Methods for Achieving Fast Query Times in Point Location Data Structures

Michael T. Goodrich*
Center for Geometric Computing
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
E-mail: goodrich@cs.jhu.edu
URL: http://www.cs.jhu.edu/~goodrich

Mark Orletsky
Signature Modeling Branch
Army Research Laboratory
10235, Burbeck Road, Ste 110
Fort Belvoir, VA 22060
E-mail: orletsky@cs.jhu.edu

Kumar Ramaiyer†
Informix Software, Inc.
1111, Broadway, Suite 2000
Oakland, CA 94607
E-mail: rk@informix.com
URL: http://www.cs.jhu.edu/~kumar

Abstract

Given a collection $\mathcal{S}$ of n line segments in the plane, the planar point location problem is to construct a data structure that can efficiently determine for a given query point p the first segment(s) in $\mathcal{S}$ intersected by vertical rays emanating out from p. It is well known that linear-space data structures can be constructed so as to achieve $O(\log n)$ query times. But applications, such as those common in geographic information systems, motivate a re-examination of this problem with the goal of improving query times further while also simplifying the methods needed to achieve such query times. In this paper we perform such a re-examination, focusing on the issues that arise in three different classes of point-location query sequences:

- sequences that are reasonably uniform spatially and temporally (in which case the constant factors in the query times become critical),
- sequences that are non-uniform spatially or temporally (in which case one desires data structures that adapt to spatial and temporal coherence), and
- sequences that must be performed in space-limited environments (in which case one desires sub-linear space data structures).

We present and analyze simple methods for adapting previous point location approaches to each of these environments. Our analysis consists of both a theoretical analysis of the constant factors in asymptotic query times as well as an experimental analysis over a range of subdivision and query domains.

1 Introduction

Planar point location is a classic and important problem in computational geometry. In this problem one is given a collection $\mathcal{S}$ of n line segments in the plane, and asked to construct a data structure that can efficiently determine for a given query point p the first segment(s) in $\mathcal{S}$ intersected by vertical rays emanating out from p. This problem is very well-studied [13, 14, 17, 24, 25, 31, 36], and there are a number of solutions that asymptotically achieve query times of $O(\log n)$ using $O(n)$ space, which is optimal. The query time in each of these solutions is bounded by the number of *point-line comparisons*, where one is given a point p and an oriented line L and asked to determine if p is on L, to the left of L, or to the right of L.

We are interested in re-examining the point location problem, focusing on issues that arise in important application areas, such as geographic information systems (GIS) [6, 7, 18, 35, 40]. Our re-examination of the point-location problem focuses on a number of practical issues that have not been explicitly explored in previous work, including an examination of the exact number of point-line comparisons used in comparison-based point location methods, as well as an examination of scenarios where the set of queries is non-uniform and one desires a data structure that can "adapt" over time to the non-uniformities. Specifically, we consider non-uniform query sequences that satisfy one of the following properties:

Spatial Coherence: This is the property that some regions (defined implicitly by certain trapezoids) are accessed more frequently than other regions. Such an environment could arise, for example, in a service-dispatch GIS where certain high-population regions create more demand requests than low-population regions.

Temporal Coherence: This is the property that between any two consecutive queries to the same cell the set of queries, which we refer to as the *working set*, is fairly small. Such an environment could arise, for example, in a robot motion tracking system where a moving robot periodically requests

*This research supported by the NSF under Grant CCR-9300079, and by ARO under grant DAAH04-96-1-0013.

†This research supported by the NSF under Grant CCR-9300079, and by ARO under grant DAAH04-96-1-0013.

map information based upon its global positioning.

Our goal in each case is to provide comparison-based data structures that have $O(\log n)$ worst-case query times (with small constant factors) for all sequences, but have $o(\log n)$ query times (possibly even $O(1)$) for sequences that exhibit a sufficient amount of one of the above properties.

1.1 Previous Work. We are not aware of any previous work on the planar point location problem that examines specialized query environments such as those described above, but there is a wealth of literature on comparison-based solutions to the general planar point location problem. The first non-trivial solution to this problem is a method by Dobkin and Lipton [14], based upon a simple, elegant *slab* method, that answers queries using at most $2\lceil \log n \rceil$ point-line comparisons[1]. The main idea of this method is to subdivide the plane into "slabs" by placing a vertical line through each segment endpoint followed by an ordering of the segments crossing each slab by the "above" relation. A location query for a point p is easily answered by performing two binary searches—one to locate the slab containing p and another to locate the cell in that slab that contains p. Using an interesting *trapezoid* technique, Preparata [31] showed that one could achieve a query time of $4\lceil \log n \rceil$ using $O(n \log n)$ space, and Kirkpatrick [24] was the first to show that one could in fact simultaneously achieve an $O(\log n)$ query time and $O(n)$ space. His method is based upon a beautiful *hierarchical subdivision* method, but the constant factor in the running time is fairly large. Because of the questions of practicality raised by this method, Edahiro, Kokubo, and Asano [16] performed a series of empirical benchmarking tests between the trapezoid and hierarchical subdivision methods, as well as a simple "bucketing" approach, which is not entirely comparison-based and actually has a worst-case query bound of $\Theta(n)$ and a worst-case space bound of $\Theta(n^2)$. Still, based upon their benchmarking tests, they argued that the best practical method at that time was the bucketing approach, with the trapezoid method being a close competitor. Following this, Cole [13], Sarnak and Tarjan [36], and Edelsbrunner, Guibas, and Stolfi [17] independently showed how to improve the constant factor in the query time for point location while still achieving linear space. These three methods are described quite differently, but the resulting data structures are remarkably similar. If optimized for the constant factors in their query times, the methods of Cole and Edelsbrunner *et al.* appear to require $3\lceil \log n \rceil$ time for point-location queries and the method of Sarnak and Tarjan appears to require $5\lceil \log n \rceil$ time[2]. We are not familiar with any previous linear-space methods that have a query time approaching the $2\lceil \log n \rceil$ time achieved by the slab method.

