line segments.



Let's take this one for instance. It only intersects with 3 other line segments.

Take this line segment. It only intersects with 1 line segment. But we need not check intersection of ① & ② since they are very far from each other. Two line segments can only intersect if they are close to each other.

Is it possible to avoid all these checkings where two lines are far away?

Of course, in the worst case, every line segment intersects with every other line segment. There may be $O(n^2)$ intersections. But we don't want every line segment to check with every other line segment with the aim to get a better algorithm.

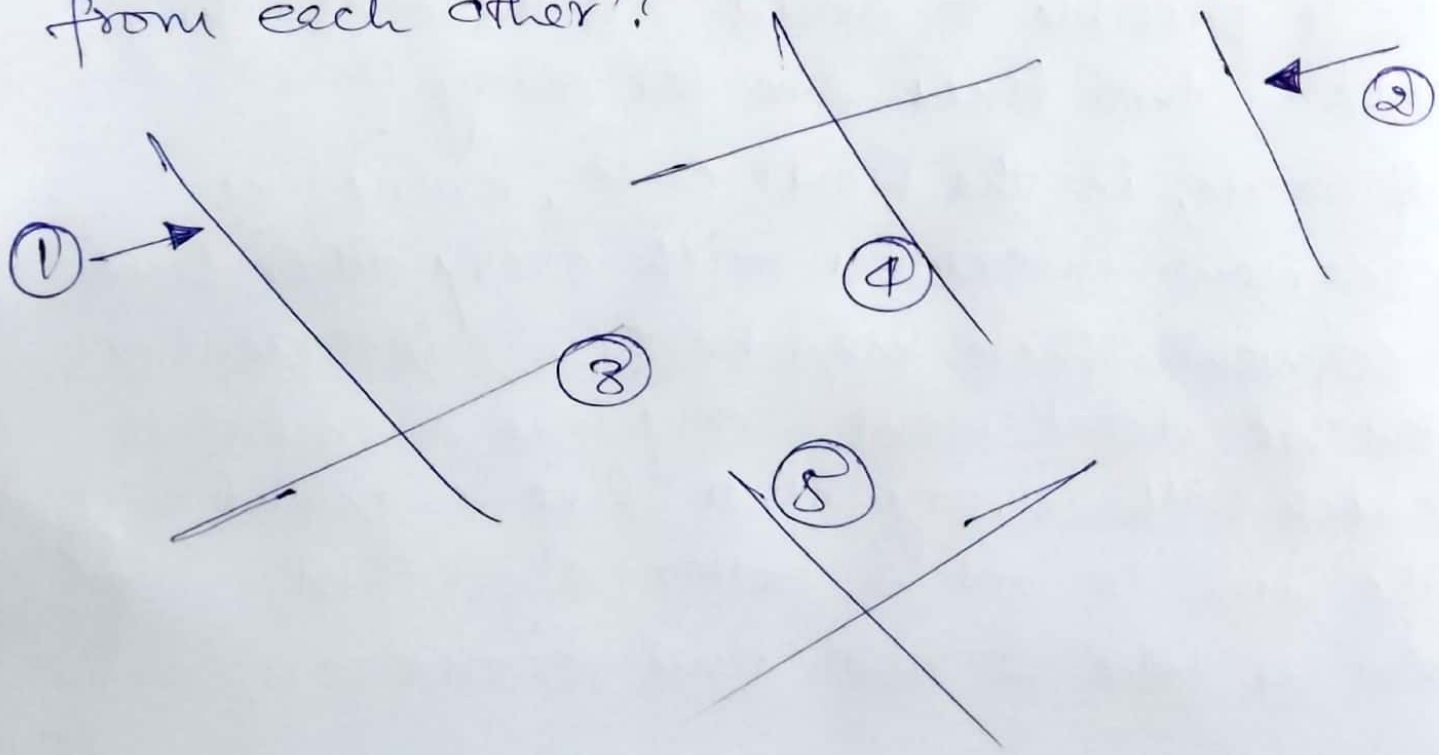
Let us take a real time scenario.

Let us take the case of road network & river network. There are huge number of lines in geographical systems of road network & river network but there are barely few intersections. Hence, checking for all possible pairs of line segments is a huge waste of time & effort.

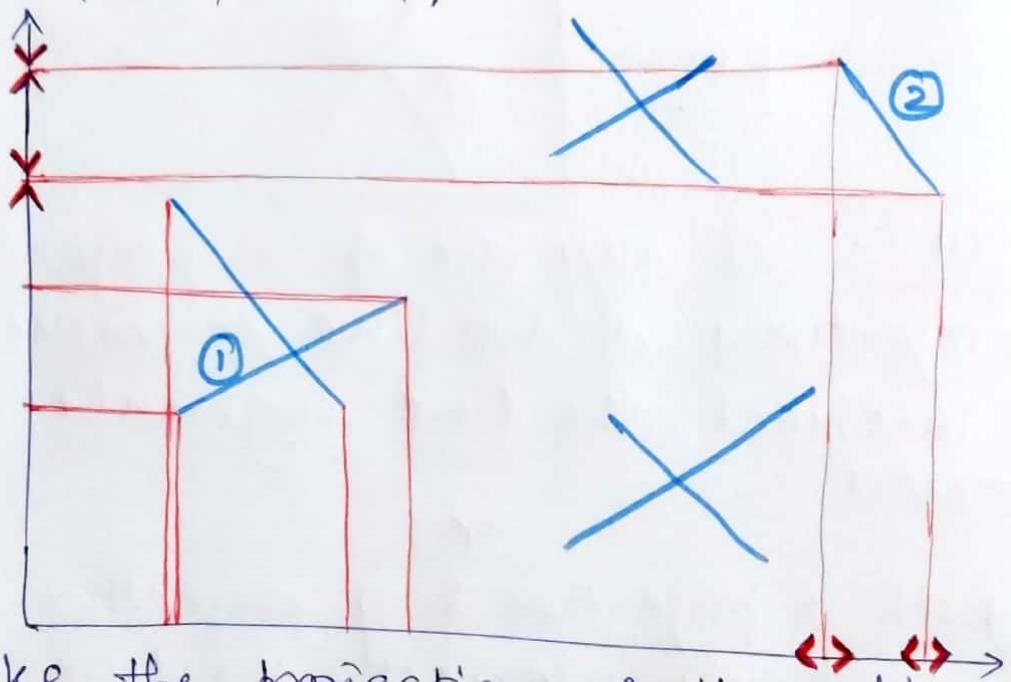
This is the key idea behind sweep line algorithm. How to avoid checking all possible pairs of line segments for intersection.

1 clear observation is that if two lines are very far away, it is very less likely that they intersect. Hence, we can avoid checking for their intersection.

Now, how to quantify this idea of 'far from each other'?



1 & 2 are very far away. So we can avoid checking for their intersection. 3, 4 & 5 are relatively close so we need to check for their intersection.

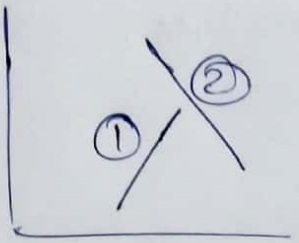


If we take the projections of these lines on the x-axis & y-axis, then check for intersections of these projections then we can say something directly about the intersection of these line segments.

Since the x projections of ① & ② are not intersecting, the line segments ① & ② will also not intersect. Cheking this is much easier.

Similarly, we can do for y projections also. If the y projections do not overlap, we can say that the line segments are not going to intersect

However, we cant say anything if both x & y projections overlap. The line segments may still not be intersecting

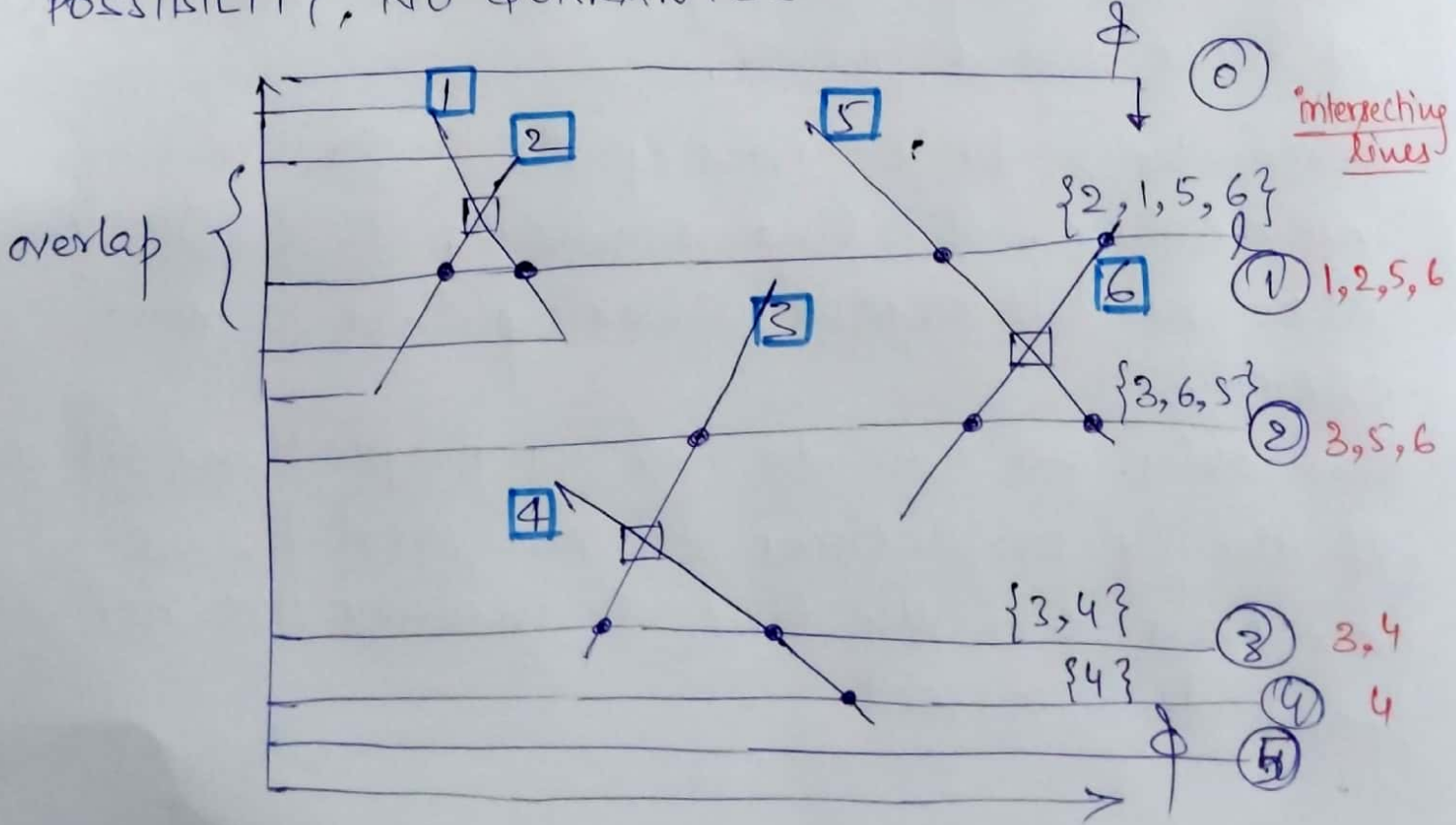
Ex:- 

RESULT:- If any one of x projections or y projections of two line segments do not intersect, the line segments do not intersect.

or

If both x projection & y projection of two line segments intersect, the line segments may intersect

POSSIBILITY, NO GUARANTEE



Now, we want to capture the property of y overlap. Look at lines 1 & 2 and their y overlap region. If we take any horizontal line l in the y overlap region, it intersects with both line segments 1 & 2.

So what we do is we take a horizontal line from the top of the plane & sweep the plane downwards (move from the top end to bottom end)

top end ⇓ above all line segments

bottom end ⇓ below all line segments

while sweeping, we will get many instances of the horizontal sweep line.

There exists an instance of this horizontal line where the line segments responsible for intersection points are adjacent to each other.

while sweeping plane from top to bottom, certainly we will encounter a position of the sweep line where we pass through the overlap of two line segments.

Certainly, there are some false alarms also there. Eg:- for line segments 1 & 5, & line segments 2 & 5 on line l. There is overlap but there is no intersection.

But for each & every intersection, overlap is there & somewhere this sweep line will capture it.

So, our algorithm is

① Move the sweep line from top to bottom.

② Find out line segments that intersect with the sweep line since they are the candidate points (y overlap). Check for intersection.

③ Move the sweep line down & repeat the procedure.

As we move down, the line segments that intersect with sweep line change.

For eg:- for line

⓪	ϕ
①	1, 2, 5, 6
②	3, 5, 6
③	3, 4
④	4
⑤	ϕ

Now we need to check intersections among the lines that the sweep line cuts. For this, we will maintain the line segments in the increasing order of x coordinate of intersection points with sweep line.

Eg:- for ①, it will be {2, 1, 5, 6}

This is called as **SWEEP LINE STATUS**

NOTE: We are only maintaining the list of line segments in left to right order. We are not maintaining the intersection points. This is because if the sweep line changes by a slight amount, the intersection points change but the list remains same.

