Point Location in $o(\log n)$ Time, Voronoi Diagrams in $o(n \log n)$ Time, and Other Transdichotomous Results in Computational Geometry

Timothy M. Chan*

Abstract

Given n points in the plane with integer coordinates bounded by $U \leq 2^w$, we show that the Voronoi diagram can be constructed in $O(\min\{n \log n / \log\log n, n\sqrt{\log U}\})$ expected time by a randomized algorithm on the unit-cost RAM with word size w. Similar results are also obtained for many other fundamental problems in computational geometry, such as constructing the convex hull of a 3-dimensional point set, computing the Euclidean minimum spanning tree of a planar point set, triangulating a polygon with holes, and finding intersections among a set of line segments.

These are the first results to beat the $\Omega(n \log n)$ algebraic-decision-tree lower bounds known for these problems. The results are all derived from a new two-dimensional version of fusion trees that can answer point location queries in $O(\min\{\log n / \log\log n, \sqrt{\log U}\})$ time with linear space. Higher-dimensional extensions and applications are also mentioned in the paper.

1. Introduction

The sorting problem requires $\Omega(n \log n)$ 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 [47, 48] can sort in $O(n \log\log U)$ time. Fredman and Willard [27] showed that $o(n \log n)$ time is possible even regardless of how U relates to n: their *fusion tree* yields a deterministic $O(n \log n / \log\log n)$-time and a randomized $O(n\sqrt{\log n})$-time sorting algorithm. Many subsequent improvements have been given (see Section 2). In all of the above, the underlying model of computation is a RAM that supports standard operations on w-bit words with unit cost, under the reasonable assumptions that $w \geq \log n$, 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. (The adjectives "transdichotomous" and "conservative" have been associated with such a word-level RAM model.)

Similarly, while searching requires $\Omega(\log n)$ time for comparison-based algorithms, sublogarithmic data structures are possible for integer input (see Section 2). These improved techniques have been applied to obtain faster graph algorithms for basic problems like minimum spanning trees and shortest paths (e.g., [28, 45]).

Applications of these techniques to geometric problems have also been considered, 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-rectilinear things (lines of arbitrary slopes, the Euclidean metric, etc.) and thus has largely remained unaffected by the breakthroughs on integer sorting and searching. It is not even known, for instance, whether the two-dimensional Voronoi diagram can be constructed in $o(n \log n)$ time, when the input points come from an integer grid of polynomial size $U = n^{O(1)}$. (In other words, no 2-d "equivalent" of radix-sort has been found.)

This paper will change all this. We show, for the first time, that the known $\Omega(n \log n)$ lower bounds for algebraic computational trees can be broken for many of the core problems in computational geometry, when the input coordinates are integers in $[0, U)$ with $U \leq 2^w$. We list our results for some of these problems below, all of which are major topics of textbooks—see [11, 25, 37, 38, 41] on the extensive history and on the previously "optimal" algorithms. (See Section 6 for more applications.)

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

*School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (tmchan@uwaterloo.ca). This work is supported by NSERC.

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

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 (just like in [27]).

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

The rest of the paper is organized as follows. Section 2 provides more background by briefly reviewing some of relevant previous work. Section 3 (and specifically Section 3.2) represents the heart of the paper and explores point location among disjoint line segments spanning a slab: this turns out to be the key subproblem, and we obtain a sublogarithmic solution by rethinking (actually simplifying) Fredman and Willard's original fusion tree and incorporating nontrivial new ideas for its 2-d extension; the resulting data structure remains simple and self-contained—its description is under two pages long. In Sections 4–6, we apply this new data structure to derive new results for general 2-d point location and other well-known geometric problems: this part of the paper is perhaps more routine and involves handpicking the right known techniques from computational geometry. Extensions and applications in higher dimensions are also mentioned. We conclude with comments in Section 7.

Added note. Independently, Mihai Pătraşcu (in these proceedings) has obtained an $O(\sqrt{\log U})$ result for 2-d point location (but not our $O(\log n/\log\log n)$ result, nor the extensions and applications to other geometric problems). His solution of the slab subproblem is similar to one of our solutions in Section 3.3, but he proposes a different method, based on persistent and exponential search trees, to reduce general point location to the slab subproblem, instead of our sampling-based or separator-based method in Section 4. (Unlike our separator-based method, the preprocessing time of his method is not linear.)

