Computational Geometry through the Information Lens

by

Mihai Pătraşcu

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author . . . .
Department of
Electrical Engineering and Computer Science
May 22, 2007

Certified by . . . .
Erik D. Demaine
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Computational Geometry through the Information Lens

by
Mihai Pătraşcu

Submitted to the Department of
Electrical Engineering and Computer Science
on May 22, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis revisits classic problems in computational geometry from the modern algorithmic perspective of exploiting the bounded precision of the input. In one dimension, this viewpoint has taken over as the standard model of computation, and has led to a powerful suite of techniques that constitute a mature field of research. In two or more dimensions, we have seen great success in understanding orthogonal problems, which decompose naturally into one dimensional problems.

However, problems of a nonorthogonal nature, the core of computational geometry, have remained uncracked for many years despite extensive effort. For example, Willard asked in SODA'92 for a $o(n \lg n)$ algorithm for Voronoi diagrams. Despite growing interest in the problem, it was not successfully solved until this thesis.

Formally, let w be the number of bits in a computer word, and consider n points with $O(w)$-bit rational coordinates. This thesis describes:

- a data structure for 2-d *point location* with $O(n)$ space, and $O\big(\min\big\{\frac{\lg n}{\lg \lg n}, \sqrt{\frac{w}{\lg w}}\big\}\big)$ query time.
- randomized algorithms with running time $n \cdot 2^{O(\sqrt{\lg \lg n})} < n \lg^{o(1)} n$ for 3-d *convex hull*, 2-d *Voronoi diagram*, 2-d *line segment intersection*, and a variety of related problems.
- a data structure for 2-d *dynamic convex hull*, with $O(\frac{\lg n}{\lg \lg n})$ query time, and $O(\lg^2 n)$ update time.

More generally, this thesis develops a suite of techniques for exploiting bounded precision in geometric problems, hopefully laying the foundations for a rejuvenated research direction.

Thesis Supervisor: Erik D. Demaine
Title: Associate Professor

Acknowledgments

Erik Demaine has been my research adviser for 4 years and a half, throughout my undergraduate and graduate work. I am profoundly grateful to him for the unconditional support that he has provided throughout these years. My work in theoretical computer science started with Erik's willingness to trust and guide a freshman with great dreams, but no real idea of what this research field meant. Throughout the years, Erik's understanding and tolerance for my unorthodox style, including in the creation of this thesis, have provided the best possible research environment for me.

This thesis is based primarily on three publications, in FOCS'06 [Pǎt06], STOC'07 [CP07], and SoCG'07 [DP07]. From my point of view, this line of work started during the *First Machu Picchu Workshop on Data Structures*. Though I shall refrain from discussing details of this event, I am profoundly grateful to John Iacono, as the main organizer, and again to Erik for supporting such research endeavors. My interactions with John during this workshop and beyond have been most gratifying. In particular, it was John who sparked my interest in sublogarithmic point location. Furthermore, his support, through research discussions and otherwise, was key to me finding a solution during the workshop.

Through a remarkable coincidence, my paper in FOCS'06 on point location was accompanied by an independent submission by Timothy Chan [Cha06] with similar results. This has lead to subsequent collaboration with Timothy, and in particular, our joint STOC'07 paper. Collaborating with Timothy has been a flawless experience, and this thesis owes a lot to his remarkable research prowess. Finally, I am once more grateful to Erik for an enjoyable and engaging collaboration on our joint SoCG'07 paper.

Beyond research, I wish to thank my friends and colleagues, among whom Alex Andoni stands out, for their constant support, good company, and the great time spent together. Last but not least, I am grateful to Akamai for sponsoring the Akamai Presidential Fellowship that supported me this year.

Contents

Chapter 1

Introduction

Sorting 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. In practice, radix sort is one of the most popular and most efficient sorting algorithms. In theory, the field of integer search provides a fascinating understanding of how information can be manipulated algorithmically, and the consequences this has on problem complexity.

Exploiting bounded precision has also been considered frequently for geometric problems, but up to now, results are essentially limited to problems about axis-parallel objects or metrics, which can be decomposed into one-dimensional problems. The bulk of computational geometry deals with non-orthogonal things (lines of arbitrary slopes, the Euclidean metric, etc.) and thus has largely remained a passive observer of the breakthroughs on integer sorting and searching. This shortcoming was highlighted by Willard in SODA'92 [Wil00], and repeatedly since then.

This thesis answers Willard's 15-year old challenge, and makes computational geometry an active player in the world of bounded precision algorithms. We provide the first $o(n \lg n)$ running times for core algorithmic problems in geometry, such as constructing Voronoi diagrams, 3-d convex hull, segment intersection, etc. We also provide the first $o(\lg n)$ query times for planar point location, 2-d nearest nearest neighbor, dynamic convex hull, etc.

More than providing an answer to an old question, our results open the door to a whole new playing field where most geometric problems do not yet have optimal solutions. At the heart of this research direction is the attempt to elucidate the fundamental ways in which bounded information about geometric objects such as points and lines can be decomposed in algorithmically useful ways. In computational geometry, this leads to many fascinating puzzles and a deeper understanding of the relative difficulty of geometric problems. In the world of RAM algorithms, our work had to develop significantly novel approaches for exploiting fixed precision. By analogy with the successes in one dimension, one can hope that an equally rich theory for multidimensional problems is waiting to be explored.

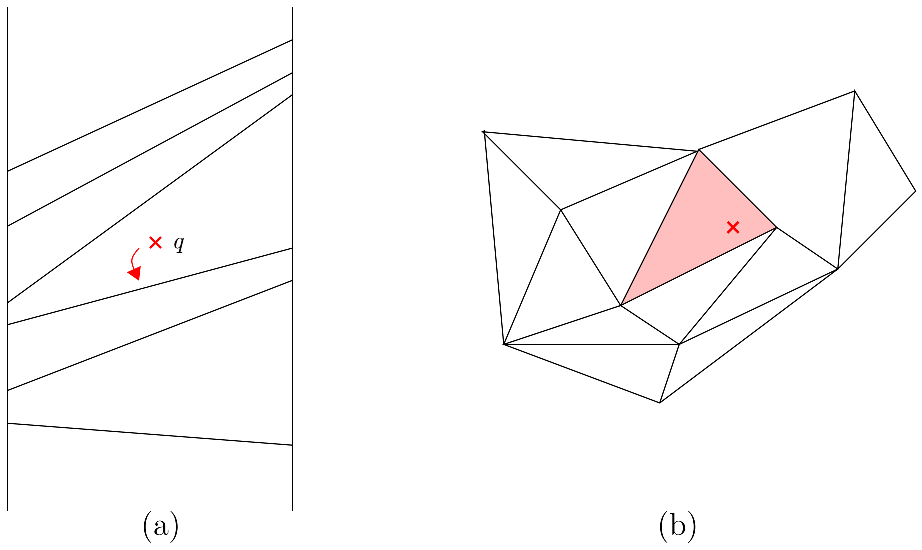

Figure 1-1: (a) Point location in a slab. (b) General point location.

1.1 Technical Overview

Model of computation. The underlying model of computation in finite-precision results is a Random Access Machine (RAM) that supports standard operations on w-bit words with unit cost. The supported operations are those available in a typical language such as `C`: additions, subtractions, multiplications, shifts, and bitwise operations. To make the model realistic, one assumes $w = \Theta(\lg U)$, i.e. the task of the algorithm is to handle numbers represented in one (or maybe $O(1)$) machine words. One can also reasonably assume that $w \geq \lg n$, so that we can have n distinct numbers, and an index or pointer can fit in a word. The adjective "transdichotomous" is often associated with this model of computation. These assumptions fit the reality of computing as it is understood and deployed today, including in common programming languages such as `C`, and standard programming practice.

Searching in 2-d. In the one-dimensional world, *predecessor search* is one of the most fundamental and well-studied problems. Given a set S of n integers in $[0, 2^w)$, the problem is to build a data structure supporting the following query: for some $q \in [0, 2^w)$, report $\max\{x \in S \mid x \leq q\}$.

Point location in a slab is perhaps the simplest toy problem which extends predecessor search to two dimensions, and captures the nonorthogonal behavior that we want to study. Refer to Figure 1-1(a). In this problem, we are given a set S of n disjoint (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 $[0, 2^w)$. The goal is to preprocess S so that given a query point q with integer coordinates inside the slab, we can quickly find the segment that is immediately below q.

If we simply store the sorted segments, the problem is immediately solved in $O(\lg n)$ query time by binary search. In Chapter 2, we describe the first data structure which can beat this simple bound. Specifically, we describe a data structure of $O(n)$ space, supporting queries in time $O\big(\min\big\{\frac{\lg n}{\lg\lg n}, \sqrt{\frac{w}{\lg w}}\big\}\big)$. This is a theoretical improvement to a sublogarithmic query time for any precision w, and a roughly quadratic improvement when the input comes from a polynomial universe ($w = O(\lg n)$).

Sorting in 2-d. The offline version of predecessor search is the problem of sorting n numbers. In 2-d, the offline version of our toy problem is an equally natural incarnation of our intuitive notion of ordering. We are given a vertical slab in the plane, m nonintersecting segments cutting across the slab, and n points inside the slab. The goal is to identify for each of the n points, the segments immediately below and above it. In other words, we would like to sort the points "relative to" the segments.

By running n queries to the online data structure, we already have a solution beating the standard $O(m+n\lg m)$. In Chapter 3, however, we provide a more efficient algorithm beating the standard bound more dramatically: we obtain a running time of $n \cdot 2^{O(\sqrt{\lg\lg m})} + O(m)$. Note that this bound does not depend on the universe at all, and it grows more slowly than $m + n\lg^{\varepsilon} m$ for any constant $\varepsilon > 0$.

Planar point location. In this problem, we are given n segments, dividing the plane into polygons; segments are only allowed to touch at end-points. The query asks for a polygon (face) which contains a given point. See Figure 1-1(b).

Point location is one of the most fundamental and well-studied search problems in computational geometry. Every introductory book in computational geometry discusses the problem at length, and the problem is almost surely the topic of hundreds of publications. There are also important applications in the offline case (such as finite-element simulations with Euler-Lagrange coupling, and overlay problems in GIS and CAD systems).

It turns out the one can reduce the problem to the slab problem discussed above, both in the online and offline case, while maintaining the same space and running times. This is the topic of Chapter 4. We thus obtain the first sublogarithmic bounds for point location, an important theoretical milestone.

In fact, we describe three independent reductions: two of them are adaptations of classic techniques in computational geometry, while the third requires significant theoretical development, and is of independent interest.

Voronoi diagrams, segment intersection etc. The "toy" problem that we study captures core aspects of nonorthogonality, and our sublogarithmic bounds have far-reaching consequences across computational geometry. In Chapter 5, we describe reductions leading to improved bounds for a wide array of problems, both algorithmic and data structural.

In particular, we obtain running times of $n \cdot 2^{O(\sqrt{\lg\lg n})}$ for problems like constructing Voronoi diagrams, 3-d convex hulls, segment intersection etc. These are the first results improving on classic $O(n\lg n)$ bounds for these bread-and-butter problems in computational geometry.

It turns out that our techniques for improved point location can be extended to higher dimensions, and they have a number of applications there, as well. Appendix A discusses results in higher dimensions.

Dynamic convex hull. So far, we have only discussed static data structures and algorithms. Dynamic problems bring an entirely different, but very interesting, set of challenges, addressed in Chapter 6. We study the *dynamic convex hull* problem, as the most important example of a dynamic nonorthogonal problem.

The problem is to maintain a set S, $|S| \leq n$, of points in 2-d under:

INSERT(p): insert a point p into S. Points in S are not necessarily in convex position.

DELETE(s): delete a point p from S.

TANGENT(p): return the two tangents through point p to the convex hull of S. It is guaranteed that p is outside the convex hull.

LP(v): return an extreme point in the direction given by vector v (linear programming).

As our main result in the dynamic world, we show how to support queries in $O(\frac{\lg n}{\lg\lg n})$ time, while supporting updates in $O(\lg^2 n)$. The highlight of this result is the unusual flavor of the dynamic context.

