Sublogarithmic Algorithms for Planar Point Location

Computational Geometry, Spring 2021 - Project Report

Aécio Santos
aecio.santos@nyu.edu

1 Introduction

Planar point location is one of the most well-studied and fundamental problems in computational geometry: given a map and a query point (i.e., its coordinates), find the region of the map containing the query point. This problem has found applications in many areas ranging from computer graphics and geographic information systems to motion planning and computer-aided design. In its classic definition, the planar point location problem asks the following question:

Definition 1 *Given a planar polygonal subdivision S and a query point p, how can we preprocess S to construct a data structure that occupies low space and supports queries that efficiently find the face in S where the point p is located.*

Point location problem variants. Many variants of this problem have also been described in the literature [13, 5, 4] that differ in terms of their settings and *supported operations*. For instance, while in the *static* variant all changes to the data structured are known at the time of the data structure construction, in the *dynamic* version of the problem the data structure must also be able to receive modifications to the planar subdivision and support operations that efficiently update an existing data structure.

Besides the differences in supported operations, solutions to the problem also make a wide range of different assumptions about the *input data*. For example, while the problem from Definition 1 accepts arbitrary polygonal subdivisions, the *orthogonal* (or *rectangular*) [4] point location problem is a variant in which all the edges of the planar subdivision are either vertical or horizontal. As another example, the solution given by Iacono and Langerman [13] works only for hyperrectangles and their space and construction bounds depend on a property of the input data, namely, the *fatness* of the hyperrectangles.

Our work. In this report, we focus on the planar point location problem from Definition 1. We studied the algorithm proposed by Chan and Pătraşcu [5], which, to the best of our knowledge, has the best known asymptotic bounds in the *worst-case scenario*. We contrast their results with classic results for point location algorithms (Section 2), and we describe how their algorithm achieves sublogarithmic query time by building on algorithmic techniques developed by Fredman and Willard in [11] for Fusion Trees (Section 3). In addition, we discuss the limitations of this algorithm from a *practical* point of view, which are reminiscent of Fusion Trees – which, currently, are mostly of theoretical interest (Section 3.3). Finally, we describe our efforts in devising an alternative algorithm that also operates on the word RAM model but that does not rely on the Fusion Tree's techniques (Section 4).

2 Classic Results for Planar Point Location

Results on the comparison-based model. The long history of theoretical research on planar point location has produced several important results. In particular, existing comparison-based algorithms are able to achieve $O(\log n)$ time queries using only $O(n)$ space. It can be shown, using an adversary argument, that *these results are optimal for comparison-based query algorithms*, i.e., a minimum of $\Omega(\log n)$ comparisons are required [23].

To achieve these results, several algorithmic techniques have been developed and used such as the *slab method* [8], *persistent trees* [22], *fractional cascading* [6, 9], *trapezoid graphs* [21], *hierarchical triangulations* [14], and *monotone subdivisions* [9]. Table 1 provides a non-exhaustive list of these results, along with their query time, preprocessing time, and space usage. Note that the four algorithms in this table that achieve optimal query time and space use substantially different approaches. As we will see in next section, these information-theoretic lower bounds for querying only hold for comparison-based algorithms that operate with real numbers and can be improved using other computational models.

Method	Query	Space	Preprocessing
Slab method [8]	$O(\log n)$	$O(n^2)$	$O(n^2 \log n)$
Trapezoids [21]	$O(\log n)$ (w.h.p.)	$O(n \log n)$	$O(n \log n)$ (exp.)
Slab + Persistent Trees [22]	$O(\log n)$	$O(n)$	$O(n \log n)$
Randomized incremental [17, 12]	$O(\log n)$	$O(n)$	$O(n \log n)$ (exp.)
Monotone subdivisions [9]	$O(\log n)$	$O(n)$	$O(n \log n)$
Hierarchical triangulations [14]	$O(\log n)$	$O(n)$	$O(n)$

Table 1: A list of classic algorithms under the comparison model that achieve optimal query time. Only four of them achieve the optimal linear space usage.