Shortly after exchanging papers, Pătraşcu has noticed that both his method and our method can lead to an $O(\sqrt{\log U/\log\log U})$ bound for general 2-d point location—in his case, by just invoking our better slab result as a subroutine, and in our case, by just setting parameters of the recurrences a little more carefully, without changing the method itself. We have added this small improvement in Section 4 and incorporated it into the statements of all subsequent results in Sections 5–6.

2. Related Work

In 1-d. In addition to the work already mentioned, many integer-sorting results have been published; currently, the best linear-space deterministic and randomized algorithms (independent of U and w) have running time $O(n\log\log n)$ and $O(n\sqrt{\log\log n})$ respectively, due to Han [29] and Han and Thorup [30]. A linear randomized time bound [6] is also known for $w \geq \log^{2+\varepsilon} n$ for any fixed $\varepsilon > 0$.

For the problem of maintaining a data structure for *successor search* (finding the smallest element greater than a query value), van Emde Boas trees [47, 48] yield $O(\log\log U)$ expected update and query time with linear space, and Fredman and Willard's fusion trees yield an $O(\log n/\log\log n)$ deterministic and $O(\sqrt{\log n})$ randomized time bound with linear space. In the static case, Beame and Fich [10] improved the query time to $O(\min\{\sqrt{\log n/\log\log n}, \log\log U/\log\log\log U\})$, with $n^{O(1)}$ space; even more importantly, they proved a matching lower bound in a cell probe model. (See [40] for further tradeoff results.) Combined with Andersson's *exponential search trees* [5, 8], this also leads to the currently best result for dynamic successor search: an update/query bound of $O(\min\{\sqrt{\log n/\log\log n}, \log\log U\log\log n/\log\log\log U, \log n/\log w + \log\log n\})$, with linear space.

Other 1-d data structure problems for integer input have also been studied. The classic problem of designing a dictionary to answer *membership queries* can be solved in $O(1)$ deterministic time statically, with linear space, and $O(1)$ expected time dynamically, by hashing. There are also known data structures for other related problems like 1-d range searching [1] and priority queues [46].

In 2-d and beyond. As mentioned, known algorithms from the computational geometry literature that exploit the power of the word RAM mostly deal with rectilinear 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 [12, 33, 34, 39] for a sample) are about van-Emde-Boas-type results, with only a few exceptions (e.g., [49, 36]). For instance, Karlsson [34] obtained an $O(n\log\log U)$-time algorithm for the L_∞-Voronoi diagram in 2-d (Chew and Fortune [22] later showed how to construct the Voronoi diagram under any fixed convex polygonal metric in 2-d in $O(n\log\log n)$ time after sorting the points along a fixed number of directions). De Berg *et al.* [12] gave $O((\log\log U)^{O(1)})$ 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); e.g., see [13] for an $O(n \log \log n)$-time 2-d approximate Euclidean minimum spanning tree algorithm.

There is one notable non-rectilinear 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 [42] solved the problem in $O(n)$ expected time (using hashing). Deterministically, the author [19] has given a reduction from closest pair to sorting (using one nonstandard but AC^0 operation on the RAM). This implies an $O(n \log \log n)$ deterministic time bound by Han's result [29], and for the special case of points from a polynomial-size grid, an $O(n)$ deterministic bound by radix-sorting. Similarly, the dynamic closest pair problem and (static or dynamic) approximate nearest neighbor queries reduce to successor search [19] (see also [3, 15] 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 [17], and an $O(nk)$-time randomized algorithm for finding the smallest circle enclosing k points in 2-d [31]. For the asymptotically tightest possible grid, i.e., $U = O(n^{1/d})$, the *discrete Voronoi diagram* [14, 20] can be constructed in linear time and can be used to solve static nearest neighbor problems.

The 2-d convex hull problem is another exception, due to its simplicity: Graham's scan [11, 41] 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 [43].)

Chazelle [21] 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 (how to adapt Beame and Fich's techniques [10]). (Un)fortunately, he did not address upper bounds for more challenging variants like intersecting a convex polygon with a query line (see Section 6, item 8).

In recent years, interest seems to have developed in the quest for faster integer-RAM algorithms for the core geometric problems like the standard Voronoi diagram and planar point location,[1] but attempts by researchers so far have apparently failed to materialize.

In the remainder of the paper, we assume without loss of generality that $U = 2^w$ (by shrinking the word size w). Henceforth, w is equated with $\log U$.