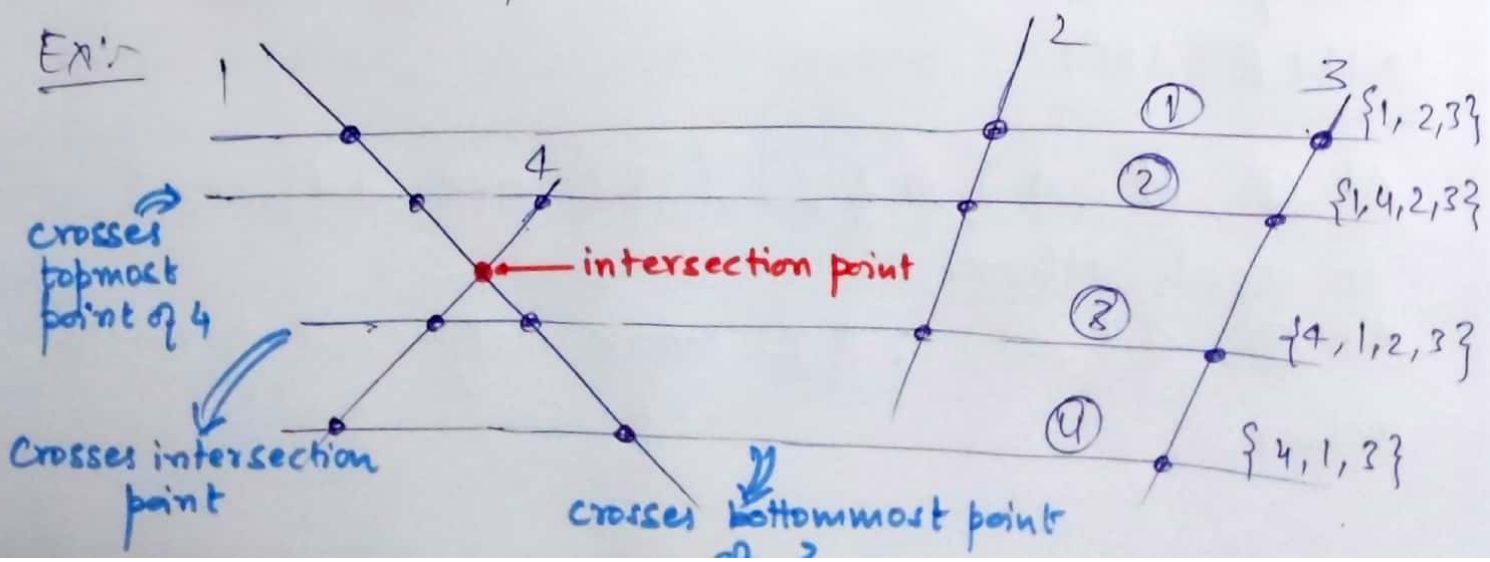
Change continuously hence not possible to maintain.

Sweep line status : left to right order of line segments intersected by sweep line.

Now, we ask what are the events when the sweep line status changes.

1. when the sweep line crosses the topmost point of a line segment
2. when the sweep line crosses an intersection point.
3. when the sweep line crosses the bottommost point of a line segment.

Ex:-



We know the topmost points & the bottommost points of the line segments, but we do not know the intersection points. So, we will find intersection points on the fly while moving from top to bottom & update the sweep line status.

We need a data structure to represent the sweep line status. We identify events at which the sweep line status changes. We need another data structure to store the top to bottom order of these events. This is called an event Queue.

↳ like a heap

↳ topmost event always the root.

Since we know the topmost & bottommost points of line segments, we insert a sorted list of these from top to bottom in the event Queue initially before the start of the algorithm.

For the sweep line status, a height balanced Binary Search Tree is used as the data structure

IMPORTANT

At ②, (1 & 4), (4 & 2), (2 & 3) are adjacent to each other.

At ①, (1 & 2), (2 & 3) were adjacent to each other.

After the sweep line crosses the topmost point of 4, 4 is inserted in sweep line status. Now, we need to check if newly formed adjacent pairs {(1 & 4), (4 & 2)} intersect. If they do, we need to push that into the event Queue.

Similarly, when sweep line crosses the bottommost point, a removal happens & a new adjacent pair is formed.

eg: (1 & 3) is formed when sweep line crosses bottommost point of 2. Hence, we need to check if 1 & 3 intersect & insert into event Queue accordingly (if intersection point is below current position of sweep line).

Similarly, in moving from ② to ③, a new adjacent pair (1 & 2) is introduced. Hence, we check for the intersection of these two line segments.

So basically, whenever sweep line status changes, new adjacent pairs are formed. We check the intersection of these line segments & report the intersection if it is there.

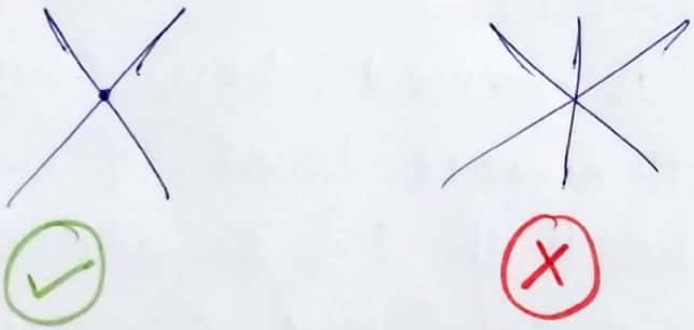
COMPUTATIONAL GEOMETRY

Lec 7

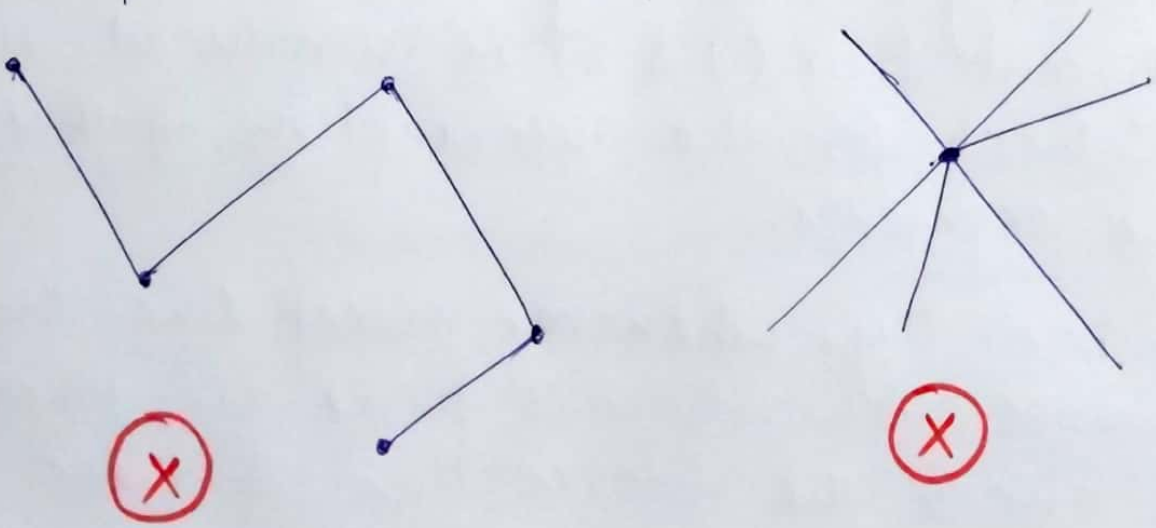
Assumptions:

For sake of simplicity of algorithm, we make the following assumptions

① Each intersection point is determined by exactly two line segments.



② Topmost points & bottommost points are distinct (i.e., no two line segments have an endpoint in common)



Algorithm:-

I/P : $\{S_1, S_2, \ldots S_n\}$

endpoints of n line segments

$Q = \phi$ (event Queue)

$L = \phi$ (sweep line status)

$Q = \{$ endpoints of line segments $\}$

"2n points inserted"

while $(Q \neq \phi)$ do

{

$q = Ext.Min(Q)$

case(i) q is top endpoint of S_i

insert S_i into L

S_l = segment left of S_i in L

S_r = segment right of S_i in L

if (S_i and S_l) are intersecting then
insert the intersection point into Q if it is not present in Q and below L

if (S_r and S_i) are intersecting then
insert the intersection point into Q if it is not present in Q and below L.

case (ii) q is bottom endpoint of S_i

S_l = segment left of S_i in L

S_r = segment right of S_i in L

Delete (S_i, L)

if (S_l and S_r) are intersecting then

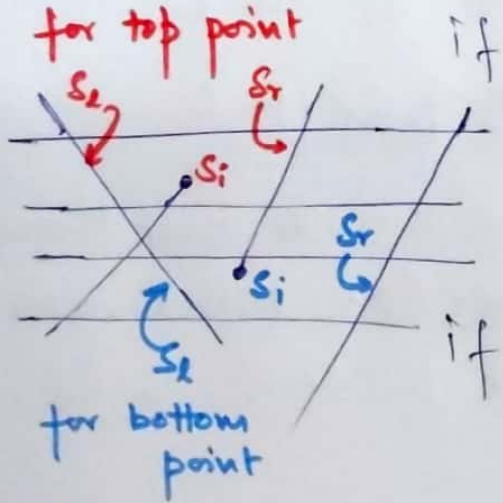
insert the intersection point into Q if it is not present in Q & below L

case (iii) q is intersection point between S_i and S_j

S_l = left of S_i and S_j in L

S_r = right of S_i and S_j in L

swap S_i & S_j in L & Report (q)



if (S_l, S_j) are intersecting then insert the intersection point into Q if it is not present in Q & below L

if (S_r, S_i) are intersecting then insert the intersection point into Q if it is not present in Q & below L.

TIME COMPLEXITY

$|Q| = 2n + k$

$2n$ endpoints of line segments

k is the no. of intersection points

k can be worst case $O(n^2)$

Hence, Ext.Min(Q)

takes $O(\log n)$ per extraction.

All elements in queue will be extracted once. Hence, total time complexity of Ext.Min(Q) = $O((2n+k)\log n)$

Case(i) is invoked n times (once for every line segment)

insert S_i into L takes $O(\log n)$ time

S_l and S_r can be found in $O(\log n)$ time

Finding intersection & inserting into Q takes $O(\log n)$ time

Hence, since it is invoked n times, total time complexity of topmost points becomes $O(n \log n)$.

By similar analysis, complexity of bottommost points becomes $O(n \log n)$.

Similarly, for intersection points, time complexity per intersection is $O(\log n)$

Hence, total time complexity becomes $O(k \log n)$.

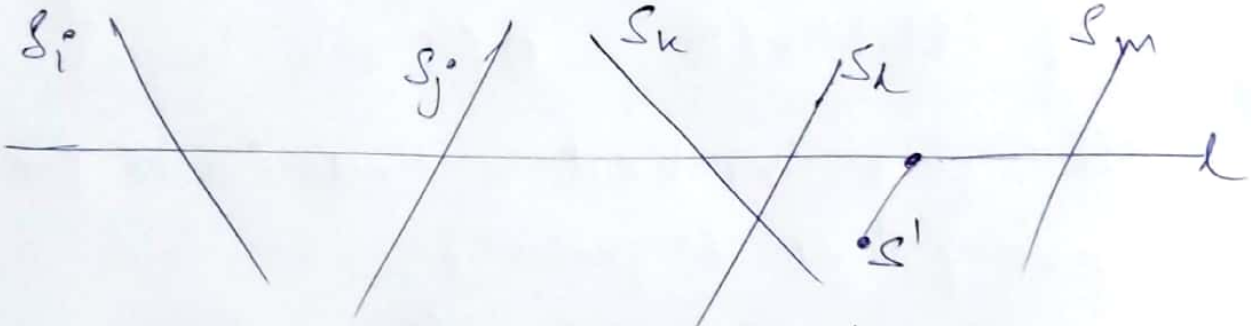
OVERALL TIME COMPLEXITY

(All steps combined)

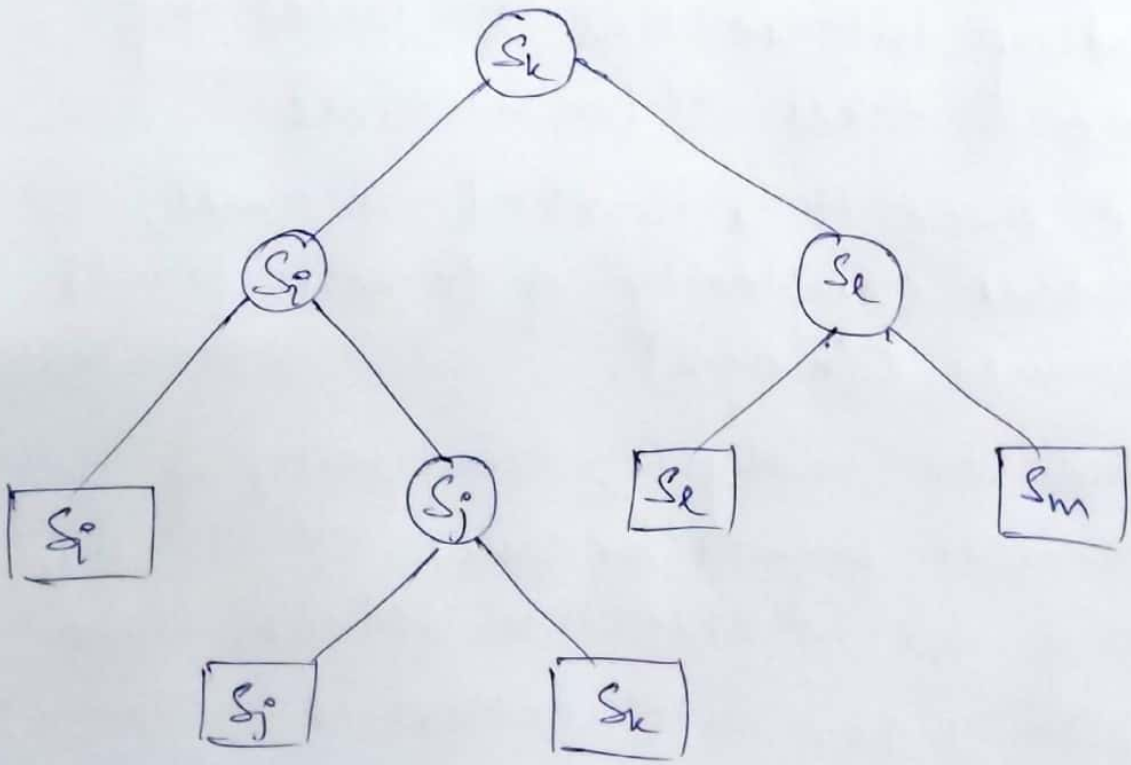
$= O((n+k) \log n)$

where k is output size.

