Computational Geometry

Lecture - 13, 14
Tarun Garg - 2009MCS2085 [*]

September 3, 2010

1 Lower Bounds For Geometric Problems

1.1 Element Uniqueness Problem

Given n elements$(x_1, x_2, ..., x_n)$. Determine if all of them are unique or there is a repetition. It is a kind of decision problem because here we just want to know whether all the elements are unique or not irrespective of that which elements are repeated. Decision problems have answers only "yes" or "no".

Suppose $\forall i, x_i \in \{1, 2, ..., n\}$. So by using the array of size n (Indirect addressing or Hashing) we can solve this problem in $O(n)$ time. Similarly if $\forall i, x_i \in \{1, 2, ..., n^c\}$ then also we can solve this problem in time $O(n)$ using Radix Sort, where c is constant.

But if c is not constant or $\forall i, x_i \in \mathbb{R}$(set of real numbers), this problem can be solved by sorting the n elements in time $O(n \log n)$. So Element Uniqeness problem is reducible to Sorting problem. Hence lower bound of Element Uniqueness problem applies to Sorting.

Lower bound of sorting assumes a certain model of computation. This model is Comparision Tree model. Each node in the tree corresponds to a comparision($<$ *or* $\geq$). Each leaf node corresponds to a specific ordering of comparision

$\Rightarrow$ Number of leaves in tree $\geq n!$

$\Rightarrow$ Length of the longest path in tree $\geq \log_2(n!) \approx \Omega(n \log n)$

A simple comparision is like whether $(x_i - x_j) \geq$ (0 or not). But a general inequality could be $a_1x_i + a_2x_j + b \geq$ (0 or not).

Model of computation for element uniqueness problem is Linear Decision Tree. Each node corresponds to some linear inequality. Each leaf node is attached with either "yes" or "no" answer, depending upon all the elements are unique or not.

In element uniqueness problem we are given a set of n elements. Consider this set as a single point in n-demensional space i.e. $\mathbb{R}^n$. Let some subset $W \subset \mathbb{R}^n$ contains all the "yes" answers and the complement $\mathbb{R}^n - W$ contains all the "no" answers. So this complement set $\mathbb{R}^n - W$ is set of all hyperplanes defined by $x_i = x_j$ but $i \neq j$, where x_i and x_j are i^{th} and j^{th} axis of the n-demensional space. Hence if any point lies on any of these hyperplanes then answer is "no"

[*]Department of Computer Science and Engineering, IIT Delhi, New Delhi 1100116, India.

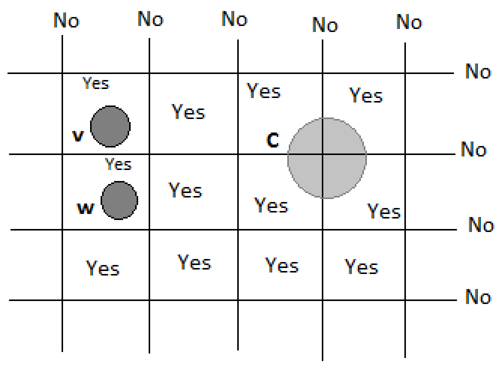
otherwise answer is “yes”.

Consider a 3D space containing x, y and z axis. Here 3 hyperplanes($x = y, y = z$ and $z = x$) contain all the “no” answers. These 3 planes divide the space $\mathbb{R}^3$ into 6 regions(because $x = y$ and $y = z$ so it follows $x = z$ that means $x = z$ has to pass through the intersection of $x = y$ and $y = z$). Each region corresponds to one of the $3! = 6$ ordering of the 3 elements.

If we consider n-demensional space then there will we $n!$ regions corresponding to each permutation. Each of these regions contains only “yes” answers. So here is a important observation about linear decision tree:

Given any input point $p \in \mathbb{R}^n$, we follow some path from the root node to some leaf node $\Rightarrow$ Any decision tree also implies a partitioning of the $\mathbb{R}^n$ in the following way: “Each node (leaf/internal) t corresponds to some subset $w(t) \subset \mathbb{R}^n$, namely all points that pass through t”. It means each leaf node corresponds to some partitioning of the $\mathbb{R}^n$.

Now recall that if any leaf node corrsponds to “yes” answer then it must lie on one of the the $n!$ regions and if it corresponds to “no” answer then it must lie on one of the hyperplanes. Consider the problem space prtitionig as shown in figure:

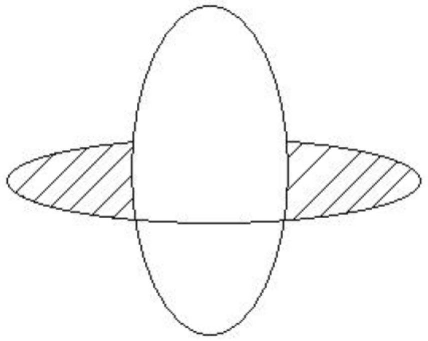


Here each straight line corresponds to one of the hyperplanes. Regions created by these lines containes all the “yes” answers and lines corresponds to the “no” answers. Now a leaf node can not correspond to cirlcle “c” as shown in figure because a leaf node can not contain both “yes” and “no” answers. Even a leaf can not be a combination of circle “v” and “w”(contains only “yes” answers) because in Linear Decision Tree every node is associated with a convex region(intersection of linear inequalities). Thus every leaf node is associated to only one region because of the convexity property. According to the above discussion the number of leaf node must exceed the number of connected components of the solution space which in $n!$.
$\Rightarrow$ Runnung time or height of this linear decision tree is $\Omega(\log w)$. Where w is number of connected components in the solution space.