1.2 Our results. Because of the great diversity of uses for planar point location, we consider a variety of domains for point-location queries, based upon the environment in which point location queries will be made. In the first domain point location queries are fairly uniform spatially and temporally. That is, location queries are fairly well spread across all cells defined by "slabs" and the i-th query has little or no relation to previous queries. For such environments we show how to achieve a linear-space data structure that can answer queries in at most $2\lceil \log n \rceil + o(\log n)$ point-line comparisons. For query sequences with spatial or temporal coherences, however, we show how to design an adaptive point location method, so that the amortized time[3] complexity for accessing a cell i in a sequence of m queries is $\bar{O}(\min\{\log n, \log(t(i)+1), \log(m/f(i))\})$ time, where $t(i)$ is the working set size for cell i (i.e., the number of different queries between two accesses of cell i) and $f(i)$ is the frequency of accesses to cell i. Note that this method simultaneously adapts itself to query environments that satisfy spatial or temporal coherence. In addition, we consider environments in which the collection of segments $\mathcal{S}$ is too large to conveniently accommodate a data structure requiring even just a linear amount of additional memory. In this case we examine an approach based on a simple "walk-through" technique often used in GIS applications, which can be used, for example, to achieve a sub-linear amount of additional space while still allowing for $O(\log n)$-time queries. Recently Mücke *et. al.* [29] proposed a randomized point location method which is also based on "walk-through" idea, but their analysis was restricted to Delaunay triangulations.

Because of the motivation of our re-examination of point location from practical considerations, we have implemented a number of different point-location methods, some based upon previous approaches and others based upon our techniques. We compare each of

[1] Most of the previous papers we review do not actually report the constant factors in the number of point-line comparisons needed to answer a query. We give here our best estimates for the constant factors in the worst-case point-line comparison counts of the previous methods.

[2] Although the resulting methods are quite similar, the differences in the constant factors come from the fact that the Cole and Edelsbrunner *et al.* methods use static complete binary tree as their primary structures, whereas the Sarnak and Tarjan method uses a dynamic red-black tree as its primary structure.

[3] We us $\bar{O}(f(n))$ to denote an amortized complexity bound that is $O(f(n))$.

these methods in a fairly comprehensive set of empirical benchmarking tests, comparing them over a variety of scenarios for subdivision and query distributions. Our experimental results give strong evidence supporting our techniques being competitive with "bucketing" approaches for "evenly-distributed" subdivisions and queries, while being significantly better than the bucket method in non-uniform environments.

In the sections that follow we outline the main ideas behind our methods.

2 Point Location in Uniform Query Environments

We begin by describing our method for environments where the queries are fairly uniform spatially and temporally, although the subdivisions themselves may be quite "unbalanced." We first present a simple method for deterministically constructing a data structure of size $O(n \log n)$ that answers point location queries with $2\lceil \log n \rceil$ point-line comparisons, and we then show how to reduce the space to be $O(n)$.

2.1 A Simple Persistent Method. Let $\mathcal{S}$ be a collection of n line segments in the plane. We can construct a planar point location data structure for $\mathcal{S}$ by applying the *persistence* [15] paradigm to a left-to-right plane sweep of $\mathcal{S}$ using a vertical line, L, as noted by Sarnak and Tarjan [36] (and Cole [13] as well, using a different terminology). Applying the persistence paradigm to a data structure D allows one to perform a sequence of updates to D and then look up the information stored in any previous version of D. Driscoll *et al.* [15] show that any linked data structure D with fixed in- and out-degree can be made persistent, with the space bound being proportional to the total number of changes to D, while keeping the access time proportional to its previous value (but multiplying the time by a constant factor that is at least 2). Several researchers (e.g., see Sarnak and Tarjan [36] for a survey) have noted that a simple *path copying* technique can make any binary tree data structure persistent with an access time exactly what it was in its non-persistent version. In this technique one simply copies the entire root-to-leaf path for a leaf-node change, keeping unchanged pointers pointing to their old neighbors.

Following an observation of Cole [13], we note that the segments in $\mathcal{S}$ form a partial order by the "above" relation. Moreover, this relation can easily be constructed in $O(n \log n)$ time, and then linearized via a topological sort in $O(n)$ additional time. We can therefore label each edge in $\mathcal{S}$ with its rank in this linear order and these ranks will define a consistent ordering of the segments in every slab defined by the endpoints of the segments in $\mathcal{S}$. This implies that we can use a static $\lceil \log n \rceil$-height tree T to represent the segments intersecting the sweep line L as it moves from left to right. Moreover, if we use the simple path copying technique to make this sweep persistent, then we derive the following lemma:

Lemma 2.1: *Given a set $\mathcal{S}$ of n non-intersecting line segments (except possibly at segment endpoints), one can construct a point location data structure for $\mathcal{S}$ in $O(n \log n)$ time and space that achieves a query time of $2\lceil \log n \rceil$.*

Proof. The time and space bound follow from the discussion above. The query time is derived as for the slab method—there is one binary search to find the appropriate previous version of the search tree followed by a $\lceil \log n \rceil$ time search down this tree. ■

We note that this simple method achieves the same space bound as that of the trapezoid method [31] but has a worst-case query time that is twice as fast. We can further improve the space bound of our method to be linear, however, while still keeping very close to the $2\lceil \log n \rceil$ query time.