In fact, tangent and linear programming queries are only two examples of queries we may wish to support. In Appendix B, we discuss the various queries that are typically considered, and prove additional upper and lower bounds. In particular, we show tangent and linear programming queries require time $\Omega(\log_w n)$ when the update time is polylog(n). This proves optimality of our upper bound for precision $w = \text{polylog}(n)$.

1.2 Motivation

On a global level, our results open more problems than they close. We are now in front of a whole new playing field where most geometric problems do not yet have optimal solutions. By analogy with one dimension, one can hope for an equally rich theory for multidimensional problems. But before proceeding through these open gates, we need to make sure that we are going down a meaningful and worthy path. In the following sections, we discuss several of our motivations for considering geometric problems in the bounded-precision context:

1.2.1 Real computers and therefore their inputs have bounded precision.

1.2.2 Bounded precision is already exploited in practice.

1.2.3 The fundamental problems under consideration have been studied and optimized extensively.

1.2.4 Exploiting bounded precision for these problems has been posed and attempted extensively.

1.2.5 New, faster algorithms may impact engineering practice.

1.2.6 The problems require the development of fascinating new techniques that force us to rethink our understanding of both bounded precision, and geometry.

1.2.1 Theory for Reality

A central issue in computational geometry is the discrepancy between the idealized *geometric view* of limited objects with infinite precision, and the realistic *computational view* that everything is represented by (finitely many) bits.

The geometric view is inspired by Euclidean geometric constructions from circa 300 BC. In computational geometry, this view is modeled by the real RAM and related models considered for lower bounds, e.g., linear/algebraic decision/computation trees. These models postulate a memory of infinite-precision cells, holding real values, and specify the operations that can be performed on these values (typically, a subset of addition, multiplication, division, roots, and comparisons). Inputs and, depending on the problem, outputs consist of real values.

The computational view matches the reality of digital computers as we know them today and as set forth by Turing in 1936 [Tur36]. This view assumes input is given with some finite precision, and that memory cells (words) have a precision comparable to the input. The preferred model for upper bounds is the word RAM, which allows operations commonly available in a programming language such as C, including bitwise operations. Lower bounds are usually shown in the cell-probe model, which allows *any* operation on a constant number of words. Thus, lower bounds have a deep information-theoretic meaning, and apply regardless of the exotic operations that might be implemented on some machine.

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

Given that algorithms must eventually be implemented in a bounded-precision model, it seems only natural not to tie our theoretical hands by using the real RAM. By recognizing that actual input data has bounded precision, and by designing algorithms for the word RAM, one could potentially obtain significantly better bounds. This ability is demonstrated for some key problems by our work. In addition, this approach has the advantage that it does not hide a large implementation cost by idealizing the model, and therefore has the potential to lead more directly to fast practical algorithms.

A question that we wish to touch on briefly is whether an integer (or rational) 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-invariant, 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.

1.2.2 Theory for Practice

When scrutinizing a direction of theoretical research, it is important to understand not only whether it studies an aspect of reality, but also whether it studies an *interesting* aspect of reality. After all, the greatest payoff for a real application is often achieved by considering an appropriate abstraction of the object of study, not all the real but irrelevant details.

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., [HT02]. 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, implementing search by comparisons is inconceivable [DBCP97].

In nonorthogonal geometry, the object of our study, there are at least two classes of examples showing the benefit of using bounded precision.

First, as discussed in a survey by Snoeyink [Sno04], the most efficient and popular approaches for planar point location use pruning heuristics on the grid. These heuristics are similar in spirit to the point-location algorithms we develop in this thesis. To some extent, our work justifies the advantage of these algorithms compared to the traditional approaches taking logarithmic time, which have been regarded as optimal.

Second, there is extensive study and implementation of the approximate nearest neighbor problem, even in two dimensions; see, e.g., [AEIS01, Cha02]. This is hard to understand when viewed through the real-RAM abstraction, because the exact nearest neighbor problem should be equally hard, both taking logarithmic time. However, the approximate version turns out to be equivalent to one-dimensional predecessor search. When dealing with bounded precision, this problem admits an exponentially better solution compared to exact search in two dimensions — $O(\lg w)$ versus $O(\sqrt{w/\lg w})$.

Thus, some of our work so far can already be seen as a theoretical justification for practically proven approaches. Though giving a theoretical understanding of practical algorithms is not our foremost goal, it is a normal and important outcome of theoretical research, in line with a significant body of modern work.

We should note, however, that not all aspects of engineering practice are yet understood. For example, practical implementations of grid-based search typically work with the original instance of planar point location. By contrast, our theoretical analysis assumes that the input is first reduced to the slab case, which incurs a constant but practically significant overhead.

1.2.3 The Problems in Historical Context

The problems that we have studied are some of the most fundamental in computational geometry. Problems like planar point location or constructing Voronoi diagrams appear in virtually every textbook and are the topic of numerous surveys. Interest in these problems has not waned throughout the years, and new techniques have been developed continuously.

Taking planar point location as an example, we note that there are at least five fundamentally different ways to achieve $O(\lg n)$ query time with $O(n)$ space: planar separators [LT80], persistence [Col86, ST86], triangulation refinement [Kir83], separating chains plus fractional cascading [EGS86], and randomized incremental construction of a trapezoidal decomposition [Mul90]. Taking this bound even further, Seidel and Adamy [SA00] obtain a running time of *exactly* $\log_2 n + 2\sqrt{\log_2 n} + O(\sqrt[4]{\log_2 n})$ point–line comparisons, with expected linear space.

In recent years, there has been a lot of interest in obtaining adaptive (but still comparison-based) bounds for point location, which can sometimes be sublogarithmic. 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 2001 saw no less than three results in this direction: Arya et al. [AMM01b] and Iacono [Iac04] independently achieve expected $O(H)$ comparisons with $O(n)$ space, while Arya et al. [AMM01a] achieves $H+o(H)$ comparisons but with $O(n \lg^* n)$ space.

Historically, such directions have also been pursued intensively in one dimension (e.g., static and dynamic optimality). Some central problems in this field remain open. Despite this, almost two decades after Fredman and Willard's fusion trees [FW93], it appears that bounded precision safely occupies the center stage, and has grown into a much larger and more cohesive theory.

1.2.4 The Model in Historical Context

RAM Algorithms in 1-d

Work in one dimension dates back at least to 1977, when van Emde Boas [vEBKZ77] showed how to support predecessor queries in $O(\lg\lg U)$ time with linear space, and thus sort in $O(n \lg\lg U)$ time. Fredman and Willard [FW93] showed that $o(\lg n)$ searching and $o(n \lg n)$ sorting is possible even regardless of how U relates to n: their *fusion tree* can search in $O(\log_w n) \le O(\frac{\lg n}{\lg\lg n})$.

Many integer-sorting results have been published since then, and a survey is beyond the scope of this work. 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 [Han04] and Han and Thorup [HT02]. A linear randomized time bound [AHNR98] is known for the case when $w \ge \lg^{2+\varepsilon} n$, for any fixed $\varepsilon > 0$. Thorup [Tho02a] showed a black-box transformation from sorting to *priority queues*, which makes the above bounds carry over to this dynamic problem.

For predecessor search, note that taking the minimum of fusion trees and van Emde Boas search yields a bound of $O(\sqrt{\lg n})$. As opposed to sorting, only small improvements are possible, and only for polynomial space [BF02]. Together with Thorup [PT06, PT07], we showed optimal upper and lower bounds for this problem, giving an exact understanding of the time-space tradeoffs.

Most importantly, our 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 when precision is (very) large. We note that point location is harder than predecessor 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., [FKS84, DadH90]). *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 [ABR01]. Even for the dynamic problem, exponential improvements over predecessor search are known [MPP05].

(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 works are about van-Emde-Boas-type results, with only a few exceptions (e.g., [Wil00]). For instance, Chew and Fortune [CF97] 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. [dBvKS95] gave $O((\lg \lg U)^{O(1)})$ results for point location in an axis-parallel rectangular subdivisions in 2- and 3-d.

There are also *approximation* results (not surprisingly, since arbitrary directions can be approximated by a fixed number of directions); for example, see [BKRS92] 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 [Rab76] solved the problem in $O(n)$ expected time, using hashing. Deterministically, Chan [Cha02] has given a reduction from closest pair to sorting (using one nonstandard operation on the RAM). Similarly, the dynamic closest pair problem and (static or dynamic) approximate nearest neighbor queries reduce to predecessor search [Cha02]. Rabin's original approach itself has been generalized to obtain an $O(n + k)$-time randomized algorithm for finding k closest pairs [Cha01b], and an $O(nk)$-time randomized algorithm for finding the smallest circle enclosing k points in 2-d [HPM05].

The 2-d convex hull problem is another exception, due to its simplicity: Graham's scan [dBSvKO00, PS85] 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.

Chazelle [Cha99] 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 predecessor 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 5.1).

For the asymptotically tightest possible grid, i.e., $U = O(n^{1/d})$, the *discrete Voronoi diagram* [Cha04] can be constructed in linear time and can be used to solve static nearest neighbor problems.

Nonorthogonal Problems

The quest for faster word-RAM algorithms for the core geometric problems dates back at least to 1992, when Willard [Wil00] 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. [BDP05], 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. [AEIS01] or Iacono and Langerman [IL00]. 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.

1.2.5 Practice for Theory

Though we have not been able to test the practicality of the new ideas from this work, we believe they may have a practical impact for two reasons. First, as noted above, algorithms of the same flavor (e.g., gridding heuristics) are already used successfully in practice. Mathematically founded ideas for grid search could lead to better practical algorithms.

Second, for the algorithmic problems we have considered (e.g., constructing Voronoi diagrams), the theoretical improvement over classic solutions is rather significant — $o(n \lg^{\varepsilon} n)$ versus $O(n \lg n)$. In the similar case of sorting, it is widely appreciated that algorithms based on bounded precision (radix sort) outperform $O(n \lg n)$ sorting algorithms.

1.2.6 Theory for Theory

All things considered, it is perhaps the theoretical view of this emerging field that we find most compelling. It is a general phenomenon that tools developed in one dimension for exploiting bounded precision seem foreign to two or more dimensions, and simply do not help. As such, our results have had to develop interesting new techniques, rebuilding our understanding of bounded precision from scratch. By analogy with the one-dimensional case, a mature research field with many dozens of publications, we can hope that in the multidimensional case lurks a similarly rich theory.

A similar account can be made from the geometric side. Our work has forced us to re-analyze many “standard” ideas and techniques in geometry through a new, information-theoretic lens. This view of geometry seems interesting in its own right, and the algorithmic puzzles it raises are very appealing.

Stepping back for a moment, we believe that there are compelling reasons for our study even outside the realm of bounded precision. An overwhelming number of fundamental problems in computational geometry are far from understood even in classic models of computation. For example, in the real RAM, or even in the simpler group model, we do not know how to prove *any* lower bound exceeding $\Omega(\lg n)$ for a static data structure (and often not even that).

We believe progress even in these models requires a firm information-theoretic understanding of the problems. A good strategy seems to start developing such lower bound tools from the ground up, starting with smaller lower bounds for the problems where bounded precision helps. Though these lower bounds are asymptotically small, they are forced to be in the right spirit of understanding geometry through information. Starting with these computationally easier problems provides a natural programme for gradual development of increasing lower bounds.

Chapter 2

Point Location in a Slab

In this chapter, we study the special case of the 2-d point location problem in which the input consists of n disjoint, nonvertical line segments inside a vertical slab, where the endpoints all lie on the boundary of the slab. Given a query point q inside the slab, the goal is to quickly find the segment that is immediately above q. We will obtain the following result:

Theorem 2.1. *Consider a sorted list of n disjoint line segments spanning a vertical slab in the plane with $O(w)$-bit rational coordinates. For any $h \geq 1$, we can build a data structure with space and preprocessing time $O(n \cdot 2^h)$, so that point location queries take time:*

$$O\left(\min\left\{\lg n/\lg\lg n,\ \sqrt{w/\lg w},\ w/h\right\}\right)$$

2.1 Technical Overview

We begin with a few words to explain intuitively the difficulty of the problem, and the contribution of our solution. One of the most influential ideas in one-dimensional search is given by fusion trees [FW93]. This structure shows how to sketch $B = w^\varepsilon$ words, each w bits long, into a single word of w bits such that a predecessor query among the B numbers can be answered in constant time after seeing the sketch. Thus, to find the predecessor among n numbers, one simply builds a balanced B-tree with such a sketch at every node. A query walks down a root-to-leaf path in constant time per node.

