On Implementing an $o(\log n)$ Planar Point Location Algorithm

Yifei Zhang 2006210806
Wei Yu 2006310472
Decheng Dai 2006310473

January 11, 2007

Contents

1 Introduction

Planar Point Location Problem is a fundamental subproblem in computational geometry, usually needed as an ingredient to solve larger geometric problems. In a dispatch system to assign policemen to the scene of a crime, the city will be partitioned into different precincts or districts. Given a map of regions and a query point (the crime scene), the system must identify which region contains the point. This is exactly the problem of planar point location.

Planar Point Location Problem is also a fundamental subprogram in other areas, such as speech identification, face reorganization and handwriting identification. The widely used algorithms in these areas are all similar. First, thousands of characters are drew out then a space with many dimensions are built up. Second, A Point Location Searching in this high-dim space were run for classification (Sometimes, they will use the nearest neighbor searching instead).

Its usage could also be found in applications that deal with geometrical data: computer graphics, geographical information systems (GIS), motion planning, and CAD. In the data structure world, the problem can be used to solve the nearest and farthest neighbor problems in the Euclidean space, by employing a Voronoi diagram as the polygonal subdivision. Since the Voronoi diagram has $O(n \log n)$ complexity, this reduction can be used to achieve optimal space bounds.

In its most general form, the problem is, given a point in the plane or in the space, to determine which area the point belongs to. Each time you click a mouse to follow a web link, the problem is to be solved, namely, to determine which area of the computer screen is under the mouse pointer.

Different variations of *Planar Point Location Problem* include:

Is a given point inside or outside of polygon P?

The simplest version of Point Location involves only two regions, one finite, inside-P and the other infinite, outside-P; and asks which contains a given query point. For polygons with lots of narrow spirals, this can be surprisingly difficult to tell by inspection. The secret to doing it both by eye or machine is to draw a ray starting from the query point and ending beyond the furthest extent of the polygon. Count the number of times the polygon crosses through an edge. If this number is odd, we must be within the polygon. If it is even, we must be outside. The case of the line passing through a vertex instead of an edge is evident from context, since we are counting the number of times we pass through the boundary of the polygon. Testing each of the n edges for intersection against the query ray takes $O(n)$ time. Faster algorithms for convex polygons are based on binary search and take $O(\log n)$ time.

How many queries will have to be performed?

When we have a subdivision with multiple regions, it is always possible to repeat the inside-polygon test above on each region in the subdivision. However, this will waste a lot of time if we will perform many such Point Location queries on the same subdivision. More cleverly, we can construct a grid-like or tree-like data structure on top of our subdivision to get us near the correct region quickly. Such kind of search structures are discussed in more detail below.

How complicated are the regions of your subdivision?

More sophisticated inside-outside tests are required when the regions of your subdi-

vision are arbitrary polygons. By triangulating all polygonal regions first, each inside-outside test reduces to testing whether a point is in a triangle. Such a test can be made particularly fast and simple, at the minor cost of recording the full-polygon name for each triangle. An added benefit is that the smaller your regions are, the better grid-like or tree-like superstructures are likely to perform.

How regularly sized and spaced are your regions?

If all resulting triangles are about the same size and shape, the simplest point location method imposes a regularly-spaced $k \times k$ grid of horizontal and vertical lines over the entire subdivision. For each of the k^2 rectangular regions, we maintain a list of all the regions that are at least partially contained within the rectangle. Performing a point location query in such a *grid file* involves a binary search or hash table lookup to identify which rectangle contains query point q and then searching each region in the resulting list to identify the right one.

Such grid files will perform very well, provided that each triangular region overlaps only a few rectangles (thus minimizing storage space) and each rectangle overlaps only a few triangles (thus minimizing search time). Whether it will perform well is a function of the regularity of your subdivision. Some flexibility can be achieved by spacing the horizontal lines irregularly, as a function of the regions of the subdivision. The *slab method*, discussed below, is a variation on this idea that guarantees worst-case efficient point location at the cost of quadratic space.

How many dimensions will you be working in?

In three or more dimensions, some flavor of *KD-tree* will almost certainly be the point-location method of choice. They are also likely to be the right answer for planar subdivisions too irregular for grid files.

KD-trees decompose the space into a hierarchy of rectangular boxes. At each node in the tree, the current box is split into a small number (typically 2 or 4 or d, where d is the dimension) of smaller boxes. At the leaves of the tree, each box is labeled with the small number of regions that are at least partially contained in the box. The point location search starts at the root of the tree and keeps traversing down the child whose box contains the query point. When the search hits a leaf, we test each of the relevant regions against q to see which one of them contains the point. As with grid files, we hope that each leaf contains a small number of regions and that each region does not cut across too many leaf cells.

1.1 Problem Definition

In two dimensions, the point location problem restricts itself to a plane.In general, The *Planar Point Location Problem* is defined by a set of n segments, dividing the plane into polygons(finite and infinite). Given a query point, the goal is to find the polygon which contains it. Segments are only allowed to touch at end-points.

Definition 1 (Planar Subdivision)**.** *A planar subdivision is a partition of plane S into a finite collection of polygonal regions*

$$\{F_1, F_2, F_3, ..., F_m | m < \infty, \forall i \neq j, F_i \cap F_j = \emptyset, \bigcup_i F_i = S\}$$

by finite collection of line segments whose pairwise intersections are restricted to segment end-points.