2.2 Space improvement via ϵ-cuttings. In the context of planar point location data structures an ϵ-cutting of a set $\mathcal{S}$ of line segments is a partitioning of the plane into trapezoids so that the number of segments in $\mathcal{S}$ intersecting any trapezoid is at most $\epsilon|\mathcal{S}|$. For any trapezoid τ we use $\mathcal{S}_\tau$ to denote the *conflict list* for τ—the set of segments in $\mathcal{S}$ that intersect τ's interior. In this subsection we make use of the following lemma about ϵ-cuttings to reduce the space needed for our planar point location data structure to $O(n)$:

Lemma 2.2: *Let $\mathcal{S}$ be a set of n non-crossing line segments in the plane. One can construct a $(1/r)$-cutting $\mathcal{C}$ of $\mathcal{S}$, together with the conflict lists for all its trapezoids, so as to have $O(r)$ cells and total size $O(n)$, for $2 \leq r \leq n$. This construction can be implemented by a randomized algorithm in time $O(n \log n + r\delta \log n + n\delta)$ with probability $1 - 1/2^\delta$, for any $\delta \geq 1$, and deterministically in polynomial time.*

Our proof of this lemma (which we omit in this preliminary version) is based upon the general theory of *geometric range spaces* [2, 9, 20, 26]. In the context of point location data structures a range space is defined by a set $\mathcal{S}$ of line segments and $\mathcal{R}$, the set of all combinatorially distinct ways of intersecting segments of $\mathcal{S}$ with trapezoids that have vertical parallel edges. The sets in $\mathcal{R}$ are called *ranges*. Let Y be a subset of $\mathcal{S}$, and let a parameter $r \in [1, n]$ be given. Further, let $N_Y(s, \mathcal{S})$ denote the number of ranges R in $\mathcal{R}|_Y$ such that $s = |R|$ and $Y \cap R = \emptyset$ (we say such ranges are

missed by Y). Define $f_0(r)$ to be the expected number of missed ranges generated by an r-sized random sample S of $\mathcal{S}$ (with all such samples equally likely). Y is a $(1/r)$-*semi-net*[4] of order $\omega \geq 0$ if

$$\sum_{0 \leq t \leq r} N_Y(tn/r, \mathcal{S}) \max\{t^\omega, 1\} \;=\; O(f_0(r)),$$

where the sum ranges over all values of t from 0 to r for which $N_Y(tn/r, X)$ is non-zero. (Y is simply an ϵ-*net* [23] if $N_Y(tn/r, \mathcal{S}) = 0$ for $t > 1$.)

Lemma 2.3: *Let $(\mathcal{S}, \mathcal{R})$ be a segment-trapezoid range space. If Y is a subset of X defined by n mutually-independent indicator random variables, each of which is 1 with probability r/n, then, with probability at least 1/2, Y is a $(1/r)$-semi-net of order $\omega \leq n/2$ with size $O(r)$, provided that f_0 is non-decreasing.*

Proof. The proof follows from applications of general proof techniques of Chazelle and Friedman [9] and Clarkson and Shor [12] for range spaces with finite VC-dimension (see also [27]). ∎

In our case, $f_0(r)$ is equal to the number of trapezoids defined by a trapezoidal decomposition of an r-segment subset $S \subseteq \mathcal{S}$; hence $f_0(r)$ is $O(r)$. We omit the details of the proof of Lemma 2.2 in this preliminary version. We note, however, that the main challenge in establishing this lemma is proving the time and high-probability bounds, as the combinatorial part of the proof is based upon using a known double-sampling technique of Chazelle and Friedman [9] in conjunction with Lemma 2.3.

Our method for constructing a point-location structure for $\mathcal{S}$, then, is as follows:

1. Apply Lemma 2.2 with $\delta = c \log n$, for some constant $c \geq 1$, to find a $(1/r)$-cutting $\mathcal{C}$ consisting of $O(r)$ trapezoids and their conflict lists, so as to have $O(n)$ total size, for $r = n/\log^2 n$. This step takes $O(n \log n)$ time with probability at least $1 - 1/n^c$.

2. Apply Lemma 2.1 to form a point-location data structure D for $\mathcal{C}$ of size $O(r \log r) = O(n/\log n)$.

3. For each range R_τ defined by a trapezoid τ in $\mathcal{C}$, apply Lemma 2.2 with $\delta = c \log n / \log\log n$, for some constant $c \geq 1$, to find a $(1/r_\tau)$-cutting $\mathcal{C}_\tau$ for τ's conflict list consisting of $O(r_\tau)$ trapezoids and their respective conflict lists so as to have $O(n/r) = O(\log^2 n)$ total size, for $r_\tau = r/(\log\log n)$. Note that each conflict list in this second cutting is of size at most $O((\log\log n)^2)$. We implement this step with a "termination condition," however, that terminates the computation for τ if its running time exceeds $C \log^3 n$, which will occur, of course, with probability at most $1/2^\delta$. If this occurs, then we just restart the computation of this step for τ.