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 1(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 [47, 48] and Andersson's exponential search trees [5] 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 [27] relies on "compression" of the input numbers and query point (i.e.,

[1] For example, Jonathan Shewchuk (2005) in a blog comment wondered about the possibility of computing Delaunay triangulations in $O(n)$ time; earlier at SODA'92, Willard [49] asked for an $o(n \log n)$ algorithm for Voronoi diagrams. Demaine and Iacono (2003) in lecture notes (and more recently [9]) asked for an $o(\log n)$ method for 2-d point location.

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 author's 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 readers, since better 1-d data structures are known). 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, where the numbers are assumed to be integers in $[0, 2^w)$. Our main idea is very simple and is encapsulated in the observation below—roughly speaking, in divide-and-conquer, allow progress to be made not only by reducing the number of elements, n, but alternatively by reducing the length of the enclosing interval, i.e., reducing the number of required bits, ℓ. (Beame and Fich [10] adopted a similar philosophy in the design of their data structure, though, in a much more intricate way.)

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*

(i) *each subinterval contains at most n/b elements of S* or *has length $2^{\ell-h}$; and*

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

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 1(b)). Note that any "gap" between two such consecutive grid subintervals do not contain elements of B and so can contain $\leq n/b$ elements. □

The data structure. The observation suggests a simple tree structure for 1-d successor search. Because of (ii), 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 $\ell < 0$). Initially, $\ell = w$. Because of (i), 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 answer is not there, it must be the first element to the right of the subinterval; this element can be stored during preprocessing.) By choosing $b = \lfloor \sqrt{\log n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\log n / \log\log n)$.

Implementing the word operation. We have assumed above that the subinterval containing q can be found in constant time, given $O(b)$ subintervals satisfying (ii), 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 (ii), by translation and scaling (namely, dividing by $2^{\ell-h}$), 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 [27, 7], 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 $\langle 1 \,|\, 1 \,|\, \cdots \rangle$ to get the word $\langle \tilde{q} \,|\, \tilde{q} \,|\, \cdots \rangle$. 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 $\langle (2^h + \tilde{a}_1 - \tilde{q}) \,\&\, 2^h \,|\, (2^h + \tilde{a}_2 - \tilde{q}) \,\&\, 2^h \,|\, \cdots \rangle$. 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 [27].

3.2. A New 2-d Fusion Tree

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

Observation 3.2 *Fix b and h. Let S be a set of n disjoint segments, where all left endpoints lie on an interval I of length 2^ℓ on a vertical line, and all right endpoints lie on an interval J of length 2^m on another vertical line. We can find $O(b)$ segments $s_0, s_1, \ldots \in S$ in sorted order, which include the lowest and highest segment of S, such that*

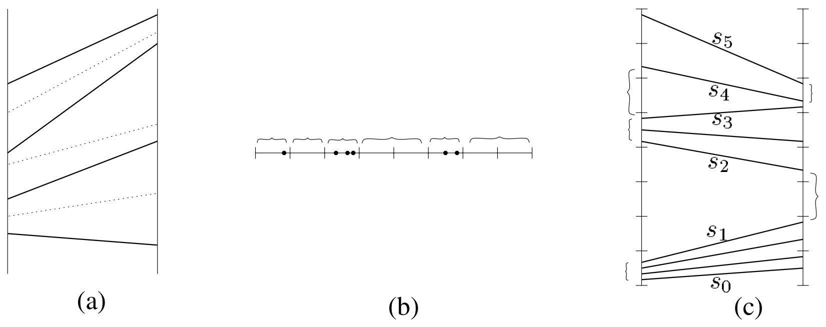

Figure 1. (a) The rounding idea: locating among the solid segments reduces to locating among the dotted segments. (b) Proof of Observation 3.1: elements of B are shown as dots. (c) Proof of Observation 3.2: segments of B are shown, together with the constructed sequence $s_0, s_1, \ldots$

(i) *for each i, there are at most n/b segments of S between s_i and s_{i+1},* or *the left endpoints of s_i and s_{i+1} lie on a subinterval of length $2^{\ell-h}$,* or *the right endpoints of s_i and s_{i+1} lie on a subinterval of length 2^{m-h}; and*

(ii) *there exist segments $\tilde{s}_0, \tilde{s}_2, \ldots$, with $s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec \cdots$ and endpoints on I and J, such that distances between left endpoints of the $\tilde{s}_i$'s are all multiples of $2^{\ell-h}$ and distances between right endpoints are all multiples of 2^{m-h}.*