For planar point location, we suspect that this type of sketching is impossible for any superconstant B, so a sublogarithmic query algorithm has to be based on a novel approach. To best understand our idea, we define an information theoretic measure for a subregion in a vertical slab. Refer to Figure 2-1. The key observation is that, while we may not be able to efficiently locate the query point among *any* B segments, we can do so as long as the region between segments does not have too small entropy.

More specifically, the search starts by assuming that the query point can lie anywhere in a slab, a region with high entropy. By appropriate sketching or tabulation techniques, each step of the algorithm reduces the entropy of

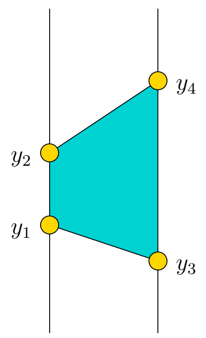

Figure 2-1: $H = \lg(y_2 - y_1) + \lg(y_4 - y_3)$

the region where the query could lie, by some fixed increment ΔH. The answer is certainly found when the entropy of the region becomes small enough, so we can bound the query time in terms of the entropy progress in each step. We note that information-progress arguments of this flavor are rather common in lower bounds, but generally their use in designing algorithms has not been widely appreciated.

We observe that this search strategy behaves quite differently from fusion trees. For instance, the $O(\log_w n)$ bound of fusion trees means the problem actually gets *easier* for very high precision. For point location, on the other hand, it is not known how to improve the $O(\lg n/\lg\lg n)$ bound for any high precision.

The remainder of this chapter is organized as follows. We start in Section 2.2 by describing a new fusion tree. Though this yields worse bounds than the regular fusion tree, the effort best describes the intuition about our search strategy. This section may be skipped by the impatient readers. The actual data structure for point location in a slab is presented in Section 2.3, with further time/space trade-offs described in Section 2.4.

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

We first re-solve the standard 1-d problem of performing predecessor 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 [BF02] adopted a similar philosophy in the design of their data structure, though in a rather different way.)

Observation 2.2. *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 $2^{\ell-h}$.*

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 2-2(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 predecessor 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

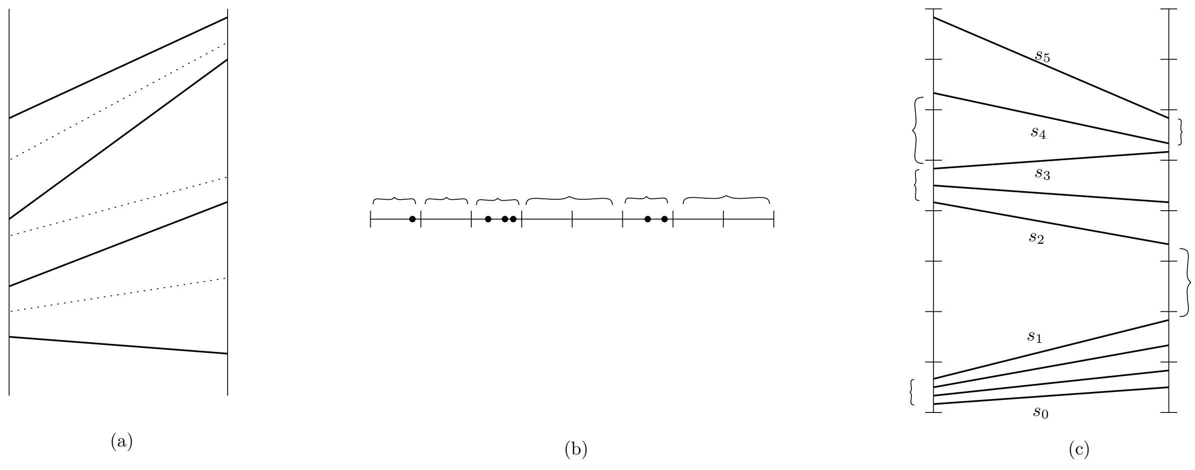

Figure 2-2: (a) The rounding idea: locating among the solid segments reduces to locating among the dotted segments. (b) Proof of Observation 2.2: elements of B are shown as dots. (c) Proof of Observation 2.3: segments of B are shown, together with the constructed sequence $s_0, s_1, \ldots$

subinterval. We stop the recursion when $n \le 1$ (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 $b = \lfloor\sqrt{\lg n}\rfloor$, 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 (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 $bh \le \varepsilon w$.

This subproblem was considered before [FW93, AMT99], 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 $\langle (2^h + \tilde{a}_1 -$

$\tilde{q}) \,\&\, 2^h \mid (2^h + \tilde{a}_2 - \tilde{q}) \,\&\, 2^h \mid \cdots \rangle$. 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 [FW93].

2.3 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 2.3. *Fix b and h. Let S be a set of n sorted disjoint 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 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 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, satisfying (1a). (See Figure 2-2(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). $\square$

The data structure. 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 (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 $[0, 2^h)$. Specifically, the mapping below transforms two intervals $I_L = \{0\} \times [B, B + 2^{\ell_L})$ and $I_R = \{C\} \times [D, D + 2^{\ell_R})$ to $\{0\} \times [0, 2^h)$ and $\{2^h\} \times [0, 2^h)$ respectively:

$$(x, y) \ \mapsto \ \left(\frac{2^{h+\ell_R} \cdot x}{2^{\ell_L}(C - x) + 2^{\ell_R} \cdot x}, \ \frac{2^h[C \cdot (y - B) - (D - B) \cdot x]}{2^{\ell_L}(C - x) + 2^{\ell_R} \cdot x} \right). \tag{2.1}$$

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 $\langle \tilde{a}_0(2^h - \tilde{x}) + \tilde{b}_0 \tilde{x} \mid \tilde{a}_2(2^h - \tilde{x}) + \tilde{b}_2 \tilde{x} \mid \cdots \rangle$. 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 [FW93] (as reviewed in Section 2.2).

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 the segment pq with I_R (which has rational coordinates) and perform a 1-d search on I_R.

Proposition 2.4. *Given a sorted list of n disjoint line segments spanning a vertical slab in the plane with $O(w)$-bit rational coordinates, we can build a data structure in $o(n \lg n)$ time and $O(n)$ space so that point location queries can be answered in $t(n) := O(\lg n/\lg\lg n)$ time.*

2.4 Alternative Bounds

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

Proposition 2.5. *Consider a sorted list of n disjoint line segments spanning a vertical slab in the plane with $O(w)$-bit rational coordinates. For any $h \geq 1$, we can build a data structure of size $O(n \cdot 4^h)$ 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 2.3 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 2.3, it suffices to locate a point $\tilde{q}$ that has h-bit integer coordinates. Thus, we can precompute the answers for all 2^{2h} such points during preprocessing. This takes time $O(2^{2h})$ time: trace each segments 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 2.5 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 $r = \lfloor n/(w/h+4^h)\rfloor$ 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 2.4 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 2.4 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 2.5 can be reduced to $O(n \cdot 4^h)$ as well, by choosing $r = \lfloor n/(w/h)\rfloor$.

We have thus obtained the bounds in Theorem 2.1.

Chapter 3

Sorting Points and Segments

In this chapter, we deal with the offline version of point location in a slab. We are given a vertical slab in the plane, m nonintersecting segments cutting across the slab, and n points inside the slab. The goal is to identify for each of the n points, the segments immediately above and below it. In other words, we would like to sort the points "relative to" the segments.

This problem can easily be solved through n queries to our online point location data structure, giving a running time of $O\big(m+n\cdot\min\big\{\frac{\lg m}{\lg\lg m}, \sqrt{\frac{w}{\lg w}}\big\}\big)$. However, we will now show a significantly better algorithm for the offline problem. The relation to the online problem is parallel to that for 1-d integer sorting. There, 1-d location (predecessor search) is known to require comparatively large running times (e.g. in terms of n alone, an $\Omega(\sqrt{\frac{\lg n}{\lg\lg n}})$ lower bound per point is known [BF02]). Yet, one can find ways of manipulating information in the offline problem (1-d sorting), such that the bottleneck of using the online problem is avoided (e.g. we can sort in $O(n\sqrt{\lg\lg n})$ expected time [HT02]). As for sorting, the highlight of our pursuit is not to study "bit tricks" in the word RAM model, but to study how information about points and lines can be decomposed in algorithmically useful ways.

Theorem 3.1. *Consider a sorted list of m disjoint line segments spanning a vertical slab in the plane, and n points inside the slab. Assuming all points and segment endpoints have $O(w)$-bit rational coordinates, we determine the segment immediately below each point in time $n \cdot 2^{O(\sqrt{\lg\lg m})} + O(m)$.*

Note that this bound does not depend on the universe (aside from assuming a coordinate fits in a word), and is deterministic. The bound is a dramatic improvement over the online bounds — note, for example, that the new bound grows more slowly than $n\lg^{\varepsilon} m+m$ for any constant $\varepsilon > 0$. In addition, the new bound represents a much more convincing improvement over the standard $O(n\lg m)$ bound based on binary search, demonstrating the power granted by bounded precision.

The remainder of this chapter is organized as follows. In Section 3.1, we describe a simple algorithm running in time $O(n\sqrt{\lg m} + m)$. This demonstrates the basic divide-and-conquer strategy behind our solution. In Section 3.2, we implement this strategy much more carefully to obtain an interesting recurrence that ultimately leads to the stated time bound of Theorem 3.1. The challenges faced by this improvement are similar to issues in integer sorting, and indeed we borrow (and build upon) some tools from that field.

Unfortunately, the implementation of Section 3.2 requires a nonstandard word operation. In Section 3.3, we describe how to implement the algorithm on a standard word RAM, using only addition, multiplication, bitwise-logical operations, and shifts. Interestingly, the new implementation requires some new geometric observations that affect the design of the recursion itself.

3.1 An Initial Algorithm

3.1.1 The Basic Recursive Strategy

We begin with a recursive strategy based on Observation 2.3, which was also the basis of our online algorithm. (Later in Section 3.3, we will replace this with a more complicated recursive structure.) We repeat the observation here for ease of reference:

Observation 3.2. *Fix b and h. Let S be a set of n sorted disjoint 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 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 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 $2^{\ell_R - h}$.*

This naturally suggests a recursive algorithm. In the pseudocode in Figure 3-1, the input is a set Q of n points and a sorted set S of m disjoint segments, where the left and right endpoints lie on intervals I_L and I_R of length 2^{ℓ_L} and 2^{ℓ_R} respectively. At the end, ANS[q] stores the segment from S immediately below q for each $q \in Q$. A special NULL value for ANS[q] signifies that q is below all segments. We assume a (less efficient) procedure SLAB$_0(Q, S)$, with the same semantics as SLAB(Q, S), which is used as a bottom case of the recursion. The choice of SLAB$_0()$ is a crucial component of the analysis.

We first explain why the pseudocode works. In step 2, an explicit formula for the transform φ has already been given as (2.1) in Chapter 2; this mapping preserves the belowness relation. According to property (2) in Observation 3.2, we know that the transformed segments $\varphi(\tilde{s}_0), \varphi(\tilde{s}_2), \ldots$ all have h-bit integer coordinates from $[2^h]$. After rounding, the n points $\varphi(Q)$ will lie in the same universe.

```
SLAB($Q, S$):
  0. if $m = 0$, set all answers to NULL and return
  1. let $s_0, s_1, \dots$ be the $O(b)$ segments from Observation 3.2
  2. let $\varphi$ be the projective transform mapping $I_L$ to $\{0\} \times [0, 2^h]$ and $I_R$ to $\{2^h\} \times [0, 2^h]$.
       Compute ROUND($\varphi(Q)$) and $\varphi(\tilde{s}_0), \varphi(\tilde{s}_2), \dots$
  3. SLAB$_0$(ROUND($\varphi(Q)$), $\{\varphi(\tilde{s}_0), \varphi(\tilde{s}_2), \dots\}$)
  4. for each $q \in Q$ with ANS[ROUND($\varphi(q)$)] $= \varphi(\tilde{s}_i)$ do
       set ANS[$q$] = the segment from $\{s_{i-4}, \dots, s_{i+7}\}$ immediately below $q$
  5. for each $s_i$ do
       SLAB($\{q \in Q \mid$ ANS[$q$] $= s_i\}$, $\{s \in S \mid s_i \prec s \prec s_{i+1}\}$)
```

Figure 3-1: A recursive algorithm for the slab problem. Parameters b and h are fixed in the analysis; ROUND($\cdot$) maps a point to its nearest integral point.