3 Planar Point Location in Sublogarithmic Time

The planar point location algorithm proposed by Chan and Pătraşcu in [5] was the first algorithm to achieve sublogarithmic query time with a linear space usage. In order to break the $O(\log n)$ query time lower bound from classical comparison-based algorithms, they rely on algorithms that operate on the word RAM computational model. Before describing the main ideas behind this state-of-the-art algorithm, we first provide a brief introduction to this model and some key results that serve as building blocks for sublogarithmic algorithm from [5].

3.1 Transdichotomous Models and the Word RAM

Transdichotomous models [11] refers to variations of the traditional random access machine (RAM) model that support bit-wise operations on words and assumes that the machine word size matches the problem size. As opposed to the *real RAM model*, which allows computations with exact real numbers, transdichotomous models, such as the *word RAM* model, assume that both the problem input and the values stored in memory are integers with a fixed number of bits. This additional power allows algorithms to break through information-theoretic lower bounds that apply to comparison-based models on the real RAM.

The Word RAM Computational Model. In the word RAM model, it is assumed that the input can be represented as integers in the bounded universe $\mathcal{U} = \{0, 1, ..., 2^w - 1\}$ of size $U = |\mathcal{U}| = 2^w$, where w is the machine word size. It is also assumed that pointers fit in a single word and that there is enough space to fit the input, so the universe size $U \geq n$ and $w \geq \log n$. Finally, the model assumes that common mathematical operations (e.g., $+$, $-$, $*$, $/$) *and* bit-wise operations (e.g., $\&$, $|$, $\neg$, $\gg$, $\ll$) can be done in constant time for integers that fit in a word. These assumptions fit more closely the capabilities of current computers and C-like programming languages.

Predecessor Search on the Word RAM. One of the problems that are typically analyzed under the word RAM model is the *predecessor problem*, which can be solved optimally in $O(\log n)$ using binary search or binary search trees (BST) under the comparison model. In the word RAM model, this problem can be solved in sublogarithmic time by two prominent data structures known as *van Emde Boas Trees* [10] and *Fusion Trees* [11].

A *van Emde Boas Tree* (vEB) for the universe U is a recursive tree structure that splits the universe U into $\sqrt{U}$ buckets, each storing $\sqrt{U}$ items using vEB data structures that hold the items. In his original paper [10], van Emde Boas has shown that vEB trees can achieve update and query operations in $\theta(\log w)$ time with $\theta(U)$ space usage. Today, it is well-known that the $\theta(U)$ space can be reduced to $\theta(n)$ with randomization techniques. Subsequent work by Willard [24] proposed similar data structures similar to vEB trees, *x-fast tries* and *y-fast tries*, that achieve the same bounds as vEB trees. More recent works, have further developed these ideas and proposed z-fast tries [3, 2], which have not only good theoretical properties, but have also achieved good performance in practical applications.

The *Fusion Tree*, proposed by Fredman and Willard [11], is a data structure similar to B-Trees. The basic idea behind it, is to use a branching factor of size k – as opposed to the size-2 branching factor of a Binary Search Tree – and, at query time, to perform a simultaneous comparison between the query and all k keys in a node in *constant-time* using word-level operations. In order to do that, fusion trees need to carefully "*pack*" (or "*fuse*") k integers into a single machine word that allows for multiple comparisons in constant time. To do so, it needs to exploit operations and techniques such multiplication, most-significant set bit (MSB), sketch compression, and word-level parallelism (for parallel comparisons).

More specifically, a fusion tree is a k-ary search tree where every node holds roughly $k = \theta(w^{1/5})$ sorted keys. Therefore, the tree height is $O(\log_{w^{1/5}} n) = O\left(\frac{\log n}{1/5 \log_w}\right) = O(\log_w n)$. Given that the comparisons necessary to descend a node take constant time, the total query time is $\theta(\log_w n)$. This is always better than a binary search tree because w is assumed to be at least $\log n$. Therefore, the following relations hold: $\theta(\log_w n) = \theta\left(\frac{\log n}{\log w}\right) < \theta\left(\frac{\log n}{\log \log n}\right) < \theta(\log n)$.