4. Apply Lemma 2.1 to form a point-location data structure D_τ for each $\mathcal{C}_\tau$ of size $O(r_\tau \log r_\tau) = O(\log^2 n/\log\log n)$.

We can perform a point location for some query point p, then, as follows. We search in D to locate the trapezoid τ in $\mathcal{C}$ containing p. We then search in D_τ to locate the trapezoid γ in $\mathcal{C}_\tau$ containing p. Finally we search in the conflict list for γ (with respect to $\mathcal{C}_\tau$) to complete the search for p. By Lemma 2.1, the total time for this search is

$$\begin{aligned} & 2\lceil \log |\mathcal{C}| \rceil + 2\lceil \log |\mathcal{C}_\tau| \rceil + \lceil (\log\log n)^2 \rceil \\ \leq \;& 2\left\lceil \log\left(\frac{bn}{\log^2 n}\right)\right\rceil + 2\left\lceil \log\left(\frac{b\log^2 n}{(\log\log n)^2}\right)\right\rceil \\ & + \lceil (\log\log n)^2 \rceil \\ \leq \;& 2\lceil \log n \rceil + o(\log n), \end{aligned}$$

where b is some constant. The total space needed by this data structure is $O(n)$. Let us, therefore, analyze the time needed to construct this data structure. We have already noted that Steps 1, 2, and 4 can be implemented in $O(n \log n)$ time with probability at least $1 - 1/n^c$ for any constant $c \geq 1$. Likewise, by a simple Chernoff bound analysis [22], we can show that Step 3 also runs in time $O(n \log n)$ with probability at least $1 - 1/n^c$ for any constant $c \geq 1$. Therefore, we have established the following:

Theorem 2.1: *Given a set $\mathcal{S}$ of n non-intersecting line segments (except possibly at segment endpoints), one can construct an $O(n)$-space comparison-based point-location data structure for $\mathcal{S}$ in $O(n \log n)$ time, with probability at least $1 - 1/n^c$, for any constant $c \geq 1$, that can answer point-location queries using at most $2\lceil \log n \rceil + o(\log n)$ point-line comparisons.*

Thus, we can achieve a query time very close to that of the slab method [14] with just a linear amount of additional space. Indeed, we conjecture that $2\lfloor \log n \rfloor$ is a lower bound on the query time for any linear-space comparison-based planar point-location data structure.

3 Adaptive Point Location

Let us now address the query environment where we anticipate that the sequence of queries is non-uniform with respect to space or time. In this section we show that in such environments one can achieve $o(\log n)$

[4] This definition of a semi-net is similar to the $(1/r)$-semi-cutting notion introduced by Chazelle [8], as well as proof techniques given in [9, 20, 26].

(amortized) query times while still maintaining fast preprocessing bounds and linear space.

3.1 Splay Trees. We achieve our results concerning adaptive point location queries by employing the splay tree data structure of Sleator and Tarjan [37]. Splay trees are self-adjusting binary trees, and they form a simple and very interesting class of "balanced" binary search trees. We highlight the term balanced, because, unlike other binary search trees such as AVL trees [1] or red-black trees [21, 39], splay trees do not enforce explicit global structural constraints. Sleator and Tarjan define a *splay* operation at a node v, which involves a series of rotations to take v to the root (but care is taken here, as the naive sequence of rotations will not derive the desired results). A splay is performed at a node v in a splay tree T after node v is accessed, independent of whether v was the objective for a search or for an update. Other than this, the access and update operations are performed as for other binary search trees. Sleator and Tarjan show that under these conditions, they can achieve an $\bar{O}(\log n)$ time complexity for all the access and update operations. The space requirement for a splay tree is clearly $O(n)$. Since no explicit information (like rank or height) is maintained to achieve the balance, the constants in the space bound are actually better than those for other search trees. Also Sleator and Tarjan prove a vast number of useful theorems regarding a sequence of m accesses in a splay tree, which we summarize below.

Theorem 3.1 Balance Theorem [37]: *The total access time is $O((m+n)\log n)$.*

Theorem 3.2 Static Optimality Theorem [37]: *If every item is accessed at least once, then the total access time is $O\left(m + \sum_{i=1}^{n} f(i)\log\left(\frac{m}{f(i)}\right)\right)$, where $f(i)$ is the access frequency of item i in m accesses.*

Let i_j be the item accessed at search j, where j ranges from 1 to m. Let $t(j)$ denote the number of different queries before search j and since the last access of item i_j or since the beginning of the sequence if j is the first access of the item i_j.

Theorem 3.3 Working Set Theorem [37]: *The total access time is*

$$O(n\log n + m + \sum_{j=1}^{m}\log(t(j)+1)).$$

An important fact here is that splay trees achieve all the above behaviors *automatically*. The splay heuristic that realizes the above behaviors is blind to the properties of the access sequence and to the global structure of the tree.

3.2 Persistence of Splay Trees. In this section we discuss how to apply the persistence paradigm of Driscoll *et al.* [15] to the splay operations that we later use in our algorithm for adaptive point location queries.