Therefore any element uniqueness algorithm using linear decision tree takes atleast $O(n \log n)$ comparisions.

Let cosider this equation $\prod_{i \neq j}(x_i - x_j) = 0$ iff answer is “no”. If we use this kind of computaion model then we do not need to compute any linear inequalities. Therefore our lower bounds may change according the model we are using. So lower bounds need to be proved corresponding to the models.

Higher Degree Polynomial Inequalities: Consider a two degree polynomial inequality like this $\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq r^2$. If we have two such kind of inequalities then their intersection will be like as shown in figure:



Here intersection is not a single connected component. Therefore a leaf of the linear decision tree may belong to more than one region. So we can can not make simple argument that number of leaf node must exceed the number of connected components of the solution space. Hence the previously discussed lower bound does not hold here. There is a relation between degree of inequalities and number of connected components corresponding to a node given by Milnor Thom Bounds.

1.2 Application to Convex Hull

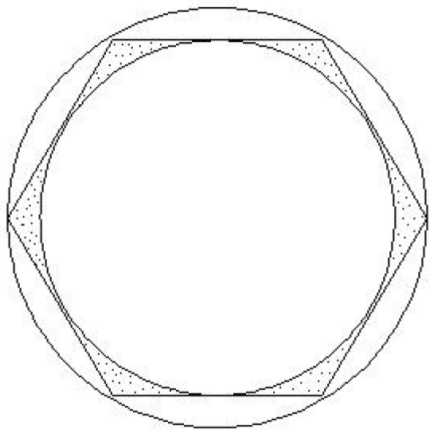
If we want the corner points of convex hull in sorted order then sorting can be reduced to convex hull, therefore lower bound of sorting is followed by convex hull problem. But here are some relaxed version of the convex hull problem whose lower bounds may differ:

1. Enumerate (in any order) the corner points of convex hull.
2. Given n points, do all points appear on the convex hull. This problem is a kind of Decision Problem.

Version of convex hull problem that takes into account the output size:

Problem 1: Given a set of n points on a plane and a number $h \leq n$, are there exactly h points on the convex hull?

Problem 2: Cosider the figure as shown:



Here is a regular h-gon, a circle inscribed in the h-gon say C_I and another circle circumscribing the h-gon say C_C. Now fix h points to be on the regular h-gon and the remaining n-h points are choosen arbitrarily in the annular region between C_I and C_C. Are there h points on the boundary of the convex hull?

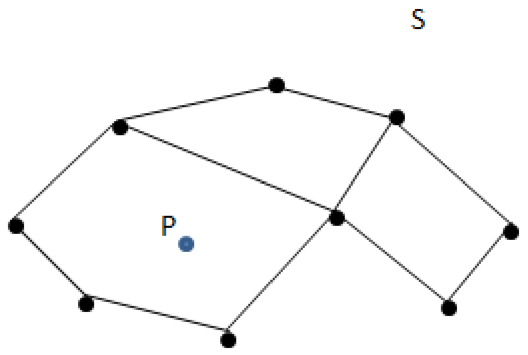
Here all n-h points should lie in the wedges between h-gon and C_I otherwise it will not be a convex polygon. So number of ways to distribute the n-h points in h wedges are h^{n-h}. Hence lower bound for this problem is $\log(h^{n-h})$ or $(n-h)\log h$.

Claim : Problem 2 can be reduced to problem 1 in linear time because for each point we have to just check whether it lies between C_I and C_C or not which is a contant time operation. Therefore the lower bound of Problem 2 also applies to Problem 1. So Problem 1 also has the lower bound $(n-h)\log h$.

2 Point Location Problem

A planar subdivision is a partition of plane into polygonal regions by a finite collection of line segments whose pairwise intersections are restricted to segment end-points.

Point Location Problem: Given a planar subdivision S and a query point P, determine which region of S contains P.

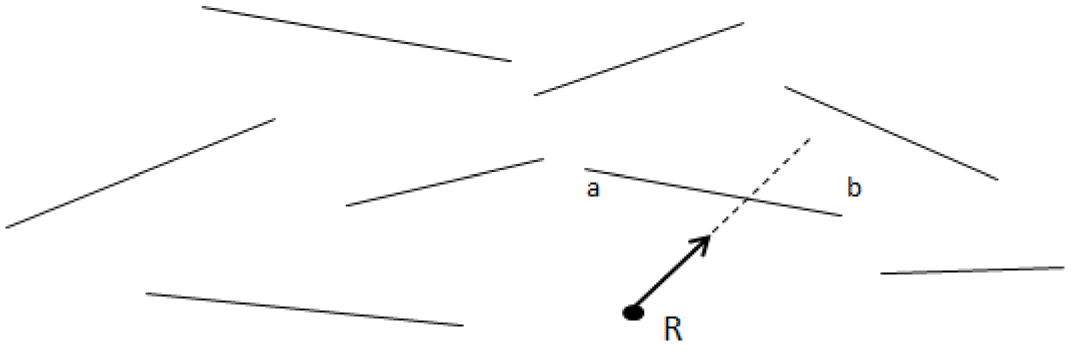


To solve this problem we need to build a data structure that supports queries of the following kind "for a point $P = (x', y')$, which region contains P". We can make simple assumption that P does not lies on any line segment.