Definition 2 (Planar Point Location)**.** *In a plane and a planar subdivision* $\{F_1, F_2, ...\}$*, given a query point* q*, decide an* i*, s.t.* $F_i/capq \neq \emptyset$ *and* $F_j/capq = \emptyset, \forall j \neq i$*,as figure 1.*

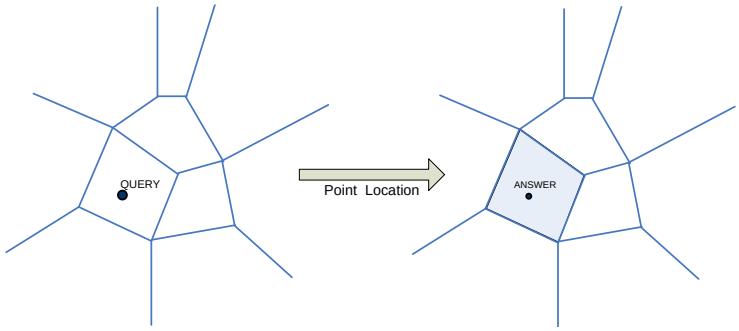

Figure 1: For a planar subdivision and a query point, the *Planar Point Location Problem* answers which polygon this query point locats in.

For that the storage for the polygons is much more complicated than the segments, we are not always signal every polygon with an unique number. In the other point, the right-hand-side polygon of a segment is unique. So we can simply output a segment for the *Planar Point Location Problem* instead of a polygon. Another definition of this problem is below,

Definition 3 (Predecessor Search)**.** *In a plane and a planar subdivision* $\{F_1, F_2, ...\}$*, given a query point* q*, decide an segment* l*, s.t. all the polygons except the one below this segment insect* q *comes to an empty set,as figure 2.*

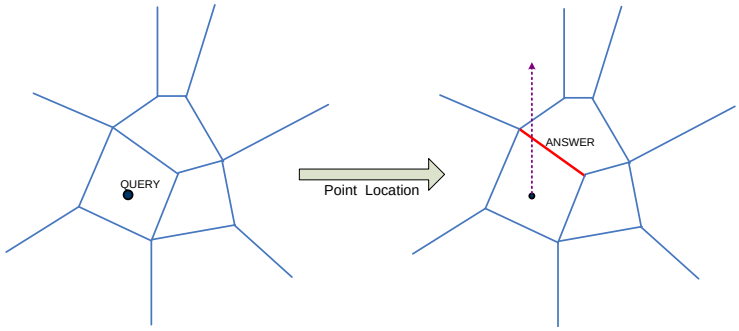

Figure 2: The output polygon can be simply replaced by a segment, which is nearest-upward from query point.

For that we only discuss the problem in planar, we won't extend this definition to higher-dimension space.

1.2 Related Work

Since we are working in a bounded universe, we can draw an enclosing box, making the problem finite.

Moreover, *Planar Point Location Problem* is one of the most fundamental and well studied search problems in computational geometry and variants of solutions were found for this problem. Traditional solutions achieved $O(\log n)$ query time and $O(n)$ space [14]. Recently in IEEE FOCS'06, Timothy Chan [1] and Mihai Patrascu [11] showed how this result could be improved, assuming coordinates have bounded precision.

Dobkin and Lipton [3] proposed the now-standard idea of dividing the plane into vertical slabs, and using binary search inside each slab. This gave an $O(\log n)$ running time with $O(n^2)$ space.

Sarnak and Tarjan [14] developed the first linear space data structure which could search in logarithmic time, based on planar separators. Their solution for *Planar Point Location Problem* is to sweep the plane with a vertical line, inserting and deleting segments in a binary search tree. The structure for each slab is the image of the dynamic structure at some point in time. Using persistent binary search trees, one can record all these images with $O(1)$ amortized space per update.

And since then, many algorithms have been invented to solve the *Planar Point Location Problem* with nearly the same query time and space complexity.

Recently, Timothy Chan [1] and Mihai Patrascu [11] have independently developed a static solution for the problem in vertical slabs. It is based on a constant-time routine which can (essentially) reduce the universe by a factor of $\sqrt{\sigma}$, using space σ. Their works are both based on the basic idea of *Fusion Tree* [6, 16] which was first invented to accelerate searching on RAM model.

The *Fusion Tree* was invented to improve worst-case algorithms for sorting and searching, surpassing the limitation of the information theoretic lower bound. That is, the information theoretic bound asserts that sorting N numbers requires $N \log N$ comparisons, which is not true in the real RAM model, when including integer division and bitwise boolean operations. From there, the idea started.

For the *Planar Point Location Problem* , they used this basic structure to derive an algorithm with query time $o(\log n)$. This is based on a judicious application of classic data structuring ideas such as exponential trees, bucketing and persistence. Perhaps the most interesting idea is that a subtle modification of exponential trees makes them respect the structure of the geometric search problem considered by their solution. In fact, with careful examinations, $O\left(\min\left\{\frac{\log n}{\log\log n}, \sqrt{\frac{\log u}{\log\log u}}, \frac{\log u}{\log S}\right\}\right)$ query time can be achieved with space $O(n \cdot S)$, where u is the word-bit length of the RAM model.