Any unit square can intersect at most two of the $\varphi(\tilde{s}_i)$'s, since these segments have vertical separation at least one and thus horizontal separation at least one (as slopes are between -1 and 1). If $\varphi(\tilde{s}_i) \prec$ ROUND($\varphi(q)$) $\prec \varphi(\tilde{s}_{i+2})$, then we must have $\varphi(\tilde{s}_{i-4}) \prec \varphi(q) \prec \varphi(\tilde{s}_{i+6})$, implying that $s_{i-4} \prec \tilde{s}_{i-4} \prec q \prec \tilde{s}_{i+6} \prec s_{i+8}$. Thus, at step 4, ANS[$q$] contains the segment from $s_0, s_1 \dots$ immediately below q. Once this is determined for every point $q \in Q$, we can recursively solve the subproblem for the subset of points and segments strictly between s_i and s_{i+1} for each i, as is done at step 5. An answer ANS[q] = NULL from the i-th subproblem is interpreted as ANS[q] $= s_i$.

Let $\ell = (\ell_L + \ell_R)/2$, where $\ell \le w$. Denote by $T(n, m, \ell)$ the running time of SLAB(), and $T_0(n, b', h)$ the running time of the call to SLAB$_0$() in step 3. Steps 1, 2 and 4 can be implemented naively in $O(n + m)$ time. We have the recurrence:

$$T(n, m, \ell) \;=\; T_0(n, b', h) \;+\; O(n + m) \;+\; \sum_{i=0}^{b'} T(n_i, m_i, \ell_i), \tag{3.1}$$

where $b' = O(b)$, $\sum_i n_i = n$, $\sum_i m_i = m - b'$. Furthermore, according to property (1) in Observation 3.2, for each i we either have $m_i \le \frac{m}{b}$ or $\ell_i \le \ell - \frac{h}{2}$. This implies that the depth of the recursion is $O(\log_b m + \frac{\ell}{h})$.

3.1.2 An $O(n\sqrt{\lg m} + m)$ Algorithm

In Chapter 2, we noticed that for $b'h \approx w$, SLAB$_0$() can be implemented in $T_0(n, b', h) = O(n)$ time by packing b' segments from an h-bit universe into a word. By setting $b \approx \log^{\varepsilon} m$ and $h \approx w/\log^{\varepsilon} m$, this leads to an $O((n + m)\frac{\lg m}{\lg\lg m})$ algorithm.

Instead of packing multiple segments in a word, our new idea is to pack *multiple points* in a word. To understand why this helps, remember that the canonical implementation for SLAB$_0$() runs in time $O(n \lg m)$ by choosing the middle segment and recursing on points above and below this segment. By packing t segments in a word, we can hope to reduce this time to $O(n \log_t m)$. However, by packing t points in a word, we can potentially reduce this to $O(\frac{n}{t} \lg m)$, a much bigger gain. (One can also think of packing both points and segments, for a running time of $O(\frac{n}{t} \log_t m)$. Since we will ultimately obtain a much faster algorithm, we ignore this slight improvement.)

To implement this idea, step 2 will pack $\textsc{round}(\varphi(Q))$ with $O(w/h)$ points per word. Each point is allotted $O(h)$ bits for the coordinates, plus $\lg b = O(h)$ bits for the answer $\textsc{ans}[\textsc{round}(\varphi(q))]$ which $\textsc{Slab}_0()$ must output. This packing can be done in $O(n)$ time, adding one point at a time.

Working on packed points, $\textsc{Slab}_0()$ has the potential of running faster, as evidenced by the following lemma. For now, we do not concern ourselves with the implementation on a word RAM, and assume nonstandard operations (an operation takes two words as input, and outputs one word).

Lemma 3.3. *If $\lg b \le h \le w$, $\textsc{Slab}_0()$ can be implemented on a RAM with nonstandard operations with a running time of $T_0(n, b, h) = O(n\frac{h}{w} \lg b + b)$.*

Proof. Given a segment and a number of points packed in a word, we can postulate two operations which output the points above (respectively below) the segment, packed consecutively in a word. Choosing a segment, we can partition the points into points above and below the segment in $O(\lceil n\frac{h}{w} \rceil)$ time. In the same asymptotic time, we can make both output sets be packed with $\lfloor \frac{w}{h} \rfloor$ points per word (merging consecutive words which are less than full).

We now implement the canonical algorithm: partition points according to the middle segment and recurse. As long as we are working with $\ge \frac{w}{h}$ points, the cost is $O(\frac{h}{w})$ per point for each segment, and each point is considered $O(\lg b)$ times. If we are dealing with less than $\frac{w}{h}$ points, the cost is $O(1)$, and that can be charged to the segment considered. Thus, the total time after packing is $O(n\frac{h}{w} \lg b + b)$.

The last important issue is the representation of the output. By the above, we obtain the sets of points which lie between two consecutive segments. We can then trivially fill in the answer for every point in the $\lg b$ bits allotted for that. However, we want an array of answers for the points in the original order. To do that, we trace the algorithm from above backwards in time. We use an operation which is the inverse of splitting a word into points above and below a segment. □

Plugging the lemma into (3.1), we get $T(n, m, \ell) = O(n\frac{h}{w} \lg b + n + m) \cdot O(\log_b m + \frac{\ell}{h})$. Setting $\lg b = \sqrt{\lg m}$ and $h = w/\sqrt{\lg m}$, we obtain $T(n, m, w) = O((n+m)\sqrt{\lg m})$. This can be improved to $O(m + n\sqrt{\lg m})$ by the standard trick of considering only one in $O(\sqrt{\lg m})$ consecutive segments. For every point, we finish off by binary searching among $O(\sqrt{\lg m})$ segments, for a negligible additional time of $O(n \lg \lg m)$.

3.2 An $n \cdot 2^{O(\sqrt{\lg\lg m})} + O(m)$ Algorithm

3.2.1 Preliminaries

To improve on the $O(m + n\sqrt{\lg m})$ bound, we *bootstrap*: we use an improved algorithm for Slab() as Slab$_0$(), obtaining an even better bound for Slab(). To enable such improvements, we can no longer afford the $O(n)$ term in the recurrence (3.1). Rather, a call to Slab() is passed Q in word-packed form, and we want to implement the steps between recursive calls in *sub*linear time (close to the number of words needed to represent Q, not to $n = |Q|$).

This task will require further ideas and more sophisticated word-packing tricks. To understand the complication, let us contrast implementing steps 2 and 5 of Slab() in sublinear time. Computing round$(\varphi(Q))$ in Step 2 is solved by applying a function in parallel to a word-packed vector of points. This is certainly possible, at least using nonstandard word operations. However, step 5 needs to group elements of Q into subsets (i.e. sort Q according to ans$[q]$). This is a deeper information-theoretic limitation, and it is rather unlikely that it can always be done in time linear in the number of words needed to store Q. The problem has connections to applying permutations in external memory, a well-studied problem which is believed to obey similar limitations [AV88].

To implement step 5 (and also step 4), we will use a subroutine Split(Q, label). This receives a set Q of ℓ-bit elements, packed in $O(n\frac{\ell}{w})$ words. Each elements $q \in Q$ has a $(\lg m)$-bit label label$[q]$ with $\lg m \le \ell$. The labels are stored in the same $O(n\frac{\ell}{w})$ words. We can think of each word as consisting of two portions, the first containing $O(\frac{w}{\ell})$ elements and the second containing the corresponding $O(\frac{w}{\ell})$ labels. The output of Split(Q, label) is a collection of sublists, so that all elements of Q with the same label are put in the same sublist (in arbitrary order).

In addition, we will need Split() to be reversible. Suppose the labels in the sublists have been modified. We need a subroutine Unsplit(Q), which outputs Q in the original order before Split(), but with the modified labels attached.

The following lemma states the time bound we will use for these two operations. The implementation of Split() is taken from a paper by Han [Han01] and has been also used as a subroutine in several integer sorting algorithms [Han04, HT02]. As far as we know, the observation that Unsplit() is possible in the same time bound has not been stated explicitly before.

Lemma 3.4. *Assume* label$[q] \in [m]$ *for all* $q \in Q$, *and let* M *be a parameter. If* $\frac{w}{\ell}\lg m \le \frac{1}{2}\lg M$ *and* $\lg m \le \ell \le w$, *both* Split() *and* Unsplit() *require time* $O(n\frac{\ell}{w}\lg\frac{w}{\ell} + M)$.

Proof. Let $g = \frac{w}{\ell}$. Each word contains g elements, with $g \lg m$ bits of labels. Put words with the same label pattern in the same bucket. This can be done in $O(n/g + \sqrt{M})$ time, since the number of different label patterns is at most $2^{g \lg m} \le \sqrt{M}$. For each bucket, we form groups of g words and *transpose* each group to get g new words, where the i-th element of the j-th new word is the j-th element of the i-th old word. Transposition can be implemented in

$O(\lg g)$ standard word operations [Tho02b]. Elements in each new word now have identical labels. We can put these words in the correct sublists, in $O(n/g + m)$ time. There are at most g leftover elements per bucket, for a total of $O(\sqrt{M}g) = o(M)$; we can put them in the correct sublists in $o(M)$ time. The total time is therefore $O((n/g) \lg g + M)$.

To support unsplitting, we remember information about the splitting process. Namely, whenever we transpose g words, we create a record pointing to the g old words and the g new words. To unsplit, we examine each record created and transpose its g new words again to get back the g old words (with labels now modified). We can also update the leftover elements by creating $o(M)$ additional pointers. □

A particularly easy application of this machinery is to implement the algorithm of Section 3.1 with standard operations (with a minor $\lg \lg m$ slowdown). This result is not interesting by itself, but it will be used later as the base case of our bootstrapping strategy.

Corollary 3.5. *If $\frac{w}{h} \lg b \leq \frac{1}{2} \lg M$ and $\lg b \leq h \leq w$, the algorithm for* Slab$_0$() *from Lemma 3.3 can be implemented on a word RAM with standard operations in time $T_0(n, b, h) = O(n \frac{h}{w} \lg b \lg \frac{w}{h} + bM)$.*

Proof. The nonstandard operations used before were splitting and unsplitting a set of points packed in a word, depending on sidedness with respect to a segment. It is not hard to compute sidedness of all points from a word in parallel using standard operations: we apply the linear map defining the support of the segment to all points (which is a parallel multiplication and addition), and keep the sign bits of each result. The sign bits define 1-bit labels for the points, and we can apply Split() and Unsplit() for these. □

Since the algorithm is used with $b = \sqrt{\lg m}$ and $h = w/\sqrt{\lg m}$, we incur a slowdown of $O(\lg \frac{w}{h}) = O(\lg \lg m)$ per point compared to the implementation with nonstandard operations. By setting $M = m^2$, the algorithm of the previous section would then run in time $O(n\sqrt{\lg m} \lg \lg m + m^3)$ if implemented with standard operations. (The dependence of the second term on m can be lowered as well.)

3.2.2 The Improved Algorithm

Our fastest algorithm follows the same pseudocode of Figure 3-1, but with a more careful implementation of the individual steps. Let $\widetilde{\ell}$ be the number of bits per point and $\widetilde{m}$ the original number of segments in the root call to Slab(). We have $\lg \widetilde{m} \leq \widetilde{\ell} \leq w$. In a recursive call to Slab(), the input consists of some n points and $m \leq \widetilde{m}$ segments, all with coordinates from $[2^\ell]$, where $\ell \leq \widetilde{\ell}$. Points will be packed in $O(\widetilde{\ell})$ bits each, so the entire set Q occupies $O(n\frac{\widetilde{\ell}}{w})$ words. At the end, the output ans$[q]$ is encoded as a label with $\lg \widetilde{m}$ bits, stored within each point $q \in Q$, with the order of the points unchanged in the list Q. Note that one could think of repacking more points per word as ℓ and m decrease, but this will not yield an asymptotic advantage, so we avoid the complication (on the other hand, repacking before the call to Slab$_0$() is essential).