A comparison between the query complexity of these two classic data structures shows that van Emde Boas Trees work better when the word size w is small, whereas Fusion Trees are better when w is large. If we know the value of w a priori, we can choose the best between fusion trees and van Emde Boas Trees to achieve $O(\min\{\log w, \log_w n\})$. It is easy to show that the two approaches become equivalent (i.e., $\log w = \log_w n$) when $\log w = \sqrt{\log n}$. Thus, it is always possible to achieve $O(\sqrt{\log n})$ query time by combining the two approaches.

3.2 Chan and Pătraşcu's Algorithm

3.2.1 Planar Point Location via Point Location in a Slab

The approach taken by Chan and Pătraşcu in [5] to solve the planar point location problem is based on the *slab method*. They argue that the problem of *locating a point among disjoint line segments spanning a vertical slab is the key sub-problem* in planar point location. They also show that it is possible convert any given solution for slab problem, with $O(t(n))$ query time and $O(n)$ space, into a solution for the general planar point location problem. Therefore, *improving the runtime of point location in a slab implies in improvements to the general planar point location problem* (and also other fundamental computational geometry problems).

Problem reductions. In order to go from the point location in a slab to a general planar location algorithm, Chan and Pătraşcu described how to adapt three different existing planar point location algorithms to use data structure for point location in a slab:

1. *Planar separators* [16]: The first solution relies on the planar graph separator theorem by Lipton and Tarjan [16]. While this approach has the best theoretical properties (e.g., deterministic bounds, linear-time construction), the authors in [5] note that it is the least practical because of large hidden constants.

2. *Random sampling* [18]: This method is a randomized algorithm that takes time $O(n \cdot t(n))$ and it is the simplest of the three proposed methods. The method basically uses random sampling to find suitable trapezoidal decompositions for which to build a slab point location data structure.

3. *Persistent search trees* [22]: This method adapts the classic approach of persistent trees to support inserting and deleting segments into the data structure for the slab problem over time. This approach requires ideas from *exponential search trees* [1] and resulted in a deterministic construction time equivalent to a sorting algorithm.

Sublogarithmic query time. Because the reductions described above make minimal assumptions on slab data structure, these results can be used as black-boxes to turn slab data structures into planar point location solutions. By proposing an more efficient data structure for the slab problem that answers queries in $O(\lg n/\lg\lg n)$ with $O(n)$ space, Chan and Pătraşcu were able to obtain a planar point location algorithm that reaches query time of $O(\min\{\lg n/\lg\lg n, \sqrt{\lg U/\lg\lg U}\})$ for planar subdivisions whose segments have w-bit rational coordinates.

3.2.2 Point Location in a Slab

At the core of Chan and Pătraşcu's approach is a new method for solving the slab problem in sub-logarithm time on the word RAM model. This sub-problem can be formally defined as follows (transcribed from [5]):

Definition 2 (Point Location in a Slab) *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 $[0, 2w)$, preprocess S so that given a query point q with integer coordinates, we can quickly find the segment that is immediately above q.*

Their approach performs a b-ary search on subdivisions of the slab using Fusion Trees' techniques that we described in Section 3.1. The challenge in making these techniques work is that the search is over line segments (2D), and not simply numeric values (1D). Hence, it is not clear how to pack segments into words and make comparisons in constant time.

Key algorithmic ideas. The initial idea is that locating a query point q among segments $s_1, ..., s_b$ reduces to locating q among any set of segments $\tilde{s}_1, ..., \tilde{s}_b$ that satisfy $s_1 \prec \tilde{s}_1 \prec s_2 \prec \tilde{s}_2 \prec ...$, where $\prec$ denotes the (strict) belowness. This flexibility in the segment endpoints is important because it allows rounding them. This, in turn, allows finding new endpoints that can be encoded in a sufficiently small number of bits, which is helps packing multiple endpoints into a single word.

