Transdichotomous Results in Computational Geometry, I: Point Location in Sublogarithmic Time*

Timothy M. Chan† Mihai Pătraşcu‡

September 23, 2008

Abstract

Given a planar subdivision whose coordinates are integers bounded by $U \leq 2^w$, we present a linear-space data structure that can answer point location queries in $O(\min\{\lg n/\lg\lg n, \sqrt{\lg U/\lg\lg U}\})$ time on the unit-cost RAM with word size w. This is the first result to beat the standard $\Theta(\lg n)$ bound for infinite precision models.

As a consequence, we obtain the first $o(n \lg n)$ (randomized) algorithms for many fundamental problems in computational geometry for arbitrary integer input on the word RAM, including: constructing the convex hull of a three-dimensional point set, computing the Voronoi diagram or the Euclidean minimum spanning tree of a planar point set, triangulating a polygon with holes, and finding intersections among a set of line segments. Higher-dimensional extensions and applications are also discussed.

Though computational geometry with bounded precision input has been investigated for a long time, improvements have been limited largely to problems of an orthogonal flavor. Our results surpass this long-standing limitation, answering, for example, a question of Willard (SODA'92).

Key words. Computational geometry, word-RAM algorithms, data structures, sorting, searching, convex hulls, Voronoi diagrams, segment intersection

AMS subject classifications. 68Q25, 68P05, 68U05

Abbreviated title. Point location in sublogarithmic time

1 Introduction

The sorting problem requires $\Omega(n \lg n)$ time for comparison-based algorithms, yet this lower bound can be beaten if the n input elements are integers in a restricted range $[0, U)$. For example, if $U = n^{O(1)}$, radix-sort runs in linear time. Van Emde Boas trees [66, 67] can sort in $O(n \lg\lg U)$ time. Fredman and Willard [35] showed that $o(n \lg n)$ time is possible even regardless of how U relates to n: their *fusion tree* yields a deterministic $O(n \lg n/\lg\lg n)$-time and a randomized $O(n\sqrt{\lg n})$-time sorting algorithm. Many subsequent improvements have been given (see Section 2).

*This work is based on a combination of two conference papers that appeared in *Proc. 47th IEEE Sympos. Found. Comput. Sci.*, 2006: pages 333–342 (by the first author) and pages 325–332 (by the second author).

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). This work has been supported by an NSERC grant.

‡MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA (mip@mit.edu).

In all of the above, the underlying model of computation is a Random Access Machine (RAM) that supports standard operations on w-bit words with unit cost, under the reasonable assumptions that $w \geq \lg n$, i.e., an index or pointer fits in a word, and that $U \leq 2^w$, i.e., each input number fits in a word[1]. The adjective "transdichotomous" is often associated with this model of computation. These assumptions fit the reality of common programming languages such as `C`, as well as standard programming practice (see Section 2.1).

Applications of these bounded-precision techniques have also been considered for geometric problems, but up to now, all known results are limited essentially exclusively to problems about axis-parallel objects or metrics (or those that involve a fixed number of directions). The bulk of computational geometry deals with non-orthogonal things (lines of arbitrary slopes, the Euclidean metric, etc.) and thus has largely remained unaffected by the breakthroughs on sorting and searching.

For example, it is not even known how to improve the $O(n \lg n)$ running time for constructing a Voronoi diagram when the input points come from an integer grid of polynomial size $U = n^{O(1)}$, in sharp contrast to the trivial radix sort in one dimension. Obtaining a $o(n \lg n)$ algorithm for Voronoi diagrams is a problem posed at least since SODA'92 [68].

Our results. We show, for the first time, that the known $\Omega(n \lg n)$ 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 $U \leq 2^w$. We list our results for some of these problems below, all of which are major topics of textbooks—see [12, 32, 51, 52, 55] on the extensive history and on the previously "optimal" algorithms. (See Section 7 for more applications.)

- We obtain $O(n \lg n / \lg \lg n)$-time randomized algorithms for the 3-d *convex hull*, 2-d *Voronoi diagram*, 2-d *Delaunay triangulation*, 2-d *Euclidean minimum spanning tree*, and 2-d *triangulation of a polygon with holes*.

- We obtain an $O(n \lg n / \lg \lg n + k)$-time randomized algorithm for the 2-d *line segment intersection* problem, where k denotes the output size.

- We obtain a data structure for the 2-d *point location* problem with $O(n)$ preprocessing time, $O(n)$ space, and $O(\lg n / \lg \lg n)$ query time. The same space and query bound hold for 2-d *nearest neighbor queries* (also known as the static "post office" problem).

If the universe size U is not too large, we can get even better results: all the $\lg n / \lg \lg n$ factors can be replaced by $\sqrt{\lg U / \lg \lg U}$. For example, we can construct the Voronoi diagram in $O(n\sqrt{\lg n / \lg \lg n})$ expected time for 2-d points from a polynomial-size grid ($U = n^{O(1)}$).

Our algorithms use only standard operations on w-bit words that are commonly supported by most programming languages, namely, comparison, addition, subtraction, multiplication, integer division, bitwise-and, and left and right shifts; a few constants depending only on the value of w are assumed to be available (a standard assumption made explicit since Fredman and Willard's paper [35]).

A new direction. Our results open a whole new playing field, where we attempt to elucidate the fundamental ways in which bounded information about geometric objects such as points and

[1] Following standard practice, we will actually assume throughout the paper that $U = 2^w$, i.e. the machine does not have more precision than it reasonably needs to.

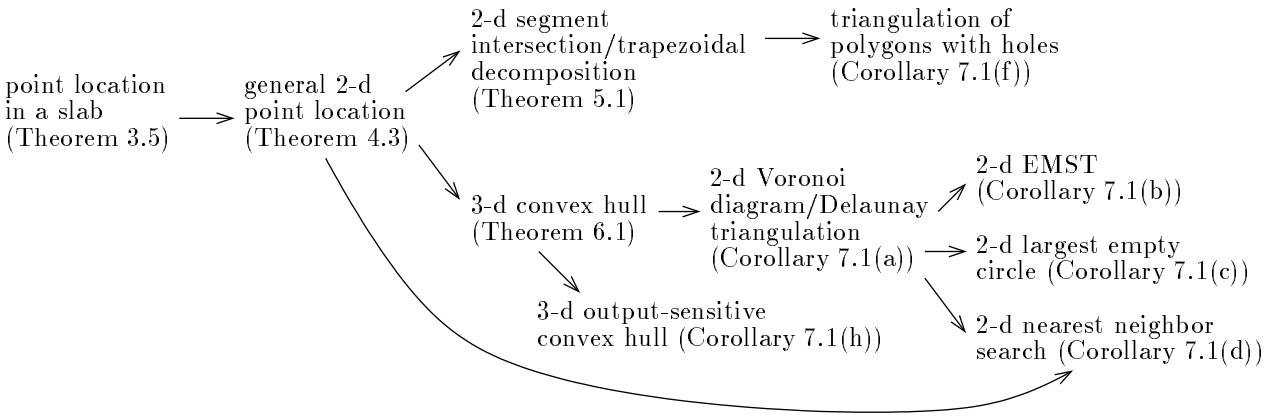

Figure 1: Dependency of results in Sections 3–7.

lines can be decomposed in algorithmically useful ways. In computational geometry, this may lead to a finer distinction of the relative complexities of well-studied geometric problems. In the world of RAM algorithms, this direction requires an expansion of existing techniques on one-dimensional sorting and searching, and may perhaps lead to an equally rich theory for multidimensional problems.

Since the publication of the conference version of this paper, two follow-up works have furthered this direction of research, refining techniques from the present paper. In [22], we show how to solve the *offline* point-location problem faster, which is sufficient to lead to faster algorithms for 2-d Voronoi diagrams, 3-d convex hulls, and other static problems. (The difference between online and offline versions of the problems is analogous to the difference between sorting and predecessor search in 1-d.) In [29], the authors construct a dynamic 2-d convex hull data structure, with $O(\lg n / \lg \lg n)$ query time and polylogarithmic update time.

Organization. The rest of the paper is organized as follows. Section 2 provides more background by briefly reviewing some of relevant previous work. Section 3 represents the heart of the paper and explores point location among disjoint line segments spanning a slab, which turns out to be the key subproblem. To introduce the structure of our search strategy, we first describe a simple alternative to Fredman and Willard's original fusion tree, which achieves sublogarithmic bounds. Even after incorporating nontrivial new ideas for extending this strategy to 2-d, the resulting data structure remains simple and self-contained: its description (Section 3.2) is about two pages long. In Section 4, we show how to reduce the general 2-d point location problem to the slab subproblem; in fact, we give three different ways to accomplish this reduction (each with its advantages and disadvantages). In Sections 5–7, we apply our point-location data structures to derive new results for other well-known geometric problems. (See Figure 1.) Extensions and applications in higher dimensions are also described in Section 8. We conclude with comments in Section 9.

2 Background

2.1 The Model versus Practice

Traditionally, computational geometry has seen the negative side of the contrast between finite and infinite precision. Algorithms are typically designed and analyzed in the real RAM, which makes the theoretical side easier. However, practitioners must eventually deal with finite precision, making theoretical algorithms notoriously difficult to implement.

In the applied literature, there has been considerable attention on robustness issues in geometric computation; for instance, much work is focused on examining the predicates used by comparison-based algorithms that were originally designed for the real RAM, and implementing these operations in a safe and effective way under finite precision. Here, we take a different perspective. By assuming that actual input data have bounded precision in the first place, we show that one can actually design asymptotically better algorithms.

A common theoretical fallacy is that it is irrelevant to study algorithms in a bounded universe because "only comparison-based algorithms are ever implemented". However, this thesis has been attacked forcefully in one dimension; see, e.g., [39]. It is well known, for instance, that the fastest solutions to sorting are based on bounded precision (radix sort). Furthermore, when search speed matters, such as for forwarding packets in Internet routers, search is certainly not implemented by comparisons [28].

Even for the geometric problems we are studying, there are examples showing the benefit of using bounded precision. A survey by Snoeyink [62] indicates that the most efficient and popular approaches for planar point location use pruning heuristics on the grid. These heuristics are similar in spirit to the algorithms we develop in this paper, so, to some extent, our work justifies engineering practice.

As another example, we note that there is considerable interest in approximate nearest neighbor search, even in two dimensions. This is hard to understand when viewed through the real-RAM abstraction, because both the approximate and exact nearest neighbor problem have the same logarithmic complexity. However, for approximate nearest neighbor one can give better (and simpler) solutions based on bounded precision [3, 20].