2 Traditional Algorithms

The simplest algorithm to guarantee $O(\log n)$ worst-case access is the *slab* method,which draws horizontal lines through each vertex,thus creating $n + 1$ 'slabs' between the lines.the slabs are defined by horizontal lines, finding the slab containing a particular query point can be done using a binary search on the y-coordinate of q,which denotes the query point.Since there can be no vertices within any slab, looking for the region containing a point within a slab can be done by a second binary search on the edges that cross the slab.The key point is that a binary search tree must be maintained for each slab, for a worst-case of space if each region intersects each slab. See figures 3, 4, 5

While this algorithm allows for point location in logarithmic time and is easy to implement, the space required to build the slabs and the regions contained within the slabs is at worst $O(n^2)$. While in most cases it is actually smaller than this, it is usually significantly larger than $O(n)$. To solve this problem, other data structures and algorithms are used that have $O(\log n)$ point location and $O(n)$ space requirements, such as trapezoidal decompositions.

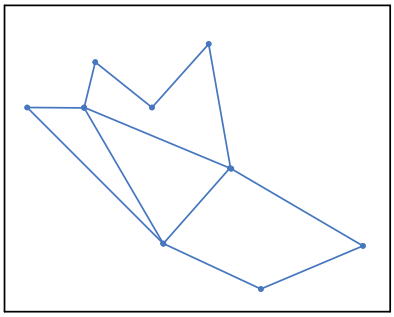

Figure 3: The simple algorithm begins with a subdivision for a bounded space.

A more space-efficient approach based on building a hierarchy of triangulations over the regions also achieves for search.Worst-case efficient computational geometry methods either require a lot of storage or are fairly complicated to implement. However, kd-trees are recommended for most general point-location. applications.

The similar problems were also discussed before.The inside-outside test for convex polygons is described in, which has a very thorough treatment of deterministic planar point location data structures. Expositions on the inside-outside test for simple polygons include [10],[12].

An experimental study of algorithms for planar point location is described in [4]. The winner was a bucketing technique akin to the grid file.

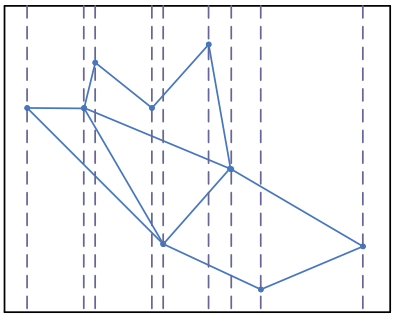

Figure 4: Horizontal lines were drew through each vertex, creating $n + 1$ slabs. A binary searching was used for locate query point in these slabs.

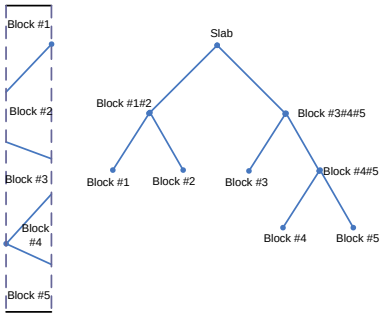

Figure 5: A second binary search on the edges that cross the slab can locate query point finally.

The elegant triangle refinement method of Kirkpatrick [8] builds a hierarchy of triangulations above the actual planar subdivision such that each triangle on a given level intersects only a constant number of triangles on the following level. Since each triangulation is a fraction of the size of the subsequent one, the total space is obtained by summing up a geometric series and hence is linear. Further, the height of the hierarchy is , ensuring fast query times. An alternative algorithm realizing the same time bounds is [5]. The slab method described above is due to [3] and is presented in [12].

More recently, there has been interest in dynamic data structures for point location, which support fast incremental updates of the planar subdivision (such as insertions and deletions of edges and vertices) as well as fast point location. Chiang and Tamassia's [2] survey is an appropriate place to begin.

3 The $o(\log n)$ Algorithm

While searching requires $\Omega(\log n)$time for comparison-based algorithms, sublogarithmic data structures are possible for integer input(see the next section *Fusion Tree*). These improved techniques have been applied to obtain faster graph algorithms for basic problems like minimum spanning trees and shortest paths. in Chan's [1] and Patrascu's [11], it shows, for the first time, that the known $\Omega(n \log n)$ lower bounds for algebraic computational trees can be broken for many of the core problems in computational geometry, when the input coordinates are integers in $[0, U)$ with $U \leq 2^w$.

3.1 Point Location in a Slab

In this section, we study a special case of the 2-d point location problem: given a static set S of n disjoint closed (nonvertical) line segments inside a vertical slab, where the endpoints all lie on the boundary of the slab and have integer coordinates in the range $[0, 2^w)$, preprocess S so that given a query point q with integer coordinates, we can quickly find the segment that is immediately above q. We begin with a few words to explain (vaguely) the difficulty of the problem.

The most obvious way to get sublogarithmic query time is to store a sublogarithmic data structure for 1-d successor search along each possible vertical grid line. However, the space required by this approach would be prohibitively large $O(n2^w)$, since unlike the standard comparison-based approaches, these 1-d data structures heavily depend on the values of the input elements, which change from one vertical line to the next.