Consider the right and left endpoints of the line segments crossing the slab, which lie in the intervals I_R and I_L, of lengths 2^{l_R} and 2^{l_L}, lying in the vertical lines x_R and x_L, respectively. Chan and Pătraşcu [5], have shown that it is possible to find $O(b)$ line segments $\{s_i \in S\}$ in sorted order, that include the lowest and highest segments of S, with the following two properties (which we transcribe from [5]):

1. *For each i, at least one of the following holds:*
   (a) *there are at most n/b line segments of S between s_i and s_{i+1};*
   (b) *the left endpoints of s_i and s_{i+1} lie on a subinterval of length 2^{l_L-h};*
   (c) *the right endpoints of s_i and s_{i+1} lie on a subinterval of length 2^{l_R-h}.*
2. *There exist $O(b)$ line segments $\tilde{s}_0, \tilde{s}_2, ...$ cutting across the slab, satisfying all of the following:*
   (a) $s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec ...;$
   (b) *distances between the left endpoints of the $\tilde{s}$'s are all multiples of 2^{l_L-h};*
   (c) *distances between right endpoints are all multiples of 2^{l_R-h}.*

These properties are important because they imply that all segments between s_i and $\tilde{s}_i$ can always make progress by reducing the number of segments of the length of the searched interval. The reason why this is the case is that either:

1. There are only a few segments (i.e., up to n/b) between s_i and $\tilde{s}_i$ (as guaranteed by property 1a), which means that we can reduce the problem size by a factor of b.
2. If there are many segments, the distances between these segments is small (as guaranteed by property 1b and 1c). Thus, endpoints can be represented with a few bits: by dividing endpoints of each subinterval by 2^{l-h}, they can be represented by an integer in $[0, 2^h)$ with only h bits. This reduces the interval length by h.

In other words, a search tree can decrease the sub-problem size by either decreasing the number of segments in a interval subdivision or by decreasing the interval length that needs to be searched (and hence, increasing the number of segments that can be packed into a single fusion tree node).

Finding Slab Subdivisions. The algorithm to find the of segments $\{s_i\}$ and $\{\tilde{s}_i\}$ is as follows:

1. Create a grid over I_L with 2^h subintervals of length 2^{l_L-h} (resp. for I_R);
2. Let B be a set containing every $(\frac{n}{b})^{th}$ segment of S, starting with the lowest segment;
3. Choose a set $\{s_i\}$ from B as follows:
   (a) First, let s_0 be the lowest segment;
   (b) Let s_{i+1} be highest segment of B such that either the left or the right endpoints of s_i and s_{i+1} are in the same grid subinterval, if it exists;
   (c) If no such segment above s_i exists, let s_{i+1} be the successor of s_i in B;
   (d) Let $i = i + 1$ and repeat 3b and 3c until the highest segment is reached;
4. For each s_i, create a $\tilde{s}_i$ by rounding each endpoint to the grid point immediately above s_i.

Given that the segments in the sets s_i and $\tilde{s}_i$ created with this procedure satisfy the two properties described above, they can be encoded in a few bits and used to create a 2D version of the fusion tree structure. Specifically, the $O(b)$ segments $\tilde{s}_0, \tilde{s}_2, ...$ can be encoded in $O(bh)$ bits, which can be packed into a single word of size w if we let $h = \lfloor \epsilon w/b \rfloor$ for a sufficiently small constant $\epsilon > 0$. Building a tree by picking the segments using this algorithm yields a tree with height at most $\log_b n + 2w/h = O(\log_b n + b)$.