A question that we wish to touch on briefly is whether an integer universe is the right model for bounded precision. In certain cases, the input is on an integer grid by definition (e.g., objects are on a computer screen). One might worry, however, about the input being a floating point number. We believe that in most cases this is an artifact of representation, and numbers should be treated as integers after appropriate scaling. One reason is to note that the "floating-point plane" is simply a union of bounded integer grids (the size depending on the number of bits of the mantissa), at different scale factors around the origin. Since the kind of problems we are considering are translation-independent, there is no reason the origin should be special, and having more detail around the origin is not particularly meaningful. Another reason is that certain aspects of the problems are not well-defined when inputs are floating point numbers. For example, the slope of a line between two points of very different exponents is not representable by floating point numbers anywhere close to the original precision.

2.2 RAM Algorithms in 1-d

In addition to the work already mentioned, many integer-sorting results have been published (e.g., [5, 47, 57, 63, 65]). Currently, the best linear-space deterministic and randomized algorithms (independent of U and w) have running time $O(n \lg\lg n)$ and $O(n\sqrt{\lg\lg n})$ respectively, due to Han [38] and Han and Thorup [39]. A linear randomized time bound [5] is known for the case when $w \geq \lg^{2+\varepsilon} n$, for any fixed $\varepsilon > 0$. Thorup [64] showed a black-box transformation from sorting to *priority queues*, which makes the above bounds carry over to this dynamic problem.

For the problem of maintaining a data structure for *successor search* (finding the smallest element greater than a query value), van Emde Boas trees [66, 67] yield $O(\lg\lg U) = O(\lg w)$ query time with linear space, and Fredman and Willard's fusion trees yield an $O(\log_w n)$ query time with linear space. (This is certainly $O(\lg n/\lg\lg n)$, and by balancing with the van Emde Boas bound, $O(\sqrt{\lg n})$.) For polynomial space, some improvements are possible [11]. Pătraşcu and Thorup [54] show optimal upper and lower bounds for this problem, giving an exact understanding of the time-space tradeoffs.

Most importantly, their lower bounds show that for near linear space (say, space $n \lg^{O(1)} n$), the optimal query time is $\Theta(\min\{\log_w n,\ \lg w/\lg\frac{\lg w}{\lg\lg n}\})$. The first branch is achieved by fusion trees, while the second branch is a slight improvement to van Emde Boas, which is only relevant for rather large precision. We note that point location is harder than successor search, so the lower bounds apply to our problems as well.

Other 1-d data structure problems for integer input have also been studied. The classic problem of designing a dictionary to answer *membership queries*, typically addressed by hashing, can be solved in $O(1)$ deterministic query time with linear space, while updates are randomized and take $O(1)$ time with high probability (see e.g., [30, 34]). *Range queries* in 1-d (reporting any element inside a query interval) can be solved with $O(1)$ query time by a linear-space data structure [2]. Even for the dynamic problem, exponential improvements over successor search are known [50].

2.3 (Almost) Orthogonal Problems

As mentioned, known algorithms from the computational geometry literature that exploit the power of the word RAM mostly deal with orthogonal-type special cases, such as orthogonal range searching, finding intersections among axis-parallel line segments, and nearest neighbor search under the L_1- or L_∞-metric. Most of these work (see [13, 43, 44, 45, 53] for a sample) are about van-Emde-Boas-type results, with only a few exceptions (e.g., [49, 68]). For instance, Karlsson [44] obtained an $O(n \lg\lg U)$-time algorithm for the L_∞-Voronoi diagram in 2-d. Chew and Fortune [25] later showed how to construct the Voronoi diagram under any fixed convex polygonal metric in 2-d in $O(n \lg\lg n)$ time after sorting the points along a fixed number of directions. De Berg *et al.* [13] gave $O((\lg\lg U)^{O(1)})$ results for point location in an axis-parallel rectangular subdivisions in 2- and 3-d. They also noted that certain subdivisions built from *fat* objects can be "rectangularized", though this is not true for general objects.

There are also *approximation* results (not surprisingly, since arbitrary directions can be approximated by a fixed number of directions); for example, see [14] for an $O(n \lg\lg n)$-time 2-d approximate Euclidean minimum spanning tree algorithm.

There is one notable non-orthogonal problem where faster exact transdichotomous algorithms are known: finding the *closest pair* of n points in a constant-dimensional Euclidean space. (This is also not too surprising, if one realizes that the complexity of the exact closest pair problem is linked to that of the approximate closest pair problem, due to packing arguments.) Rabin's classic

paper on randomized algorithms [56] solved the problem in $O(n)$ expected time, using hashing. Deterministically, Chan [20] has given a reduction from closest pair to sorting (using one nonstandard but AC^0 operation on the RAM). This implies an $O(n \lg\lg n)$ deterministic time bound by Han's result [38], and for the special case of points from a polynomial-size grid, an $O(n)$ deterministic bound by radix-sorting (with standard operations only). Similarly, the dynamic closest pair problem and (static or dynamic) approximate nearest neighbor queries reduce to successor search [20] (see also [3, 16] for previous work). Rabin's original approach itself has been generalized to obtain an $O(n+k)$-time randomized algorithm for finding k closest pairs [19], and an $O(nk)$-time randomized algorithm for finding the smallest circle enclosing k points in 2-d [40].

The 2-d convex hull problem is another exception, due to its simplicity: Graham's scan [12, 55] takes linear time after sorting the x-coordinates. In particular, computing the diameter and width of a 2-d point set can be reduced to 1-d sorting. (In contrast, sorting along a fixed number of directions does not help in the computation of the 3-d convex hull [60].)

Chazelle [24] studied the problem of deciding whether a query point lies inside a convex polygon with w-bit integer or rational coordinates. This problem can be easily reduced to 1-d successor search, so the study was really about lower bounds. (Un)fortunately, he did not address upper bounds for more challenging variants like intersecting a convex polygon with a query line (see Corollary 7.1(g)).

For the asymptotically tightest possible grid, i.e., $U = O(n^{1/d})$, the *discrete Voronoi diagram* [15, 21] can be constructed in linear time and can be used to solve static nearest neighbor problems.

2.4 Nonorthogonal Problems

The quest for faster word-RAM algorithms for the core geometric problems dates back at least to 1992, when Willard [68] asked for a $o(n \lg n)$ algorithm for Voronoi diagrams. Interest in this question has only grown stronger in recent years. For example, Jonathan Shewchuk (2005) in a blog comment wondered about the possibility of computing Delaunay triangulations in $O(n)$ time. Demaine and Iacono (2003) in lecture notes, as well as Baran *et al.* [10], asked for a $o(\lg n)$ method for 2-d point location.

Explicit attempts at the point location problem have been made by the works of Amir *et al.* [3] or Iacono and Langerman [43]. These papers achieve an $O(\lg\lg U)$ query time, but unfortunately their space complexity is only bounded by measures such as the quad-tree complexity or the fatness. This leads to prohibitive exponential space bounds for difficult input instances.

There has also been much interest in obtaining adaptive sublogarithmic bounds in the decision-tree model. The setup assumes queries are chosen from a biased distribution of entropy H, and one tries to relate the query time to H. Following some initial work on the subject, SODA'01 saw no less than three results in this direction: Arya *et al.* [8] and Iacono [42] independently achieved expected $O(H)$ comparisons with $O(n)$ space, while Arya *et al.* [9] achieved $H + o(H)$ comparisons. We note that a similar direction of research has also been pursued intensively for searching in 1-d (e.g., static and dynamic optimality), but has lost in popularity, with integer search rising to prominence.

3 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 $s_1, \ldots, s_b$ could be done in constant time. In our 2-d problem, this seems possible, at least for certain selected segments $s_1, \ldots, s_b$, because of the following "input rounding" idea: locating q among $s_1, \ldots, s_b$ reduces to locating q among any set of segments $\tilde{s}_1, \ldots, \tilde{s}_b$ that satisfy $s_1 \prec \tilde{s}_1 \prec s_2 \prec \tilde{s}_2 \prec \cdots$, where $\prec$ denotes the (strict) belowness relation (see Figure 2(a)). Because the coordinates of the $\tilde{s}_i$'s are flexible, we might be able to find some set of segments $\tilde{s}_1, \ldots, \tilde{s}_b$, 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 q can be rounded as well.)

Unfortunately, previous 1-d data structures do not seem compatible with this idea. Van Emde Boas trees [66, 67] and Andersson's exponential search trees [4] require hashing of the rounded input numbers and query point—it is unclear what it means to hash line segments in our context. Fredman and Willard's original fusion tree [35] relies on "compression" of the input numbers and query point (i.e., extraction of some carefully chosen bits)—the compressed keys bear no geometric relationship with the original.

We end up basing our data structure on a version of the fusion tree that appears new, to the best of the authors' knowledge, and avoids the complication of compressed keys. This is described in Section 3.1 (which is perhaps of independent interest but may be skipped by the impatient reader). The actual data structure for point location in a slab is presented in Section 3.2, with further variants described in Section 3.3.

3.1 Warm-Up: A Simpler 1-d Fusion Tree

We first re-solve the standard 1-d problem of performing successor search in a static set of n numbers in sublogarithmic time, where the numbers are assumed to be integers in $[0, 2^w)$. Although faster solutions are known, our solution is very simple. Our main idea 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, which we denote by ℓ. Initially, $\ell = w$. (Beame and Fich [11] adopted a similar philosophy in the design of their data structure, though, in a much more intricate way, as they aimed for better query time.)

Observation 3.1 *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 $2^{\ell-h}$; and*

(2) *the subinterval lengths are all multiples of $2^{\ell-h}$.*

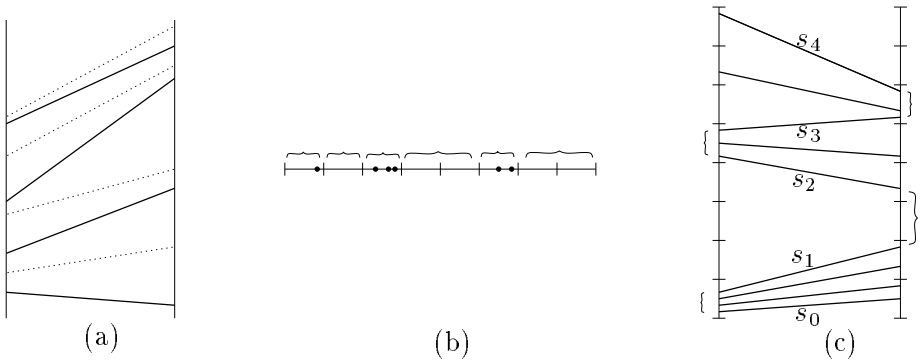

Figure 2: (a) The rounding idea: locating among the solid segments reduces to locating among the dotted segments. (b) Proof of Observation 3.1: elements of B are shown as dots. (c) Proof of Observation 3.2: segments of B are shown, together with the constructed sequence $s_0, s_1, \ldots$

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

The data structure. The observation suggests a simple tree structure for 1-d successor search. Because of (2) (by dividing by $2^{\ell-h}$), 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 (or "fused") 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 $\ell < 0$). 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 (see the next paragraph for more details). We then recursively search inside this subinterval. (If the sucessor is not there, it must be the first element to the right of the subinterval; this element can be stored during preprocessing.) By choosing $b = \lfloor \sqrt{\lg n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\lg n / \lg \lg n)$.