DATA STRUCTURE REQUIRED FOR SWEEP LINE STATUS



y' = current y coordinate of sweep line



LINE SEGMENTS ARE STORED IN LEAF NODES. The non-leaf nodes contain values which are an aid to direction as to where to go (left or right) in order to find a line/segment.

Let's suppose we have to insert s'. (Refer to figure).

We find the value of s_k at y'. Let the point be (x_k, y').

We already know the x-coordinate of topmost point of s'.

Let that be x.

If x coordinate of current node is left of x, search on right child.

else, search on left child.

Proceed similarly on reaching a leaf node, check the value with leaf node value & make the left & right childs of that node accordingly.

The value of leaf node will be value of left child.

Balance the tree.

ENDING NOTE

Time complexity that we achieved is $O((n+k)\log n)$
However this is not optimal.
There is an algorithm with time complexity $O(n\log n + k)$. But in this, intersection points are outputted in arbitrary order, not from top to bottom, unlike the sweep line algorithm we discussed above.

NOTE: We looked at the simplest problem in Geographical Systems. But in actual scenario, there are regions not lines. They have some property associated to them. Many times we have to take intersection of these regions from 2 different layers & the intersection region gets its properties from both the regions. This problem can also be solved using sweep line algorithm.

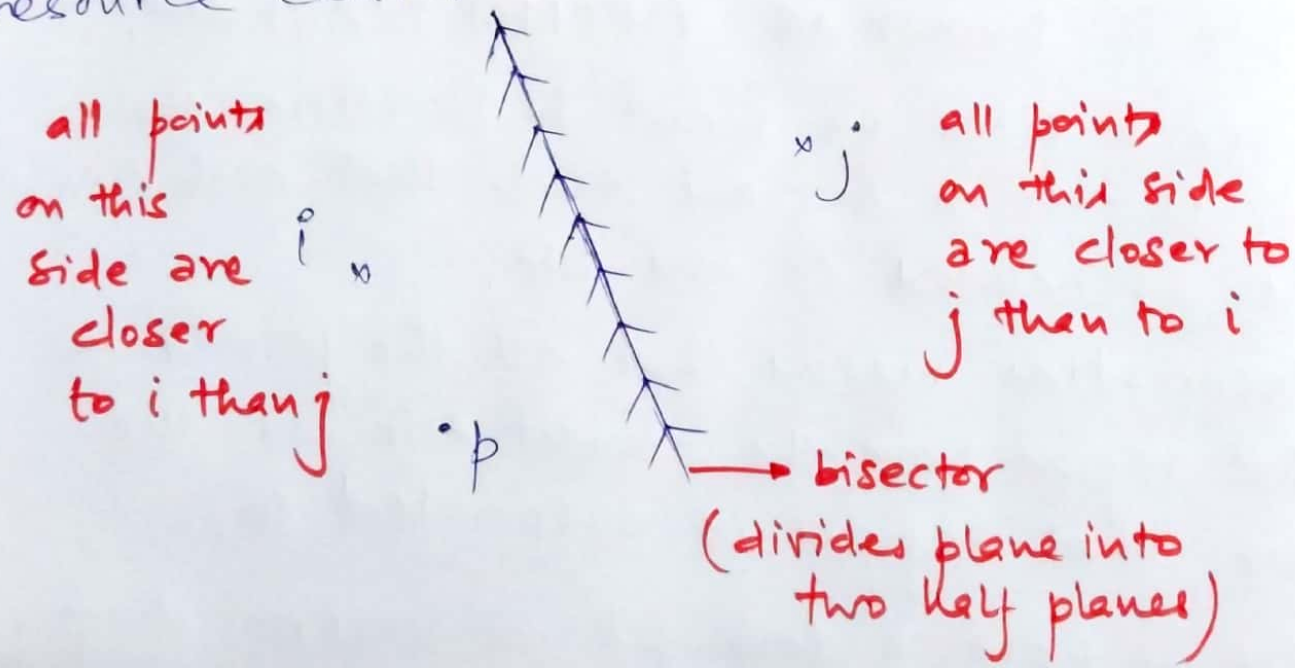
COMPUTATIONAL GEOMETRY

Lec 8

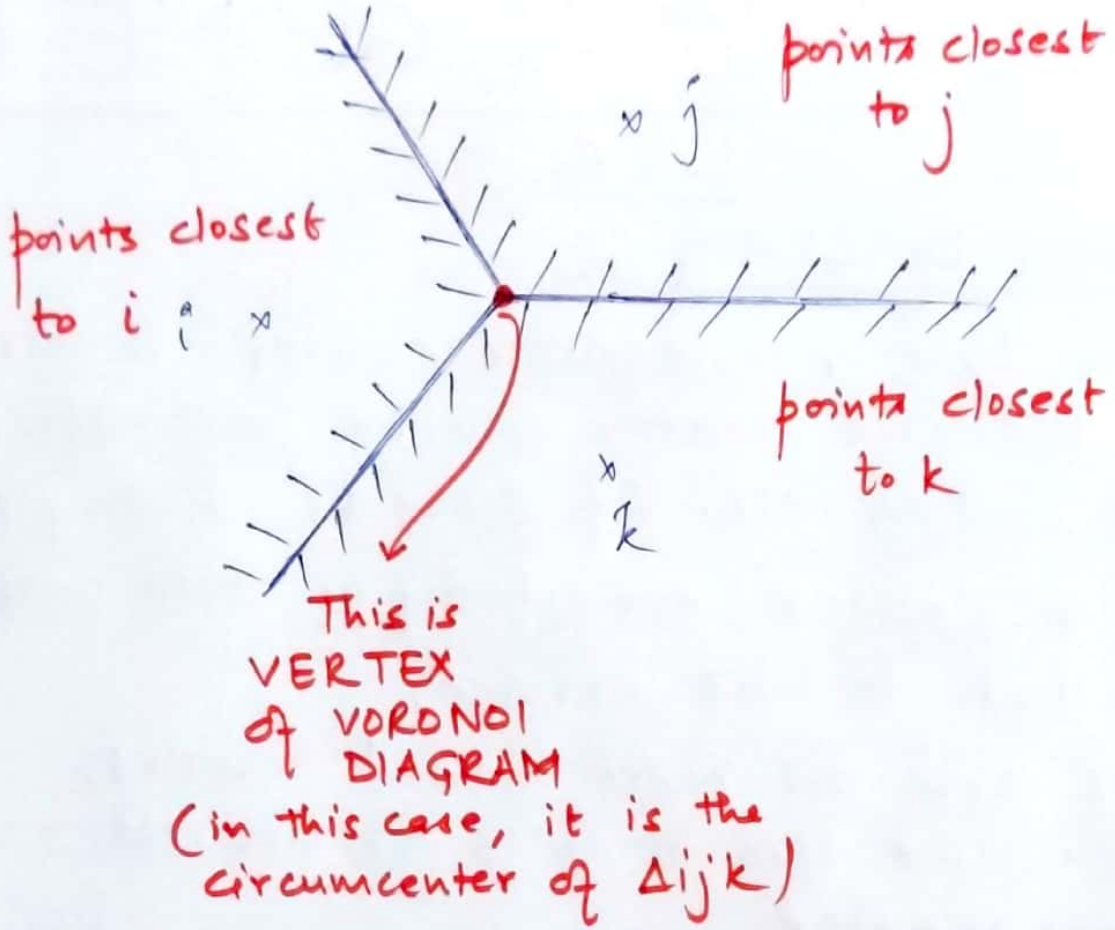
VORONOI DIAGRAM

Suppose there is a resource center in our city. Resource center can be post office, petrol station, etc. If there is a single resource center, everybody in town will visit that to take service.

But if there are more than 1 centers, people would like to go to the nearest resource center



But if there are more than 2 sites. Sites is same as resource centers.

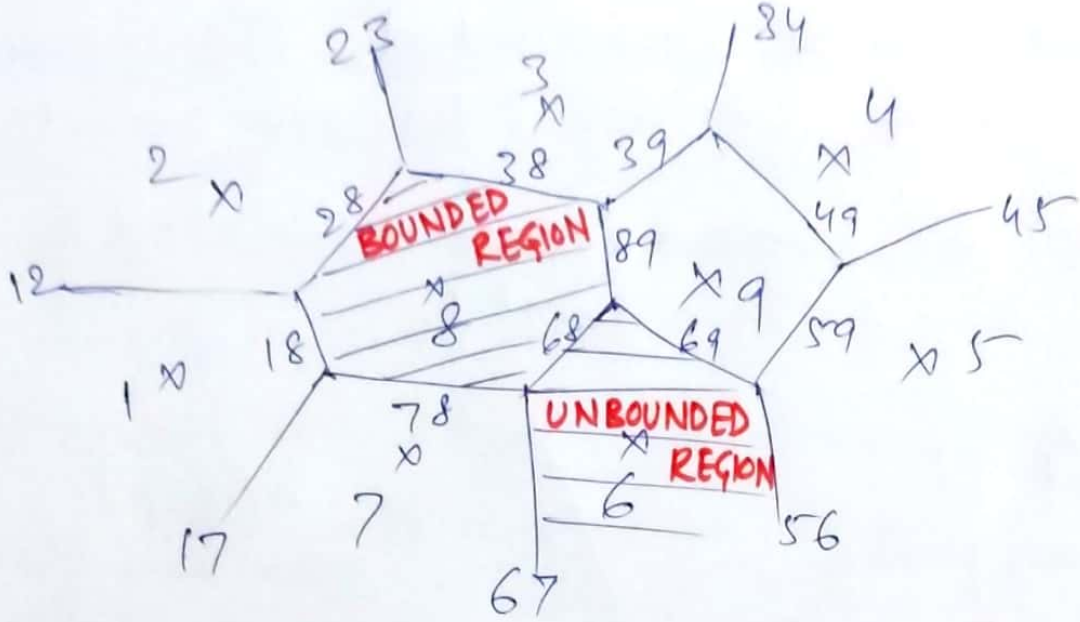


This is known as VORONOI DIAGRAM.

Given n sites, we want to partition the planes into n regions such that each region is associated to one site.

Association means that all the points in that region have the closest site as the one that region is associated to.

Let us have a look at a voronoi diagram with more points on next page



VORONOI DIAGRAM

If no 4 points are co circular, then each vertex is exactly defined by 3 points.

If query point

(i) lies inside region
closest point is site associated with that region

(ii) lies on bisector but isn't vertex
close to two points on whose bisector this point lies on.

(iii) is a vertex
close to 3 points which define that vertex

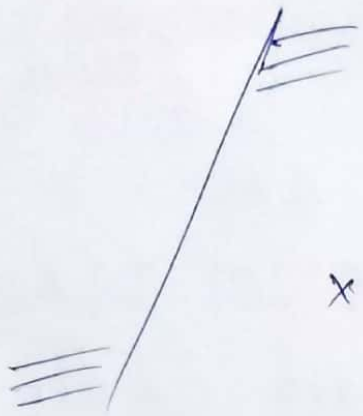
OBSERVATION

(1) Each voronoi region is convex.

(2) If a site lies on the convex hull of all these sites, the region associated

with that will be unbounded. The region associated to internal sites are bounded.

Vor(p_i) ⇐ voronoi region associated to point p_i



$h(p,q)$
open half plane on side of p made by bisector of p & q

$h(q,p)$
open half plane on side of q made by bisector of p & q.

$$P = \{p_1, p_2, \ldots, p_n\}$$

$$Vor(p_i) = \bigcap_{\substack{i \neq j \\ (1 \leq j \leq n)}} h(p_i, p_j)$$

Intersection of half planes.

NOTE:- We do not need all bisectors for defining the voronoi region. But the definition will still be valid.

For eg:-

In figure, Voronoi region of 9 can be defined by using

$$3\text{9}, 4\text{9}, 5\text{9}, 6\text{9}, 8\text{9}.$$

Involving other bisectors is redundant but the result will remain the same

even if we include them in calculation.

* Since every site is associated to one voronoi region, therefore no. of regions is linear in n (equal to n).

* VORONOI DIAGRAM is a planar graph.