Proof: Form a grid over I consisting of 2^h subintervals of length $2^{\ell-h}$, and a grid over J consisting of 2^h subintervals of length 2^{m-h}. Let B contain the $(\lfloor in/b \rfloor)$-th lowest segment of S for all $i = 1, \ldots, b$. Set s_0 to be the lowest segment. For $i = 0, 2, 4 \ldots$ (until the highest segment is reached), set s_{i+1} to be the highest segment of B such that the left endpoints of s_i and s_{i+1} are in the same grid subinterval *or* the right endpoints of s_i and s_{i+1} are in the same grid subinterval. Set s_{i+2} to be the successor of s_{i+1} in B. (See Figure 1(c).) Since the left endpoints of s_i and s_{i+2} are in different grid subintervals *and* the right endpoints of s_i and s_{i+2} are in different grid subintervals, we can round s_i to a new segment $\tilde{s}_i$ to satisfy (ii). □

The data structure. Because of (ii), 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 $\ell < 0$ or $m < 0$). Initially, $\ell = m = w$. Because of (i), in each subproblem, n is decreased by a factor of b, *or* ℓ is decreased by h, *or* m is decreased by h. Thus, the height of the tree is at most $\log_b n + 2w/h = O(\log_b n + b)$.

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 $b = \lfloor \sqrt{\log n} \rfloor$, for instance, we get a query time of $O(\log_b n + b) = O(\log n / \log\log n)$.

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

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

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

(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 $\langle z_1 \mid z_2 \mid \cdots \rangle$ 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 $\langle \tilde{a}_0 \mid \tilde{a}_2 \mid \cdots \rangle$ and $\langle \tilde{b}_0 \mid \tilde{b}_2 \mid \cdots \rangle$ 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 $\langle \tilde{a}_0(2^h - \tilde{x}) + \tilde{b}_0\tilde{x} \mid \tilde{a}_2(2^h - \tilde{x}) + \tilde{b}_2\tilde{x} \mid \cdots \rangle$. We want to compute the rank of $2^h\tilde{y}$ among the values encoded in the blocks of this word. This subproblem was solved before [27] (as reviewed in Section 3.1).

3.3. Variants

A larger-space data structure. We now present results that are sensitive to w. We first describe a variant of our 2-d fusion tree that is even simpler and provides a time-space tradeoff. Namely, we just apply Observation 3.2 recursively, this time, with $b = n$ (i.e., we care only about reducing ℓ and m, not n). The height of the resulting tree is now at most $2w/h$.

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 $\tilde{q}$ that has h-bit integer coordinates. The idea is to precompute the answers for all 2^{2h} such points during preprocessing in $O(2^{2h})$ time, for example, by 2^h scans along each vertical grid line, and subsequently answer a query by table lookup. The total extra cost for the table precomputation is $O(2^{2h}n)$. We immediately obtain $O((w/h + 2^{2h})n)$ preprocessing time (after initial sorting), $O(2^{2h}n)$ space, and $O(w/h)$ query time, for any given h.

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

The preprocessing time excluding initial sorting is $O(n + (w/h + 2^{2h})r)$, the space requirement is $O(n + 2^{2h}r)$, and the query time is $O(\log(n/r) + w/h)$. Setting $r = \lfloor n/(w/h + 2^{2h}) \rfloor$ leads to $O(n)$ preprocessing time and space and $O(h + w/h)$ 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 the data structure from Section 3.2 to store each subset of $O(n/r)$ elements. The query time becomes $O(\log(n/r)/\log\log(n/r) + w/h) = O(h/\log h + w/h)$. Setting $h = \lfloor \sqrt{w \log w} \rfloor$ instead yields a query time of $O(\sqrt{w/\log w})$.

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 easily adapted for disjoint open segments that may share endpoints.

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.

The result of the previous section naively yields an $O(n^2)$-space data structure with $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\})$ query time: Divide the plane into $O(n)$ slabs through the x-coordinates of the endpoints and build our 2-d fusion tree inside each slab (note that the endpoints of the segments clipped to the slab indeed are 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. The preprocessing time is $o(n^2 \log n)$.