In step 2, we can compute ROUND($\varphi(Q)$) in time linear in the number of words $O(n\frac{\widetilde{\ell}}{w})$, by using a nonstandard word operation that applies the projective transform (and rounding) to multiple points packed in a word. Unfortunately, it does not appear possible to implement this efficiently using standard operations. We will deal with this issue in Section 3.3, by changing the algorithm for SLAB() so that we only require affine transformations, not projective transformations.

Before the call to SLAB$_0$() in step 3, we need to condense the packing of the points ROUND($\varphi(Q)$) to take up $O(n\frac{h}{w})$ words. Previously, we had $O(\frac{w}{\widetilde{\ell}})$ points per word, but after step 2, only $O(h)$ bits of each point were nonzero. We will stipulate that points always occupy an number of bits which is a power of 2. This does not affect the asymptotic running time. Given this property, we obtain a word of ROUND($\varphi(Q)$) by condensing $\widetilde{\ell}/h$ words. This operation requires shifting each old word, and ORing it into the new word.

Note that the order of ROUND($\varphi(Q)$) is different from the order of Q, but this is irrelevant, because we can also reverse the condensing easily. We simply mask the bits corresponding to old word, and shift them back. Thus, we obtain the labels generated by SLAB$_0$() in the original order of Q. Both condensing and its inverse take $O(n\frac{\widetilde{\ell}}{w})$ time.

For the remainder of the steps, we need to SPLIT() and UNSPLIT(). For that, we fix a parameter M satisfying $\frac{w}{\widetilde{\ell}}\lg\widetilde{m} \le \frac{1}{2}\lg M$. In step 4, we first split the list ROUND($\varphi(Q)$) into sublists with the same ANS labels. For each sublist, we can perform the constant number of comparisons per point required in step 4, and then record the new ANS labels in the list, in time linear in the number of words $O(n\frac{\widetilde{\ell}}{w})$. It is standard to implement this in the word RAM by parallel multiplications (see the proof of Lemma 3.4). To complete step 4, we UNSPLIT() to get back the entire list ROUND($\varphi(Q)$), and then copy the ANS labels to the original list Q in $O(n\frac{\widetilde{\ell}}{w})$ time. Since both lists are in the same order, this can be done by masking labels and ORing them in.

To perform step 5, we again split Q into sublists with the same ANS labels. After the recursive calls, we unsplit to get Q back in the original order, with the new ANS labels.

3.2.3 Analysis

For $\frac{w}{\widetilde{\ell}}\lg\widetilde{m} \le \frac{1}{2}\lg M$ and $\lg\widetilde{m} \le \widetilde{\ell} \le w$, the recurrence (3.1) now becomes:

$$T(n,m,\ell) \;=\; T_0(n,b',h) \,+\, O\left(n\frac{\widetilde{\ell}}{w}\lg\frac{w}{\widetilde{\ell}} + M\right) \,+\, \sum_{i=0}^{b'} T(n_i,m_i,\ell_i), \tag{3.2}$$

where $b' = O(b)$, $\sum_i n_i = n$, $\sum_i m_i = m - b'$, and for each i, we either have $m_i \le \frac{m}{b}$ or $\ell_i \le \ell - \frac{h}{2}$. As before, the depth of the recursion is bounded by $O(\log_b\widetilde{m} + \frac{\widetilde{\ell}}{h})$.

Assume that for $\frac{w}{h}\lg b \le \frac{1}{2}\lg M$ and $\lg b \le h \le w$, an algorithm with running time

$$T_0(n,b,h) \;\le\; c_k\left(n\frac{h}{w}\lg^{1/k} b\,\lg\left(\frac{w}{h}\lg b\right) \,+\, bM\right)$$

is available to begin with. This is true for $k = 1$ with $c_1 = O(1)$ by Corollary 3.5.

Then the recurrence (3.2) yields:

$$T(n, \widetilde{m}, \widetilde{\ell}) \;=\; O(c_k) \cdot \left(\left[n\frac{h}{w} \lg^{1/k} b \, \lg\left(\frac{w}{h}\lg b\right) \;+\; n\frac{\widetilde{\ell}}{w} \lg \frac{w}{\widetilde{\ell}} \right] \cdot \left(\log_b \widetilde{m} + \frac{\widetilde{\ell}}{h} \right) \;+\; mM \right).$$

Set $\lg b = \lg^{k/(k+1)} \widetilde{m}$ and $h = \widetilde{\ell} / \lg^{1/(k+1)} \widetilde{m}$. Notice that indeed $\frac{w}{h} \lg b = \frac{w}{\widetilde{\ell}} \lg \widetilde{m} \le \frac{1}{2} \lg M$ and $\lg b \le h \le w$. Thus, we obtain an algorithm with running time:

$$T(n, \widetilde{m}, \widetilde{\ell}) \;\le\; c_{k+1} \left(n\frac{\widetilde{\ell}}{w} \log^{1/(k+1)} \widetilde{m} \; \lg\left(\frac{w}{\widetilde{\ell}} \lg \widetilde{m} \right) \;+\; \widetilde{m}M \right)$$

for some $c_{k+1} = O(1) \cdot c_k$.

Iterating this process k times, we get:

$$T(n, \widetilde{m}, \widetilde{\ell}) \;\le\; 2^{O(k)} \left(n\frac{\widetilde{\ell}}{w} \lg^{1/k} \widetilde{m} \; \lg\left(\frac{w}{\widetilde{\ell}} \lg \widetilde{m} \right) \;+\; \widetilde{m}M \right)$$

for any value of k. Choosing $k = \sqrt{\lg\lg \widetilde{m}}$ to asymptotically minimize the expression, and plugging in $\widetilde{\ell} = w$ and $M = \widetilde{m}^2$ (so that indeed $\frac{w}{\widetilde{\ell}} \lg \widetilde{m} \le \frac{1}{2} \lg M$), we get:

$$T(n, \widetilde{m}, w) \;=\; 2^{O(\sqrt{\lg\lg \widetilde{m}})}\,(n \;+\; \widetilde{m}^3).$$

We can reduce the dependence on $\widetilde{m}$ to linear as follows. First, select one out of every $\widetilde{m}^{3/4}$ consecutive segments of S, and run the above algorithm on just these $\widetilde{m}^{1/4}$ segments. This takes time $2^{O(\sqrt{\lg\lg \widetilde{m}})}(n + \widetilde{m}^{3/4})$ time. Now recurse between each consecutive pair of selected segments. The depth of the recursion is $O(\lg\lg \widetilde{m})$, and it is straightforward to verify that the running time is $n \cdot 2^{O(\sqrt{\lg\lg \widetilde{m}})} + O(\widetilde{m})$.

3.3 Avoiding Nonstandard Operations

The only nonstandard operation used by the algorithm of Section 3.2 is applying a projective transform in parallel to points packed in a word. Unfortunately, it does not seem possible to implement this in constant time using standard word RAM operations, since, according to the formula for projective transform (2.1), this operation requires multiple divisions where the divisors are all different.

One idea is to simulate the special operation in slightly superconstant time. We can use the circuit simulation results of Brodnik et al. [BMM97] to reduce the operation to $\lg w \cdot (\lg\lg w)^{O(1)}$ standard operations. For the version of the slab problem in dimension 3 or higher (see Appendix A), this is the best approach we know.

However, in two dimensions we can get rid of the dependence on the universe, obtaining a time bound of $n \cdot 2^{O(\sqrt{\lg\lg m})} + O(m)$ on the standard word RAM. This constitutes the object of this section.

3.3.1 The Center Slab

By horizontal translation, we can assume the left boundary of our vertical slab is the y-axis. Let the abscissa of the right boundary be Δ. For some h to be determined, let the *center slab* be the region of the plane defined by $\Delta/2^h \le x \le \Delta \cdot (1 - 2^{-h})$. The lateral slabs are defined in the intuitive way: the left slab by $0 \le x \le \Delta/2^h$ and the right slab by $\Delta \cdot (1 - 2^{-h}) \le x \le \Delta$.

The key observation is that distances are somewhat well-behaved in the center slab, so we will be able to decrease both the left and right intervals at the same time, not just one of them. This enables us to use (easier to implement) affine maps instead of projective maps.

The following is a replacement for Observation 3.2:

Observation 3.6. *Fix b and h. Let S be a set of m sorted disjoint segments, such that all left endpoints lie on an interval I_L and all right endpoints lie on an interval I_R, where both I_L and I_R have length 2^ℓ. In $O(b)$ time, we can find $O(b)$ segments $s_0, s_1, \ldots \in S$ in sorted order, which include the lowest segment of S, such that:*

(1) *for each i, at least one of the following holds:*
   - (1a) *there are at most m/b segments of S between s_i and s_{i+1}.*
   - (1b) *both the left and right endpoints of s_i and s_{i+1} are at distance at most $2^{\ell-h}$.*

(2) *there exist segments $\tilde{s}_0 \prec \tilde{s}_1 \prec \cdots$ cutting across the slab, satisfying all of the following:*
   - (2a) *distances between the left endpoints of the $\tilde{s}_i$'s are multiples of $2^{\ell-2h}$.*
   - (2b) *ditto for the right endpoints.*
   - (2c) *inside the center slab, $s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec \cdots$.*

Proof. Let B contain every $\lfloor m/b \rfloor$-th segment of S, starting with the lowest segment s_0. We define s_{i+1} inductively. If the next segment after s_i has either the left or right endpoints at distance greater than $2^{\ell-h}$, let s_{i+1} be this segment, which satisfies (1a). Otherwise, let s_{i+1} be the *highest* segment of B which satisfies (1b).

Now impose grids over I_L and I_R, both consisting of 2^{2h} subintervals of length $2^{\ell-2h}$. We obtain $\tilde{s}_i$ from s_i by rounding each endpoint to the grid point immediately above. This immediately implies $\tilde{s}_0 \prec \tilde{s}_1 \prec \cdots$ and $s_i \prec \tilde{s}_i$. Unfortunately, $\tilde{s}_i$ and s_{i+k} may intersect for arbitrarily large k (e.g. $s_i, \ldots, s_{i+k}$ are very close on the left, while each consecutive pair is far on the right). However, we will show that inside the center slab, $\tilde{s}_i \prec s_{i+2}$. (See Figure 3-2(a).)

By construction, s_i and s_{i+2} are vertically separated by more than $2^{\ell-h}$ either on the left or on the right. Since lateral slabs have a fraction of 2^{-h} of the width of the entire slab, the vertical separation exceeds $2^{\ell-h}/2^h = 2^{\ell-2h}$ anywhere in the center slab. Rounding s_i to $\tilde{s}_i$ represents a vertical shift of less than $2^{\ell-2h}$ anywhere in the slab. Hence, $\tilde{s}_i \prec s_{i+2}$ in the center slab. $\square$

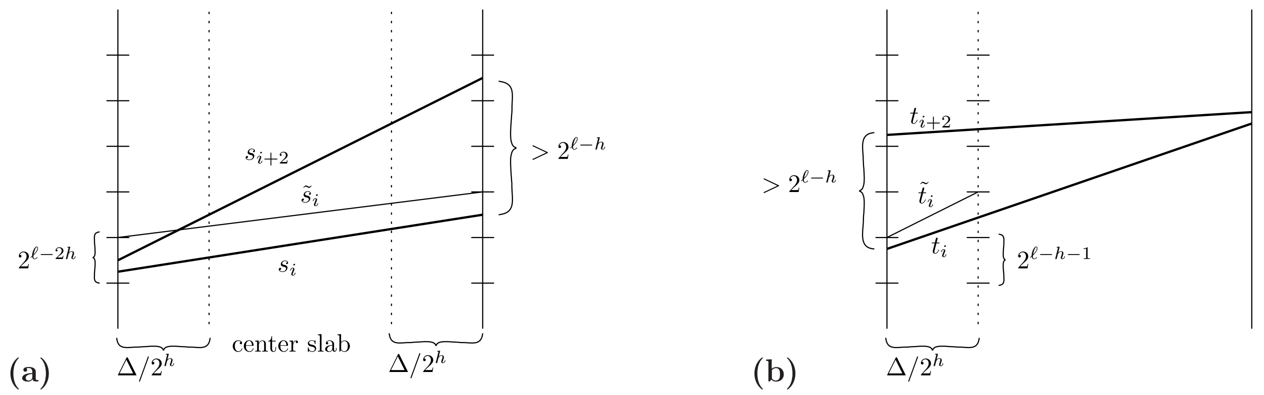

Figure 3-2: (a) A center slab, as in Observation 3.6. (b) A left slab, as in Observation 3.7.