These two facts enable us to put a bound on the number of edges & number of vertices.

$$n_e \leq 3n - 6$$
$$n_v \leq 2n - 5$$

→ linear in n

n_e → no. of edges

n_v → no. of vertices

Proof

For this, we use Euler's formula

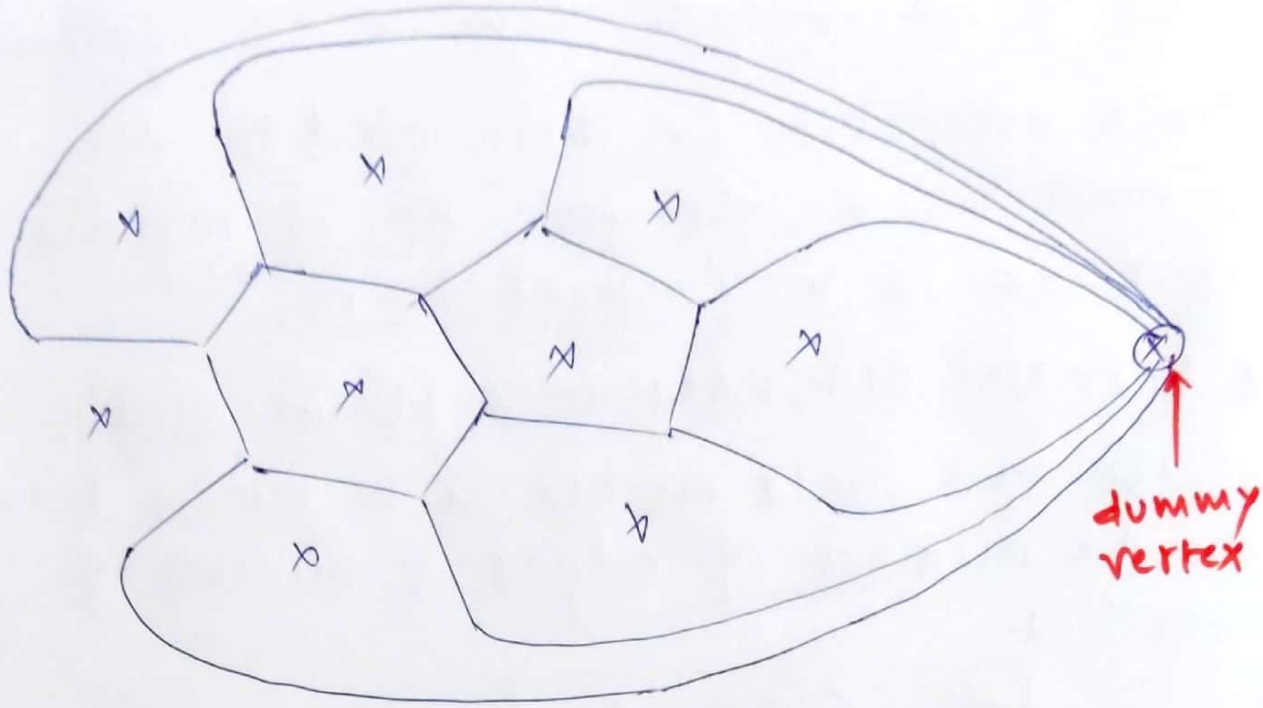
EULER'S FORMULA :-

$$n_v - n_e + n_f = 2$$

→ valid for all planar embedded graphs

The formula cannot be applied directly due to the presence of unbounded edges. (In a graph, every edge has 2 endpoints, but this is not true for unbounded edges).

To apply this formula here, we create a dummy vertex & connect all the unbounded edges to that vertex



number of regions remains same.
number of edges remains same.
no. of vertices increases by 1

$$n_v - n_e + n_f = 2$$

$$\Rightarrow (n_v + 1) - n_e + n = 2 \quad \text{——— eq (1)}$$

Sum up degrees of all vertices.
This will be equal to $2n_e$ since every edge is counted twice (once for every endpoint)

$$2n_e \geq \underbrace{3(n_v + 1)} \quad \text{——— eq (2)}$$

$\Rightarrow$ degree of any vertex is 3 except for dummy vertex which has a greater degree

$$2(n_v + 1) + 2n = 4 + 2n_e \geq 4 + 3(n_v + 1)$$

$$\Rightarrow 2 + 2n \geq n_v + 7$$

$$\Rightarrow \boxed{n_v \leq 2n - 5}$$

$$1 + n_e - n \leq 2n - 5$$

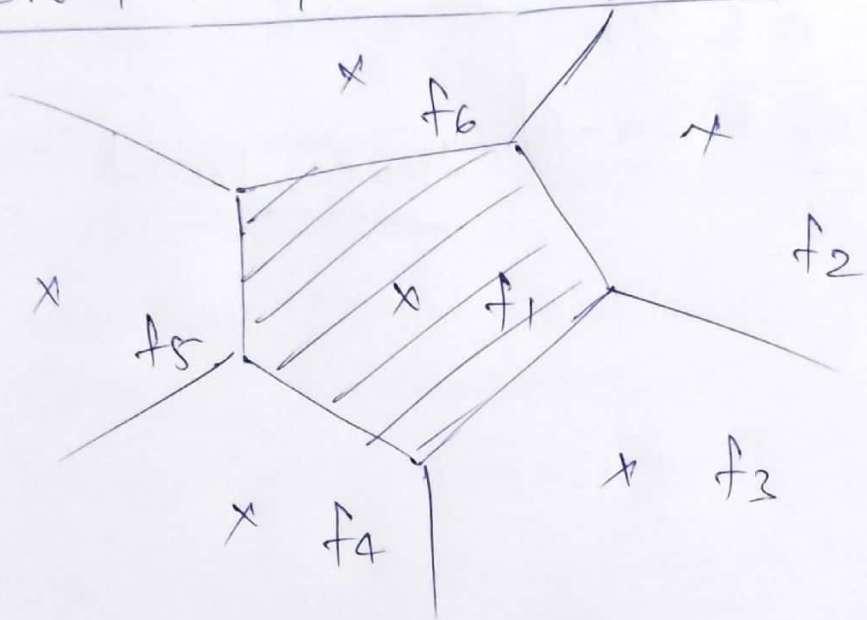
$$\Rightarrow \boxed{n_e \leq 3n - 6}$$

Hence, Proved

COMPUTATIONAL GEOMETRY

Lec 9

DATA STRUCTURE FOR STORING VORONOI DIAGRAM REGIONS



Voronoi diagram with one region bounded & all others unbounded.

Queries on Voronoi diagram

① Let us consider the closed region. We want to report the boundary of the face of this region.

② Given a vertex, we want to report all the edges incident ~~on~~ that ~~edge~~ vertex.

③ Given an edge, report all the faces incident on that edge.

Our data structure should support these operations. There is a Data structure called Doubly Connected Edge List (DCEL). It is not only useful for the case of Voronoi diagram, but can be used for any planar embedded graph.

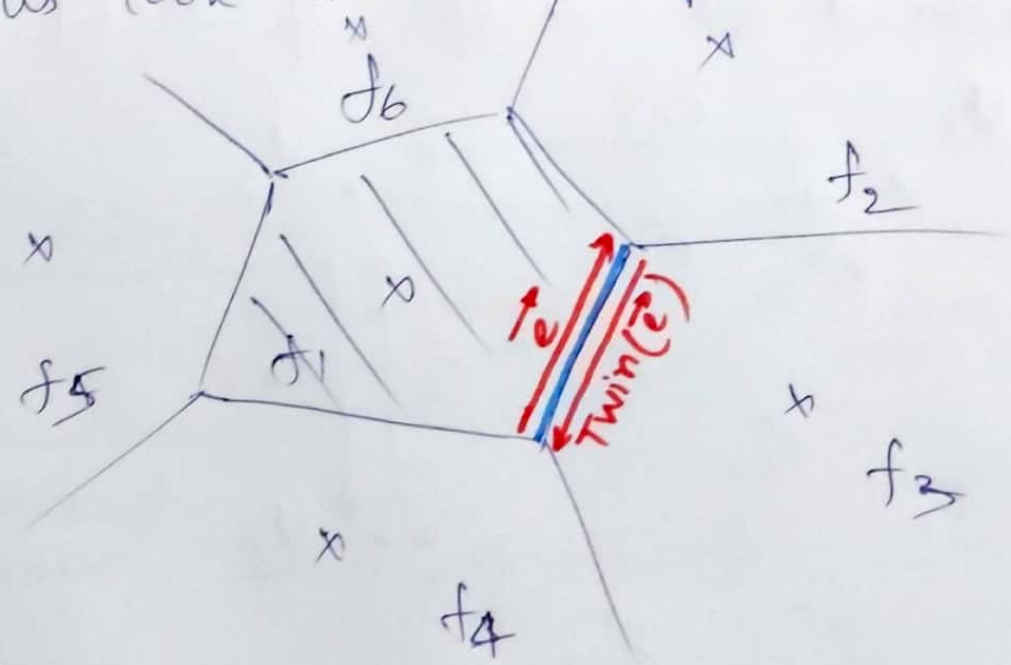
Voronoi diagram consists of 3 different types of objects

1. Vertices (0-dimensional)
2. Edges (1-dimensional)
3. Faces (2-dimensional)

In this DCEL, there will be 1 record for each of these entities.

We divide edges into 2 half edges.

Let us look at an example.

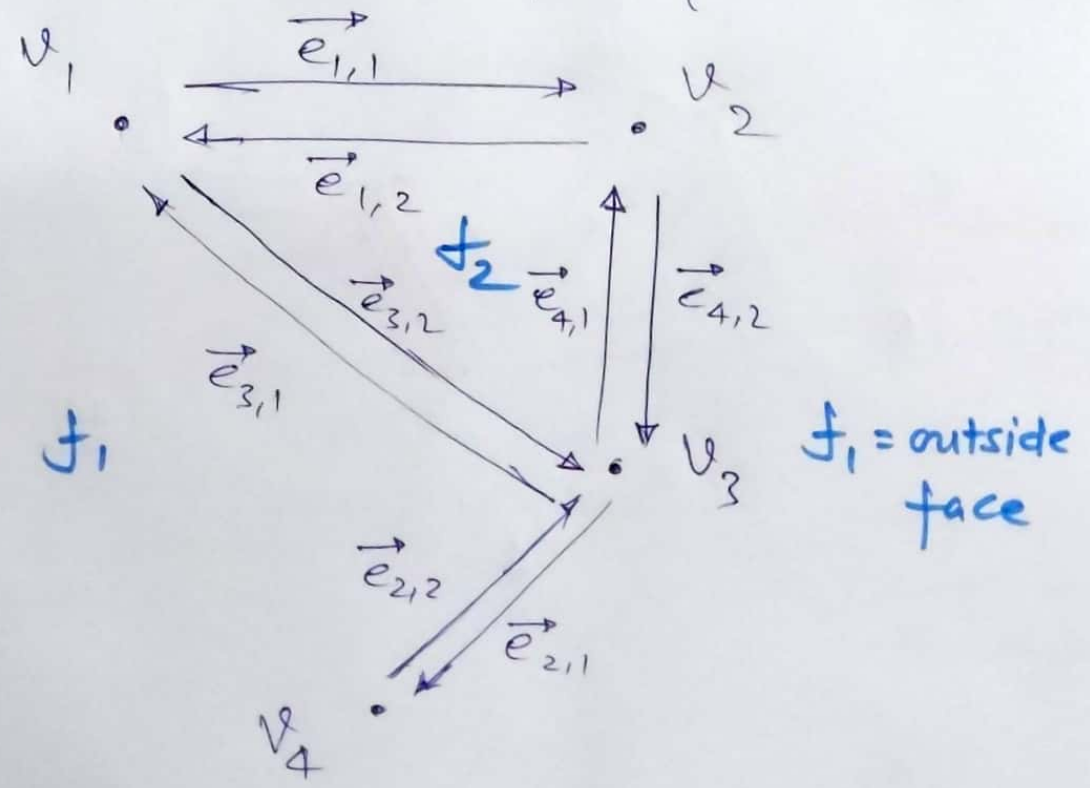


Let us look at the edge separating f_1 & f_3. We divide that edge into two edges, first is $\vec{e}$ & the other edge is called the twin of $\vec{e}$ denoted by $Twin(\vec{e})$.

Note:- The two vectors are Twins of each other, i.e., $Twin(Twin(\vec{e})) = \vec{e}$

This division into 2 half edges is not required. But for the sake of simplicity & to understand in a better way, we divide edges into 2 half edges. We assign directions to each of these half edges & store each half edge as one record in DCEL. Due to this, edge records become double the size.

We take a much simpler graph to understand the concepts of DCEL.



VERTEX RECORD

	Coordinates	Incident Edge (the source of this edge should be vertex)
v_1		$\overrightarrow{e_{1,1}}$
v_2		$\overrightarrow{e_{4,2}}$
v_3		$\overrightarrow{e_{2,1}}$
v_4		$\overrightarrow{e_{2,2}}$

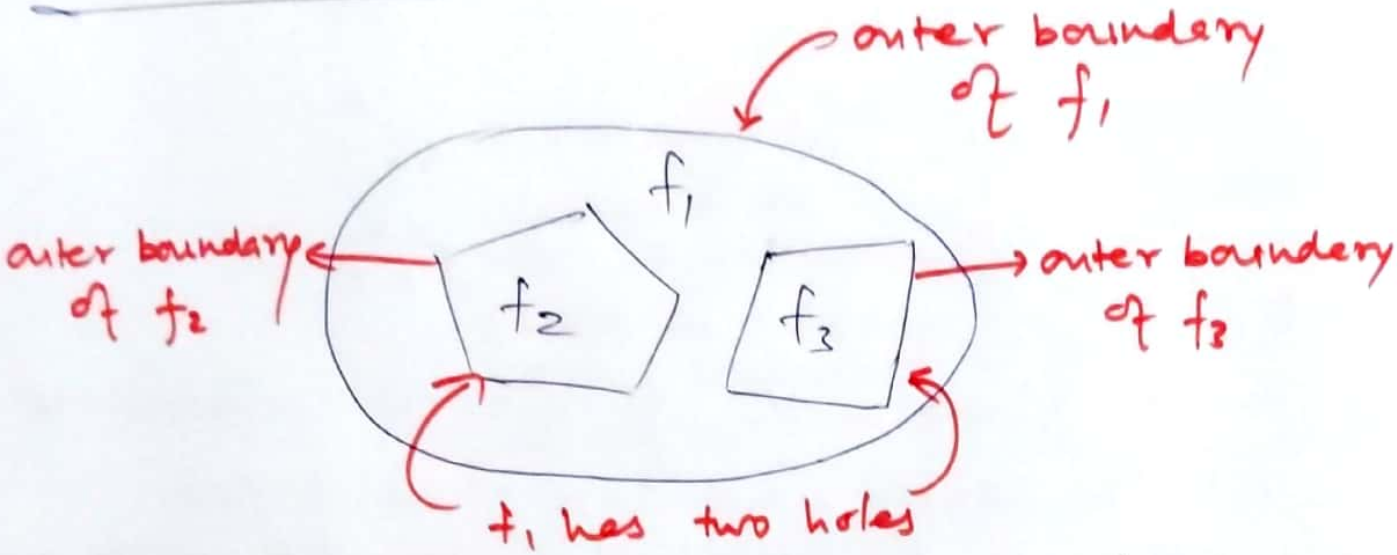
Of course, along with geometric information about each vertex, we can also store some important attributes for the vertices (eg. famous junction in a geographical information system). In our perspective, attributes aren't very important & do not play any role in the algorithmic aspect. Hence, we will omit attributes.