Querying. At query time, similarly as done with lines segments, we can map the query point q to a point $\tilde{q}$ with integer coordinates in $[0, 2^h]$ in $O(1)$ time. Then, we first locate $\tilde{q}$ among $\tilde{s}_i$'s. Next, we can use word operations to find the position of $\tilde{q}$ between the segments $\{\tilde{s}_0, \tilde{s}_2, \tilde{s}_4, ...\}$, and then, another constant-time comparison can be done to locate $\tilde{q}$ among all $\{s_i\}$ and answer the query by recursively searching the sub-trees. In [5], Chan and Pătraşcu described in high level how to implement these comparisons using constant-time operations such as multiplications, divisions, shifts, and bitwise ANDs. To get a sublogarithmic query time, they suggest to choose $b = \lfloor \sqrt{\log n} \rfloor$, which yields $O(\log_b n + b) = O(\log n / \log \log n)$.

3.3 Practical Limitations

Chan and Pătraşcu's algorithm reached a *significant* theoretical speed-up for planar point location. In practice, however, the algorithm has limitations reminiscent of the fusion tree approach. The algorithm relies on packing multiple numbers into a single word w, which means that we need a large word size to achieve large branching factors.

In practice, the word size is fixed in common computer architectures (currently, $w = 64$ in most computers), thus we can not freely choose the branching factor b. For instance, if we fix $h = 7$, we get a integer grid with small ranges in $[0, 2^h] = [0, 2^7] = [0, 128]$. The *actual space* required to store a line segment is $2(h + 1)$ bits [5], then the branching factor becomes $b \leq \lfloor w/2(h + 1) \rfloor = 64/16 = 4$, which is close to to the branching factor of a binary search tree. In this case, b is only less than $\sqrt{\log n}$ for input of size $n \leq 2^8 = 65,536$.

Finally, Chan and Pătraşcu's algorithm conceivably has larger hidden constant factors than a simple binary search tree or binary search, and therefore it may not pay off in practice in current computer architectures. Similar limitations have been noted by Navarro for the predecessor search problem using 1D fusion trees [19] and even in Fredman and Willard's original paper [11].

4 Point Location using 1D Predecessor Search

The limitations described in Section 3.3 motivate the design of alternative algorithms on the Word RAM model that are able to achieve sublogarithmic query time, while not relying on the large machine word sizes. In this project, we explored several ideas of possible algorithms based on efficient (word RAM) predecessor search data structures. While none of these ideas were able guarantee $O(n)$ space and $o(\log n)$ query time in the worst case, they still may provide sublogarithmic query time if the input data meets certain conditions.

Point location via slab rotation. Our first observation is that, for certain inputs, we can create slab subdivisions that contain a single line segment. For instance, the slab from Figure 1a shows an example of a slab subdivision by horizontal lines where three subdivisions contain a single line segment. This suggests that an algorithm based on a 1D predecessor search (e.g., vEB trees [10], y-fast tries [24], and z-fast tries [3, 2]) may be used as a first step: we can build a 1D predecessor search structure for locating points among vertical subdivisions over the y-axis. At query time, whenever we find that a query point lies on a subdivision that contains a single segment, we only need a single constant-time point-line comparison to determine the face where the point query lies on. If there is more than one segment, we can fall back to a binary search over line segments. If we use data structures such as vEB-like trees, we get a query time of $O(\log\log U + 1) = O(\log w)$ for certain queries, and $O(\log n)$ in the worst-case queries. For typical computer architectures, where $w = 64$, this means a small constant time query cost (given that $\log_2 w = 6$).

The second observation is that some slab subdivisions that contain more than one segment, such as the top (green) subdivision in Figure 1b, can be rotated through the origin by an angle of θ degrees, such that it becomes possible to locate the point among the segments using 1D predecessor search (see, for example, the slab in Figure 1c). This suggests another improvement to the algorithm above: we can create a search tree structure where each node leads to a slab subdivision. A 1D predecessor search structure can be used again for locating which slab should be followed. When we locate a subdivision that contains s segments, we compare the query point to the segment $s/2$, a perform a recursive search over half of the slabs in the query subdivision.

This approach implies that with a predecessor search plus one segment comparison, we reduce the problem size from $O(n)$ to $O(n/4)$ in the *worst case* and $\Omega(1)$ in the best case. We get the $n/4$ reduction because the segment comparison can divide the problem by 2, and the predecessor search by another 2 (but possibly more). It easy to see that we can always find a rotation that subdivides the set of segments in at most $n/2$ segments (it suffices to pick the angle that transforms the segment in the middle into a horizontal line).