Here each region is a simple polygon(not necessarily convex). If we represent each region separately then finding regions adjacent to a given region will be $O(n)$ time query which is not desirable. We want to answer this kind of query in time O(number of adjacent regions). For this we can simply maintain Doubly Connected Edge List(DCEL). In this structure we store oriented edges around the vertex. Each vertex stores all the edges that incident on it in a counter clockwise manner.

Similar problem: Ray Shooting: Given a set of line segments(non-intercesting). Build a data structure to support queries of the following form:

"If we shoot a ray, which line segment does it hit first."



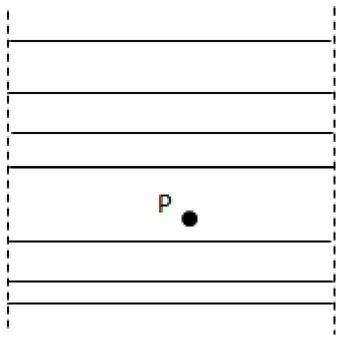
In figure if we shoot a ray from a point R as shown in figure then it will hit line segment (a,b) first.

Point location problem is similar to a ray shooting problem. We can solve point location problem as follows - shoot a ray vertically from point P, depending upon which line segment it hits first, we can determine the region which contains point P. In the dynamic version of algorithm, segments can be inserted or deleted. So our goal is to obtain:

1. A linear space data structure
2. Logarithimic query time (insert, delete etc.)
3. Optimal preprocessing/construction time

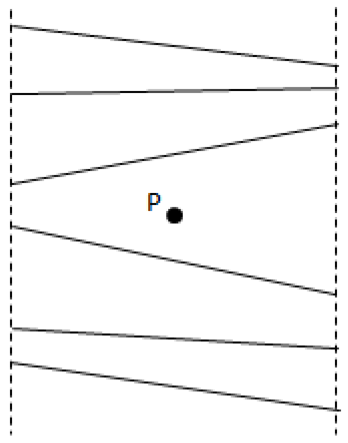
preprocessing time is least important among these because it occurs only once. There is always a trade-off between space and query time but space is more important than query time.

Consider an example of a planar subdivision as shown in figure



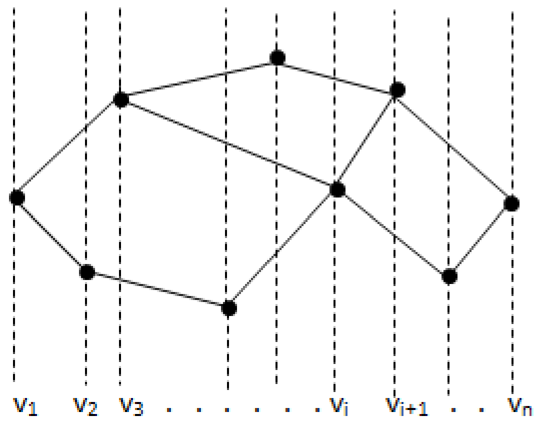
Here all horizontal line segments are of equal length and are parallel to each other. To solve point location problem, project all line segments on vertical axis and sort them. Now use simple binary search to find the region which contains point P. Here space used is $O(n)$, query time is $O(\log n)$ and preprocessing time is $O(n \log n)$.

Consider another example of planar subdivision, where line segments are not parallel to each other but have equal x-axis length as shown in figure:



In this case all the line segments are tilted but they are ordered because none of the line segments intersects with another line segment. So we can still apply binary search. If line segments intersect or they are not of equal x-axis length then we can not apply binary search.

Using this strategy we can solve the original point location problem. Draw a vertical line on each vertex of the planar subdivision as shown in figure:



Between two consecutive vertical lines v_i and v_{i+1}, there can not be any vertex of planar subdivision(by definition). So within a slab the subproblem is similar to the problem as discussed in earlier example. We can summarize the complete solution as follows:

Arbitrary planar subdivision can be partitioned into vertical slabs by drawing vertical lines on each vertex(assume no two vertices lies on same vertical line). Now two binary searches suffice for planar point location, one to determine which slab contains point and another within slab to determine which region contains point. So query time for this solution is $O(\log n)$ however space required can be $\Omega(n^2)$. But if we see two consecutive slabs then change in number of line segments is very small or constant. So instead of storing each slab separately we want to somehow store the common information only once. For this we have a data structure called

Persistent Data Structure. This kind of data structure is used for storing similar kind of lists, here in our case slabs. So using this data structure we can store information of all the slabs in linear space and still support $O(\log n)$ query time.