FACE RECORD

	Outer component (outer boundary)	Inner component (holes inside face)
f_1 (contains a hole in b/w & is unbounded)	nil (since it is unbounded)	$\overrightarrow{e_{1,1}}$ (here we store any edge on the inner component)
f_2	$\overrightarrow{e_{4,1}}$	nil

inner component ↑

Here, we choose $\vec{e}_{1,1}$ purposefully because we want f_1 to be on the left side of the chosen vector. Similarly for f_2.



For each inner boundary, we will have 1 edge in the inner component section of the face which contains the inner boundary as holes.
(Here, f_1 will contain 2 vectors in its inner component section).

Half edge record is shown on next page

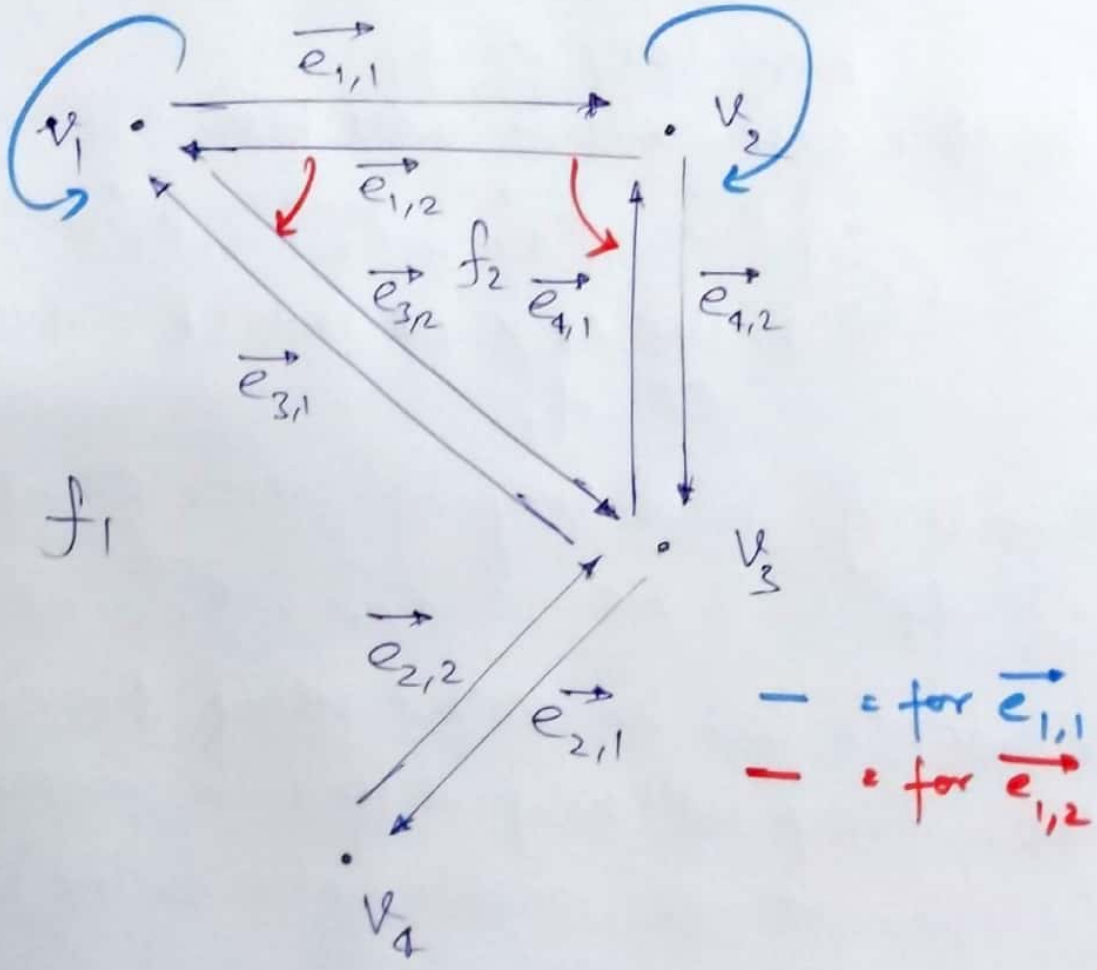
HALF EDGE RECORD

face on left side of the half edge

	origin	twin	incident face	next edge	previous edge
$\overrightarrow{e_{1,1}}$	v_1	$\overrightarrow{e_{1,2}}$	f_1	$\overrightarrow{e_{4,2}}$	$\overrightarrow{e_{3,1}}$
$\overrightarrow{e_{1,2}}$	v_2	$\overrightarrow{e_{1,1}}$	f_2	$\overrightarrow{e_{2,2}}$	$\overrightarrow{e_{4,1}}$

choose edge in clockwise dir^n at destination. Choose the twin edge having destination as origin

choose edge in counter clockwise dir^n at source. Choose the twin edge having source as ending point



There is one record for each vertex, each face, each half edge.

Suppose we want to output the boundary of f_1.

- Go to record of f_1
- outer component is nil
- Look at inner components
- $\overrightarrow{e_{1,1}}$ is stored
- output this & traverse over next edges until you get to initial half edge again
- The output will be inner boundary of f_1

In this case, output will be

$\{\overrightarrow{e_{1,1}}, \overrightarrow{e_{1,2}}, \overrightarrow{e_{2,1}}, \overrightarrow{e_{2,2}}, \overrightarrow{e_{3,1}}\}$

BOUNDARY OF INNER COMPONENT OF f_1

Similarly, boundary of outer component of f_2 will be $\{\overrightarrow{e_{4,1}}, \overrightarrow{e_{1,2}}, \overrightarrow{e_{3,2}}\}$

Boundaries can be output using previous edges too. They will only differ in orientation (clockwise or counter clockwise).

OUTPUTTING EDGES INCIDENT TO A VERTEX

- Go to vertex record
- Look at its incident edge
- output that
- Consider the twin edge of current edge. Make next of twin edge as current edge.
- Repeat till you return to orignal edge.

Ex:- V_3

output $\overrightarrow{e_{2,1}}$

Consider next of $\overrightarrow{e_{2,2}}$ which is $\overrightarrow{e_{3,1}}$

output $\overrightarrow{e_{3,1}}$

consider next of $\overrightarrow{e_{3,2}}$ which is $\overrightarrow{e_{4,1}}$

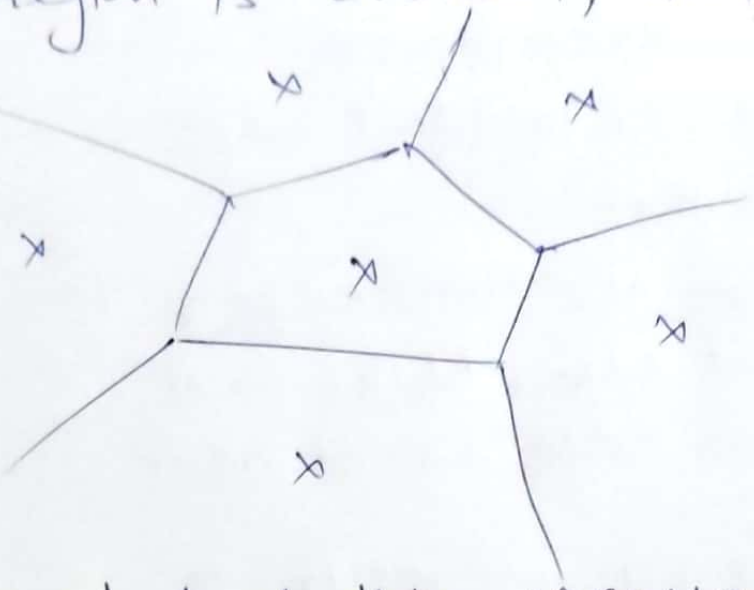
output $\overrightarrow{e_{4,1}}$

consider next of $\overrightarrow{e_{4,2}}$ which is $\overrightarrow{e_{2,1}}$

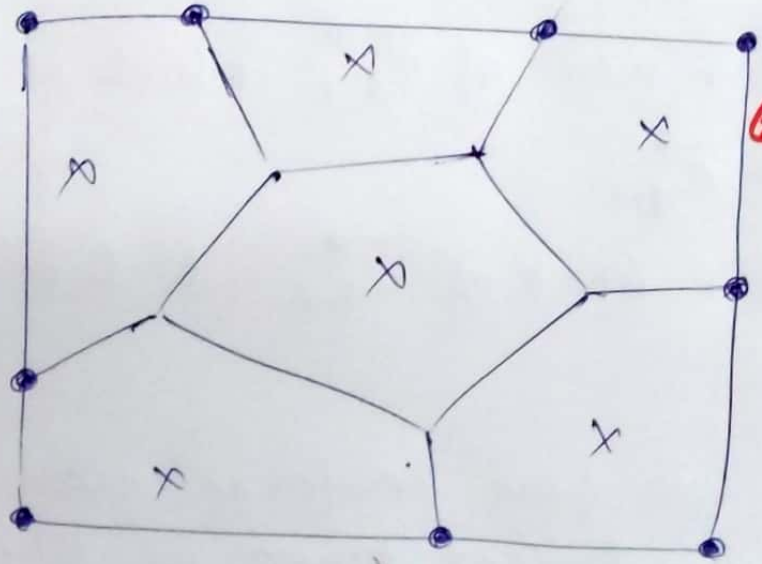
Halt.

Like this, we can perform many such operations in linear time in this planar st. line embedded graph using Doubly Connected Edge Lists.

NOTE:- DCEL is useful to store planar straight line embedded graphs where each region is bounded, except outer face



But if we look at this VORONOI DIAGRAM, there are many faces which are unbounded. DCEL cannot store this information. In this case, we will bound this voronoi diagram in a rectangular box & store its information.



In this case, all faces are bounded except for the outer face.

Some new vertices & edges are introduced. We keep them & store the information regarding this bounded voronoi diagram in Doubly Connected Edge List (DCEL).