So, to obtain sublogarithmic query time with a reasonable space bound, we need to directly generalize a 1-d data structure to 2-d.The common approach to speed up binary search is a multiway search, i.e., a "b-ary search" for some nonconstant parameter b. The hope is that locating a query point q among b given elements $s_1, s_2, ..., s_b$ could be done in constant time. In our 2-d problem, this seems possible, at least for certain selected segments $s_1, s_2, ..., s_b$, because of the following "input rounding" idea: locating q among $s_1, s_2, ..., s_b$ reduces to locating q among any set of segments $\tilde{s_1}, \tilde{s_2}, ..., \tilde{s_b}$ that satisfy $s_1 \prec \tilde{s_1} \prec s_2 \prec \tilde{s_2} \prec s_3 \prec \tilde{s_3} \prec ...$, where $\prec$ denotes the (strict) belowness relation. Because the coordinates of the $\tilde{s_i}$'s are flexible, we might be able to find some set of segments $\tilde{s_1}, \tilde{s_2}, ..., \tilde{s_b}$, which can be encoded in a sufficiently small number of bits, so that locating among the $\tilde{s_i}$s can be done quickly by table lookup or operations on words. (After the s_i's have been "rounded", we will see later that the query point can be rounded as well.)

Unfortunately, previous 1-d data structures do not seem compatible with this idea. So the Chen describe a new data structure built by the 1-d fusion tree for slab distribution and 2-d fusion subtree for the distribution in a slab. These two data structures will be described below separately.

3.2 The Fusion Tree

We first review the standard 1-d problem of performing successor search in a static set of n numbers, where the numbers are assumed to be integers in $[0, 2^w)$. The main idea is very simple and is encapsulated in the observation below–roughly speaking, in divide-and-conquer, allow progress to be made not only by reducing the number of elements, n, but alternatively by reducing the length of the enclosing interval, i.e.,reducing the number of required bits,l.

Lemma 1. *Fix b and h. Given a set S of n numbers in an interval I of length 2^l, we can divide I into $O(b)$ subintervals such that*

1. *each subinterval contains at most n/b elements of S or has length 2^{l-h} ,and*
2. *the subinterval lengths are all multiples of 2^{l-h}*

Proof. Form a grid over I consisting of 2^h subintervals of length 2^{l-h}. Let B contain the $(\lfloor in/b \rfloor)^{th}$ smallest element of S for $i = 1, ..., b$. Consider the grid subintervals that contain elements of B. Use these $O(b)$ grid subintervals to subdivide I. Note that any "gap" between two such consecutive grid subintervals do not contain elements of B and so can contain $\leq n/b$ elements(See Figure 6). □

The lemma above suggests a simple tree structure for 1-d successor search. Because of **2.**, we can represent each endpoint of the subintervals by an integer in $[0, 2^h)$, with h bits. We can thus encode all $O(b)$ subintervals in $O(bh)$ bits, which can be packed into a single word if we set $h = \lfloor \varepsilon w/b \rfloor$ for a sufficiently small constant $\varepsilon > 0$.We recursively build the tree structure for the subset of all elements inside each subinterval. We stop the recursion when $n \leq 1$(in particular, when $l > 0$).Initially, $l = w$. Because of **1.**,in each subproblem,n is decreased by a factor of b or l is decreased by h. Thus, the height of the tree is at most $\log_b n + w/h = O(\log_b n + b)$.

To search for a query point q, we first find the subinterval containing q by a word operation (for that word operation is not the important detail in the algorithm and is associated by the hardware, we do not discuss it here). We then recursively search inside this subinterval. (If the answer is not there, it must be the first element to the right of the subinterval; this element can be stored during preprocessing.) By choosing $b = \lfloor \sqrt{\log n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\log n / \log \log n)$.

3.3 The 2-D Fusion Tree

We now present the data structure for point location in a slab. The idea is to allow progress to be made either combinatorially(in reducing n) or geometrically (in reducing the length of the enclosing interval for either the left or the right endpoints).

Lemma 2. *Fix b and h. Let S be a set of n disjoint segments, where all left endpoints lie on an interval I of length 2^l on a vertical line, and all right endpoints lie on an interval J of length 2^m on another vertical line. We can find $O(b)$ segments $s_0, s_1, ... \in S$ in sorted order, which include the lowest and highest segment of S, such that*

1. *for each i, there are at most n/b segments of S between s_i and s_{i+1}, or the left endpoints of s_i and s_{i+1} lie on a subinterval of length 2^{l-h}, or the right endpoints of s_i and s_{i+1} lie on a subinterval of length 2^{m-h}, and*
2. *there exist segments $\tilde{s_0}, \tilde{s_2}, ...$, with $s_0 \prec \tilde{s_0} \prec s_2 \prec \tilde{s_2} \prec ...$and endpoints on I and J, such that distances between left endpoints of the $\tilde{s_i}$'s are all multiples of 2^{l-h} and distances between right endpoints are all multiples of 2^{m-h}.*

Proof. Form a grid over I consisting of 2^h subintervals of length 2^{l-h} , and a grid over J consisting of 2^h subintervals of length 2^{m-h}. Let B contain the $(\lfloor in/b \rfloor)^{th}$ lowest segment of S for all $i = 1, ..., b$. Set s_0 to be the lowest segment. For $i = 0, 2, 4, ...$(until the highest segment is reached), set s_{i+1} to be the highest segment of B such that the left endpoints of s_i and s_{i+1} are in the same grid subinterval or the right endpoints of s_i and s_{i+1} are in the same grid subinterval. Set s_{i+2} to be the successor of s_{i+1} in B. Since the left endpoints of s_i and s_{i+2} are in different grid subintervals

and the right endpoints of s_i and s_{i+2} are in different grid subintervals, we can round s_i to a new segment $\tilde{s_i}$ to satisfy **2.**(See Figure 6). □