Implementing the word operation. We have assumed above that the subinterval containing q can be found in constant time, given $O(b)$ subintervals satisfying (2), all packed in one word. We now show that this nonstandard operation can be implemented using more familiar operations like multiplications, shifts, and bitwise-ands (&'s).

First, because of (2), we may assume that the endpoints of the subintervals are integers in $[0, 2^h)$. We can thus round q to an integer $\tilde{q}$ in $[0, 2^h)$, without changing the answer. The operation then reduces to computing the rank of an h-bit number $\tilde{q}$ among an increasing sequence of $O(b)$ h-bit numbers $\tilde{a}_1, \tilde{a}_2, \ldots$, with $bh \leq \varepsilon w$.

This subproblem was considered before [35, 6], and we quickly review one solution. Let $\langle z_1 \,|\, z_2 \,|\, \cdots \rangle$ denote the word formed by $O(b)$ blocks each of exactly $h+1$ bits, where the i-th block holds the value z_i. We precompute the word $\langle \tilde{a}_1 \,|\, \tilde{a}_2 \,|\, \cdots \rangle$ during preprocessing by repeated shifts and

additions. Given $\tilde{q}$, we first multiply it with the constant $\langle 1 \mid 1 \mid \cdots \rangle$ to get the word $\langle \tilde{q} \mid \tilde{q} \mid \cdots \rangle$. Now, $\tilde{a}_i < \tilde{q}$ iff $(2^h + \tilde{a}_i - \tilde{q}) \,\&\, 2^h$ is zero. With one addition, one subtraction, and one & operation, we can obtain the word $\langle (2^h + \tilde{a}_1 - \tilde{q}) \,\&\, 2^h \mid (2^h + \tilde{a}_2 - \tilde{q}) \,\&\, 2^h \mid \cdots \rangle$. The rank of $\tilde{q}$ can then be determined by finding the most significant 1-bit (msb) position of this word. This msb operation is supported in most programming languages (for example, by converting into floating point and extracting the exponent, or by taking the floor of the binary logarithm); alternatively, it can be reduced to standard operations as shown by Fredman and Willard [35].

3.2 A Solution for 2-d

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).

Observation 3.2 *Fix b and h. Let S be a set of n sorted disjoint line segments, where all left endpoints lie on an interval I_L of length 2^{ℓ_L} on a vertical line, and all right endpoints lie on an interval I_R of length 2^{ℓ_R} on another vertical line. In $O(b)$ time, we can find $O(b)$ segments $s_0, s_1, \ldots \in S$ in sorted order, which include the lowest and highest segments of S, such that:*

(1) *for each i, at least one of the following holds:*

  (1a) *there are at most n/b line segments of S between s_i and s_{i+1}.*

  (1b) *the left endpoints of s_i and s_{i+1} lie on a subinterval of length $2^{\ell_L - h}$.*

  (1c) *the right endpoints of s_i and s_{i+1} lie on a subinterval of length $2^{\ell_R - h}$.*

(2) *there exist $O(b)$ line segments $\tilde{s}_0, \tilde{s}_2, \ldots$ cutting across the slab, satisfying all of the following:*

  (2a) *$s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec \cdots$.*

  (2b) *distances between the left endpoints of the $\tilde{s}_i$'s are all multiples of $2^{\ell_L - h}$.*

  (2c) *distances between right endpoints are all multiples of $2^{\ell_R - h}$.*

Proof: Let B contain every $\lfloor n/b \rfloor$-th segment of S, starting with the lowest segments s_0. Impose a grid over I_L consisting of 2^h subintervals of length $2^{\ell_L - h}$, and a grid over I_R consisting of 2^h subintervals of length $2^{\ell_R - h}$. We define s_{i+1} inductively based on s_i, until the highest segment is reached. We let s_{i+1} be the 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. This will satisfy (1b) or (1c). If no such segment above s_i exists, we simply let s_{i+1} be the successor of s_i in B, satisfying (1a). (See Figure 2(c) for an example.)

Let $\tilde{s}_i$ be obtained from s_i by rounding each endpoint to the grid point immediately above (ensuring (2b) and (2c)). By construction of the s_i's, both the left and right endpoints of s_i and s_{i+2} are in different grid subintervals. Thus, $\tilde{s}_i \prec s_{i+2}$, ensuring (2a). □

The data structure. With Observation 3.2 to replace Observation 3.1, we can now proceed as in the previous section. Because of (2b) and (2c), 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 $\tilde{s}_0, \tilde{s}_2, \ldots$ 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 $\ell_L < 0$ or $\ell_R < 0$). Initially, $\ell_L = \ell_R = w$. Because of (1), in each subproblem, n is decreased by a factor of b, *or* ℓ_L is decreased by h, *or* ℓ_R is decreased by h. Thus, the height of the tree is at most $\log_b n + 2w/h = O(\log_b n + b)$.

Given a query point q, we first locate q among the $\tilde{s}_i$'s by a word operation. With one extra comparison we can then locate q among $s_0, s_2, s_4 \ldots$, 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 $b = \lfloor \sqrt{\lg n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\lg n / \lg \lg n)$.

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., $O(n \lg n / \lg \lg n)$ (which can easily be improved to $O(n)$ as we will observe in the next subsection). Sorting naively takes $O(n \lg n)$ time, which can be improved by known results.

Implementing the word operation. We have assumed above that we can locate q among the $\tilde{s}_i$'s in constant time, given $O(b)$ segments $\tilde{s}_0, \tilde{s}_2 \ldots$, 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 $(2^h, \tilde{b}_i)$, where the $\tilde{a}_i$'s and $\tilde{b}_i$'s are increasing sequences of integers in $[0, 2^h)$. Specifically, the mapping below transforms two intervals $I = \{A\} \times [B, B + 2^\ell)$ and $J = \{C\} \times [D, D + 2^m)$ to $\{0\} \times [0, 2^h)$ and $\{2^h\} \times [0, 2^h)$ respectively:

$$(x, y) \;\mapsto\; \left(\frac{2^{h+m}(x - A)}{2^\ell(C - x) + 2^m(x - A)},\; \frac{2^h[(C - A)(y - B) - (D - B)(x - A)]}{2^\ell(C - x) + 2^m(x - A)} \right).$$

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 $\tilde{x}$ and $\tilde{y}$, we proceed as follows. Let $\langle z_1 \,|\, z_2 \,|\, \cdots \rangle$ 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 $\langle \tilde{a}_0 \,|\, \tilde{a}_2 \,|\, \cdots \rangle$ and $\langle \tilde{b}_0 \,|\, \tilde{b}_2 \,|\, \cdots \rangle$ 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 $\langle \tilde{a}_0(2^h - \tilde{x}) + \tilde{b}_0\tilde{x} \;|\; \tilde{a}_2(2^h - \tilde{x}) + \tilde{b}_2\tilde{x} \;|\; \cdots \rangle$. We want to compute the rank of $2^h\tilde{y}$ among the values encoded in the blocks of this word. As we have reviewed in Section 3.1, this subproblem can be solved using a constant number of standard operations [35].

Remarks. The above data structures can be extended to deal with $O(w)$-*bit rational* coordinates, i.e., coordinates that are ratios of integers in the range $[-2^{cw}, 2^{cw}]$ for some constant c. (This extension will be important in subsequent applications.) The main reason is that the coordinates have bounded "spread": namely, the difference of any two such distinct rationals must be at least

$1/2^{2cw}$. Thus, when ℓ or m reaches below $-2cw$, we have $n \leq 1$. The point-segment comparisons and projective transformations can still be done in constant time, since $O(w)$-bit arithmetic can be simulated by $O(1)$ w-bit arithmetic operations.

The data structures can also be adapted for disjoint open segments that may share endpoints: We just consider an additional base case, when all segments pass through one endpoint p, say, on I_L. To locate a query point q among these segments, we can compute the intersection of $\overrightarrow{pq}$ with I_R (which has rational coordinates) and perform a 1-d search on I_R.

Proposition 3.3 *Given a sorted list of n disjoint line segments spanning a vertical slab in the plane where the endpoints have $O(w)$-bit rational coordinates, we can build a data structure with space $O(n)$ in time $o(n \lg n)$ so that point location queries can be answered in time $O(\lg n / \lg \lg n)$.*

3.3 Alternative Bounds

We now describe some alternative bounds which depend on the universe size and the space.

Proposition 3.4 *Consider a sorted list of n disjoint line segments spanning a vertical slab in the plane where the endpoints have $O(w)$-bit rational coordinates. For any $h \geq 1$, we can build a data structure with space $O(n \cdot 4^h)$ in time $O(n \cdot (w/h + 4^h))$ so that point location queries can be answered in time $O(w/h)$.*

Proof: This is a simple variant of our previous data structure, relying on table lookup instead of word packing. We apply Observation 3.2 recursively, this time with $b = 2^h$. The height of the resulting tree is now at most $O(w/h + \log_b n) = O((w + \lg n)/h) = O(w/h)$.

Because the segments $\tilde{s}_0, \tilde{s}_2, \ldots$ can no longer be packed in a word, we need to describe how to locate a query point q among the $\tilde{s}_i$'s in constant time. By the projective transformation and rounding as described in Section 3.2, it suffices to locate a point $\tilde{q}$ whose x- and y-coordinates are h-bit integers. Thus, we can precompute the answers for all 2^{2h} such points during preprocessing. This takes time $O(2^{2h})$ time: trace each segment horizontally in $O(b \cdot 2^h)$ time, and fill in the rest of the table by 2^h scans along each vertical grid line.

The total extra cost for the table precomputation is $O(n \cdot 4^h)$. We immediately obtain preprocessing time $O(n \cdot (w/h + 4^h))$ starting with sorted segments, space $O(n \cdot 4^h)$ and query time $O(w/h)$, for any given parameter h. □

Now we can obtain a linear-space data structure whose running time depends on w, by a standard space reduction as follows:

Let R contain the $\lfloor in/r \rfloor$-lowest segment for $i = 1, \ldots, r$, and apply the data structure of Proposition 3.4 only for these segments of R. To locate a query point q among S, we first locate q among R and then finish by binary search in a subset of $O(n/r)$ elements between two consecutive segments in R.