The main idea is a variant of the path copying technique. We give the details in the full version. The space of the data structure increases by $O(1)$ per rotation during a splay. But the number of rotations is bounded nicely for the temporal and spatial coherent queries as shown in section 3.1. Thus we have the following:

Lemma 3.1: *The splay operation in version i of a persistent search tree data taking time $t(n_i)$ increases the space of the data structure by $\bar{O}(t(n_i))$, where n_i is the number of nodes in the persistent structure at version i.*

Our adaptive point location method, then, involves the following steps:

1. First construct a persistent search structure for the given planar subdivision. Initialize all the cells of the subdivision which correspond to one or more nodes in the data structure to have a weight of one. Build a splay tree on top of the roots of the different versions of persistent structure. We refer to this structure as the *horizontal* structure.

2. We keep a count and record of the queries performed on the subdivision and also maintain a history information as weights in the cells into which the query points fall. For queries numbering from one to n, we perform the query operation using the horizontal and persistent structures. We do a splay operation in both the horizontal and persistent structures, but do path copying only in the persistent structure. We increase the weight of the node in the persistent structure that contains the query point by one.

3. After n query operations, we reconstruct the persistent data structure and also build the horizontal structure. For building the persistent search structure we use the weights of the nodes to *bias* the depth of the nodes. As a result, the nodes of larger weights will be at smaller depth. Our persistent tree is therefore now a *globally biased search tree* [5] rather than a simple complete balanced binary tree, but we can still bound its height to be $O(\log n)$. This step takes $O(n)$ time and space, which can be amortized by charging $O(1)$ time to each of the previous n queries.

4. We then re-perform the last $\sqrt{n}$ query and splay operations on the persistent and the horizontal structures. We do this to reestablish the temporal

coherence and the working set on the reconstructed data structure.

5. We repeat the previous three steps for the next n query operations on the newly constructed data structure.

We now analyze the complexity of the construction of our data structure, the space requirement, and the cost of the query operations. Step 1 takes $O(n \log n)$ time and is performed only once. The data structure created uses $O(n)$ space. Step 2 takes time proportional to the total access time in all the splay trees. The increase in space for spatial and temporal coherent queries is $\bar{O}(1)$ per query (see static finger theorem and working set theorem in section 3.1), and for other arbitrary queries the increase in space is $\bar{O}(\log n)$. Therefore the increase in space for n queries in step two is $O(n)$ for adaptive point location queries. The rest of the steps clearly require linear time. This result combined with the properties of the splay trees outlined in section 3.1 achieve the following result. Suppose we label the cells in the region as $1, 2, \cdots, O(n)$. Let i_j be the cell accessed during the search j, let $t(j)$ be the number of different cells accessed before search j since the last access of cell i_j, and let $f(i_j)$ be the frequency of access of cell i_j. Then we have,

Theorem 3.4: *Given a set $\mathcal{S}$ of n non-intersecting line segments (except possibly at segment endpoints), one can construct an $O(n)$-space point location data structure for $\mathcal{S}$ in $O(n \log n)$ time that achieves a query time over a sequence of m queries that is*

$$\bar{O}\left(\log \min\left\{n, m/f(i_j), t(j)+1\right\}\right),$$

where the cells in region of $\mathbb{R}^2$ are defined by the "slabs".

Thus, we have achieved an adaptive point location method.

4 Space-Limited Environments and Epsilon Cutting Methods

The final environment we consider is that in which the amount of additional space is limited to be sub-linear. In this case, we assume the subdivision defined by $\mathcal{S}$ is already a trapezoidal map or a triangulation, with the adjacencies given. For such cases we can easily adapt our Theorem 2.1 to use a two-level $(1/r)$-cutting of size $O(r)$ with $r = n/\log n$. We can perform a point location query, then, by searching down a point location structure defined upon this set (say, using Theorem 2.1 itself) and then resolve the last $\lceil \log n \rceil$ segments by traversing through $\mathcal{S}$ itself towards the query point. In fact, this approach gives rise to an extremely simple method for performing point locations in small subdivisions: just store a random sample of the edges of size $O(\sqrt{n})$. One can answer a query, then, in $O(\sqrt{n})$ expected time (independent of input distribution) by finding the closest segment in this random sample and then traversing $\mathcal{S}$ from this edge to the query point.

We implemented this method and other variations which we discuss in the next section.

5 Experimental Results

In this section we describe experiments conducted on new and existing methods of planar point location and give some discussion as to the results observed.

5.1 Experimental Setup. The simple persistence method based on path copying, epsilon net methods of planar point location (which are simple variants of our ϵ-cutting method), adaptive point location methods, as well as the Edahiro *et al.* [16] bucket method, were implemented and experiments were conducted to compare the query times of the various methods.

All algorithms were implemented in C++ using the LEDA [28, 30] library of data structures and algorithms (v3.2.1), and they were compiled on a *SUN SPARC station ELC* running SUN OS Release 4.1.1 with the g++ compiler (v4.2).

Three different classes of input subdivisions were used to test the performance of the above algorithms. These are the *uniform* subdivision or Delaunay triangulation, a random triangulation produced by the LEDA package which we call the *LEDA* subdivision, and a highly non-uniform subdivision that we call *diagonal graphs*. Numerous instances of each of these types of subdivisions were generated and they ranged in size from 1000 vertices to 30,000 vertices. Our motivation behind this selection of inputs is that we wanted a method whose performance is oblivious to the distribution of the edges and the shapes of the faces in the subdivision.