For the w-bit computing we need to finish the algorithm, the data structure must be designed for it. Because of **2.**, we can represent each endpoint of the $\tilde{s_i}$'s as an integer in $[0, 2^h)$, with h bits. We can thus encode all $O(b)$ segments $\tilde{s_0}, \tilde{s_2}, ...$ in $O(bh)$ bits, which can be packed in a single word if we set $h = \lfloor \varepsilon w/b \rfloor$ for a sufficiently small constant $\varepsilon > 0$. We recursively build the tree structure for the subset of all segments strictly between s_i and s_{i+1}. We stop the recursion when $n \leq 1$ (in particular, when $l < 0$ or $m < 0$). Initially, $l = m = w$. Because of **1.**, in each subproblem, n is decreased by a factor of b, or l is decreased by h, or m is decreased by h. Thus, the height of the tree is at most $\log_b n + 2w/h = O(\log_b n + b)$.

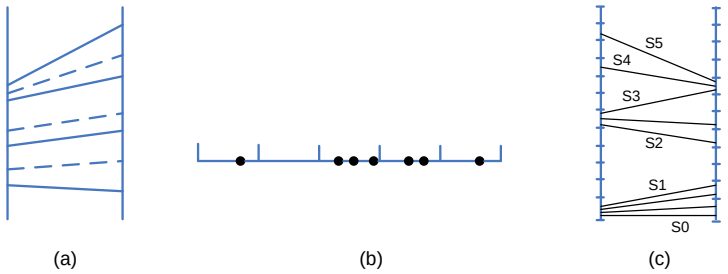

Figure 6: (a) The rounding idea: locating among the solid segments reduces to locating among the dotted segments. (b) Proof of Lemma 1: elements of B are shown as dots. (c) Proof of Lemma 2: segments of B are shown , together with the constructed sequence $s_0, s_1, ...$

Given a query point , we first locate q among the $\tilde{s_i}$'s by a word operation. With one extra comparison we can then locate q among $s_0, s_2, s_4, ...$, and with one more comparison we can locate q among all the s_i's and answer the query by recursively searching in one subset. By choosing $b = \lfloor \sqrt{\log n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\log n / \log \log n)$.

The data structure clearly requires $O(n)$ space. Since the segments s_i's and $\tilde{s_i}$'s can be found in linear time for pre-sorted input, the preprocessing time after initial sorting can be bounded naively by $O(n)$ times the tree height, i.e.,$O(n \log n / \log \log n)$(which can actually be improved to $O(n)$ with more work). Sorting naively takes $O(n \log n)$ time, which can be improved by known results.

3.4 Word Operations

For that the word operation is also a key point in this algorithm and there are many problems in the operations which defined by the author. But we only describe the advantages here and the disadvantage will be discussed in the next section.

We have assumed above that we can locate q among the $\tilde{s_i}$'s in constant time, given $O(b)$ segments $\tilde{s_1}, \tilde{s_2}, ...$, satisfying **2.**, all packed in one word. We now show that this nonstandard operation can be implemented using more familiar operations like multiplications, divisions, shifts, and bitwise-ands.

First, by a projective transformation, we may assume that the left endpoint of $\tilde{s_i}$ is $(0, \tilde{a_i})$ and the right endpoint is $(2^h, b_i)$, where the $\tilde{a_i}$'s and $\tilde{b_i}$'s are increasing se-

quences of integers in $[0, 2^h)$. For example, the mapping below transforms two intervals $I = \{0\} \times [0, 2^h)$ and $J = \{C\} \times [D, D + 2^m)$to $\{0\} \times [0, 2^h)$and $\{2^h\} \times [0, 2^h)$ respectively:

$$(x, y) \mapsto (\frac{2^{h+m}x}{2^l(C - x) + 2^m x}, \frac{2^h[Cy - Dx]}{2^l(C - x) + 2^m x})$$

(The line segments $\tilde{s_i}$'s are mapped to line segments, and the belowness relation is preserved.)

We round the query point q, after the transformation, to a point $\tilde{q}$ with integer coordinates in $[0, 2^h)$. (Note that $\tilde{q}$ can be computed exactly by using integer division in the above formula.) Observe that a unit grid square can intersect at most two of the $\tilde{s_i}$'s, because the vertical separation between two segments (after transformation) is at least 1 and consequently so is the horizontal separation (as slopes are in the range $[-1, 1]$). This observation implies that after locating $\tilde{q}$, we can locate q with $O(1)$ additional comparisons.