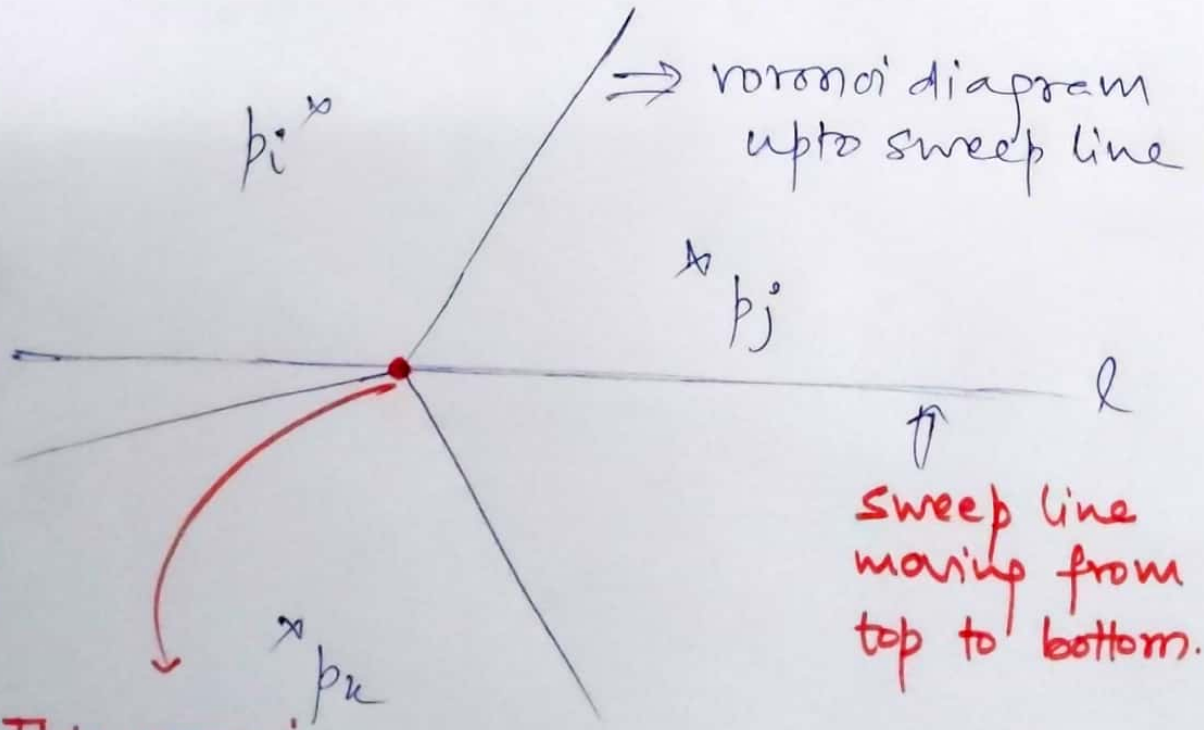
COMPUTATIONAL GEOMETRY

Lec 10

SWEEP LINE ALGORITHM FOR VORONOI DIAGRAM

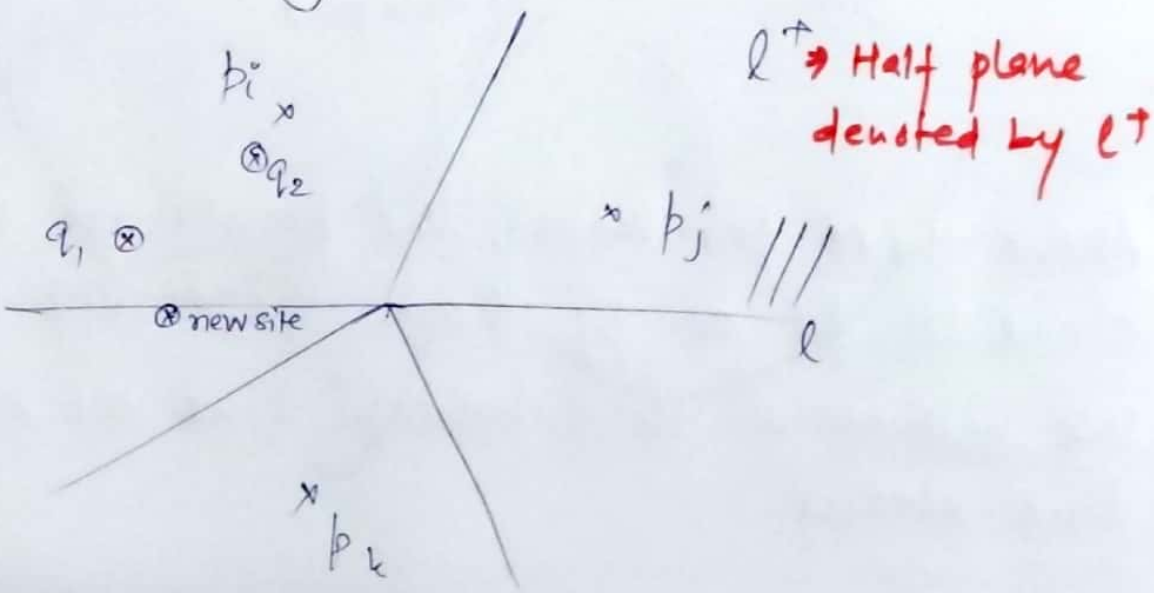
Whenever the sweep line reaches upto a position, we want to construct voronoi diagram upto that position & as a sweep line status, we want to maintain the intersections b/w voronoi diagram & current position of sweep line.

Constructing voronoi diagram upto a position is not easy.



But we do not have any info on p_k to find out this voronoi vertex.
The voronoi diagram upto sweep line also depends on future info, which is yet to be seen. Hence, it is not possible to construct voronoi diagram upto sweep line.

Hence, we cannot apply standard sweep line algorithm in the case of voronoi diagram computation.
Now that raises a question.
Is it possible to find a voronoi diagram for the points above l, which doesn't depend on any site below l.

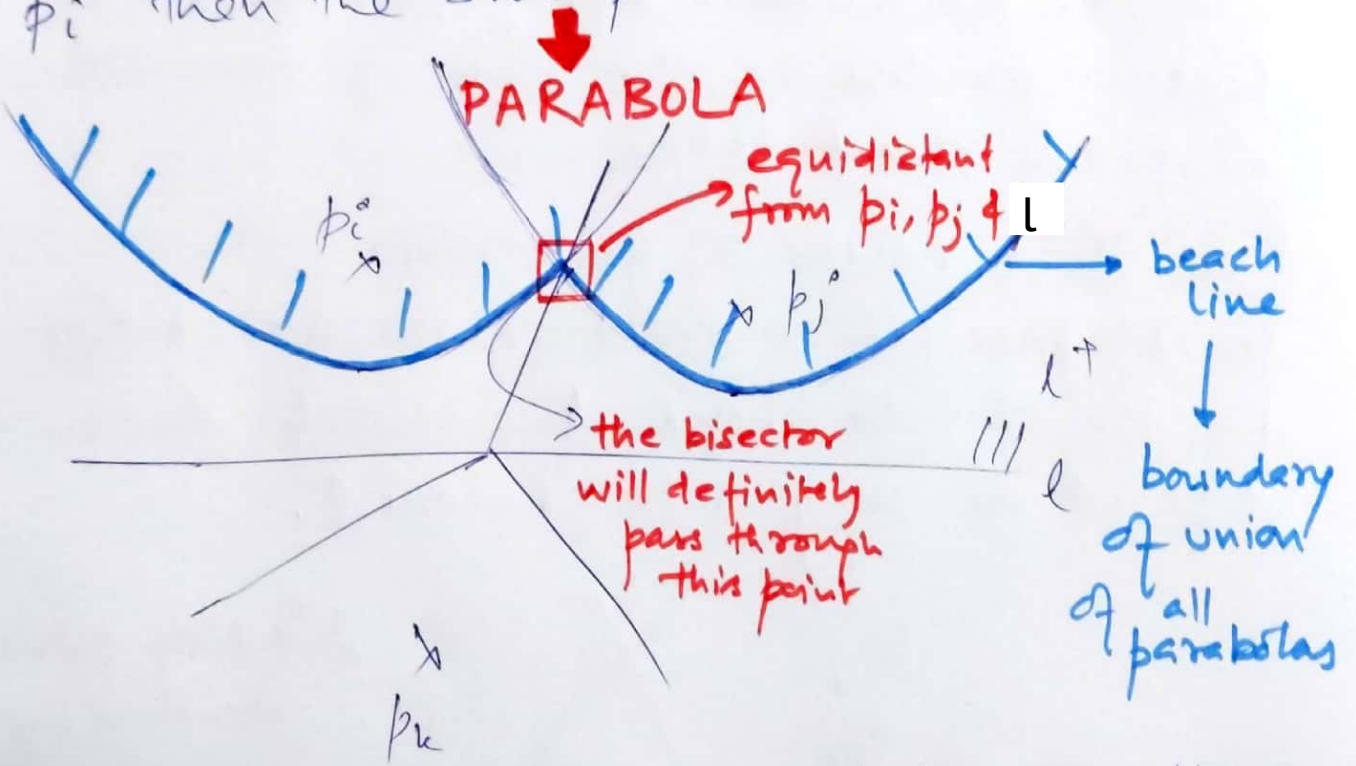


In this diagram, if there exists a site just below l, then q_1, which was closest to p_i till now will become closer to newsite.

But for q_2, even if a new site is present just below the sweep line, it will still be closest to p_i. Hence, it will not change as encountering any new sites below l irrespective of its position.

$\Rightarrow q_2$ surely belongs to $Vor(p_i)$

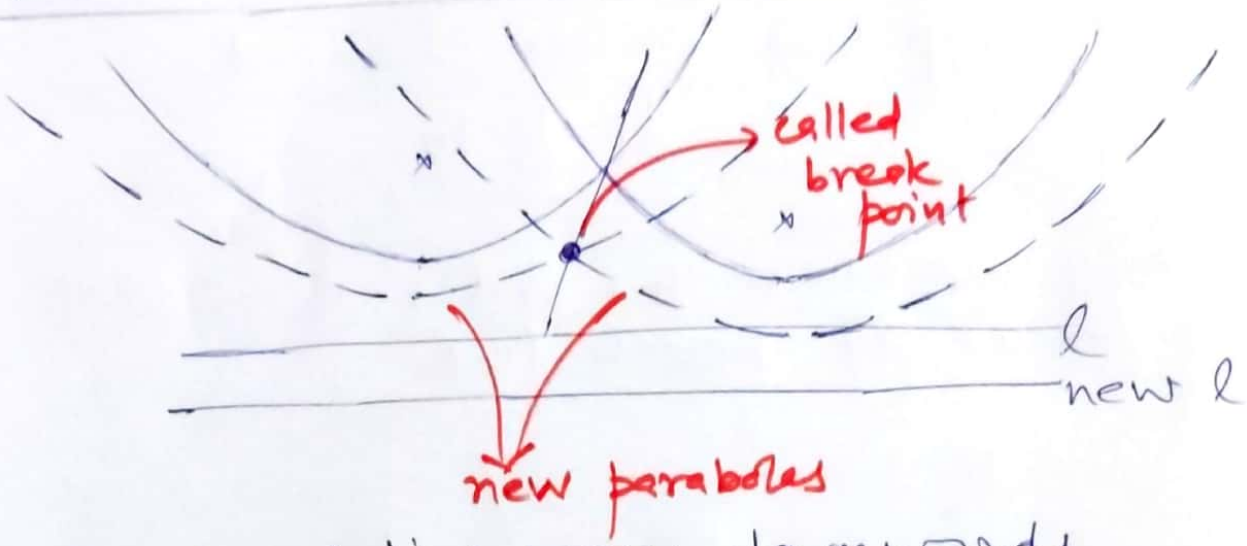
Locus of all such points which are closer to p_i than the sweep line.



Above these parabolas, the points are either close to p_i or p_j than the sweep line.

We maintain this beach line as sweep line status

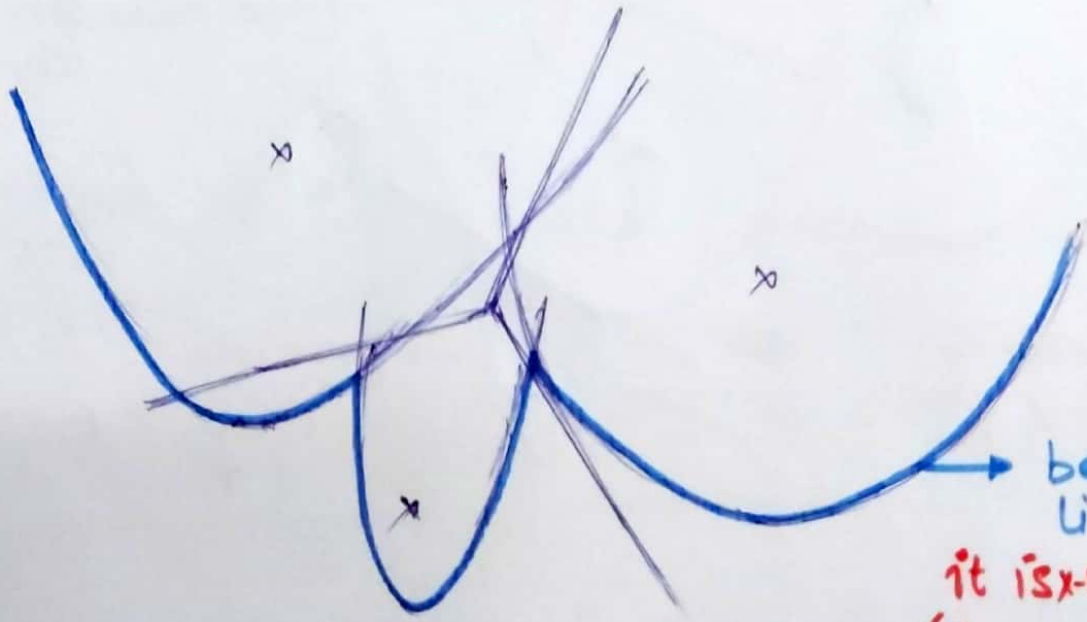
EFFECT OF SWEEP LINE ON PARABOLAS



As the sweep line moves downwards, parabolas expand & the intersection point b/w the parabolas traces out the voronoi edge.

This keeps on happening until structure of beach line changes

topology

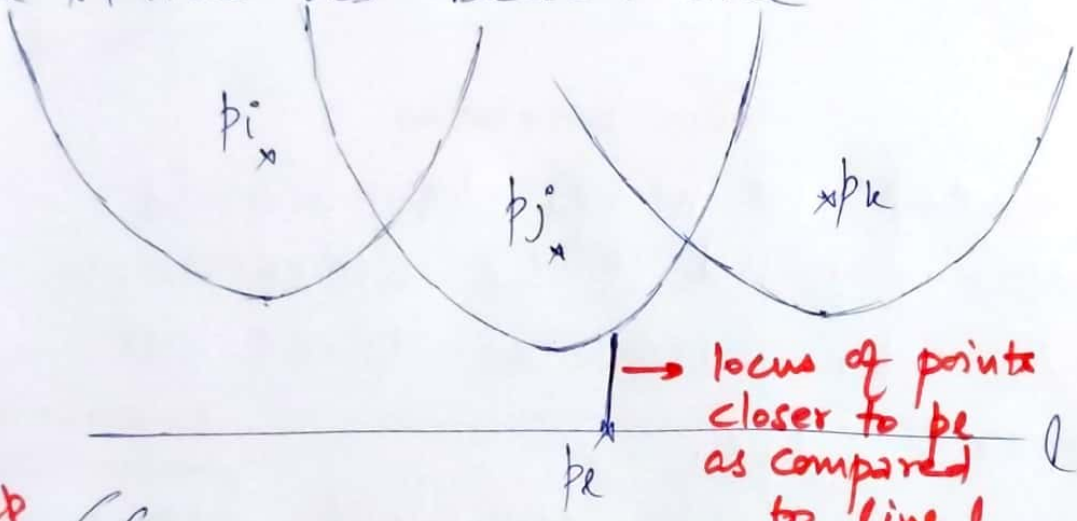


it is x-monotone (for each x value, there will only be 1 point)

Now question is where does this beach line / sweep line status changed.