We now describe how to implement Slab(), assuming the intervals containing the left endpoints (I_L) and the right endpoints (I_R) both have length 2^ℓ. In this section, we only deal with points in the center slab. It is easy to Split() Q into subsets corresponding to the center and lateral slabs, and Unsplit() at the end.

We use Observation 3.6 instead of Observation 3.2. Since I_L and I_R have equal length, the map φ is affine. Thus, it can be implemented using parallel multiplication and parallel addition. This means step 2 can be implemented in time $O(n\frac{\tilde{\ell}}{w})$ using standard operations.

Because we only deal with points in the center slab, and there $s_i \prec \tilde{s}_i \prec s_{i+2}$ (just like in the old Observation 3.2), steps 4 and 5 work in the same way.

3.3.2 Lateral Slabs

To deal with the left and right slabs, we use the following simple observation, which we only state for the left slab by symmetry. Note that the guarantees of this observation (for the left slab) are virtually identical to that of Observation 3.6 (for the center slab). Thus, we can simply apply the algorithm of the previous section for the left and right slabs.

Observation 3.7. *Fix b and h. Let S be a set of m sorted disjoint segments, such that all left endpoints lie on an interval I_L and all right endpoints lie on an interval I_R, where both I_L and I_R have length 2^ℓ. In $O(b)$ time, we can find $O(b)$ segments $t_0, t_1, \ldots \in S$ in sorted order, which include the lowest segment of S, such that:*

(1) *for each i, at least one of the following holds:*
   - (1a) *there are at most m/b segments of S between s_i and s_{i+1}.*
   - (1b) *anywhere in the left slab, the vertical separation between s_i and s_{i+1} is less than $2^{\ell-h+1}$.*

(2) *there exist segments $\tilde{t}_0 \prec \tilde{t}_1 \prec \cdots$ cutting across the left slab, satisfying all of the following:*
   - (2a) *distances between the left endpoints of the $\tilde{t}_i$'s are multiples of $2^{\ell-2h}$.*
   - (2b) *ditto for the right endpoints.*

(2c) *inside the left slab,* $t_0 \prec \tilde{t}_0 \prec t_2 \prec \tilde{t}_2 \prec \cdots$.

Proof. Let I_A be the vertical interval at the intersection of the right edge of the left slab with the parallelogram defined by I_L and I_R. Note I_A also has size 2^ℓ.

Let B contain every $\lfloor m/b \rfloor$-th segment of S, starting with the lowest segment t_0. Given t_i, we define t_{i+1} to be the highest segment of B which has the left endpoint at distance at most $2^{\ell-h}$ away. If no such segment above t_i exists, let t_{i+1} be the successor of t_i in B (this will satisfy (1a)). In the first case, (1b) is satisfied because the right endpoints of t_i and t_{i+1} are at distance most 2^ℓ, so on I_A, the separation is at most $2^{\ell-h}(1-2^{-h}) + 2^\ell \cdot 2^{-h} < 2^{\ell-h+1}$.

Now impose grids over I_L and I_A, both consisting of 2^{h+1} subintervals of length $2^{\ell-h-1}$. We obtain $\tilde{t}_i$ from t_i by rounding the points on I_L and I_A to the grid point immediately above. Note that the vertical distance between t_i and $\tilde{t}_i$ is less than $2^{\ell-h-1}$ anywhere in the left slab. On the other hand, the left endpoints of t_i and t_{i+2} are at distance more than $2^{\ell-h}$. The distance on I_A (and anywhere in the left slab) is at least $2^{\ell-h}(1-2^{-h}) \geq 2^{\ell-h-1}$. Thus $t_i \prec \tilde{t}_i \prec t_{i+2}$. (See Figure 3-2(b).) □

3.3.3 Bounding the Dependence on m

Our analysis needs to be modified, because segments are simultaneously in the left, center and right slabs, so they are included in 3 recursive calls. In other words, in recurrence (3.2), we have to replace $\sum_i m = m - b'$ with a weaker inequality $\sum_i m \leq 3m$. Recall that for our choice of b and h, the depth of the recursion is bounded by $O(\log_b \widetilde{m} + \frac{\widetilde{\ell}}{h}) = O(\lg^{1/(k+1)} \widetilde{m})$. Thus, the cost per segment is increased by an extra factor of $3^{O(\lg^{(1/(k+1)} \widetilde{m})} = 3^{O(\sqrt{\lg \widetilde{m}})}$ for each bootstrapping round; the cost per point does not change. With $k = \sqrt{\lg \lg \widetilde{m}}$ rounds, the overall dependence on $\widetilde{m}$ is now increased slightly to $2^{O(\sqrt{\lg \lg \widetilde{m}})} \cdot \widetilde{m}^3 \cdot 3^{O(\sqrt{\lg \widetilde{m}} \lg \lg \widetilde{m})}) = O(\widetilde{m}^{3+\varepsilon})$. As before, this can be made $O(\widetilde{m})$ by working with $\widetilde{m}^{1/4}$ segments and recursing.

Chapter 4

General Point Location via Reductions

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 (x, y), we can first locate the slab containing (x, y) by a 1-d predecessor search on x, and then search in this slab. Since point location among horizontal segments solves predecessor search, we know predecessor search takes at most $t(n)$ time.

Note that this reduction also relates the two offline problems. If we can sort n points and m segments in a slab with running time $O(m + n \cdot t(m))$, general offline point location is solved in time $O(m^2 + n \cdot t(m))$, as each point appears in exactly one slab.

Using more advanced techniques, we can obtain reductions from the general case to the slab problem without a loss in the bounds. Formally, our results are as follows:

Theorem 4.1. *Let all coordinates by $O(w)$-bit rationals. 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.*

Then given any planar connected subdivision defined by n disjoint line segments, we can build a data structure in $O(n)$ time and space so that point location queries can be answered in time asymptotically:

$$\min \begin{cases} t(n) \cdot O(\lg \lg n) \\ t(n), \quad \textit{if } t(n)/\lg^{\delta} n \textit{ is monotone increasing for constant } \delta > 0 \\ \sqrt{w/\lg w} \end{cases}$$

Observe that the running time is unaffected if it is a not-too-small function of n, making this a very general reduction. At present, the best bound is of course $t(n) = O(\lg n/\lg\lg n)$ by Theorem 2.1. If the running time depends on w, we do not get such a nice guarantee. However, we can still recover the $O(\sqrt{w/\lg w})$ bound from Theorem 2.1, which is the best known to date. This needs a somewhat more careful analysis in the reduction, crafter for the precise bounds of Theorem 2.1.

Theorem 4.2. *Suppose there is an algorithm with running time $O(m) + n \cdot \tau(m)$, which can answer offline point location for m segments cutting across a slab, and n points inside the slab. Then we can locate n given points in an arbitrary planar subdivision with m segments in time $O(m + n \cdot \tau(m) \cdot \lg\lg m)$.*

We could obtain a result similar to the online case, including a bound of $\sqrt{w/\lg w}$, and without the $\lg\lg m$ factor for $\tau(m)$ large enough. However, we already know $\tau(m) \leq 2^{O(\sqrt{\lg\lg m})}$, so these cases are irrelevant. Note that $2^{O(\sqrt{\lg\lg m})} \cdot \lg\lg m = 2^{O(\sqrt{\lg\lg m})}$, so again we recover the best known bounds for the slab problem (in the qualitative sense of the Oh-notation).

4.1 Technical Overview

We describe three different reductions from general point location to point location in a slab. The first two are careful adaptations of classic techniques in computational geometry, while the third is a technical contribution of independent interest:

planar separators [LT80] This method has the best theoretical properties, including deterministic bounds, a linear-time construction for the online problem, and a linear dependence on m for the offline problem. However, it is probably the least practical because of large hidden constants.

random sampling [Mul00] This method is the simplest, but the construction algorithm is randomized, and takes time $O(n \cdot \tau(n))$. For the offline case, it gives an expected running time of $O((m + n) \cdot \tau(m) \cdot \lg\lg m)$.

persistent search trees [ST86] This is the least obvious to adapt and requires an interesting construction of an *exponential search tree* with nice geometric properties. It yields a deterministic construction time of $O(\text{sort}(n))$, and an offline algorithm with running time $O(\text{sort}(m) + n \cdot \tau(m) \cdot \lg\lg m)$.

Most importantly, this result shows how our sublogarithmic results can be used in sweep-line algorithms, which is important for some later applications.

Since the planar separator approach yields the best bounds (Theorems 4.1 and 4.2), we describe it with all calculations necessary for the theorems. In the other cases, we only aim to convey the technical ideas, so we only describe the online reduction, and further assume $t(n)/\lg^{\delta} n$ is increasing. The calculations are essentially the same as for the separator method.

4.2 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. Note that for any connected subdivision, we can first triangulate it in linear deterministic time by Chazelle's algorithm [Cha91].

Our deterministic method is based on the planar graph separator theorem by Lipton and Tarjan [LT80] (who also noted its possible application to the point location problem).

Lemma 4.3. *Given a planar graph G with n vertices and a parameter r, we can find a subset of $O(\sqrt{rn})$ vertices in $O(n)$ time, such that each connected component of $G \setminus S$ has at most n/r vertices.*

Proof. The combinatorial bound follows by applying the planar separator theorem recursively. We can get a linear-time algorithm from the results by Aleksandrov and Djidjev [AD96] or Goodrich [Goo95]. □

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.

If the problem is offline, we have to solve an offline problem in the subdivision induced by R, and the an offline problem in each subregion. We will make sure the offline running time for m segments and n points can be expressed as $P(m) + n \cdot Q(m)$. This is true for the slab problem by assumption. Then Q_0, Q_1 represent the cost per point, and P_0, P_1 the additive cost depending on the number of segments (in R, and each subregion).

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 naïve method with $P_0(n) = P_1(n) = O(n^2)$ and $Q_0(n) = Q_1(n) = O(t(n))$ (or $\tau(n)$ for the offline problem). 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 $r = \lfloor n^{1/4} \rfloor$, 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 $t(n) \geq (4/3)^\delta t(n^{3/4})$ 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 bootstrap 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 [Kir83], $P_1(n) = O(n)$ and $Q_1(n) = O(\lg n)$. We obtain the following bounds, where $\sum n_i \leq n$ and $n_i \leq n/r$:

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

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.

We note that it is possible to avoid the last bootstrapping step by observing that the total cost of the recursive separator computations is linear [Goo95]. The first bootstrapping step could also be replaced by a more naïve method that divides the plane into $\sqrt{n}$ slabs.

Alternative bounds. We can also obtain bounds that depend on the precision w, although parameters have to be set differently (to avoid increasing the query time by a $\lg\lg n$ factor). Using the h-sensitive bounds from Theorem 2.1, we start with $P_0(n) = P_1(n) = O(n^2 \cdot 2^h)$ and $Q_0(n) = Q_1(n) = O(w/h)$. The first bootstrapping step with $r = \lfloor\sqrt{n}\rfloor$ yields $P(n) = O(n^{3/2} \cdot 2^h)$ and $Q(n) = O(w/h)$.

In the next step, we use $P_0(n) = O(n^{3/2} \cdot 2^h)$ and $Q_0(n) = O(w/h)$ and apply recursion to handle each subregion. We set $r = \lfloor n^{1/4} \rfloor$ and $h = \lfloor \varepsilon \lg n \rfloor$ for a sufficiently small constant $\varepsilon > 0$ (so that $(\sqrt{rn})^{3/2} \cdot 2^h = o(n)$). 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 $\sum_i n_i \leq n$ and $n_i = O(n^{3/4})$. We stop the recursion when $n \leq n_0$ and handle the base case using Theorem 2.1, 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.