We can improve the preprocessing time and space by applying known computational-geometric techniques for point location, for example, using a b-ary version of the *segment tree* or the *trapezoid method* [11, 41, 44]. To get linear space, though, we adopt a *random sampling* method [37] or a *planar separator* method [35]. The former is simpler to implement, but the latter is deterministic and also has linear preprocessing time for connected subdivisions. Due to space limitation, we will only describe the sampling method here; see the full paper for more on the separator method.

Reducing space by sampling... 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 $O(r \log r)$ time. We store $T(R)$ in a point-location data structure, with $P_0(r)$ preprocessing time, $S_0(r)$ space, and $Q_0(r)$ query time.

For each segment $s \in S$, we first find the trapezoid of $T(R)$ containing the left endpoint of s in $Q_0(r)$ time. By a walk in $T(R)$, we can then find all trapezoids of $T(R)$ that intersects s in time linear in the number of such trapezoids (note that s does not intersect any segment of R and can only cross vertical walls of $T(R)$). As a result, for each trapezoid $\Delta \in T(R)$, we obtain the subset S_Δ of all segments of S intersecting Δ (the 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 [24, 37],

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

holds with probability 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 $n_i = O((n/r)\log r)$:

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

We have already described a method with $P_0(r) = o(r^2 \log r)$, $S_0(r) = O(r^2)$, and $Q_0(r) = O(\log r/\log\log r)$. By setting $r = \lfloor\sqrt{n}\rfloor$, the above recurrence solves to $P(n), S(n) = O(n \log^{O(1)} n)$ and $Q(n) = O(\log n/\log\log n)$.

...and resampling. To reduce space further, we bootstrap using the new bounds $P_0(r), S_0(r) = O(r \log^c r)$ and $Q_0(r) = O(\log r/\log\log r)$ for some constant c. This time, we replace recursion by directly invoking a known planar point location method [44] with $P_1(n) = O(n \log n)$ preprocessing time, $S_1(n) = O(n)$ space, and $Q_1(n) = O(\log n)$ query time. We then obtain the following bounds, where $\sum_i n_i = O(n)$ and $n_i = O((n/r)\log r)$:

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

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

Variant. Alternatively, to get a result sensitive to w, we can use the larger-space variant from Section 3.3 and start with the bounds $P_0(r) = o((w/h + 2^{2h})r^2 \log r)$, $S_0(r) = O(2^{2h}r^2)$, and $Q_0(r) = O(w/h)$. We set $r = \lfloor n^{1/4}\rfloor$ and $h = \lfloor\varepsilon \log n\rfloor$ for a sufficiently small constant $\varepsilon > 0$ (so that $2^{2h}r^2 \log r = o(n)$).[2] The recurrences become

$$\begin{aligned} P(n) &= \textstyle\sum_i P(n_i) + O(nw/\log n) \\ S(n) &= \textstyle\sum_i S(n_i) + o(n) \\ Q(n) &= \max_i Q(n_i) + O(w/\log n), \end{aligned}$$

where $\sum_i n_i = O(n)$ and $n_i = O(n^{3/4}\log n)$. We stop the recursion when $n \le n_0$ and handle the base case using our earlier method with $O(n_0 \log^{O(1)} n_0)$ expected preprocessing time, $O(n_0)$ space, and $O(\log n_0/\log\log n_0)$ query time. As a result, the recurrences solve to $P(n) = O(nw\log^{O(1)} n)$, $S(n) = O(n\log^{O(1)} n)$, and $Q(n) = O(\log n_0/\log\log n_0 + w/\log n_0)$ (because the $Q(\cdot)$ recurrence expands to a geometric series). Setting $n_0 = 2^{\sqrt{w \log w}}$ yields $Q(n) = O(\sqrt{w/\log w})$.

Now, substituting $P_0(r) = O(rw\log^c r)$, $S_0(r) = O(r\log^c r)$, $Q_0(r) = O(\sqrt{w/\log w})$, $P_1(n) = O(n\log n)$, $S_1(n) = O(n)$, $Q_1(n) = O(\log n)$, and $r = \lfloor n/(w\log^c n)\rfloor$ yields $O(n\sqrt{w/\log w})$ expected preprocessing, $O(n)$ space, and $O(\sqrt{w/\log w})$ query time.

5. 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 $O(r\log r)$ time by a known algorithm

[2]This choice of h is based on a suggestion by Mihai Pătraşcu. Originally, we used a fixed value of h during recursion, which resulted in a slightly larger query bound of $O(\sqrt{w})$ instead of $O(\sqrt{w/\log w})$.