The preprocessing time starting with sorted segments is $O(n + r \cdot (w/h + 4^h))$, the space requirement $O(n + r \cdot 4^h)$, and the query time is $O(w/h + \lg(n/r))$. Setting $r = \left\lfloor n/(w/h + 4^h) \right\rfloor$ leads to $O(n)$ preprocessing time and space and $O(w/h + h)$ query time. Setting $h = \lfloor \sqrt{w} \rfloor$ yields $O(\sqrt{w})$ query time.

We can reduce the query time further by replacing the binary search with a point location query using Proposition 3.3 to store each subset of $O(n/r)$ elements. The query time becomes

$O(w/h + \lg(n/r)/\lg\lg(n/r)) = O(w/h + h/\lg h)$. Setting $h = \lfloor\sqrt{w \lg w}\rfloor$ instead yields a query time of $O(\sqrt{w/\lg w})$.

Incidentally, the preprocessing time in Proposition 3.3 can be improved to $O(n)$ using the same trick, for example, by choosing $r = \lfloor n/\log n \rfloor$. The preprocessing time in Proposition 3.4 can be reduced to $O(n \cdot 4^h)$ as well, by choosing $r = \lfloor n/(w/h) \rfloor$.

Our results for the slab problem are summarized by the following:

Theorem 3.5 *Consider a sorted list of n disjoint (open) line segments spanning a vertical slab in the plane where the endpoints have $O(w)$-bit rational coordinates. For any $h \geq 1$, we can build a data structure with space and preprocessing time $O(n \cdot 4^h)$, so that point location queries take time: $O\left(\min\left\{\lg n/\lg\lg n,\ \sqrt{w/\lg w},\ w/h\right\}\right)$.*

4 General 2-d Point Location

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 q. By associating each segment with an incident face, it suffices to find the segment that is immediately above q.

Assuming a solution for the slab problem with $O(n)$ space and preprocessing time and $t(n)$ query time, we can immediately obtain a data structure with $O(n^2)$ space and preprocessing time, which supports queries in $O(t(n))$ 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 rationals 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. (Since point location among horizontal segments solves successor search, we know successor search takes at most $t(n)$ time.)

We can improve the preprocessing time and space by applying known computational-geometric techniques for point location; for example, we could attempt a b-ary version of the *segment tree* or the *trapezoid method* [12, 55, 62], though the resulting structure would not have linear space. We describe three different linear-space, $O(t(n))$-time solutions by adapting the following techniques:

planar separators [48] This method has the best theoretical properties, including determinstic bounds and linear-time construction. However, it is probably the least practical because of large hidden constants.

random sampling [51] This method is the simplest, but the construction algorithm is randomized, and takes time $O(n \cdot t(n))$.

persistent search trees [59] This is the least obvious to adapt and requires some interesting use of ideas from *exponential search trees*, but results in a deterministic construction time of $O(\text{sort}(n))$, where $\text{sort}(n)$ denotes the time to sort n numbers on the word RAM. This result shows how our sublogarithmic results can be used in sweep-line algorithms, which is important for some applications (e.g., see Corollary 7.1(f)).

Our results can be stated as black-box reductions that make minimal assumptions about the solution to the slab problem. In general, the query time increases by an $O(\lg\lg n)$ factor. However,

for many natural cases for $t(n)$, we get just a constant-factor slow-down. By the slab result of the previous section, we obtain $O(t(n)) = O(\lg n / \lg\lg n)$ query time. With some additional effort, we can recover the alternative $O(\sqrt{w/\lg w})$ query time bound as well, using any of the three reductions. In the first reduction (planar separators), we discuss all these implications formally. For the other two, which are not achieving asymptotically better bounds, we omit the w-sensitive results and concentrate only on the most natural case for $t(n)$.

4.1 Method 1: Planar Separators

We describe our first method for reducing general 2-d point location to point location in a slab. We assume that the given subdivision is a triangulation. If the input is an arbitrary connected subdivision, we can first triangulate it in linear deterministic time by Chazelle's algorithm [23] (in principle).

Our deterministic method is based on the planar graph separator theorem by Lipton and Tarjan [48] (who also noted its possible application to the point location problem). We use the following version, which can be obtained by applying the original theorem recursively (to get the linear running time, see [1, 36]):

Lemma 4.1 *Given a planar graph G with n vertices and a parameter r, we can find a subset R of $O(\sqrt{rn})$ vertices in $O(n)$ time, such that each connected component of $G \setminus R$ has at most n/r vertices.*

Deterministic divide-and-conquer. Let n denote the number of triangles in the given triangulation T. We apply the separator theorem to the dual of T to get a subset R of $O(\sqrt{rn})$ triangles, such that the removal of these triangles yields subregions each comprising at most n/r triangles. We store the subdivision induced by R (the number of edges is $O(|R|)$), using a point-location data structure with $O(P_0(\sqrt{rn}))$ preprocessing time, and $O(Q_0(\sqrt{rn}))$ query time. For each subregion with n_i triangles, we build a point-location data structure with $P_1(n_i)$ preprocessing time and $Q_1(n_i)$ query time.

As a result, we get a new method with the following bounds for the preprocessing time $P(n)$ and query time $Q(n)$ for some n_i's with $\sum_i n_i \le n$ and $n_i \le n/r$:

$$\begin{aligned} P(n) &= \sum_i P_1(n_i) + O(n + P_0(\sqrt{rn})) \\ Q(n) &= \max_i Q_1(n_i) + O(Q_0(\sqrt{rn})). \end{aligned}$$

Calculations. To get started, we use the naive method with $P_0(n) = P_1(n) = O(n^2)$ and $Q_0(n) = Q_1(n) = O(t(n))$. Setting $r = \lfloor\sqrt{n}\rfloor$ then yields $P(n) = O(n^{3/2})$ and $Q(n) = O(t(n))$.

To reduce preprocessing time further, we bootstrap using the new bound $P_0(n) = O(n^{3/2})$ and $Q_0(n) = O(t(n))$ and apply recursion to handle each subregion. By setting $r = \left\lfloor n^{1/4} \right\rfloor$, the recurrences

$$\begin{aligned} P(n) &= \sum_i P(n_i) + O(n + P_0(\sqrt{rn})) = \sum_i P(n_i) + O(n) \\ Q(n) &= \max_i Q(n_i) + O(Q_0(\sqrt{rn})) = \max_i Q(n_i) + O(t(n)) \end{aligned}$$

have depth $O(\lg\lg n)$. Thus, $P(n) = O(n \lg\lg n)$. If $t(n)/\lg^\delta n$ is monotone increasing for some constant $\delta > 0$, the query time is $Q(n) = O(t(n))$, because the assumption implies that $t(n) \ge$

$(4/3)^\delta t(n^{3/4})$ and so $Q(\cdot)$ expands to a geometric series. (If the assumption fails, the upper bound $Q(n) = O(t(n) \log\log n)$ still holds.)

Lastly, we bootstap one more time, using $P_0(n) = O(n \lg\lg n)$ and $Q_0(n) = O(t(n))$, and by Kirkpatrick's point location method [46], $P_1(n) = O(n)$ and $Q_1(n) = O(\lg n)$. We obtain the following bounds, where $\sum n_i \le n$ and $n_i \le n/r$:

$$\begin{aligned} P(n) &= \sum_i P_1(n_i) + O(n + P_0(\sqrt{rn})) = O(n + \sqrt{rn}\lg\lg n) \\ Q(n) &= \max_i Q_1(n_i) + O(Q_0(\sqrt{rn})) = O(\lg(n/r) + t(n)). \end{aligned}$$

Setting $r = \lfloor n/\lg n \rfloor$ then yields the final bounds of $P(n) = O(n)$ and $Q(n) = O(t(n))$ (as $t(n)$ exceeds $\lg\lg n$ under the above assumption). The space used is bounded by the preprocessing cost and is thus linear as well.

(Note: it is possible to avoid the last bootstrapping step by observing that the total cost of the recursive separator computations is linear [36]. The first bootstrapping step could also be replaced by a more naive method that divides the plane into $\sqrt{n}$ slabs.)

Proposition 4.2 *Suppose there is a data structure with $O(n)$ preprocessing time and space that can answer point location queries in $t(n)$ time for n disjoint line segments spanning a vertical slab in the plane where the endpoints have $O(w)$-bit rational coordinates.*

Then given any planar connected subdivision defined by n disjoint line segments whose endpoints have $O(w)$-bit rational coordinates, we can build a data structure in $O(n)$ time and space so that point location queries can be answered in $O(t(n))$ time, assuming that $t(n)/\lg^\delta n$ is monotone increasing for some constant $\delta > 0$. (If the assumption fails, the query time is still bounded by $O(t(n)\lg\lg n)$.)

Alternative bounds. By Theorem 3.5, we can set $t(n) = O(\lg n/\lg\lg n)$ in Proposition 4.2 and get $O(\lg n/\lg\lg n)$ query time. To get the alternative $O(\sqrt{w/\lg w})$ query time bound, we need to modify the above calculations, in order to avoid increasing the query time by a $\lg\lg n$ factor. Using the h-sensitive bounds from Theorem 3.5, we start with $P_0(n) = P_1(n) = O(n^2 \cdot 4^h)$ and $Q_0(n) = Q_1(n) = O(w/h)$. The first bootstrapping step with $r = \lfloor\sqrt{n}\rfloor$ yields $P(n) = O(n^{3/2} \cdot 4^h)$ and $Q(n) = O(w/h)$.

In the next step, we use $P_0(n) = O(n^{3/2} \cdot 4^h)$ and $Q_0(n) = O(w/h)$ and apply recursion to handle each subregion. We set $r = \left\lfloor n^{1/4} \right\rfloor$ and $h = \lfloor \varepsilon \lg n \rfloor$ for a sufficiently small constant $\varepsilon > 0$ (so that $(\sqrt{rn})^{3/2} \cdot 4^h = o(n)$). The recurrences become

$$\begin{aligned} P(n) &= \sum_i P(n_i) + O(n) \\ Q(n) &= \max_i Q(n_i) + O(w/\lg n), \end{aligned}$$

where $\sum_i n_i \le n$ and $n_i = O(n^{3/4})$. We stop the recursion when $n \le n_0$ and handle the base case using Proposition 4.2 (and Theorem 3.5) with $O(n_0)$ preprocessing time and $O(t(n_0)) = O(\lg n_0/\lg\lg n_0)$ query time. As a result, the recurrences solve to $P(n) = O(n\lg\lg n)$ and $Q(n) = O(w/\lg n_0 + \lg n_0/\lg\lg n_0)$, because $Q(\cdot)$ expands to a geometric series. Setting $n_0 = 2^{\lfloor\sqrt{w\lg w}\rfloor}$ yields $Q(n) = O(\sqrt{w/\lg w})$.