We used three types of query data for testing. We used query data distributed uniformly over the entire domain, spatially coherent query data, i.e., queries that are restricted to lie in a small cell in the subdivision, and temporally coherent query data wherein the number of different query points is a fixed constant.

5.2 Experiment 1: Uniformly Distributed Query Data. We conducted extensive experiments using these different methods on different types of input subdivisions. We summarize our conclusions here.

For the bucket method experiments were conducted with linear sized grids with $\alpha * \sqrt{n}$ divisions along x and y axis, and we varied the constant α to be 1, 2, and

3 [16]. We tested epsilon-net methods with net sizes $\sqrt{n}$ and $n/\log n$.

For each experiment, we used subdivisions of sizes ranging from 1000 to 30,000 vertices. For each such size, five different subdivisions of that type were selected. And for each such selection, several runs were conducted by selecting different random samples of a given size, which would expect to be epsilon nets, and for each run 10,000 queries were performed. We then computed an average of these values to obtain the probe time for a given method on a particular type of subdivision of given size and for a sample of a certain size.

Figure 1 shows the results for experiments conducted on Delaunay subdivisions. The epsilon-net method based on red-black tree with net size $n/\log n$, bucket methods with α equal to two and three, and simple persistence method all perform equally well. As the constant α is increased, the bucket method performs better as expected.

Figure 2 shows the results for experiments conducted on LEDA subdivisions which are basically random triangulations. The bucket method performs very poorly as the number of triangles intersecting a grid cell is large leading to larger probe time. The epsilon-net methods perform well as the randomness in the algorithm takes care of the bad distribution.

Figure 3 show similar results for experiments conducted on diagonal subdivisions.

We also conducted experiments to test the relative quanlity of one random sample (expected epsilon-nets) to another. We use the average number of faces traversed in the second phase of the algorithm as a measure of quality of a given net. We computed the mean and standard deviation for the number of faces traversed for fifteen different random epsilon-nets. These fifteen means are presented in Fig. 4 as Gaussian curves, each of which has a mean that is the same as the sequence being observed and a standard deviation that corresponds to that of same sequence.

The 15 means have values ranging from 6.59 through 9.21, and they have a median value of 7.17. Thus, the average number of faces crossed in the best net of the fifteen is 72% of that of the worst net and 92% of that of the median net. This suggests that if point location speed is critical, and preprocessing time is abundant, one may wish to observe the performance of a number of random samples during the preprocessing phase and choose the best of those observed for the actual net to be used. Alternatively, if preprocessing time is not so abundant, one can be confident that the randomly selected net that is chosen on the first try is not likely to be much worse than the best if a few random samples were selected and the best one was used.

5.3 Experiment 2: Spatial Coherent Query Data. We implemented the adaptive point location method using persistent splay tree and conducted several experiments. We allowed the query range (a rectangular box within the bounding box of the subdivision) to be a parameter. The query range was perturbed randomly to lie anywhere in the bounding box and the query points are then chosen randomly from the relocated query region. The subdivision was chosen to lie within a unit square. We experimented with query regions which are of squared shape and of size 0.01×0.01, 0.05×0.05, and 0.1×0.1, respectively.

Figure 5 shows results for experiments on Delaunay subdivisions of various sizes with query points restricted to lie within a region of size 0.01×0.01. This figure shows the results for three methods: namely, the bucket method, the persistence method that uses red-black trees, and the persistence method that uses splay trees.

Each method was tested on Delaunay triangulation of sizes ranging from 1000 to 15,000 vertices. For each size, five different subdivisions were selected for testing and on each subdivision three different query boxes of given size were tested. Each test consisted of 10,000 probe points. These values were averaged for each of the fifteen tests which were then averaged to get the time per probe for a subdivision of given size. We observe that the splay tree method gives a better performance than the other two methods. We also observe that the bucket method has a probe time that is essentially constant (as expected for Delaunay triangulation), but the constant is larger than the other two methods.

Figures 6 and 7 show the results for similar spatial experiments conducted on LEDA subdivisions and diagonal subdivisions, respectively. We observe that the splay tree method gives a better performance than the other two methods as expected. The bucket method performs very poorly for both subdivisions. The bucketing strategy we used here ($\sqrt{n} \times \sqrt{n}$ sized grid) fails as there are number of long triangles with bad aspect ratio in both subdivisions.

We conclude that to get consistent performance for different types of subdivision for adaptive point location queries one needs special techniques to adapt the underlying data structure storing the given subdivision.

5.4 Experiment 3: Temporal Coherent Query Data. We now discuss the adaptive point location method for temporally coherent query point distributions. In this case we allowed the working set size to be a parameter. Query points equal in number to the size of the working set were drawn randomly within the bounding box of the subdivision. These points form the set of points from which all the query points were

drawn. The subdivision was chosen to lie within a unit square. We experimented with working sets of sizes 1, 20, and 50, respectively.

Figure 8 shows results for experiments on Delaunay subdivisions of various sizes with query points drawn from a working set of size 20.

Each method is tested on Delaunay triangulation of sizes ranging from 1000 to 15000 vertices. We conducted experiments similar to the spatial coherent tests. We observe that the splay tree method gives a better performance than the other two methods.