[11, 41]. The xy-projection of the faces of the upper hull is a triangulation; store the triangulation in our point-location data structure.

For each point $s \in S$, consider the dual plane s^* [11, 25, 37]. Constructing the upper hull is equivalent to constructing the lower envelope of the dual planes. Let $T(R)$ denote a *canonical triangulation* [23, 37] of the lower envelope LE(R) of the dual planes of R, which can be computed in $O(r)$ time given LE(R). For each $s \in S$, we first find a vertex of the 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 $O(n \min\{\log r/ \log\log r, \sqrt{w/\log w}\} + \sum_{\Delta \in T(R)} |S^*_\Delta|)$. We then construct LE($S^*_\Delta$) inside S^*_Δ, by using a known $O(|S^*_\Delta| \log |S^*_\Delta|)$-time convex-hull/lower-envelope algorithm. We finally stitch these lower envelopes together to obtain the overall lower envelope/convex hull.

By a standard analysis of Clarkson and Shor [24, 37], $E\left[\sum_{\Delta \in T(R)} |S^*_\Delta|\right] = O(n)$ and $E\left[\sum_{\Delta \in T(R)} |S^*_\Delta| \log |S^*_\Delta|\right] = O(r \cdot (n/r) \log(n/r))$. The total expected running time is $O(r \log r + n \min\{\log r/ \log\log r, \sqrt{w/\log w}\} + n \log(n/r))$. Setting $r = \lfloor n/ \log n \rfloor$ yields an $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$-time randomized algorithm.

6. 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, assuming that all input coordinates/coefficients are $O(w)$-bit integers or rationals.

1. We can construct the convex hull of n points in 3-d in $O(n \log H/ \log\log H)$ expected time, where H is the number of hull vertices. This follows by using the convex hull algorithm from Section 5 within the output-sensitive algorithm from [16].

2. We can report all k intersections among a set of n line segments in 2-d in $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\} + k)$ expected time, where k is the number of intersecting pairs. Moreover, we can construct the *trapezoidal decomposition* in the same amount of time. This follows by modifying the preprocessing algorithm in Section 4 to work for non-disjoint segments. (Since the details are similar, we defer them to the full version of the paper.)

3. We can compute a triangulation of an n-vertex polygon with holes, in expected time $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$. This follows immediately from the fact that a triangulation can be constructed from the trapezoidal decomposition of the edges in linear time.

4. We can construct the *Voronoi diagram*, or equivalently the *Delaunay triangulation*, of n points in the plane in $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ expected time. This follows immediately because the Delaunay triangulation reduces to the convex hull of a 3-d point set via a lifting map (where coordinates still have $O(w)$ bits).

5. We can find the largest empty circle inside a rectangle for a set of n points in the plane in $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ expected time. This follows from the fact that the optimal circle can be determined from the Voronoi diagram in linear time.

6. We can preprocess n points in the plane, in $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ expected time and $O(n)$ space, so that *nearest/farthest neighbor queries* under the Euclidean metric can be answered in $O(\min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ time. This follows because these queries reduce to point location in the Voronoi diagram (whose coordinates have $O(w)$-bit numerators/denominators).

7. We can construct the *Euclidean minimum spanning tree* (MST) of n points in the plane in $O(n \min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ expected time. This follows because the MST is contained in the Delaunay triangulation and the MST of a planar graph can be computed in linear time, e.g., by Borůvka's algorithm.

8. We can preprocess a convex polygon P with n vertices in $O(n)$ space, so that *wrapping queries* (finding the left tangent from an exterior point) and *ray shooting queries* (intersecting P with a line) can be answered in $O(\min\{\log n/ \log\log n, \sqrt{w/\log w}\})$ time. For wrapping queries, this follows by decomposing the plane into regions (wedges) that have a common answer and performing a point location query. Ray shooting queries reduce to wrapping queries in the dual convex polygon (whose coordinates are still $O(w)$-bit rationals).

9. We can store n points in the plane in $O(n \log\log n)$ space, so that *circular range reporting queries* (finding the k points inside a query circle), and *"k nearest neighbors" queries*, can be answered in $O(\min\{\log n/ \log\log n, \sqrt{w/\log w}\} + k)$ time. This follows by combining our point location result with a data structure from [18].