In the last bootstrapping step, we use $P_0(n) = O(n\lg\lg n)$ and $Q_0(n) = O(\sqrt{w/\lg w})$, and $P_1(n) = O(n)$ and $Q_1(n) = O(\lg n)$. Setting $r = \lfloor n/\lg n \rfloor$ yields $O(n)$ preprocessing time and $O(\sqrt{w/\lg w})$ query time.

Our results for planar point location are summarized by the following:

Theorem 4.3 *Consider a planar connected subdivision defined by n disjoint line segments whose endpoints have $O(w)$-bit rational coordinates. We can build a data structure with space and preprocessing time $O(n)$, so that point location queries take time:*

$$t(n,w) \;:=\; O\left(\min\left\{\lg n/\lg\lg n,\ \sqrt{w/\lg w}\right\}\right).$$

4.2 Method 2: Random Sampling

Again, we assume a solution for the slab problem using $O(n)$ space and construction time, and supporting queries in $t(n)$ time, where $t(n)/\lg^{\delta} n$ is monotone increasing for some constant $\delta > 0$. We now describe a different data structure for general point location, using $O(n)$ space, which can be constructed in expected $O(n \cdot t(n))$ time, and supports queries in $O(t(n))$ query time. Although this method is randomized and has a slower preprocessing time, it is simpler and the idea itself has further applications, as we will see later in Sections 5–6. The method is based on random sampling. (The idea of using sampling-based divide-and-conquer, or cuttings, to reduce space in point-location data structures has appeared before; e.g., see [9, 37, 61].)

Randomized divide-and-conquer. 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 has $O(r)$ trapezoids and is known to be constructible in $O(r \lg r)$ time. We store $T(R)$ in a point-location data structure, with $P_0(r)$ preprocessing time, $S_0(r)$ space, and $Q_O(r)$ 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 so-called *conflict list* of Δ). The time required is $O(nQ_0(r) + \sum_{\Delta \in T(R)} |S_\Delta|)$.

By a standard analysis of Clarkson and Shor [27, 51], the probability that

$$\sum_{\Delta \in T(R)} |S_\Delta| \;=\; O(n) \qquad \text{and} \qquad \max_{\Delta \in T(R)} |S_\Delta| \;=\; O((n/r)\lg r)$$

is 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 then recursively build a point-location data structure inside Δ for each subset S_Δ.

To locate a query point q, we first find the trapezoid $\Delta \in T(R)$ containing q in $Q_0(r)$ time and then recursively search inside 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 $n_i = O((n/r)\lg r)$:

$$\begin{aligned} P(n) &= \sum_i P(n_i) + O(P_0(r) + nQ_0(r)) \\ S(n) &= \sum_i S(n_i) + O(S_0(r)) \\ Q(n) &= \max_i Q(n_i) + O(Q_0(r)). \end{aligned}$$

Calculations. To get started, we use the naive method with $P_0(r) = S_0(r) = O(r^2)$ and $Q_0(r) = O(t(r))$. By setting $r = \lfloor \sqrt{n} \rfloor$, the above recurrence has depth $O(\lg \lg n)$ and solves to $P(n), S(n) = O(n \cdot 2^{O(\lg \lg n)}) = O(n \lg^{O(1)} n)$ and $Q(n) = O(t(n))$, because $Q(\cdot)$ expands to a geometric series under our assumption.

To reduce space further, we bootstrap using the new bounds $P_0(r), S_0(r) = O(r \lg^c r)$ and $Q_0(r) = O(t(r))$ for some constant c. This time, we replace recursion by directly invoking some known planar point location method [62] with $P_1(n) = O(n \lg n)$ preprocessing time, $S_1(n) = O(n)$ space, and $Q_1(n) = O(\lg n)$ query time. We then obtain the following bounds, where $\sum_i n_i = O(n)$ and $n_i = O((n/r) \lg r)$:

$$\begin{aligned} P(n) &= \sum_i P_1(n_i) + O(P_0(r) + nQ_0(r)) = O(n \lg(n/r) + r \lg^c r + n \cdot t(r)) \\ S(n) &= \sum_i S_1(n_i) + O(S_0(r)) = O(n + r \lg^c r) \\ Q(n) &= \max_i Q_1(n_i) + O(Q_0(r)) = O(\lg(n/r) + t(r)). \end{aligned}$$

Remember that $t(n)$ exceeds $\lg \lg n$ under our assumption. Setting $r = \lfloor n / \lg^c n \rfloor$ yields $O(n \cdot t(n))$ expected preprocessing time, $O(n)$ space, and $O(t(n))$ query time.

4.3 Method 3: Persistence and Exponential Search Trees

We now show how to use the classic approach of persistence: perform a sweep with a vertical line, inserting and deleting segments into a *dynamic* structure for the slab problem. The structure is the same as in the naive solution with quadratic space: what used to be separate slab structures are now snapshots of the dynamic structure at different moments in time. The space can be reduced if the dynamic structure can be made persistent with a small amortized cost in space.

Segment successor. We define the *segment-successor problem* as a dynamic version of the slab problem, in a changing (implicit) slab. Formally, the task is to maintain a set S of segments, subject to:

QUERY(p): locate point p among the segments in S. It is guaranteed that the segments of S are intersected by some vertical line, and that p is inside the maximal vertical slab which does not contain any endpoint from S.

INSERT(s, s_+): insert a segment s into S, given a pointer to the segment $s_+ \in S$ which is immediately above s. (This ordering is strict in the maximal vertical slab.)

DELETE(s): delete a segment from S, given by a pointer.

If S does not change, the slab is fixed and we have, by assumption, a solution with $O(n)$ space and $t(n)$ query time. However, for the dynamic problem we have a different challenge: as segments are inserted or deleted, the vertical slab from which the queries come can change significantly. This seems to make the problem hard and we do not know a general solution comparable to the static case.

However, we can solve the *semionline* version of the problem, where INSERT is replaced by:

INSERT(s, s_+, t): insert a segment s as above. Additionally, it is guaranteed that the segment will be deleted at time t in the future.

Note that our application will be based on a sweep-line algorithm, which guarantees that the left endpoint of every inserted segment and the right endpoint of every deleted segment appear in order. Thus, by sorting all x-coordinates, we can predict the deletion time when the segment is inserted.

Exponential trees. We will use exponential trees [4, 7], a remarkable idea coming from the world of integer search. This is a technique for converting a black-box static succesor structure into a dynamic one, while maintaining (near) optimal running times. The approach is based on the following key ideas:

- *construction:* Pick B *splitters*, which separate the set S into subsets of size n/B. Build a static data structure for the splitters (the *top structure*), and then recursively construct a structure for each subset (*bottom structures*).
- *query:* First search in the top structure (using the search for the static data structure), and then recurse in the relevant bottom structure.
- *update:* First search among splitters to see which bottom structure is changed. As long as the bottom structure still has between $\frac{n}{2B}$ and $\frac{2n}{B}$ elements, update it recursively. Otherwise, split the bottom structure in two, or merge with an adjacent sibling. Rebuild the top structure from scratch, and recursively construct the modified bottom structure(s).

An important point is that this scheme cannot guarantee splitters are actually in S. Indeed, an element chosen as a splitter can be deleted before we have enough credit to amortize away the rebuilding of the top structure. However, this creates significant issues for the segment-predecessor problem, due to the changing domain of queries. If some splitters are deleted from S, the vertical slab defining the queries may now extend beyond the endpoints of these splitters. Then, the support lines of the splitters may intersect in this extended slab, which means splitters no longer separate the space of queries.

Our contribution is a variant of exponential trees which ensures splitters are always members of the current set S given semionline knowledge. Since splitters are in the set, we do not have to worry about the vertical slab extending beyond the domain where the splitters actually decompose the search problem. Thus, we construct exponential trees which respect the geometric structure of the point location problem.

Construction and queries. We maintain two invariants at each node of the exponential tree: the number of splitters B is $\Theta(n^{1/3})$; and there are $\Theta(n^{2/3})$ elements between every two consecutive splitters. Later, we will describe how to pick the splitters at construction time in $O(n)$ time, satisfying some additional properties. Once splitters are chosen, the top structure can be constructed in $O(B) = o(n)$ time and we can recurse for the bottom structures. Given this, the construction and query times satisfy the following recurrences, for $\sum_i n_i = n$ and $n_i = O(n^{2/3})$:

$$\begin{aligned} P(n) &= O(n) + \sum_i P(n_i) = O(n \lg\lg n) \\ Q(n) &= O(t(B)) + \max_i Q(n_i) \leq O(t(n^{1/3})) + Q(O(n^{2/3})) \end{aligned}$$

The query satisfies the same type of recurrence as in the other methods, so $Q(n) = O(t(n))$ assuming $t(n)/\lg^\delta n$ is increasing for some $\delta > 0$.

Handling updates. Let $\tilde{n}$ be the number of segments, and $\tilde{B}$ the number of splitters, when the segment-successor structure was created. As before, n and B denote the corresponding values at present time. We make the following twists to standard exponential trees, which leads to splitters always being part of the set:

- *choose splitters wisely:* Let an *ideal splitter* be the splitter we would choose if we only cared about splitters being uniformly distributed. (During construction, this means $\tilde{n}/\tilde{B}$ elements apart; during updates, the rule is specified below.) We will look at $\frac{1}{10}(\tilde{n}/\tilde{B})$ segments above and below an ideal splitter, and choose as the actual splitter the segment which will be deleted farthest into the future. This is the crucial place where we make use of semionline information. Though it is possible to replace this with randomization, we are interested in a deterministic solution.

- *rebuild often:* Normally, one rebuilds a bottom structure (merging or splitting) when the number of elements inside it changes by a constant factor. Instead, we will rebuild after any $\frac{1}{10}(\tilde{n}/\tilde{B})$ updates in that bottom structure, regardless of how the number of segments changed.

- *rebuild aggressively:* When we decide to rebuild a bottom structure, we always include in the rebuild its two adjacent siblings. We merge the three lists of segments, decide whether to break them into 2, 3 or 4 subsets (by the *balance rule* below), and choose splitters between these subsets. Ideal splitters are defined as the (1, 2 or 3) segments which divide uniformly the list of segments participating in the rebuild.