Figures 9 and 10 show the results for similar temporal coherent experiments conducted on LEDA subdivisions and diagonal subdivisions, respectively. We observe that the splay tree method again gives a better performance than the other two methods as expected.

We conclude that it helps to pay attention to the nature of query distribution and to adapt the data structure to the application requirements. Also, one needs to choose methods that perform well for all types of input distributions.

6 Conclusions

In this paper we outline strategies for performing point location queries very fast using linear or sub-linear space. Moreover, we define methods that are efficient for a number of different query environments, including uniform query environments, non-uniform query environments, and space-limited query environments. We leave as an open problem the proof or disproof of our conjecture that $2\lfloor \log n \rfloor$ is a lower bound for the query time in a linear-space comparison-based point location data structure.

All of our methods assume that the planar subdivision does not change over time, but environments allowing for dynamic changes to the subdivision over time are well-motivated and well-studied, as well [3, 4, 10, 11, 19, 32, 33, 34, 38]. Thus, another interesting open problem is whether one can, say, achieve the adaptive query bounds of Theorem 3.4 in such dynamic environments, where one allows insertions and deletions of vertices and edges in the subdivision $\mathcal{S}$.

References

[1] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.*, 3:1259–1262, 1962.

[2] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proc. 35th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 683–694, 1994.

[3] M. J. Atallah, M. T. Goodrich, and K. Ramaiyer. Biased finger trees and three-dimensional layers of maxima. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 150–159, 1994.

[4] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17:342–380, 1994.

[5] S. Bent, D. Sleator, and R. Tarjan. Biased search trees. *SIAM Journal of Computing*, 14(3):545–568, August 1985.

[6] T. Bernhardsen. *Geographic Information Systems*. VIAK IT AS./Norwegian Mapping Authority, 1992.

[7] J. K. Berry. *Beyond Mapping: Concepts, Algorithms, and Issues in GIS*. GIS World, Fort Collins, CO, 1993.

[8] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10:377–409, 1993.

[9] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[10] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992.

[11] Y.-J. Chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM J. Comput.*, to appear. Preprint: Technical Report CS-92-07, Comput. Sci. Dept., Brown Univ. (1992).

[12] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[13] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.

[14] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[16] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency: comparison with existing algorithms. *ACM Trans. Graph.*, 3:86–109, 1984.

[17] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[18] W. R. Franklin. Computer systems and low level data structures for GIS. In D. Maguire, D. Rhind, and M. Goodchild, editors, *GIS: Principles and Practice*, volume 1, pages 215–225. Longman Higher Education and Reference, London UK, 1991.

[19] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[20] M. T. Goodrich. Geometric partitioning made easier, even in parallel. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 73–82, 1993.

[21] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes in Computer Science, pages 8–21, 1978.

[22] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(10):305–308, 1990.

[23] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987.

[24] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[25] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[26] J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.*, 6:385–406, 1991.

[27] J. Matoušek. Epsilon-nets and computational geometry. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 69–89. Springer-Verlag, 1993.

[28] K. Mehlhorn and S. Näher. Leda, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[29] E. Mucke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *Proc. 12th ACM Symposium on Computational Geometry*, pages 274–283, 1996.

[30] S. Näher and C. Uhrig. *The LEDA User Manual Version R 3.2*. Max-Planck-Institut für Informatik 66123, Saarbrücken, Germany.

[31] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[32] F. P. Preparata and R. Tamassia. A fully dynamic planar point location technique. Report UILU-ENG-87-2266, Coordinated Sci. Lab., Univ. Illinois, Urbana, IL, 1987.

[33] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830, 1989.

[34] F. P. Preparata and R. Tamassia. Dynamic planar point location with optimal query time. *Theoret. Comput. Sci.*, 74:95–114, 1990.

[35] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.

[36] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[37] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[38] R. Tamassia. An incremental reconstruction method for dynamic planar point location. *Inform. Process. Lett.*, 37:79–83, 1991.

[39] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial Applied Mathematics, 1983.

[40] D. Tomlin. *Geographic Information Systems and Cartographic Modeling*. Prentice Hall, 1990.

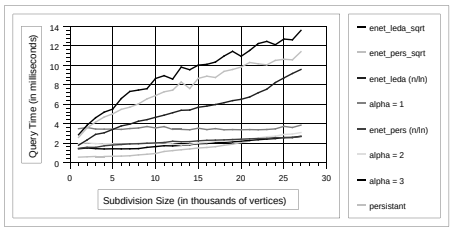

Figure 1: Average Query Times for different methods on *uniform* subdivisions. The plots labeled $\alpha = 1$, $\alpha = 2$, and $\alpha = 3$ represent the bucket method with the α parameter set to the value noted. The plot labeled *persistent* represents the path-copying persistence method. The other plots are for the ϵ-net methods, using either node-copying persistence or LEDA persistence, with net sizes of either the square root of n or $n/\log n$, as noted.

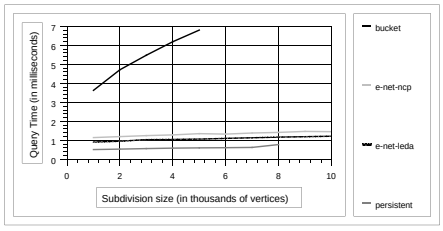

Figure 2: Average Query Times for different methods on LEDA Subdivision.