Point location via duality. We also considered an approach based on point-line duality. The slab structure has a rather interesting structure in the dual plane, which can be seen in Figure 2. The line segments in the slab generate the double-wedge structures in the dual plane. Point q, in the primary plane, that lie between two segments s_1 and s_2 translates to a line q^* that crosses the white diamond-shaped quadrilaterals created by the double-wedges of s_1 and s_2. This suggests that we can pre-compute the regions where q^* crosses a vertical line in the dual plane. However, this would lead to n^2 space given that the double-wedge lines create n^2 different subdivisions on a vertical line in the dual plane. While it seems to be possible to reduce the n^2 space factor to some sub-quadratic factor by partitioning the segments into a tree structure, it is not clear how to make this use only linear space.

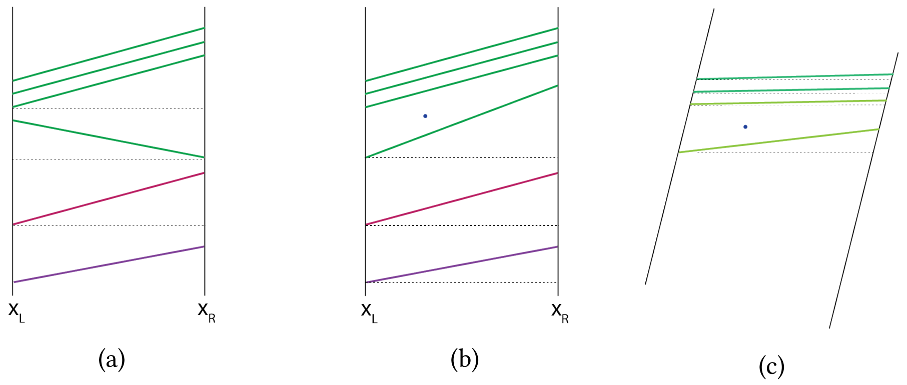

Figure 1: Examples of vertical slabs subdivisions by horizontal lines.

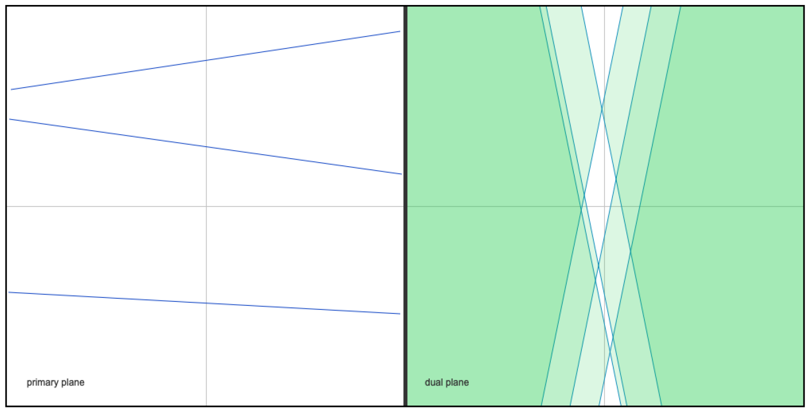

Figure 2: Structure of the slab in the dual plane. Figure generated using the app available in [15])

5 Acknowledgements

The text in this report has been written by this report's author, except for the transcribed excerpts that are explicitly noted in the text. The report is a result of the author's own bibliographic research, readings, and understanding of the cited material and papers. Besides information from the cited written material, Section 3.1 contains information exposed in multiple video lectures available online by Professors Jelani Nelson [20] and Erik Demaine [7]. **Except as described above, all the work on this project is my own. Signed: Aécio Santos.**

References

[1] Arne Andersson and Mikkel Thorup. "Dynamic ordered sets with exponential search trees". In: *Journal of the ACM (JACM)* 54.3 (2007), 13–es.