Lemma 4.4 *No segment is ever deleted while it is a splitter.*

Proof: Say a segment s is chosen as a splitter. In one of the two adjacent substructures, there are at least $\frac{1}{10}(\tilde{n}/\tilde{B})$ segments which get deleted before s. This means one of the two adjacent structures gets rebuilt before the splitter is deleted. But the splitter is included in the rebuild. Hence, a splitter is never deleted between the time it becomes a splitter and the next rebuild which includes it. □

Lemma 4.5 *There exists a balance rule ensuring all bottom structures have $\Theta(\tilde{n}/\tilde{B})$ elements at all times.*

Proof: This is standard. We ensure inductively that each bottom structure has between $0.6(\tilde{n}/\tilde{B})$ and $2(\tilde{n}/\tilde{B})$ elements. During construction, ideal splitters generate bottom structures of exactly $(\tilde{n}/\tilde{B})$ elements. When merging three siblings, the number of elements is between $1.8(\tilde{n}/\tilde{B})$ and $6(\tilde{n}/\tilde{B})$. If it is at most $3(\tilde{n}/\tilde{B})$, we split into two ideally equal subsets. If it is at most $3.6(\tilde{n}/\tilde{B})$, we split into three subsets. Otherwise, we split into four. These guarantee the ideal sizes are between $0.9(\tilde{n}/\tilde{B})$ and $1.5(\tilde{n}/\tilde{B})$. The ideal size may be modified due to the fuzzy choice of splitters (by $0.1(\tilde{n}/\tilde{B})$ on each side), and by $0.1(\tilde{n}/\tilde{B})$ updates that we tolerate to a substructure before rebuilding. Then, the number of elements stays within bounds until the structure is rebuilt. □

We can use this result to ensure the number of splitters is always $B = O(\tilde{B})$. For a structure other than the root, this follows immediately: the lemma applied to the parent shows n for the current structure can only change by constant factors before we rebuild, i.e. $n = \Theta(\tilde{n})$. For the root, we enforce this through global rebuilding when the number of elements changes by a constant factor.

Thus, we have ensured that the number of splitters and the size of each child are within constant factors of the ideal-splitter scenario.

Let us finally look at the time for an INSERT or DELETE. These operations first update the appropriate leaf of the exponential tree; we know the appropriate leaf since we are given a point to the segment (for delete) or its neighbor (for insert). Then, the operations walk up the tree, triggering rebuilds where necessary.

For each of the $O(\lg\lg n)$ levels, an operation stores $O(\lg\lg n)$ units of potential, making for a total cost of $O((\lg\lg n)^2)$ per update. The potential accumulates in each node of the tree until that node causes a rebuild of itself and some siblings. At that point, the potential of the node causing the rebuild is reset to zero. We now show that this potential is enough to pay for the rebuilds. Rebuilding a bottom structure (including the siblings involved in the rebuild) takes time $O(1) \cdot P(O(n/B)) = O(\frac{n}{B}\lg\lg\frac{n}{B})$. Furthermore, there is a cost of $O(B) = O(n^{1/3}) = o(n/B)$ for rebuilding the top structure. However, these costs are incurred after $\Omega(\tilde{n}/\tilde{B}) = \Omega(n/B)$ updates to that bottom structure, so there is enough potential to cover the cost.

Bucketing. We now show how to reduce the update time to a constant. We use the decomposition idea from above, but now with $B = O(n/(\lg\lg n)^2)$ splitters. The splitters are maintained in the previous data structure, which supports updates in $O((\lg\lg n)^2)$ time. The bottom structures have $\Theta((\lg\lg n)^2)$ elements, and we can simply use a linked list to maintain them in order. The query time is increased by $O((\lg\lg n)^2)$ because we have to search through a bottom a list, but that is a lower order term. Updating a bottom list now takes constant time, given a pointer to a neighbor. An update to the top structure only occurs after $\Omega((\lg\lg n)^2)$ updates to a bottom structure, so the updates in the top structure cost $O(1)$ amortized.

Sweep-line construction. We first sort the x-coordinates corresponding to the endpoints, taking $\mathrm{sort}(2n)$ time. To know which of the $O(n)$ slabs a query point lies in, we construct an integer successor structure for the x-coordinates. The optimal complexity of successor search cannot exceed the optimal complexity of point location, so this data structure is negligible.

We now run the sweep-line algorithm, inserting and deleting segments in the segment successor structure, in order of the x-coordinates. For each insert, we also need to perform a query for the left endpoint, which determines where the inserted segment goes (i.e. an adjacent segment in the linear order). Thus the overall construction time is $O(n \cdot t(n))$.

We can reduce the construction time to $O(\mathrm{sort}(n))$, if we know where each insert should go, and can avoid the queries at construction time. Finding the line segment immediately above/below each endpoint is equivalent to constructing the *trapezoidal decomposition* of the line segments [12]. For any connected subdivision, it is known that we can compute the decomposition in deterministic linear time by Chazelle's algorithm [23]. If the input is a triangulation or a convex subdivision, we can easily compute the decomposition directly.

Persistence. It remains to make the segment successor structure persistent, leading to a data structure with linear space. Making exponential trees persistent is a standard exercise. We augment each pointer to a child node with a 1-d successor structure (the dimension is time). Whenever the child is rebuilt, we store a pointer to the new version and the time when the new version was created. To handle global rebuilding at the root, the successor structure for the x-coordinates stores a pointer

to the current root when each slab is considered. The leaves of the tree are linked lists of $O((\lg\lg n)^2)$ elements, which can be made persistent by standard results for the pointer machine [31].

Given k numbers in $\{1, \ldots, 2n\}$ (our time universe), a van Emde Boas data structure for integer successor can be constructed in $O(k)$ time deterministically [58], supporting queries in $O(\lg\lg n)$ time. Thus, our point location query incurs an additional $O(\lg\lg n)$ cost on each of the $O(\lg\lg n)$ levels, which is a lower order term.

The space cost for persistence is of course bounded by the update time in the segment successor structure. Since we have $2n$ updates with constant cost for each one, the space is linear. The additional space due to the van Emde Boas structures for child pointers is also linear, as above.

5 Segment Intersection

In this section, we consider the problem of computing all k intersections among a set S of n line segments in the plane, where all coordinates are $O(w)$-bit integers, or more generally $O(w)$-bit rationals. We actually solve a more general problem: constructing the trapezoidal decomposition $T(S)$, defined as the subdivision of the plane into trapezoids formed by the segments of S and vertical upward and downward rays from each endpoint and intersection. Notice that the intersection points have $O(w)$-bit rational coordinates.

We use a random sampling approach, as in the previous section. Take a random sample $R \subseteq S$ of size r. Compute its trapezoidal decomposition $T(R)$ by a known algorithm [51] in $O(r \lg r + |T(R)|)$ time. Store $T(R)$ in the point-location data structure from Theorem 4.3.

For each segment $s \in S$, we first find the trapezoid of $T(R)$ containing the left endpoint of s by a point location query. By a walk in $T(R)$, we can then find all trapezoids of $T(R)$ that intersects s in time linear in the total face length of such trapezoids, where the *face length* ℓ_Δ of a trapezoid Δ refers to the number of edges of $T(R)$ on the boundary of Δ. As a result, for each trapezoid $\Delta \in T(R)$, we obtain the subset S_Δ of all segments of S intersecting Δ (the so-called *conflict list* of Δ). The time required thus far is $O(n \cdot t(r, w) + \sum_{\Delta \in T(R)} |S_\Delta| \ell_\Delta)$, where $t(n, w)$ is as defined in Theorem 4.3. We then construct $T(S_\Delta)$ inside Δ, by using a known algorithm in $O(|S_\Delta| \lg |S_\Delta| + k_\Delta)$ time, where k_Δ denotes the number of intersections within Δ (with $\sum_\Delta k_\Delta = k$). We finally stitch these trapezoidal decompositions together to obtain the trapezoidal decomposition of the entire set S.

By a standard analysis of Clarkson and Shor [27, 51],

$$E[|T(R)|] = O(r + kr^2/n^2) \qquad \text{and} \qquad E\left[\sum_{\Delta \in T(R)} |S_\Delta| \lg |S_\Delta|\right] = O((r + kr^2/n^2) \cdot (n/r) \lg(n/r)).$$

Clarkson and Shor had also specifically shown [27, Lemma 4.2] that

$$E\left[\sum_{\Delta \in T(R)} |S_\Delta| \ell_\Delta\right] = O((r + kr^2/n^2) \cdot (n/r)) = O(n \cdot (1 + kr/n^2)).$$

The total expected running time is $O(r \lg r + n \cdot t(r, w) + n \lg(n/r) + k)$. Setting $r = \lfloor n / \lg n \rfloor$ yields the following result, since $t(n, w)$ exceeds $\lg\lg n$:

Theorem 5.1 *Let $t(n, w)$ be as in Theorem 4.3. Given n line segments in the plane whose endpoints have $O(w)$-bit rational coordinates, we can find all k intersections, and compute the trapezoidal decomposition, in $O(n \cdot t(n, w) + k)$ expected time.*

6 3-d Convex Hulls

We next tackle the well-known problem of constructing the convex hull of a set S of n points in 3-d, under the assumption that the coordinates are w-bit integers, or more generally $O(w)$-bit rationals.

We again use a random sampling approach. First it suffices to construct the upper hull (the portion of the hull visible from above), since the lower hull can be constructed similarly. Take a random sample $R \subseteq S$ of size r. Compute the upper hull of R in $O(r \lg r)$ time by a known algorithm [12, 55]. The xy-projection of the faces of the upper hull is a triangulation; store the triangulation in the point-location data structure from Theorem 4.3.

For each point $s \in S$, consider the dual plane s^* [12, 32, 51]. Constructing the upper hull is equivalent to constructing the lower envelope of the dual planes. Let $T(R)$ denote a *canonical triangulation* [26, 51] of the lower envelope $\mathrm{LE}(R)$ of the dual planes of R, which can be computed in $O(r)$ time given $\mathrm{LE}(R)$. For each $s \in S$, we first find a vertex of the $\mathrm{LE}(R)$ that is above s^*, say, the extreme vertex along the normal of s^*; in primal space, this is equivalent to finding the facet of the upper hull that contains s when projected to the xy-plane—a point location query. By a walk in $T(R)$, we can then find all cells of $T(R)$ that intersect s^* in time linear in the number of such cells. As a result, for each cell $\Delta \in T(R)$, we obtain the subset S^*_Δ of all planes s^* intersecting Δ. The time required thus far is $O(n \cdot t(r,w) + \sum_{\Delta \in T(R)} |S^*_\Delta|)$. We then construct $\mathrm{LE}(S^*_\Delta)$ inside S^*_Δ, by using a known $O(|S^*_\Delta| \lg |S^*_\Delta|)$-time convex-hull/lower-envelope algorithm. We finally stitch these lower envelopes together to obtain the lower envelope/convex hull of the entire set.