p_i p_j p_k

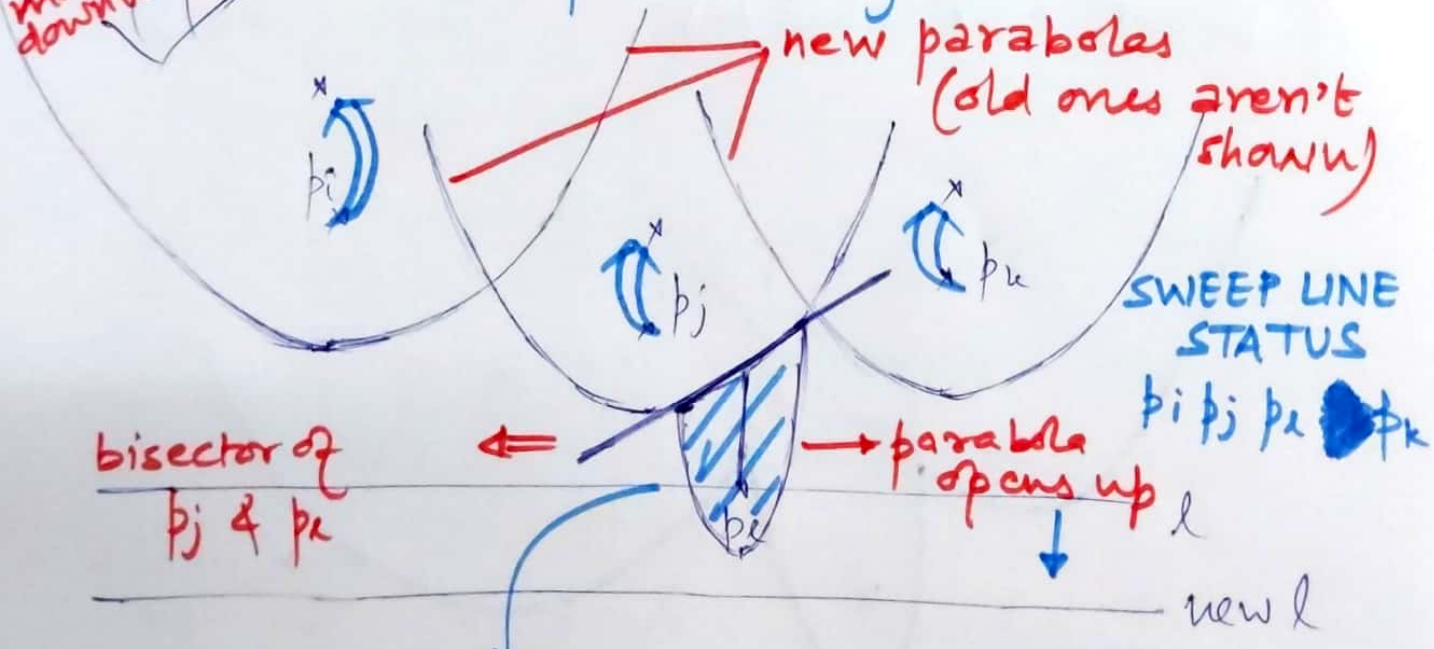
we only maintain this order in sweep line status as beach line



sweep line moves downward

when l passes through site

new parabolas (old ones aren't shown)



for all points inside this region, the closest site is p_k regardless of points above or below the line l.

The break point of parabolas of p_j & p_i trace out the bisector of p_j & p_i. Initially, the parabola was a vertical line & slowly expanded to become a real parabola. The endpoints move in opposite directions slowly tracing out the bisector of p_j & p_i.

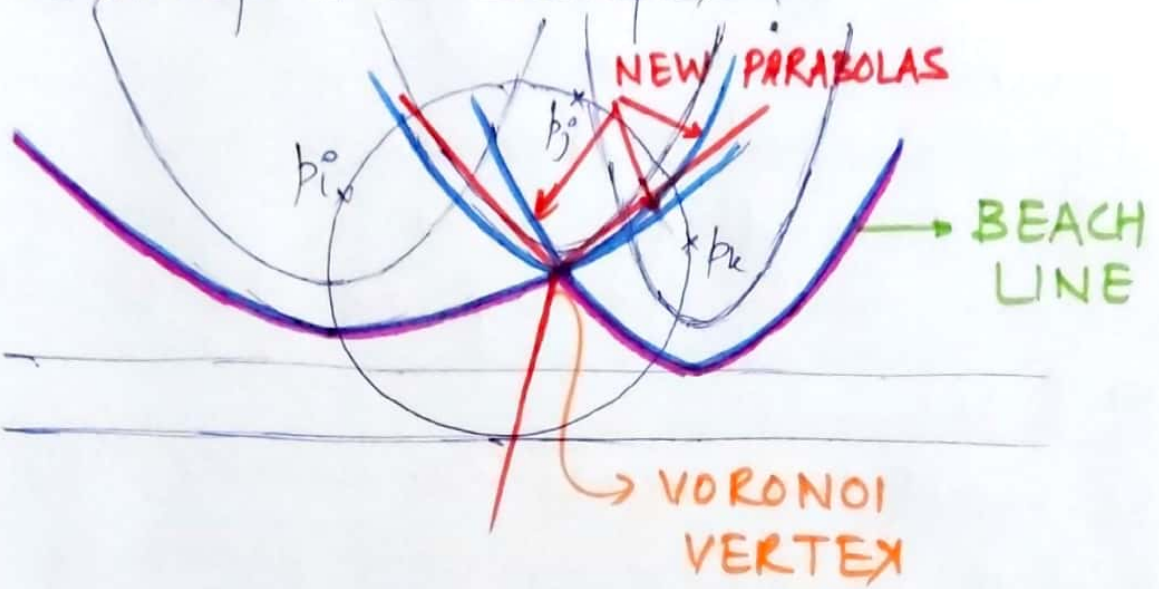
As the sweep line moves downwards, the parabola of p_i intersects with parabola of p_k & that end starts tracing out the bisector of p_i & p_k.

So this is how a new parabola enters the sweep line status (or beach line)

THIS EVENT IS KNOWN AS "SITE EVENT".

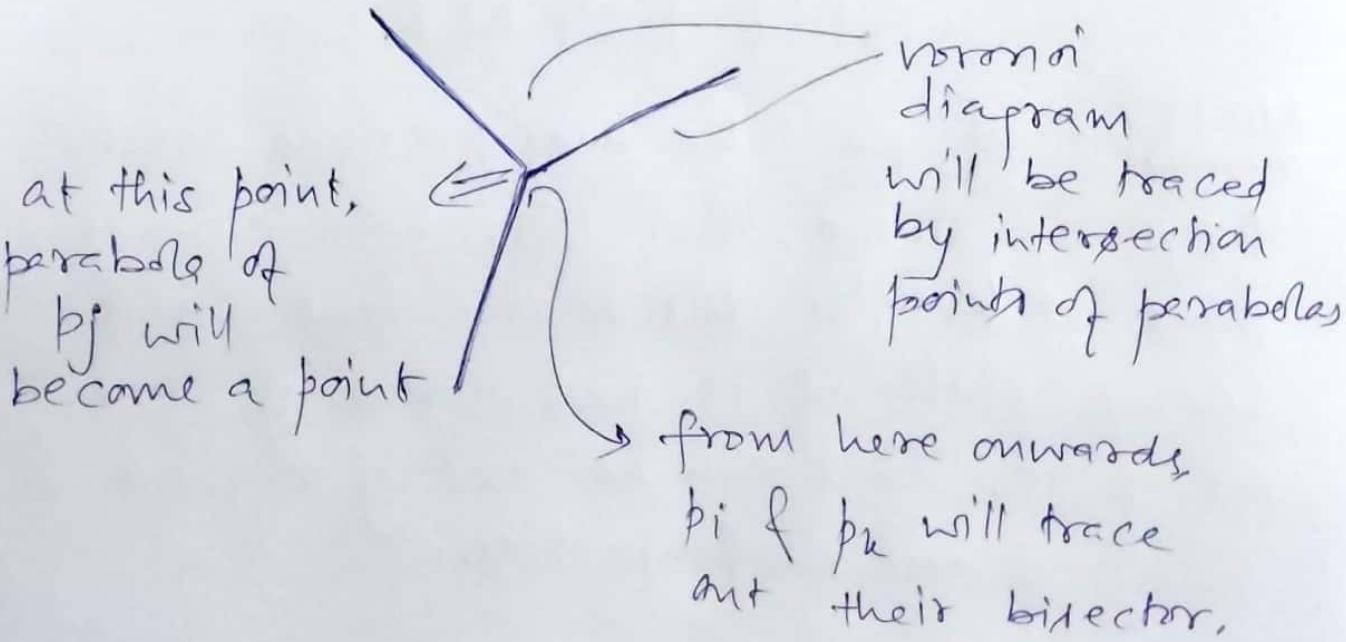
NOTE:- If you take any previous parabola that was above, that parabola cannot come back into this beach line in a future state if it has already gone out of it. This can be easily argued using basic geometric properties.

Now, the question is where else does the sweep line changes?



PARABOLA OF p_j GOES OUT OF BEACH LINE

As the line moves downwards, parabola corresponding to p_j diminishes slowly, becomes a point and then vanishes from the beach line.



A parabola exits from the sweep line status when we encounter a voronoi vertex. This happens when sweep line is a tangent to a circle passing through p_i, p_j & p_k.

Bottommost point of circle is the event at which parabola corresponding to p_j vanishes from ~~beach~~ beach line.

THIS IS CALLED "CIRCLE EVENT"

Once we have events information (Site events & circle events), then we need to figure out how to maintain the sweep line status & how to find out circle events on the fly & how to modify & update the data structures corresponding to voronoi diagram.

We needs

1. Data Structure for constructing voronoi diagram
2. for sweep line status
3. for event Queue

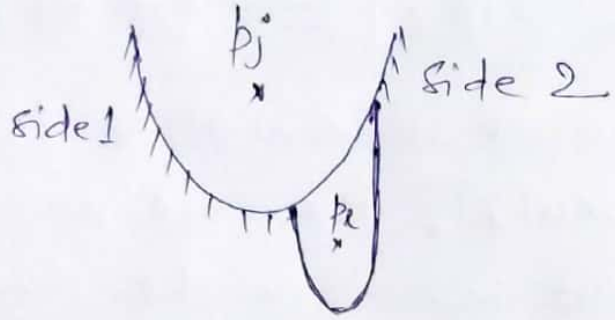
Once we figure out what are the events, how to maintain them is the issue.

New parabola can only enter through site event.

Circle event only occurs when we encounter a voronoi vertex.

Size of beach line $\leq 2n-1$

In the worst case, every parabola will have two sides except one.



2 sides of parabola pj

Hence, size of beach line is linear.

This algorithm is known as FORTUNE'S algorithm.

Earlier, people used Divide & Conquer to construct Voronoi Diagrams. It takes $O(n \log n)$ time.

Fortune's algorithm is much easier to implement having same time complexity of $O(n \log n)$

COMPUTATIONAL GEOMETRY

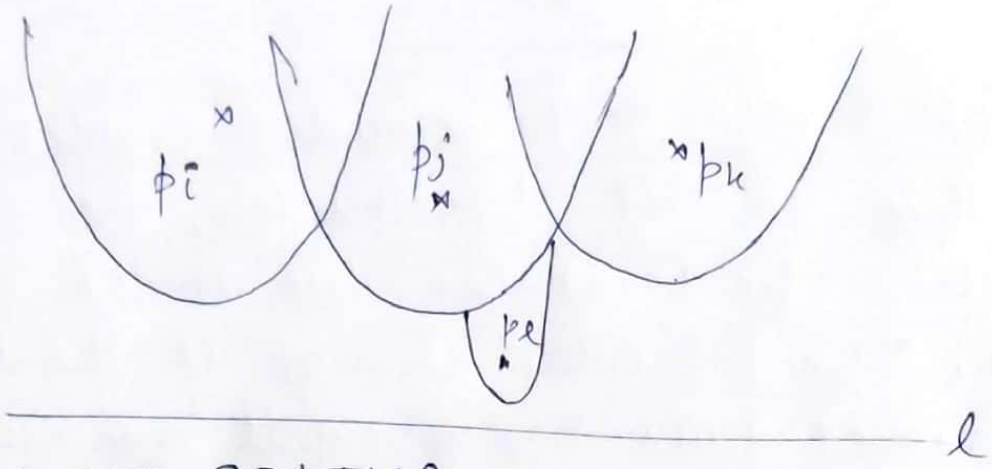
Lec 11

NOTE:- Similar to the standard sweep line algorithm, in this algorithm too, we cannot maintain the parabolas due to the same issue. The parabolas change if the sweep line moves down by a small amount. Hence, we maintain the list of points instead & we can build the parabolas from this list very quickly.

NOTE:- We know all the site events beforehand. Whenever a new site is encountered, a new parabola is added. This also marks the formation of a new Voronoi ~~diagram~~ edge. This new Voronoi edge is added to the DCEL.

NOTE:- Every 3 consecutive parabolas may define a circle event. Take the 3 adjacent parabolas & corresponding sites & look at the circle passing through these 3 sites. The bottommost point of the circle is a circle event.
If this point is below the current position of sweep line, there is a possibility that this circle event takes place later.

Hence, we insert this into the event Queue.



SWEEP LINE STATUS

p_i p_j p_l p_k

The different consecutive triplets in this sweep line status are

(p_i p_j p_l

p_j p_l p_j → not a valid circle event because the 3 sites are not distinct.

(p_l p_j p_k

These two are a possibility for future circle event

We have to check for the candidate triplets if the bottommost point of the circle passing through them is below the current position of the sweep line. If yes, add this to the event Queue. This is how we discover new events on the fly & insert into event Queue.

At a circle event, the middle parabola vanishes & we get a voronoi vertex. We need to update the DCEL for this voronoi vertex accordingly. Also, we need to update the endpoints of bisector of (p_i & p_j) & (p_j & p_k). We also need to take into account the start of a new bisector of p_i & p_k & update the DCEL accordingly. When p_j vanishes, p_i & p_k become adjacent. Let p_1 be the parabola to the left of p_i & p_2 be the parabola to the right of p_k. Two new triplets are formed:-

$p_1\ p_i\ p_k$ } we need to check
$p_i\ p_k\ p_2$ } for circle event & update accordingly.