To locate $\tilde{q} = (\tilde{x}, \tilde{y})$ for h-bit integers before $\tilde{x}$ and $\tilde{y}$, we proceed as follows. Let $\langle z_1|z_2|...\rangle$ denote the word formed by $O(b)$ blocks each of exactly $2(h+1)$ bits, where the i-th block holds the value z_i (recall that $bh \leq \varepsilon w$). We precompute $\langle \tilde{a_0}|\tilde{a_2}|...\rangle$ and $\langle \tilde{b_0}|\tilde{b_2}|...\rangle$ during preprocessing by repeated shifts and additions. The y-coordinate of $\tilde{s_i}$ at $\tilde{x}$ is given by $[\tilde{a_i}(2^h - \tilde{x}) + \tilde{b_i}\tilde{x}]/2^h$. With two multiplications and some additions and subtractions, we can compute the word $\langle \tilde{a}_0(2^h - \tilde{x}) + \tilde{b}_0\tilde{x}|\tilde{a}_2(2^h - \tilde{x}) + \tilde{b}_2\tilde{x}|...\rangle$. We want to compute the rank of $2^h\tilde{y}$among the values encoded in the blocks of this word. This subproblem was solved before [7].

3.5 The $o(\log n)$ Algorithm

We now tackle the 2-d point location problem in the general setting: given a static planar subdivision formed by a set S of n disjoint open line segments with $O(w)$-bit integer or rational coordinates, preprocess S so that given a query point q with integer or rational coordinates, we can quickly identify (a label of) the face containing . By associating each segment with an incident face, it suffices to find the segment that is immediately above q.

The result of the previous section naively yields an $O(n^2)$-space data structure with $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\}$ query time: Divide the plane into $O(n)$ slabs through the x-coordinates of the endpoints and build our 2-d fusion tree inside each slab (note that the endpoints of the segments clipped to the slab indeed are rational numbers with $O(w)$-bit numerators and denominators). Given a query point q, we can first locate the slab containing q by a 1-d successor search on the x-coordinates and then search in this slab. The preprocessing time is $o(n^2 \log n)$.

We can improve the preprocessing time and space by applying known computational-geometric techniques for point location, for example, using a b-ary version of the *segment tree* or the *trapezoid method* [13]. To get linear space, though, we adopt a random sampling method [15] or a planar separator method [9]. The former is simpler to implement, but the latter is deterministic and also has linear preprocessing time for connected subdivisions. As the author in the main paper we read, it was also mentioned that these two methods are both not easy to implement. So in our algorithm , we only use the $O(n^2)$ space but $o(\log n)$ query time algorithm for demo and testing.

4 Project Overview

4.1 Implemented Algorithm

For that the algorithm the paper proved to achieve the linear space is not fit for implementing, we only use the 2-D Fusion tree for both the slab search (while we can regard a slab a block in a bigger slab for all the slabs) and the slab-inside location and achieve the $o(\log n)$ query time algorithm. More details in our implement will be described below.

First, we split the plane to $O(n)$ slabs according to all the vertices and build a fusion tree for this structure. The two parameter in this part were the branch number b and the word length w. For that the best w is the integer length, and although we can extend w to get a more effective and lower-depth tree, but it is not easy to implement another basic arithmetic for integer longer than 64 bits in our personal computer. So we set w to be 64 and b can be alternative between $[2, 4]$. The upper bound 4 is decided by the fusion tree algorithm and w. And , we assumes that the initial points are all located on the integers, so in this step every thing go well without any precision problem.

Secondly, after we located the query point q into a slab, we will locate the query point q to a block. In this step we build a 2-d fusion tree and do a b-ary searching in this fusion tree. In this step, we have two problem to solve. The two parameter as mentioned above have to be determined. Also, we change the branch number from 2 to 4, and set the word size w to be 64 as our computer integer word size. But the other problem is the precision we don't meet in the first step. Although all initial points are all locate on integers, but the point of intersection of segments are not always on integers. We have to deal with it. For that we can easily found that the coordinate of intersection is rational number, so they can presented by a $2w$-bit integer. But it is not very easy to develop an $O(1)$ arithmetic system for $2w$-bit rational numbers(although the $O(w)$ arithmetic system is trivial), so we use a precision-time tradeoff here. We treat all numbers with the same denominator 2^w and so their error ratio will be less than 2^{-w}. In our slab decision algorithm, we first map the anomalous into a square and do the location checking in $O(1)$ time.

We use only one algorithm module for slab location in the first two step for convenience. In the first 1-D *Fusion Tree* , we treat the whole plane as a horizontal slab and use two sufficient horizontal line as the two slab bounds. In the second step, we use it as what it is.

Finally, we display the path in the fusion trees and the point location with a JAVA application program.

4.2 User Manual

The Project was compiled with J2SE 6.0.
You can :

1. **Draw polygons**
   (a) Directly draw polygons in the canvas by draw series points and connect them;
   (b) While moving on the canvas, the coordinates will be display on the right;
   (c) After drawing, the graph can be saved and reloaded;