By a standard analysis of Clarkson and Shor [27, 51],

$$E\left[\sum_{\Delta \in T(R)} |S^*_\Delta|\right] = O(n) \qquad \text{and} \qquad E\left[\sum_{\Delta \in T(R)} |S^*_\Delta| \lg |S^*_\Delta|\right] = O(r \cdot (n/r) \lg(n/r)).$$

The total expected running time is $O(r \lg r + n \cdot t(r,w) + n \lg(n/r))$. Setting $r = \lfloor n/\lg n \rfloor$ yields the following result, since $t(n,w)$ exceeds $\lg \lg n$:

Theorem 6.1 *Let $t(n,w)$ be as in Theorem 4.3. Given n points in three dimensions with $O(w)$-bit rational coordinates, we can compute the convex hull in $O(n \cdot t(n,w))$ expected time.*

7 Other Consequences

To demonstrate the impact of the preceding results, we list a sample of improved algorithms and data structures that can be derived from our work. (See Figure 1.)

Corollary 7.1 *Let $t(n,w)$ be as in Theorem 4.3. Given n points in the plane with $O(w)$-bit rational coordinates,*

(a) *we can construct the Voronoi diagram, or equivalently the Delaunay triangulation, in $O(n \cdot t(n,w))$ expected time;*

(b) *we can construct the Euclidean minimum spanning tree in $O(n \cdot t(n,w))$ expected time;*

(c) *we can find the largest empty circle that has its center inside the convex hull in $O(n \cdot t(n,w))$ expected time;*

(d) *we can build a data structure in $O(n \cdot t(n,w))$ expected time and $O(n)$ space so that nearest/farthest neighbor queries under the Euclidean metric can be answered in $O(t(n,w))$ time.*

(e) *we can build an $O(n \lg\lg n)$-space data structure so that circular range queries (reporting all k points inside a query circle) and "k nearest neighbors" queries (reporting the k nearest neighbors to a query point) can be answered in $O(t(n,w)+k)$ time.*

Furthermore,

(f) *we can triangulate a polygon with holes, having n vertices with $O(w)$-bit rational coordinates, in $O(n \cdot t(n,w))$ deterministic time;*

(g) *we can preprocess a convex polygon P, having n vertices with $O(w)$-bit rational coordinates, in $O(n)$ space, so that gift wrapping queries (finding the two tangents of P through an exterior point) and ray shooting queries (intersecting P with a line) can be answered in $O(t(n,w))$ time;*

(h) *we can compute the convex hull of n points in three dimensions with $O(w)$-bit rational coordinates in $O(n \cdot t(H^{1+o(1)},w))$ expected time, where H is the number of hull vertices.*

Proof:

(a) By a lifting transformation [12, 32, 52], the 2-d Delaunay triangulation can be obtained from the convex hull of a 3-d point set (whose coordinates still have $O(w)$ bits). The result follows from Theorem 6.1.

(b) The minimum spanning tree (MST) is contained in the Delaunay triangulation. We can compute the MST of the Delaunay triangulation, a planar graph, in linear time, for example, by Borůvka's algorithm. The result thus follows from (a).

(c) We can determine the optimal circle from the Voronoi diagram (whose coordinates still have $O(w)$-bit numerators and denominators) in linear time [55]. Again the result follows from (a). (Curiously, the 1-d version of the problem admits an $O(n)$-time RAM algorithm of Gonzalez; see [55].)

(d) Nearest neighbor queries reduce to point location in the Voronoi diagram, so the result follows from (a) and Theorem 4.3. Farthest neighbor queries are similar.

(e) The result is obtained by adopting the range reporting data structure from [18], using Theorem 4.3 to handle the necessary point location queries.

(f) It is known [33] that a triangulation can be constructed from the trapezoidal decomposition of the edges in linear time. The result follows from Theorem 5.1 if randomization is allowed. Deterministically, we can instead compute the trapezoidal decomposition by running the algorithm from Section 4.3, since that algorithm explicitly maintains a sorted list of segments that intersect the vertical sweep line at any given time.

(g) For wrapping queries, it suffices to compute the tangent from a query point q with P to the left, say, of this directed line. Decompose the plane into regions where two points are in the same region iff they have the same answer; the regions are wedges. The result follows by performing a point location query (Theorem 4.3).

Ray shooting queries reduce to gift wrapping queries in the dual convex polygon (whose coordinates are still $O(w)$-bit rationals).

(h) The result is obtained by adopting the output-sensitive convex hull algorithm from [17], using Theorem 6.1 to compute the subhull of each group. For readers familiar with [17], we note that the running time for a group size m is now $O(n \cdot t(m,w) + H(n/m)\lg m)$; we can choose $m = \lfloor H \lg H \rfloor$ and apply the same "guessing" trick. □

8 Higher Dimensions

The first approach for point location from Section 3 can be generalized to any constant dimension d. The main observation is very similar to Observation 3.2:

Observation 8.1 *Let S be a set of n disjoint $(d-1)$-dimensional simplices in $\mathbb{R}^d$, whose vertices lie on d vertical segments $I_0, \ldots, I_{d-1}$ of length $2^{\ell_0}, \ldots, 2^{\ell_{d-1}}$. We can find $O(b)$ simplices $s_0, s_1, \ldots \in S$ in sorted order, which include the lowest and highest simplex of S, such that*

(1) *for each i, there are at most n/b simplices of S between s_i and s_{i+1},* or *the endpoints of s_i and s_{i+1} lie on a subinterval of I_j of length $2^{\ell_j - h}$ for some j; and*

(2) *there exist $O(b)$ simplices $\tilde{s}_0, \tilde{s}_2, \ldots$, with $s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec \cdots$ and vertices on $I_1, \ldots, I_d$, such that distances between endpoints of the $\tilde{s}_i$'s on I_j are all multiples of $2^{\ell_j - h}$.*

Applying this observation recursively in the same manner as in Section 3.2, we can get an $O(\lg n / \lg\lg n)$-time query algorithm for point location among n disjoint $(d-1)$-simplices spanning a vertical prism, with $O(n)$ space, for any fixed constant d.

In the implementation of the special word operation, we first apply a projective transformation to make $I_0 = \{(0,\ldots,0)\} \times [0, 2^h)$, $I_1 = \{(2^h, 0, \ldots, 0)\} \times [0, 2^h)$, $\ldots$, $I_{d-1} = \{(0, , \ldots, 0, 2^h)\} \times [0, 2^h)$. This can be accomplished in three steps. First, by an affine transformation, we can make the first $d-1$ coordinates of $I_0, \ldots, I_{d-1}$ to be $(0,\ldots,0)$, $(1,0,\ldots,0)$, $\ldots$, $(0,0,\ldots,0,1)$, while leaving the d-th coordinate unchanged. Then by a shear transformation, we can make the bottom vertices of $I_0, \ldots, I_{d-1}$ lie on $x_d = 0$, while leaving the lengths of the intervals unchanged. Finally, we map $(x_1, \ldots, x_d)$ to

$$\frac{1}{2^{\ell_0}(1 - x_1 - \cdots - x_{d-1}) + 2^{\ell_1} x_1 + \cdots + 2^{\ell_{d-1}} x_{d-1}} \left(2^{h+\ell_1} x_1,\ \ldots,\ 2^{h+\ell_{d-1}} x_{d-1},\ 2^h x_d\right).$$

The coordinates of the $\tilde{s}_i$'s become h-bit integers. We round the query point q to a point $\tilde{q}$ with h-bit integer coordinates, and by the same reasoning as in Section 3.2, it suffices to locate $\tilde{q}$ among the $\tilde{s}_i$'s (since every two $(d-1)$-simplices have separation at least one along all the axis-parallel directions). The location of $\tilde{q}$ can be accomplished as in Section 3.2, by performing the required arithmetic operations on $O(h)$-bit integers in parallel, using a constant number of arithmetic operations on w-bit integers.

The alternative bounds in Section 3.3 also follow in a similar manner.

Theorem 8.2 *Consider a sorted list of n disjoint (open) $(d-1)$-simplices spanning a vertical prism in $\mathbb{R}^d$ where the vertices have $O(w)$-bit rational coordinates. For any $h \geq 1$, we can build a data structure with space and preprocessing time $O(n \cdot 2^{dh})$, so that point location queries take time: $O\left(\min\left\{\lg n / \lg\lg n,\ \sqrt{w / \lg w},\ w/h\right\}\right)$.*

We can solve the point location problem for any subdivision of $\mathbb{R}^d$ into polyhedral cells, where vertices have $O(w)$-bit integer or rational coordinates. For a naive solution with polynomial preprocessing time and space, we project all $(d-2)$-faces vertically to $\mathbb{R}^{d-1}$, triangulate the resulting arrangement in $\mathbb{R}^{d-1}$, lift each cell to form a vertical prism, and build the data structure from Theorem 8.2 inside each prism. Given a point q, we first locate the prism containing q by a $(d-1)$-dimensional point location query (which can be handled by induction on d) and then search inside this prism. The overall query time is asymptotically the same for any constant d.

As many geometric search problems can be reduced to point location in higher-dimensional space, our result leads to many more applications. We mention the following:

Corollary 8.3 *Let $t(n,w)$ be as in Theorem 4.3.*

(a) *We can preprocess n points in $\mathbb{R}^d$ with $O(w)$-bit rational coordinates, in $n^{O(1)}$ time and space, so that exact nearest/farthest neighbor queries under the Euclidean metric can be answered in $O(t(n,w))$ time.*

(b) *We can preprocess a fixed polyhedral robot and a polyhedral environment of size n in $\mathbb{R}^d$ whose vertices have $O(w)$-bit rational coordinates, in $n^{O(1)}$ time and space, so that we can decide whether two given placements of the robot are reachable by translation, in $O(t(n,w))$ time.*

(c) *Given an arrangement of n semialgebraic sets of the form $\{x \in \mathbb{R}^d \mid p_i(x) \geq 0\}$ where each p_i is fixed-degree polynomial with $O(w)$-bit rational coefficients, put two points in the same region iff they belong to exactly the same sets. (Regions may be disconnected.) We can build a data structure in $n^{O(1)}$ time and space, so that (a label of) the region containing a query point can be identified in $O(t(n,w))$ time.*

(d) *Given n disjoint x-monotone curve segments in $\mathbb{R}^2$ that are graphs of fixed-degree univariate polynomials with $O(w)$-bit rational coefficients, we can build a data structure in $n^{O(1)}$ time and space, so that the curve segment immediately above a query point can be found in $O(t(n,w))$ time.*

(e) *Part* (a) *also holds under the L_p metric for any constant integer $p > 2$.*

(f) *We can preprocess an arrangement of n hyperplanes in $\mathbb{R}^d$ with $O(w)$-bit rational coefficients, in $n^{O(1)}$ time and space, so that ray shooting queries (finding the first hyperplane hit by a ray) can be answered in $O(t(n,w))$ time.*

(g) *We can preprocess a convex polytope with n facets in $\mathbb{R}^d$ whose vertices have $O(w)$-bit rational coordinates, in $n^{O(1)}$ time and space, so that linear programming queries (finding the extreme point in the polytope along a given direction) can be answered in $O(t(n,w))$ time.*

Proof:

(a) This follows by point location in the Voronoi diagram.