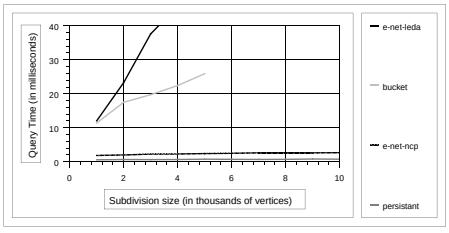

Figure 3: Average Query Times for different methods on Diagonal Subdivision.

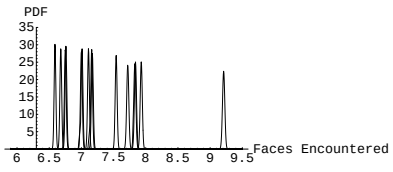

Figure 4: The mean and standard deviations, represented as Gaussian curves, for each of fifteen epsilon nets chosen.

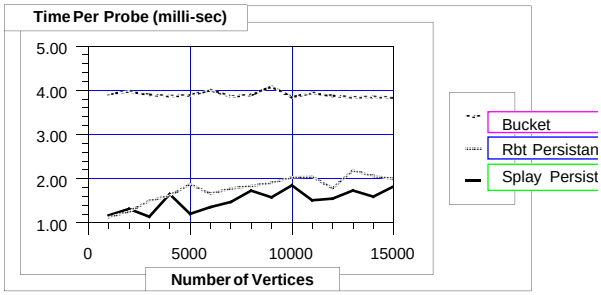

Figure 5: Spatial Coherent Experiments on Unif Subdivisions. The Query Region is a Square of 0.01×0.01.

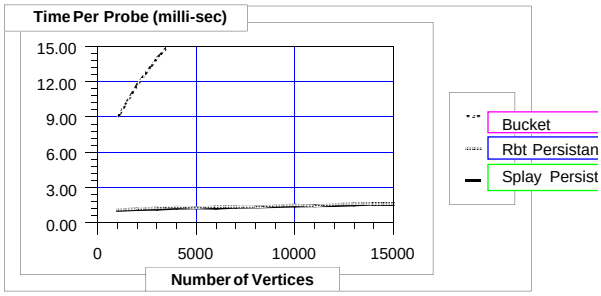

Figure 6: Spatial Coherent Experiments on LE Subdivisions. The Query Region is a Square of 0.01×0.01.

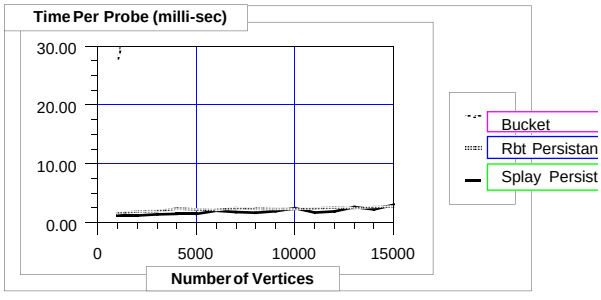

Figure 7: Spatial Coherent Experiments on Diag Subdivisions. The Query Region is a Square of 0.01×0.01.

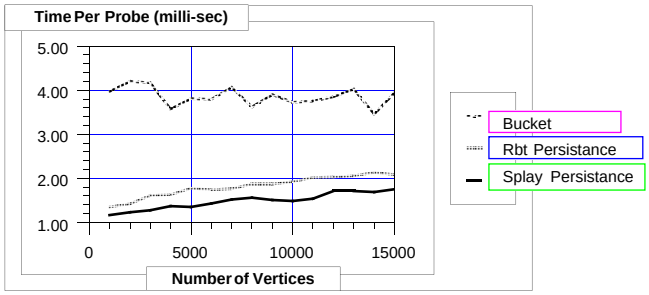

Figure 8: Temporal Coherent Experiments on Uniform Subdivisions. The Working Set Size is 20.

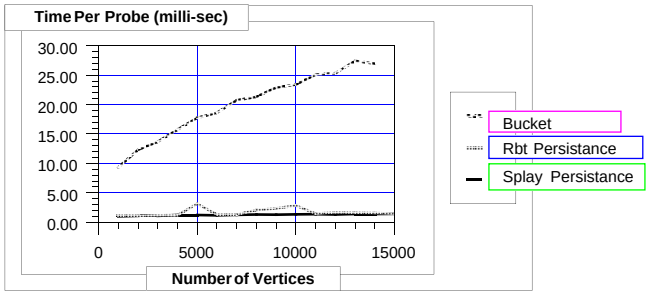

Figure 9: Temporal Coherent Experiments on LEDA Subdivisions. The Working Set Size is 20.

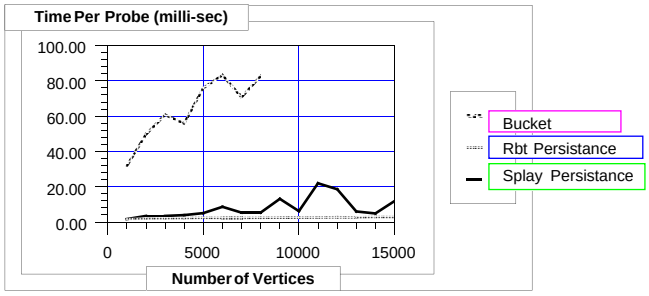

Figure 10: Temporal Coherent Experiments on Diagonal Subdivisions. The Working Set Size is 20.