NOTE:- Before removing p_j, there were three triplets.

$p_1\ p_i\ p_j$ ($p_i\ p_j\ p_k$) $p_j\ p_k\ p_2$

In our example, we took action for this triplet.

After removal of p_j,

these two are not a valid triplet anymore. So, the circle event corresponding to these 2 triplets

cannot take place because p_j is no more in the sweep line status.

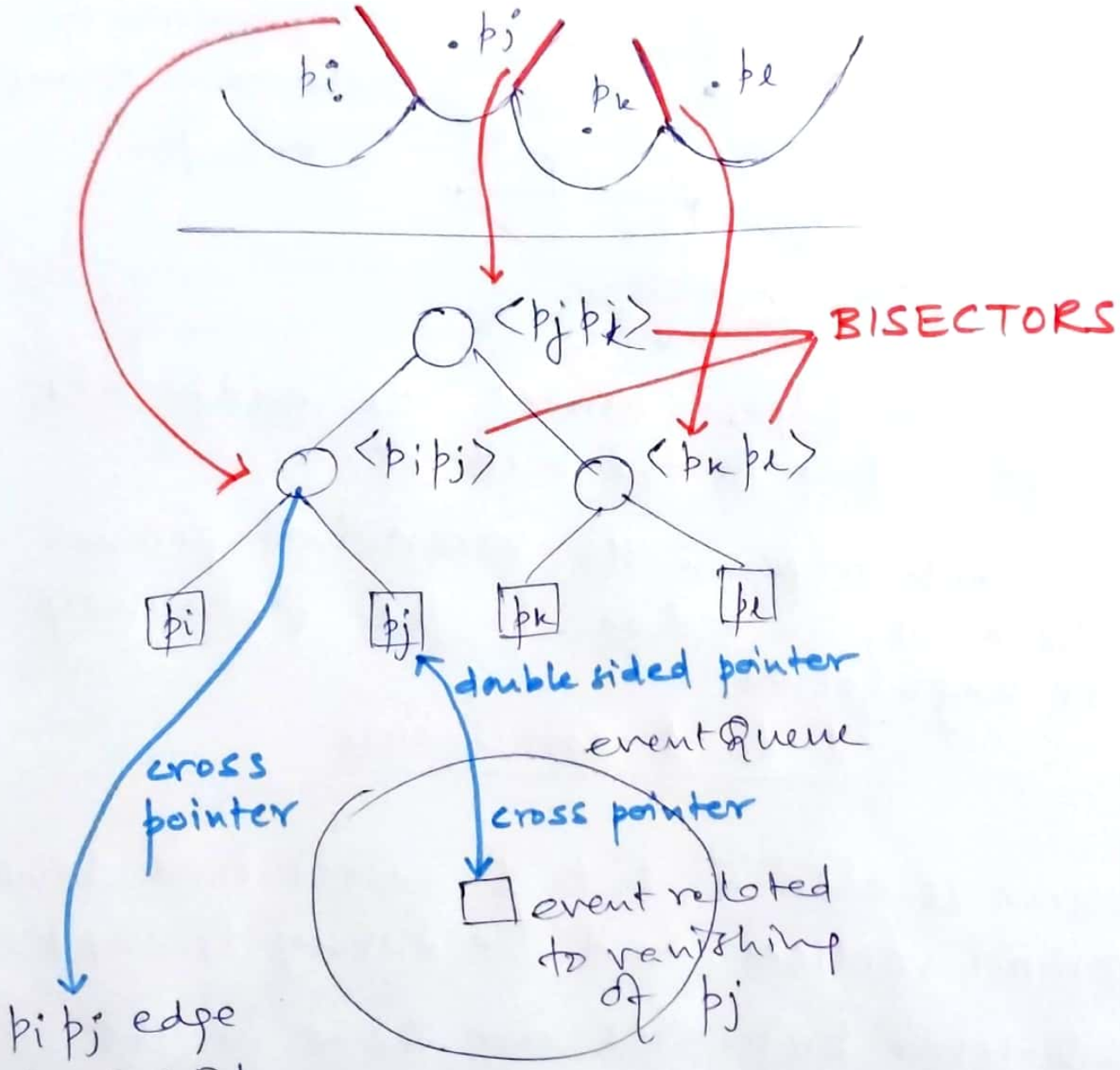
Hence, if there is an event corresponding to these 2 triplets in the event Queue, it needs to be removed.

Action of site event & circle event takes logarithmic time. Sweep line status uses a height balanced BST to store its data. event Queue is a normal priority queue. We want to do all operations in logarithmic time. Here, we also have to maintain cross pointers b/w the sweep line status & event Queue.

Whenever p_i is stored in the leaf node of sweep line status, we have to maintain a pointer from that node to (p_1, p_i, p_j) event in event Queue since this event is based on vanishing of p_i.

Similarly for p_j p_k p_2.

When we encounter a voronoi vertex, we need to update endpoints of the edges in DCEL. Hence, we need to maintain cross pointers from sweep line status to DCEL so that we can update the endpoints quickly.



Using this setup, all actions can be realized in logarithmic time

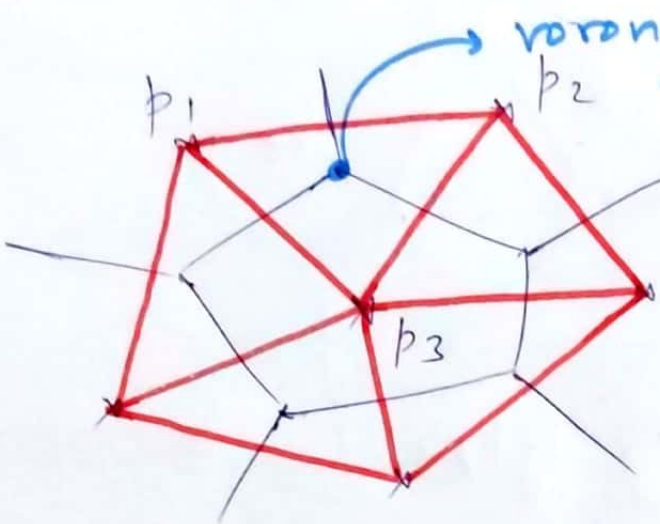
no. of edges = linear

no. of vertices / voronoi diagram = linear

at each event, logarithmic time is spent

Hence, overall time complexity = $O(n \log n)$

DELAUNAY TRIANGULATION



For each edge, connect the endpoints corresponding to the bisector.

For each edge in the voronoi diagram, there is an edge in the Delaunay triangulation.

1 to 1 association

There is also a 1 to 1 association b/w voronoi vertices and Delaunay triangle.

* Whatever properties are there in the Delaunay triangulation, are also there in the Voronoi diagram.

EUCLIDEAN MINIMUM SPANNING TREE

Vertices $V_1, V_2, \ldots, V_n$ ↑ points in a plane

All edges in the graph ($O(n^2)$) Complete graph

$wt(V_1, V_2) = dist(V_1, V_2)$

If such a graph is given & we want to find minimum spanning tree, then it is called Euclidean Minimum Spanning Tree.

EMST $\subseteq$ DT

Euclidean Minimum Spanning Tree is a subgraph of Delaunay triangulation

Delaunay triangulation → Can find in $O(n \log n)$ time

has linear no. of edges → use Delaunay triangulation to find EMST

There are many other graphs
- Nearest Neighbourhood Graph
- Gabriel Graph
- Relative Neighbourhood Graph
- Closest Pair

All these graphs are subset of Delaunay triangulation.

All these graphs & Delaunay triangulation are used in many applications where proximity (closeness) is required.

NOTE:- In Linux, there are some softwares namely qhull, geomview, rbox

⇓

program which computes Delaunay triangulation in any dimension.

⇓

can generate the output file compatible with geomview & use geomview to watch the output visually.

rbox is used to generate input in many ways.

① evenly distributed points.

② points within a rectangular box

③ points within a unit circle.

Specify number & any dimension.

((can use this input to run qhull & then see output in geomview.

2D points ⟶ 3D point (x, y, x^2+y^2)

⇓

project this in 2D ⟸ find convex hull in 3D

⇓

result will be Delaunay triangulation.

Convex hull in 3D has relation with Delaunay triangulation.

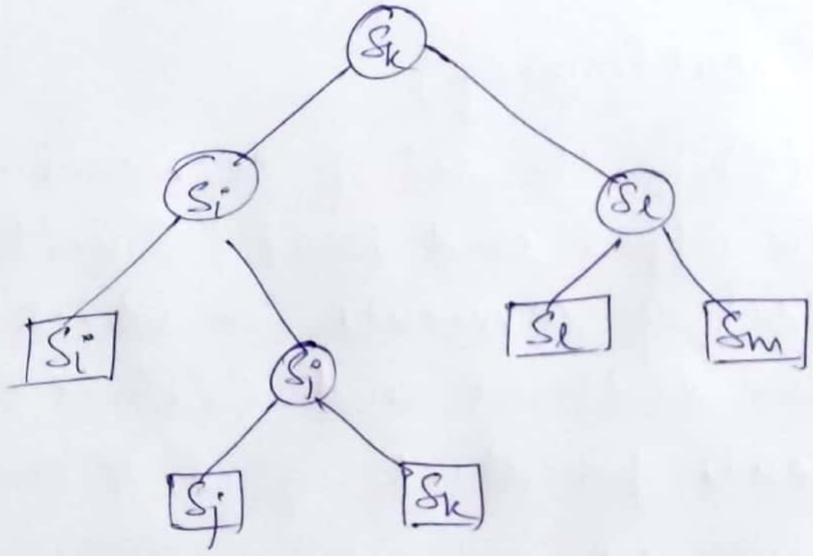
qhull uses this result to compute convex hull, delaunay triangulation, voronoi diagram

EXTRA (FROM BOOK)

Data structure for Sweep Line Status

We use a self balancing BST (AVL Tree or Red Black Tree). For demonstration, we will store all the segments only in the leaves & the nodes will only guide us to the right leaf/leaf spot. A node will store the rightmost leaf segment of its left subtree

For example, the sweep line drawn at the end of Lec 7 can be represented as



NOTE:-
However, the leaves are not needed. The segments can also be stored intrinsically in a balanced BST. This is just to make things easier.

Getting rid of assumptions

We have assumed a bunch of assumptions

like no two lines should share an endpoint etc. We could get rid of these assumptions by modifying the algo a bit.

(I) FindIntersections(S) {S is input set of line segments}
Q = ϕ {Q is min-heap representing eventQueue}
Insert endpoints into Q. for every upper endpoint, also store the segment corresponding to this endpoint
L = ϕ {Balanced BST for sweep line status}
while Q ≠ ϕ, do
q = ExtractMin(Q)
HandleEventPoint(q)

(II) HandleEventPoint(q)
1) Let U(q) be the set of segments for which q is an upper end point. Similarly, L(q) is the set of segments for which q is a lower endpoint and C(q) is the set of segments for which q is a centre point.
2) If U(q) ∪ L(q) ∪ C(q) contains more than one point, then Report q along with U(q) ∪ L(q) ∪ C(q)
3) Delete L(q) ∪ C(q) from L and insert C(q) ∪ U(q) into L such that the order of segments in L is corresponding to a left or right order just below q.

4) If $U(q) \cup C(q) = \phi$ then

4.1) s_l = rightmost segment left of q at same y coordinate

4.2) s_r = leftmost segment right of q at same y coordinate

4.3) Find New Event(s_l, s_r, p)

5) Else

s' = leftmost segment of $U(p) \cup C(p)$ in L

s_l = left neighbour of s' in L

Find New Event (s_l, s', p)

s'' = rightmost segment of $U(p) \cup C(p)$ in L

s_r = Right neigbour of s'' in L

Find New Event(s'', s_r, p)

III Find New Event(s_l, s_r, p)

If s_l and s_r intersect below the sweep line, or on it and to the left of p, and the intersection is not yet present in Q, then

⟶ Insert intersection point of s_l and s_r in Q.

Some notes about time complexity

The time complexity of above algorithm
$= O((n+I)\log n)$
where I is the number of intersection points

→ It is easy to come to $O((n+k)\log n)$ where k is the output size (intersection points and segments intersecting at those points)

→ For a point q, let $m(q)$ be the cardinality of $L(q) \cup U(q) \cup C(q)$. Let $m = \sum m(q)$ for all event points q. It is easy to see that $m = O(n+k)$ and
Time complexity $= O(m \log n)$

To prove:- $m = O(n+I)$
Consider the set of segments as a planar graph where all event points are vertices & edges are parts of the segments connecting these vertices.
Number of vertices $n_v \leq 2n+I$. $m(p)$ is bounded by the degree of vertex corresponding to p. Every edge contributes to the degree of at most two vertices $\Rightarrow m = \sum m(q) \leq 2n_e$

for a planar graph:-
1) Every face of the graph is bounded by at least three edges (if there are at least 3 segments)
2) An edge can bound at most two different faces

$\Rightarrow n_f \leq \frac{2n_e}{3}$

Euler's formula for a planar graph:-

$$n_v - n_e + n_f \geq 2$$

$\Rightarrow 2 \leq (2n+I) - n_e + \frac{2n_e}{3} = 2n + I - \frac{n_e}{3}$

$\Rightarrow n_e \leq 6n + 3I - 6.$

Since $m = O(n_e)$ and $n_e = O(n + I)$

$\Rightarrow m = O(n+I)$

Hence, Proved

TIME COMPLEXITY = $O((n+I)\log n)$