(b) This reduces to point location in the arrangement formed by the Minkowski difference [12, 52] of the environment with the robot.

(c) By linearization (i.e., by creating a new variable for each monomial), the problem is reduced to point location in an arrangement of n hyperplanes in a sufficiently large but constant dimension.

(d) This is just a 2-d special case of (c), after adding vertical lines.

(e) This follows by applying (c) to the $O(n^2)$ semialgebraic sets $\{x \in \mathbb{R}^d \mid \|x - a_i\|_p \leq \|x - a_j\|_p\}$ over all pairs of points a_i and a_j.

(f) Parametrize the query ray $\{x + ty \mid t \geq 0\}$ with $2d$ variables $x, y \in \mathbb{R}^d$. Suppose that the i-th hyperplane is $\{x \in \mathbb{R}^d \mid a_i \cdot x = 1\}$. The "time" the ray hits this hyperplane (i.e., when $a_i \cdot (x+ty) = 1$) is given by $t = (1-a_i \cdot x)/(a_i \cdot y)$. We apply (c) to the $O(n^2)$ semialgebraic sets $\{(x, y) \in \mathbb{R}^{2d} \mid (1-a_i \cdot x)/(a_i \cdot y) \leq (1-a_j \cdot x)/(a_j \cdot y)\}$ over all i, j and $\{(x, y) \in \mathbb{R}^{2d} \mid a_i \cdot x \leq 1\}$ over all i. It is not difficult to see that all rays whose parameterizations lie in the same region in this arrangement of semialgebraic sets have the same answer.

(g) Linear programming queries reduce to ray shooting inside the dual convex polytope, which has $n^{O(1)}$ facets, so the result follows from (f). □

Remarks. Actually, allowing $n^{O(1)}$ space, we can also get query time $O(w/\log n)$ in Corollary 8.3 by setting $h = \Theta(\log n)$.

Alternatively, the preprocessing time and space in Corollary 8.3 can be reduced by applying random sampling techniques [26, 27] like in Section 4.2. For example, for (a), we can achieve $O(t(n, w))$ query time with $O(n^{\lceil d/2 \rceil} \lg^{O(1)} n)$ space. For (d), we can achieve $O(t(n, w))$ query time with $O(n)$ space; by the same techniques as in Section 5, we can also obtain an $O(n \cdot t(n, w) + k)$-time randomized algorithm for segment intersection for such curve segments.

9 Conclusions

We have hardly exhausted all implications of our results. What we have shown is that the complexity of many previously "solved" problems in computational geometry may have to be revised, at least for researchers who are willing to embrace the transdichotomous RAM model. The most popular model for computational geometers is the unit-cost real RAM. While we do not intend to change this default, the transdichotomous model for integer or rational input also deserves to be studied. If one can take the example of 1-d achievements, it is reasonable to expect a rich theory, leading to a fascinating understanding of fundamental problems.

One possible complaint about our algorithms is their use of w-bit integer multiplication and division, which are not AC^0 operations and in reality take more than constant time as w grows. However, the previous real-RAM algorithms all need multiplication as well, and our algorithms use comparatively fewer multiplications. Notice that when the input is from a polynomial-size grid ($w = O(\lg n)$), multiplication and division on εw-bit words can be simulated by table lookup, so there is no dispute about the model used in this case.

If it has not been made clear already, our study here is primarily theoretical. As noted above, however, our work can also be seen as explaining a common engineering practice of using grid "heuristics" for solving point location. One can thus hope that mathematically founded ideas for grid search could lead to better practical algorithms. The greatest obstacle to this seems to be the reduction to the slab subproblem, which incurs a constant but practically significant overhead.

The main theoretical question is to determine the precise complexity of the subproblem of 2-d point location in a slab. As the 1-d successor search problem has essentially been completely solved, the 2-d problem is all the more pressing. In subsequent work, we [22] have shown how to solve the

offline problem of answering a batch of n queries more efficiently. But for *online* point location or nearest neighbor queries we do not know any improvement to the bounds of this paper, even for the special case $w = O(\lg n)$. A lower bound strictly stronger than that for 1-d successor search would also be exciting.

Finally, it would be interesting to see if our improvements help for dynamic planar point location. We note that Husfeldt *et al.* [41] have shown an $\Omega(\lg n / \lg \lg n)$ lower bound for dynamic planar point location in monotone subdivisions. However, the hardness seems to stem from non-geometric issues, namely, identifying (a label of) the face incident to a given edge, not locating which edge is immediately above a given point.

References

[1] L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM J. Discrete Math.*, 9:129–150, 1996.

[2] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd ACM Sympos. Theory Comput.*, pages 476–483, 2001.

[3] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30:164–187, 2001.

[4] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th IEEE Sympos. Found. Comput. Sci.*, pages 135–141, 1996.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Comput. Sys. Sci.*, 57:74–93, 1998.

[6] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoret. Comput. Sci.*, 215:337–344, 1999.

[7] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 32nd ACM Sympos. Theory Comput.*, pages 335–342, 2000.

[8] S. Arya, T. Malamatos, and D. M. Mount. A simple entropy-based algorithm for planar point location. *ACM Trans. Algorithms*, 3(2): article 17, 2007.

[9] S. Arya, T. Malamatos, D. M. Mount, and K. C. Wong. Entropy-preserving cuttings and space-efficient planar point location. *SIAM J. Comput.*, 37:584–610, 2007.

[10] I. Baran, E. D. Demaine, and M. Pătraşcu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50:584–596, 2008.

[11] P. Beame and F. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Sys. Sci.*, 65:38–72, 2002.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2000.

[13] M. de Berg, M. van Kreveld, and J. Snoeyink. Two-dimensional and three-dimensional point location in rectangular subdivisions. *J. Algorithms*, 18:256–277, 1995.

[14] M. W. Bern, H. J. Karloff, P. Raghavan, and B. Schieber. Fast geometric approximation techniques and geometric embedding problems. *Theoret. Comput. Sci.*, 106:265–281, 1992.

[15] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance transform algorithms. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 17:529-533, 1995.

[16] M. Cary. Towards optimal ϵ-approximate nearest neighbor algorithms. *J. Algorithms*, 41:417–428, 2001.

[17] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996.

[18] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM J. Comput.*, 30:561–575, 2000.

[19] T. M. Chan. On enumerating and selecting distances. *Int. J. Comput. Geom. Appl.*, 11:291–304, 2001.

[20] T. M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473, 2002.

[21] T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. *Comput. Geom. Theory Appl.*, 35:20–35, 2006.

[22] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, II: Offline search. In preparation. Preliminary version appeared in *Proc. 39th ACM Sympos. Theory Comput.*, pages 31–39, 2007.

[23] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[24] B. Chazelle. Geometric searching over the rationals. In *Proc. 7th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 1643, Springer-Verlag, pages 354–365, 1999.

[25] L. P. Chew and S. Fortune. Sorting helps for Voronoi diagrams. *Algorithmica*, 18:217–228, 1997.

[26] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.

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

[28] M. Degermark, A. Brodnik, S. Carlsson and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, pages 3–14, 1997.

[29] E. D. Demaine and M. Pătraşcu. Tight bounds for dynamic convex hull queries (again). In *Proc. 23rd ACM Sympos. Comput. Geom.*, pages 354–363, 2007.

[30] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.

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

[32] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[33] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. on Graphics*, 3:153–174, 1984.

[34] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31:538–544, 1984.

[35] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Sys. Sci.*, 47:424–436, 1993.

[36] M. T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Sys. Sci.*, 51:374–389, 1995.

[37] M. T. Goodrich, M. W. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 757–766, 1997.

[38] Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms*, 50:96–105, 2004.

[39] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 135–144, 2002.

[40] S. Har-Peled and S. Mazumdar. Fast algorithms for computing the smallest k-enclosing disc. *Algorithmica*, 41:147–157, 2005.

[41] T. Husfeldt, T. Rauhe, and S. Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic J. Comput.*, 3:323–336, 1996.

[42] J. Iacono. Expected asymptotically optimal planar point location. *Comput. Geom. Theory Appl.*, 29:19–22, 2004.

[43] J. Iacono and S. Langerman. Dynamic point location in fat hyperrectangles with integer coordinates. In *Proc. 12th Canad. Conf. Comput. Geom.*, pages 181–186, 2000.

[44] R. G. Karlsson. *Algorithms in a Restricted Universe*. Ph.D. thesis, TR CS-84-50, Dept. of Computer Science, University of Waterloo, 1984.

[45] R. G. Karlsson and M. H. Overmars. Scanline algorithms on a grid. *BIT*, 28:227–241, 1988.

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

[47] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoret. Comput. Sci.*, 28:263–276, 1984.

[48] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.

[49] C. W. Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35:1494–1525, 2006.

[50] C. W. Mortensen, R. Pagh, and M. Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Sympos. Theory Comput.*, pages 104–111, 2005.

[51] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, N.J., 1994.

[52] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd ed., 1998.

[53] M. H. Overmars. Computational geometry on a grid: an overview. Tech. Report RUU-CS-87-04, Utrecht U., 1987.

[54] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Sympos. Theory Comput.*, pages 232–240, 2006.

[55] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[56] M. O. Rabin. Probabilistic algorithms. In *Algorithms and Complexity* (J. F. Traub, ed.), pages 21–30, Academic Press, New York, NY, 1976.

[57] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proc. 4th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 1136, Springer-Verlag, pages 121–137, 1996.

[58] M. Ružić. Making deterministic signatures quickly. In *Proc. 18th ACM-SIAM Sympos. Discrete Algorithms*, pages 900–909, 2007.

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

[60] R. Seidel. A method for proving lower bounds for certain geometric problems. In *Computational Geometry* (G. T. Toussaint, ed.), North-Holland, Amsterdam, pages 319–334, 1985.

[61] R. Seidel and U. Adamy. On the exact worst case complexity of planar point location. *J. Algorithms*, 37:189–217, 2000.

[62] J. Snoeyink. Point location. In *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), CRC Press, Boca Raton, FL, pages 559–574, 1997.

[63] M. Thorup. Fast deterministic sorting and priority queues in linear space. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 550–555, 1998.

[64] M. Thorup. Equivalence between priority queues and sorting. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 125–134, 2002.

[65] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42:205–230, 2002.

[66] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6:80–82, 1977.

[67] D. E. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(n)$. *Inform. Process. Lett.*, 17:81–84, 1983.

[68] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29:1030–1049, 2000.