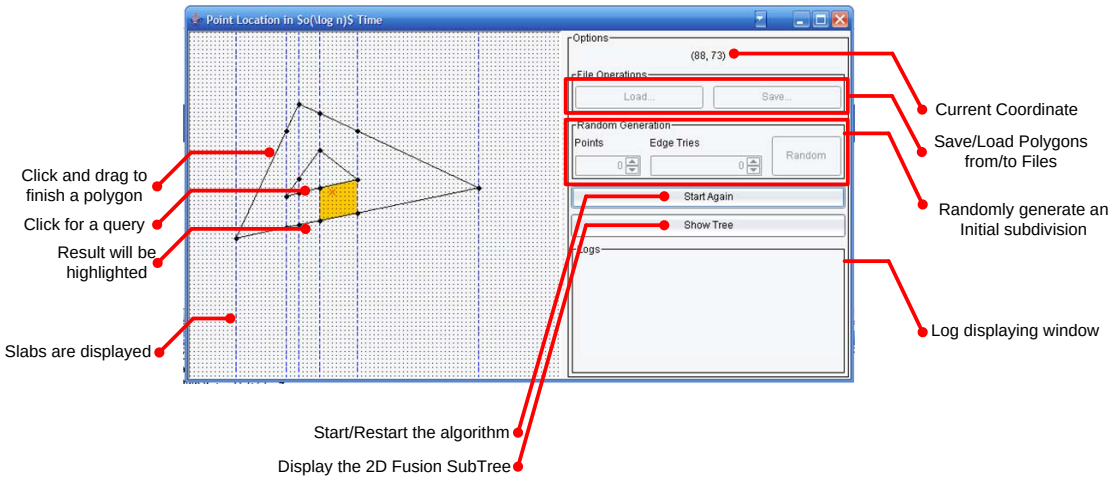

Figure 7: a screen shot for the interface: the main window

(d) Or the random generation can initialize a large set of polygons;

2. **Query**

   (a) Select a point to query its location; Click on the canvas to query directly;

   (b) Result will be displayed as Figure 7, the polygon is limited in a slab;

   (c) The 2-D Fusion Tree Can be displayed in another window;

   (d) The operation log displayed on right

3. **Checkout the data structure**

   (a) Use the ”Display Tree” button to display the Fusion Tree 8;

   (b) Use ”Ctrl+1,2,3,4” To change the orientation of the tree;

   (c) Right click to zoom in/out;

   (d) Drag to see the hidden part;

   (e) Click to spread an sub tree node;

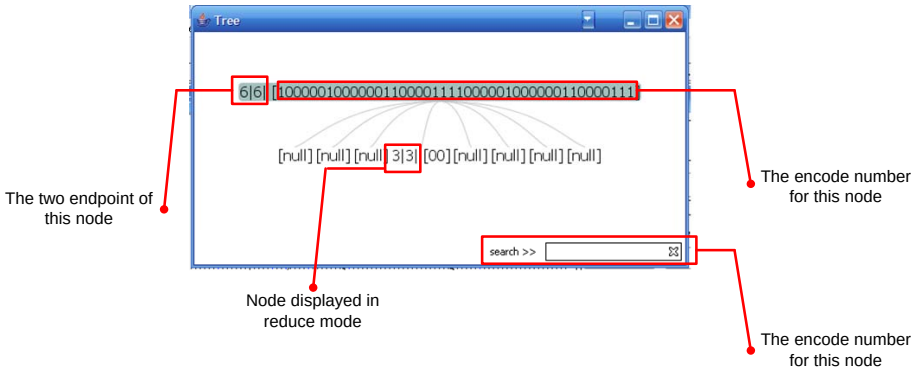

Figure 8: a screen shot for the interface: tree display

5 Discussion

5.1 Further Theoretical Result

The author use some analysis to achieve the $O(n)$ space, but it is not feasible to implement. So we perform a brief mathematical calculations it here without practicality to validate our foreground.

Take a random sample $R \subseteq S$ of size r. We first compute the *trapezoidal decomposition* $T(R)$: the subdivision of the plane into trapezoids formed by the segments of R and vertical upward and downward rays from each endpoint of R. This decomposition had $O(r)$ trapezoids and is known to be constructible in $O(r \log r)$ time. We store $T(R)$ in a point-location data structure, with $P_0(r)$ preprocessing time, $S_0(r)$ space and $Q_0(r)$ query time.

For each segment $s \in S$, we first find the trapezoid of $T(R)$ containing the left endpoint of s in $Q_0(r)$ time. By a walk in $T(R)$, we can then find all trapezoids of $T(R)$ that intersects s in time linear in the number of such trapezoids(note that s does not intersect any segment of R and can only cross vertical walls of $T(R)$). As a result , for each trapezoid $\Delta \in T(R)$, we obtain the subset S_Δ of all segments of S intersecting Δ. The time required is $O\left(nQ_0(r) + \sum_{\Delta \in T(R)} |S_\Delta|\right)$.

By a standard analysis of Clarkson and Shor, the probability that

$$\sum_{\Delta \in T(R)} |S_\Delta| = O(n)$$

and

$$max_{\Delta \in T(R)} |S_\Delta| = O((n/r) \log r)$$

is a greater than a constant. As soon as we discover that these bounds are violated, we stop the process and restart with a different sample; the expected number of trials is constant. We than recursively build a point-location data structure inside Δ for each subset S_Δ.

The expected preprocessing time $P(n)$, worst-case space $S(n)$, and worst-case query time $Q(n)$ satisfy the following recurrences for some n_i's with $\sum_i n_i = O(n)$ and $n_i = O((n/r) \log r)$

$$P(n) = \sum_i P(n_i) + O(P_0(r) + nQ_0(r))$$

$$S(n) = \sum_i S(n_i) + O(S_0(r))$$

$$Q(n) = max_i(n_i) + O(Q_0(r))$$

To reduce space further, we use the new method and create new bounds ,for some constant c. We than obtain the following bounds, where $\sum_i n_i = O(n)$ and $n_i = O((n/r)\log r)$:

$$P(n) = \sum_i P(n_i) + O(P_0(r) + nQ_0(r)) = O(n\log(n/r)) + r\log^c r + n\frac{\log r}{\log\log r})$$

$$S(n) = \sum_i S(n_i) + O(S_0(r)) = O(n + r\log^c r)$$

$$Q(n) = max_i(n_i) + O(Q_0(r)) = O(\log(n/r) + \frac{\log r}{\log\log r})$$

Setting $r = \lfloor n/\log^c n \rfloor$ this time yields $O(n\log n/\log\log n)$ expected preprocessing time, $O(n)$ space, and $O(\log n/\log\log n)$ query time.