4.3 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 monotonically increasing. The method we will describe gives a data structure of $O(n)$ space, which can be constructed in expected $O(n \cdot t(n))$ time[1], and supports queries in $O(t(n))$ query time. Although construction is slower and randomized, the method is simpler and the idea itself has further applications.

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 [CS89, Mul00], 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 $n_i =$

[1]In fact, $O(n \cdot \tau(n))$ because construction is an offline problem. We ignore this detail, since planar separators already give us a theoretically optimal bound.

$O((n/r)\lg r)$:

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

Calculations. To get started, we use the naïve 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 [Sno04] 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 than $t(n)$ 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.

4.4 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 naïve 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.

In 1-d, the pervasive approach for dynamization and persistence would be bucketing. The idea is to bucket, say, $\Theta(k)$ consecutive elements, building a bottom structure inside each bucket, and a top structure for elements separating the $\Theta(n/k)$ buckets. A particularly aggressive type of bucketing is the exponential tree, a structure that we will describe below.

In two dimensions, the equivalent of bucketing seems very difficult to achieve. Indeed, this would have important applications even for unbounded precision: it would imply logarithmic upper bounds for *dynamic* point location, a famous open problem. The difficulty in bucketing comes from the apparent impossibility to choose good separators. Say, for instance, that some segment s_1 is chosen to separate two buckets. In the very next step, s_1 could be deleted, and another s_2 which intersects s_1 can be inserted. Now, unless we rebuild the top structure to replace s_1 (a very expensive operation), we do not know in which bucket to place s_2.

Though we do not know how to make bucketing work in general, we can make it work for our application, by observing that point location only needs to construct exponential trees in the *semionline* case: for every element, we are told the time in the future when it will be deleted. This is a natural property of any sweep-line algorithm. Motivated by this property, we construct semionline geometric exponential trees by a careful interplay of geometric separation arguments and standard amortization arguments.

4.4.1 The Segment Predecessor Problem

We define the *segment-predecessor 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 p is inside the maximal vertical slab which does not contain any endpoint from S (and that this slab is always nonempty).

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 the following operation:

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, even at the time of insertion.

4.4.2 Geometric Exponential Trees

We will use exponential trees [And96, AT02], a remarkable idea coming from the world of integer search. This is a technique for converting a black-box static predecessor 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 $\frac{n}{B}$ 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-predecessor 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 structure from above over $O(n/(\lg\lg n)^2)$ splitters. The bottom structures have $\Theta((\lg\lg n)^2)$ elements, and we can simply use a linked list to represent them in order. The query time is increased by $O((\lg\lg n)^2)$ because we have to search through such a list, but that is a lower order term. Updating the bottom structure now takes constant time, given a pointer to the segment or a neighbor. As shown above, an update to the top structure only occurs after $\Omega((\lg\lg n)^2)$ updates to a bottom structure. So operations in the top structure cost $O(1)$ amortized.

4.4.3 Application to Point Location

Sweep-line construction. We first sort the x-coordinates corresponding to the endpoints, taking sort$(2n)$ time. To know which of the $O(n)$ slabs a query point lies in, we construct an integer predecessor structure for the x-coordinates. The optimal complexity of predecessor 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-predecessor 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 $O(\text{sort}(n))$, if we know where each insert should go, and can avoid the queries at construction time. To achieve this, we first convexify the outer face, adding edges of the convex hull. The convex hull can be found in linear time by gift wrapping, since we can walk the edges of the polygon in order. Then, we triangulate the planar subdivision by the deterministic linear-time algorithm of [Cha91].

Now consider any vertex at the time when the vertical sweep line hits it. Since we are working with a triangulation, the vertex must have an incident edge to the left of the sweep line, unless it is on the outer face. Since the outer face is convex, the vertex must have an edge to the left unless it is the very first vertex inserted. Thus, every segment insert immediately follows a segment delete with a shared vertex, so we know a neighbor of the inserted segment by examining the neighbors of the deleted segment. These neighbors are known in constant time since we can just maintain a linked list of all active segments in order.

Persistence. It remains to make the segment predecessor 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 predecessor 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 predecessor 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 [DSST89].

Given k numbers in $\{1, \dots, 2n\}$ (our time universe), a van Emde Boas data structure for integer predecessor can be constructed in $O(k)$ time deterministically [Ruž07], 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 predecessor 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.

Chapter 5

Reexamining Computational Geometry

It turns out the our results for point location lead to improved bounds for many fundamental problems in computational geometry. This highlights once again the center-stage position that point location occupies in nonorthogonal geometry.

In the next sections, we discuss the following applications of our results:

5.1 applications to online data structures, such as 2-d nearest neighbor queries.
5.2 application to the segment intersection problem.
5.3 a reduction from computing 3-d convex hulls and 2-d Voronoi diagrams, to offline point location.
5.4 further algorithmic consequences of the previous two results.

5.1 Improved Data Structures

Corollary 5.1. *Let all coordinates be $O(w)$-bit rationals. If the point location problem can be solved by a data structure of size $O(n)$ with query time $t(n, w)$, then:*

(a) *given n points in the plane, we can build a data structure of size $O(n)$, so that nearest/farthest neighbor queries under the Euclidean metric can be answered in $O(t(n, w))$ time.*

(b) *given n points in the plane, we can build an $O(n \lg \lg n)$-space data structure supporting the following queries in $O(t(n, w) + k)$ time:*
- *circular range queries: report all k points inside a query circle;*
- *k-nearest neighbors queries: report the k nearest neighbors to a query point.*

(c) *given a convex polygon P with n vertices, we can build an $O(n)$-space data structure supporting the following queries in $O(t(n, w))$:*
- *tangent queries: find the two tangents of P through an exterior point;*
- *line stabbing queries: find the intersections of P with a line.*

Proof. (a) Nearest neighbor queries reduce to point location in the Voronoi diagram. (From the results below, the construction time of the Voronoi diagram, and hence the data structure, relates to *offline* point location.) Farthest neighbor queries also reduce to point location.

(b) The result is obtained by adopting the range reporting data structure from [Cha00], using Theorem 4.1 to handle the necessary point location queries.

(c) For tangent 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.

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

□

5.2 Segment Intersection

We now 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 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.

This problem turns out to be related to *offline* point location, by the random sampling approach used in Section 4.3. Unfortunately, we are unable to adapt the separator or persistence approaches. Thus, we can only beat classic $O(n \lg n)$ bounds with randomized algorithms.

Theorem 5.2. *Assume offline point location with n points and m segments can be solved in time $O(m + n \cdot \tau(m))$, and that $\tau(n) = \Omega(\lg \lg n)$. In particular, $\tau(m) \leq 2^{O(\sqrt{\lg \lg m})}$.*

Then, given n line segments in the plane, we can find all k intersections, and compute the trapezoidal decomposition, in $O(n \cdot \tau(n) + k)$ expected time.

Proof. Take a random sample $R \subseteq S$ of size r. Compute its trapezoidal decomposition $T(R)$ by a known algorithm [Mul00] in $O(r \lg r + |T(R)|)$ time. 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. This is an offline problem with $O(|T(R)|)$ segments and n query points, so it takes time $O\big(|T(R)| + n \cdot \tau(|T(R)|)\big)$. Since $|T(R)| \leq n^2$ and $\tau(m) < O(\lg m)$, this running time is $O(|T(R)| + n \cdot \tau(n))$.

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 $O(n \cdot \tau(n) + \sum_{\Delta \in T(R)} |S_\Delta| \ell_\Delta)$.

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 [CS89, Mul00],

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

Clarkson and Shor had also specifically shown [CS89, 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 $O(r\lg r + n\cdot\tau(n) + n\lg(n/r) + k)$. Setting $r = \lfloor n/\lg n\rfloor$ yields the desired result. $\square$

5.3 Convex Hulls in 3-d and Voronoi Diagrams in 2-d

We next tackle the well-known problem of constructing the convex hull of a set S of n points in 3-d. It is well known that constructing a Voronoi diagram in 2-d reduces to this problem. As before, our reduction is randomized, and we do not know how to achieve $o(n\lg n)$ algorithms deterministically.

Theorem 5.3. *Let $\tau(n)$ be as in Theorem 5.2. Given n points in three dimensions with $O(w)$-bit rational coordinates, we can compute their convex hull in $O(n\cdot\tau(n))$ expected time.*

Proof. 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 [dBSvKO00, PS85]. The xy-projection of the faces of the upper hull is a triangulation T_π.

For each point $s\in S$, consider the dual plane s^* [dBSvKO00, Ede87, Mul00]. Constructing the upper hull is equivalent to constructing the lower envelope of the dual planes. Let $T(R)$ denote a *canonical triangulation* [Cla88, Mul00] 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, i.e. a point location query in T_π. 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 $O(n\cdot\tau(n) + \sum_{\Delta\in T(R)} |S^*_\Delta|)$.

We then construct LE(S^*_Δ) 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 [CS89, Mul00],

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

The total expected running time is $O(r \lg r + n\cdot\tau(n) + n\lg(n/r))$. Setting $r = \lfloor n/\lg n\rfloor$, the running time is dominated by $O(n\cdot\tau(n))$. ☐

5.4 Other Consequences

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

Corollary 5.4. *Let $\tau(n)$ be as in Theorem 5.2. Then:*

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

(b) *given n points in the plane, we can construct the Euclidean minimum spanning tree in $O(n\cdot\tau(n))$ expected time;*

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

(d) *we can triangulate an n-vertex polygon in $O(n\cdot\tau(n))$ deterministic time;*

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

Proof. (a) By a lifting transformation [dBSvKO00, Ede87, O'R98], 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 5.3.

(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) The coordinates of the optimal circle are still $O(w)$-bit rationals, and can be determined from the Voronoi diagram in linear time [PS85]. Again the result follows from (a). Curiously, the 1-d version of the problem admits an $O(n)$-time RAM algorithm by Gonzalez [PS85].

(d) It is known [FM84] that a triangulation can be constructed from the trapezoidal decomposition of the edges in linear time. The result follows from Theorem 5.2 if randomization is allowed. Deterministically, we can instead compute the trapezoidal decomposition by running the algorithm from Section 4.4, since that algorithm explicitly maintains a sorted list of segments that intersect the vertical sweep line at any given time.

(e) The result is obtained by adopting the output-sensitive convex hull algorithm from [Cha96], using Theorem 5.3 to compute the subhull of each group. For readers familiar with [Cha96], we note that the running time for a group size m is now $O(n \cdot \tau(m) + H(n/m)\lg m)$; we can choose $m = \lfloor H \lg H \rfloor$ and apply the same "guessing" trick.

$\square$

Chapter 6

Improving Dynamic Search

Dynamic data structures present some of the toughest challenges in our study. As opposed to static data structures, where we are free to organize a hierarchical structure based on our information-progress argument, dynamic data structures already need a different hierarchical structure for the sake of dynamic maintenance. Quite expectedly, the requirements of these two structures can clash in catastrophic ways.

In this chapter, we study tangent queries in the *dynamic convex hull* problem, as the prime example of a dynamic nonorthogonal problem. Recall that the tangent query asks for the two tangents through a given point to the convex hull of the set S of points in the data structure. Remember that tangent queries for a static polygon are reducible to point location; see Corollary 5.1(c). In the dynamic setting, we show:

Theorem 6.1. *Assuming points have $O(w)$-bit rational coordinates, tangent queries can be supported in time $O(\frac{\lg n}{\lg \lg n})$, while updates are supported in time $O(\lg^2 n)$.*

As opposed to the static case, we cannot hope for a substantially better bound for small universes. In Appendix B, we show that for any polylog(n) update time, the query time must be $\Omega(\lg n / \lg w)$. Thus, our query time is optimal, at least for precision $w = \text{polylog}(n)$.

Tangent queries are only one of the queries that are typically considered for the dynamic convex hull problem. In Appendix B, we discuss the various queries in more detail. While most of these can be reduced to finding tangents, it turns out that some queries (e.g. linear programming) are in essence one-dimensional, and enjoy better bounds. In this chapter, we focus only on tangent queries, which have a strong nonorthogonal, two-dimensional flavor, and use them to study techniques needed for sublogarithmic query time in a dynamic setting.