Remark on higher dimensions. For any fixed dimension $d > 2$, the slab data structure from Section 3 can be generalized to handle point location for disjoint simplices in a vertical prism, in $O(n)$ space and $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\})$ query time, by using an appropriate analog of Observation 3.2. As a result, we can solve the point location problem for any subdivision of $\mathbb{R}^d$ into polyhedral cells, with $O(w)$-bit integer/rational coordinates, naively with $n^{O(1)}$ space and preprocessing time and $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\})$ query time. We can again use sampling, as in Section 4, to reduce space. For example, we can perform point location in an arrangement of n hyperplanes in $\mathbb{R}^d$ with $O(w)$-bit rational coefficients, in $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\})$ time with $O(n^d \log^{O(1)} n)$ space; we can answer (exact) nearest neighbor queries in $\mathbb{R}^d$ in $O(\min\{\log n/\log\log n, \sqrt{w/\log w}\})$ time with $O(n^{\lceil d/2\rceil} \log^{O(1)} n)$ space [23]. Many geometric search problems (even those involving curved objects) can be reduced to point location in higher dimensions (by the technique of linearization); this leads to many more applications, such as ray shooting and motion planning. See the full paper for more details.

7. 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, for good reasons, 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 and, as some might argue, is perhaps more realistic. (In practice, we do not have infinite precision as assumed by the real RAM.) In fact, discussion on robustness and the exact arithmetic paradigm in computational geometry often assumes w-bit integer or floating-point input and operations on w-bit words. (We leave open the issue of whether our results could be extended to floats.)

One possible complaint about our algorithms is their use of w-bit integer multiplication/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 ($w = O(\log n)$), multiplication/division on εw-bit words can be simulated by table lookup, so there is no dispute about the model in this case.

If it has not been made clear already, our study here is primarily theoretical. A $\log\log n$ factor improvement would not necessarily make a huge impact in practice, especially because of possibly larger hidden constant factors. (To be more optimistic, though, we mention that Andersson [4] did implement an $O(\log n/\log\log n)$ method for 1-d successor search, supposedly with promising results.)

The most pressing theoretical question is to determine the precise complexity of the subproblem of 2-d point location in a slab. Any improved upper bound would lead to better results for the other problems. Actually, for most of the offline applications, it suffices to solve the problem of answering a batch of n queries. It is not clear how this offline problem can be solved more efficiently (unlike in 1-d by sorting). Improvement for the special case $w = O(\log n)$ alone (say, $U = n$) would be interesting.

A lower bound strictly stronger than that for 1-d successor search [10] would even be more exciting. $\Omega(\log n/\log\log n)$ query-time lower bounds have been established in the cell probe model for some dynamic problems when update time is polylogarithmic [26, 2]. For example, Husfeldt *et al.* [32] have stated such a result for dynamic planar point location in monotone subdivisions. However, this lower bound does not apply to the static problem, and the hardness seems to stem from non-geometric issues namely, identifying (a label of) the face incident to an edge, not finding the edge immediately above a point.

Acknowledgement. I thank Mihai Pătraşcu for sharing his ideas on improving $O(\sqrt{w})$ bounds to $O(\sqrt{w/\log w})$.

References

[1] 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.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. A cell probe lower bound for dynamic nearest-neighbour searching. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 779–780, 1998.

[3] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for location and proximity problems. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 160–170, 1999.

[4] A. Andersson. Sublogarithmic searching without multiplications. In *Proc. 36th IEEE Sympos. Found. Comput. Sci.*, pages 655–663, 1995.

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

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

[7] 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.

[8] 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.

[9] I. Baran, E. D. Demaine, and M. Pătraşcu. Subquadratic algorithms for 3SUM. In *Proc. 9th Workshop Algorithms Data Struct.*, Lect. Notes Comput. Sci., vol. 3608, Springer-Verlag, pages 409–421, 2005.

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] T. M. Chan. Random sampling, halfspace range reporting, and construction of $(\leq k)$-levels in three dimensions. *SIAM J. Comput.*, 30:561–575, 2001.

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

[20] T. M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. *Proc. 20th ACM Sympos. Comput. Geom.*, pages 152–159, 2004.

[21] 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.

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

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

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

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

[26] M. L. Fredman and M. R. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st ACM Sympos. Theory Comput.*, pages 345–354, 1989.

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

[28] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Sys. Sci.*, 48:533–551, 1994.

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

[30] 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.

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

[32] 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.

[33] 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.

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

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

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

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

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

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

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

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

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

[43] 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.

[44] 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.

[45] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.

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

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

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

[49] 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.