[2] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. "Dynamic z-fast tries". In: *International Symposium on String Processing and Information Retrieval.* Springer. 2010, pp. 159–172.

[3] Djamal Belazzougui et al. "Monotone minimal perfect hashing: searching a sorted table with O (1) accesses". In: *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms.* SIAM. 2009, pp. 785–794.

[4] Timothy M Chan. "Persistent predecessor search and orthogonal point location on the word RAM". In: *ACM Transactions on Algorithms (TALG)* 9.3 (2013), pp. 1–22.

[5] Timothy M. Chan and M. Pătraşcu. "Transdichotomous Results in Computational Geometry, I: Point Location in Sublogarithmic Time". In: *SIAM J. Comput.* 39 (2009), pp. 703–729.

[6] Bernard Chazelle and Leonidas J Guibas. "Fractional cascading: II. applications". In: *Algorithmica* 1.1-4 (1986), pp. 163–191.

[7] Erik Demaine. *MIT 6.851 Advanced Data Structures.* `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/lecture-videos/`. Online; accessed May 2, 2021. 2012.

[8] David Dobkin and Richard J Lipton. "Multidimensional searching problems". In: *SIAM Journal on Computing* 5.2 (1976), pp. 181–186.

[9] Herbert Edelsbrunner, Leonidas J Guibas, and Jorge Stolfi. "Optimal point location in a monotone subdivision". In: *SIAM Journal on Computing* 15.2 (1986), pp. 317–340.

[10] Peter van Emde Boas. "Preserving order in a forest in less than logarithmic time". In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975).* IEEE. 1975, pp. 75–84.

[11] Michael L Fredman and Dan E Willard. "Surpassing the information theoretic bound with fusion trees". In: *Journal of computer and system sciences* 47.3 (1993), pp. 424–436.

[12] Michael Hemmer, Michal Kleinbort, and Dan Halperin. "Optimal randomized incremental construction for guaranteed logarithmic planar point location". In: *Computational Geometry* 58 (2016), pp. 110–123.

[13] John J Iacono and Stefan Langerman. "Dynamic point location in fat hyperrectangles with integer coordinates". In: *Proceedings of the 12th Canadian Conference on Computational Geometry.* 2000, pp. 181–186.

[14] David Kirkpatrick. "Optimal search in planar subdivisions". In: *SIAM Journal on Computing* 12.1 (1983), pp. 28–35.

[15] Hengfeng Li. *Point and Lines Duality Demo.* `https://people.eng.unimelb.edu.au/henli/programs/duality-demo/`. Online; accessed May 11, 2021. 2012.

[16] Richard J Lipton and Robert Endre Tarjan. "Applications of a planar separator theorem". In: *SIAM journal on computing* 9.3 (1980), pp. 615–627.

[17] Ketan Mulmuley. "A fast planar partition algorithm, I". In: *Journal of Symbolic Computation* 10.3-4 (1990), pp. 253–280.

[18] Ketan Mulmuley. "Computational Geometry: An introduction through randomized algorithms". In: (1994).

[19] Gonzalo Navarro and Javiel Rojas-Ledesma. "Predecessor Search". In: *ACM Computing Surveys (CSUR)* 53.5 (2020), pp. 1–35.

[20] Jelani Nelson. *Harvard CS 224: Advanced Algorithms.* `https://people.seas.harvard.edu/minilek/cs224/fall14/lec.html`. Online; accessed May 2, 2021. 2014.

[21] Franco P Preparata. "A new approach to planar point location". In: *SIAM Journal on Computing* 10.3 (1981), pp. 473–482.

[22] Neil Sarnak and Robert E Tarjan. "Planar point location using persistent search trees". In: *Communications of the ACM* 29.7 (1986), pp. 669–679.

[23] Jack Snoeyink. "Point location". In: *Handbook of discrete and computational geometry.* 2017, pp. 1005–1028.

[24] Dan E Willard. "Log-logarithmic worst-case range queries are possible in space Θ (N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.