5.2 Shortage

The first disadvantage for this algorithm is the model they used. Up till now, the most popular model for computational geometry, for good reasons, is the unit-cost real RAM, but not this one. And in practice, we do not have infinite precision as assumed by the real RAM. For that discussion on robustness and the exact arithmetic paradigm in computational geometry often assumes w-bit integer or floating-point input and operations on w-bit words, the author build this w-bit integer model is also understandable.

Second, the algorithm without exact presentation for the rational number will occur errors sometimes (We assume that all the rational number all with a same but big enough denominator, but unfortunately, sometimes it will fail). This error ratio is based on the presentation for all points in the subdivision. Initially, all points are located in integers and w-bit word is enough and effective. But in our algorithm, the intersection between slab bounds and segments are not always integers and they can not presented by a w-bit words.And if we treated them as integers an error will occur.This error is very tiny but if the query point is located very near the segment between this "error" point, it will make the wrong result. The author do not mentioned this problem in his paper for some reason, but it is the existence in the realization. So we use this ZOOM-in-out technique to avoid error ratio in a majority of usual cases. See figure 5.2.

Third, the branch number b is bounded on $[2, 4]$ in our usual 64-bit computer and $b = \lfloor \log n \rfloor$ can not be achieved. Because every $\tilde{s_i}$ needs at least h bits for presentation in theory and $2h + 1$ bits in practise. and we also need $O(bh) = w$ for upper bound analyze. So in deed, if we use an common $h = 3$ parameter, $b \leq w/(2h + 1) = \lfloor 64/7 \rfloor = 8$.More strictly, we use $b \times (2h + 1) \leq w/2$ for rational numbers, we will find that $b \leq 4$. So we will have not many choices and can only build a tree with 4 branches at most in practice.

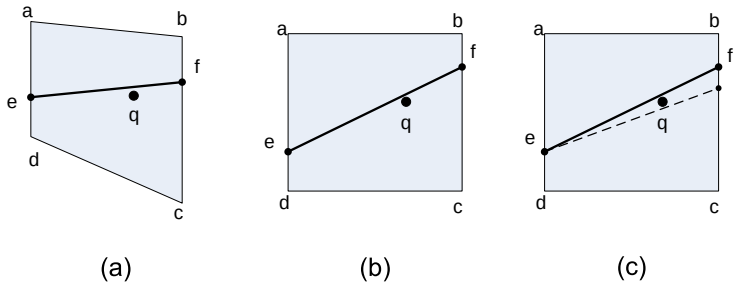

Figure 9: (a) The initial block in a slab with a segment (e, f) and a query point q (b) The linear map will not change the up/below relationship between them (c) If the precision is not enough, the rational number will occur a error.

6 Acknowledgement

Thanks the other students in Theory Group, the discussion with them make us much more clear with the algorithm. Thanks Professor Ma Bin, he pointed out many key points in the paper. Thanks for this course by Professor Junhui Deng and his wonderful lecture. We learnt not only knowledge but also principles in this course.

We are from the different group and we focus on distinct point on this problem, but on matter what we will do in the future, this course extend our view of computational geometry and computer science.

References

[1] Timothy M. Chan. Point location in o(log n) time, voronoi diagrams in o(n log n) time, and other transdichotomous results in computational geometry. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 333–344, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Yi-Jen Chiang and Roberto Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Int. J. Comput. Geometry Appl.*, 2(3):311–333, 1992.

[3] David P. Dobkin and Richard J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5(2):181–186, 1976.

[4] Masato Edahiro, I. Kokubo, and Takao Asano. A new point-location algorithm and its practical efficiency: Comparison with existing algorithms. *ACM Trans. Graph.*, 3(2):86–109, 1984.

[5] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[6] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 1–7, New York, NY, USA, 1990. ACM Press.

[7] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[8] David G. Kirkpatrick, Maria M. Klawe, and Derek G. Corneil. On pseudosimilarity in trees. *J. Comb. Theory, Ser. B*, 34(3):323–339, 1983.

[9] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

[10] U. Manber. *Introduction to Algorithms.* Addison-Wesley, 1989.

[11] Mihai Patrascu. Planar point location in sublogarithmic time. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 325–332, Washington, DC, USA, 2006. IEEE Computer Society.

[12] F. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, New York, NY, USA, 1985.

[13] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Springer, 1985.

[14] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[15] Erik Meineche Schmidt and Sven Skyum, editors. *Algorithm Theory - SWAT '94, 4th Scandinavian Workshop on Algorithm Theory, Aarhus, Denmark, July 6-8, 1994, Proceedings*, volume 824 of *Lecture Notes in Computer Science*. Springer, 1994.

[16] Dan E. Willard. Applications of the fusion tree method to computational geometry and searching. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 286–295, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.