6.1 Technical Overview

All previous data structures for dynamic planar convex hull assume infinite precision and are therefore limited to running queries in $\Theta(\lg n)$ time. The original such data structure, of Overmars and van Leeuwen [OvL81], introduced the idea of recursively representing the convex hull, leading to a $\Theta(\lg^2 n)$ update time. Eighteen years later, Chan [Cha01a] had the breakthrough idea of using techniques from decomposable search problems, reducing the update time to $\Theta(\lg^{1+\varepsilon} n)$. This approach was subsequently improved by Brodal and Jacob to update times of $\Theta(\lg n \lg\lg n)$ [BJ00] and finally $\Theta(\lg n)$ [BJ02].

Our upper bound starts from the classic dynamic convex hull structure of Overmars and van Leeuwen [OvL81]. The first idea is to convert the binary tree in this structure into a tree with branching factor $\Theta(\lg^{\varepsilon} n)$, so that its height is $\Theta\left(\frac{\lg n}{\lg\lg n}\right)$. The many years of failed attempts at sublogarithmic planar point location suggest, however, that it is impossible to solve any nontrivial query by spending $\Theta(1)$ time per node in such a tree. For example, determining which child to recurse into for a tangent query boils down to planar point location in a subdivision of complexity $\Theta(\lg^{\varepsilon} n)$, which we do not know how to solve in $o\left(\frac{\lg\lg n}{\lg\lg\lg n}\right)$ time.

Instead, by carefully exploiting the partial information that a query learns about its answer, we show that the time a query spends to determine which child to visit is proportional to the knowledge it learns about the answer. By charging the time cost to the information progress, we can use an amortization argument to show that expensive nodes are rare and thus bound the overall query cost to $O\left(\frac{\lg n}{\lg\lg n}\right)$. This type of insight does not appear in the static problems.

In contrast, the decomposition techniques of [Cha01a, BJ02], which achieve $o(\lg^2 n)$ update time, seem fundamentally incompatible with information progress arguments. The trade-off we can obtain between a node's query cost and the information it reveals relies on an essentially explicit representation of the convex hull as in the Overmars-van Leeuwen structure. Representing the convex hull as the hull of $O\left(\frac{\lg n}{\lg\lg n}\right)$ overlapping convex hulls, as in Chan's structure, means that we must make independent information progress for all hulls, which is too slow.

Our information-theoretic lens therefore highlight an even stronger contrast between the original Overmars-van Leeuwen structure and the more modern structures based on decomposable search: the latter structures are not informationally efficient. It is a fascinating open question to break the $\Theta(\lg^2 n)$ barrier (again) while achieving information efficiency.

6.1.1 Review of Overmars-van Leeuwen

Before proceeding, we quickly sketch the classic data structure of [OvL81], skipping all implementation details which are treated as a black box by our modifications.

The data structure is a binary tree, in which every node v contains the set of points S_v in a certain vertical slab. Let left(v) and right(v) be the children of v. The node v stores a vertical line, splitting S_v into $S_v = S_{\text{left}(v)} \cup S_{\text{right}(v)}$, with $\min\{|S_{\text{left}(v)}|, |S_{\text{right}(v)}|\} = \Omega(|S_v|)$. The children split the sets recursively, down to singleton sets in the leaves. Maintaining this partition is equivalent to maintaining a balanced binary search tree with values stored only in the leaves.

Although Overmars and van Leeuwen developed their structure before the invention of persistence [DSST89], it is easier to see its workings using persistent catenable search trees. Every node v stores a list of the nodes on the convex hull H_v of S_v, represented as a (partially) persistent binary search tree. Then, a query can be answered in logarithmic time based on the hull stored at the root. By standard tree-threading techniques, we can also support gift-wrapping queries in $O(1)$ time.

To maintain this hull dynamically, note that H_v is defined by a convex subchain of $H_{\text{left}(v)}$ and one from $H_{\text{right}(v)}$, plus two new edges (*bridges*) that join them. It is shown in [OvL81] that the bridges can be computed in $O(\lg |S_v|)$ time through binary search. Then, because the binary search trees storing the hulls are persistent and catenable, the information at every node can be recomputed in $O(\lg n)$ time. Thus, updates cost $O(\lg^2 n)$.

6.2 Adaptive Point Location

As hinted already, we will store the hulls in the Overmars-van Leeuwen structure as (persistent, catenable) B-trees, for some $B = \lg^\varepsilon n$. It turns out (see below) that the subproblem that must be solved at each node is point location among $O(B)$ segments. Unfortunately, we do not know how to achieve the equivalent of 1-d fusion trees, handling a superconstant B in $O(1)$ time. To obtain our improvement, one must refine the fusion tree paradigm to *hull fusion*: querying a node of the B-tree (a hull-fusion node) is allowed to take superconstant time, but averaged over all $O(\log_B n)$ nodes that are considered, we spend constant time per node. This follows from an information-progress argument: if querying one node is slow, it is because we have made a lot of progress in understanding the query, and therefore this cannot happen too often.

Our basic tool for relating the complexity of point location to information progress is the following lemma, which is an adaptive version of our point location result from Section 2.3:

Lemma 6.2. *Consider a vertical slab $\{x_L, \ldots, x_R\} \times [u]$, and a set S of $B \le w$ segments between points (x_L, ℓ_i) and (x_R, r_i), where $\ell_1 \le \cdots \le \ell_B$ and $r_1 \le \cdots \le r_B$, and ℓ_i, r_i are $O(w)$-bit rationals. In $O(B^2)$ time, we can construct a data structure, such that a query for a point between segments i and $i+1$ is supported in time $O\left(1 + \frac{B}{w}\left(\lg \frac{\ell_B - \ell_1}{\ell_{i+2} - \ell_{i-1}} + \lg \frac{r_B - r_1}{r_{i+2} - r_{i-1}}\right)\right)$.*

Proof. For convenience, if s is the i-th segment in S, let $\ell(s) = \ell_i$ and $r(s) = r_i$. As before, we start by applying Observation 2.3, with parameters $n = b = B$ and $h = \Theta(w/B)$. Then, in time $O(B)$ we select a subset of at most B segments $s_0, s_1, \ldots$ from S, such that:

(1) for each i, at least one of the following holds:
- (1a) there is at most one segment in S between s_i and s_{i+1}.
- (1b) $\ell(s_{i+1}) - \ell(s_i) \le (\ell_B - \ell_1)/2^h$.
- (1c) $r(s_{i+1}) - r(s_i) \le (r_B - r_1)/2^h$.

(2) there exist segments $\tilde{s}_0, \tilde{s}_2, \ldots$ cutting across the slab, satisfying all of the following:
- (2a) $s_0 \prec \tilde{s}_0 \prec s_2 \prec \tilde{s}_2 \prec \cdots$, where $\prec$ denotes the belowness relation.
- (2b) distances between the left endpoints of the $\tilde{s}_i$'s are all multiples of $(\ell_B - \ell_1)/2^h$.
- (2c) distances between right endpoints are all multiples of $(r_B - r_1)/2^h$.

As shown in Section 2.3, for our choice $h = \Theta(w/B)$, we can pack $\tilde{s}_0, \tilde{s}_2, \ldots$ in a word, and perform point location among them in constant time. If we know the query lies between

$\tilde{s}_i$ and $\tilde{s}_{i+2}$, by (2a) it must lie between s_i and s_{i+4}. In constant time, we can compare the point against these 5 segments. Assume that it is found to lie between s_j and s_{j+1}. If case (1a) happens, we compare the query to one more segment and point location is completed.

In cases (1b) or (1c), we recurse among the segments in S between s_j and s_{j+1}. In the preprocessing phase, we build a data structure recursively for every such interval of segments from S. Because in every recursive step we eliminate at least two segments, each original segment appears in $O(B)$ nodes of the recursion tree, so the total cost is $O(B^2)$.

At query time, every recursive step takes constant time, and reduces either the left (case 1(b)) or right (case (1c)) span of the remaining segments by a factor of $2^h = 2^{\Theta(w/B)}$. Now assume the final answer is that the query lies between original segments i and $i+1$. After

$$\frac{B}{w}\left(\lg\frac{\ell_B - \ell_1}{\ell_{i+2} - \ell_{i-1}} + \lg\frac{r_B - r_1}{r_{i+2} - r_{i-1}}\right)$$

steps, the subset we are left with cannot include *both* segments $i-1$ and $i+2$, because either the left or right span is too small. Then in at most one additional recursion, we are done. $\square$

We will think of $\lg(\ell_B - \ell_1) + \lg(r_B - r_1)$ as the entropy of the search region. If the above data structure takes time t for a query, the entropy decreases by at least $\frac{w}{B}(\Omega(t) - 1)$ bits. Thus, we can hope that the sum of the running times for $\log_B n$ applications of the lemma is bounded by $(1+t_1) + (1+t_2) + \cdots + (1 + t_{\log_B n}) \le O(\log_B n + \lg u / \frac{w}{B}) = O(\log_B n + B)$, implying a running time of $O(\lg n / \lg\lg n)$ for, say, $B = \sqrt{\lg n}$. This intuition will indeed prove to be correct, but we need to carefully analyze the geometry of the problem, and show that the information progress is maintained as we query various vertical slabs at various hull-fusion nodes.

6.3 Quantifying Geometric Information

For simplicity, we will only try to determine the right tangent, and assume it lies in the upper convex hull. Left tangents and the lower hull can be handled symmetrically.

We denote an upper convex chain P by its list of vertices from left to right: $P = \langle p_1, p_2, \ldots, p_m\rangle$ where $x(p_i) < x(p_{i+1})$ for all i. Define the *exterior* exterior(P) of an upper convex chain P to be the region bounded by the chain and by the two downward vertical rays emanating from p_1 and p_m that includes points above the chain. In other words, the exterior exterior(P) of P consists of all points left, above, or right of P.

Given an upper convex chain P and indices $1 \le i < j < m$, the *Zorro* $\mathcal{Z}_P(p_i, p_j)$ is the region of points exterior to P, strictly right of the ray from p_{i+1} to p_i, and nonstrictly left of the ray from p_{j+1} to p_j. Thus the Zorro is bounded by these two rays and by the subchain $p_i, p_{i+1}, \ldots, p_j$, as illustrated in Figure 6-1. Note that the Zorro is an object in the infinite real plane, not on the grid.

The following fact justifies our interest in this definition:

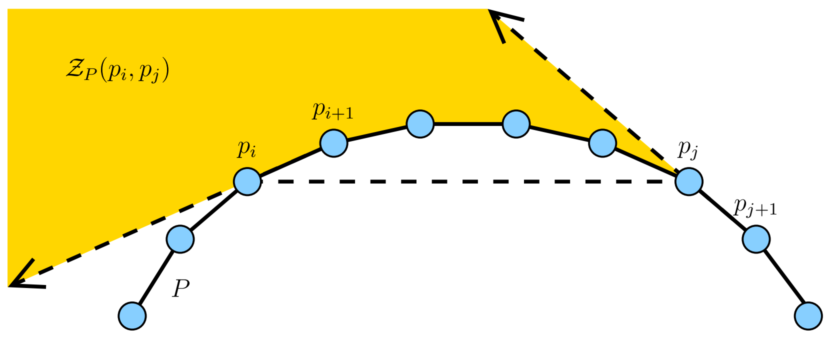

Figure 6-1: The Zorro $\mathcal{Z}_P(p_i, p_j)$ is the shaded region.

Fact 6.3. *Let q be a point exterior to an upper convex chain P. Then q is in the Zorro $\mathcal{Z}_P(p_i, p_j)$ if and only if the answer to the right tangent query for q in P is in $\{p_{i+1}, \ldots, p_j\}$.*

Proof. By definition, for each k with $i+1 \leq k \leq j$, the region of points whose right tangent query answers p_k is a cone emanating from p_k with bounding rays from p_k to p_{k-1} and from p_{k+1} to p_k. This cone is precisely the Zorro $\mathcal{Z}_P(p_{k-1}, p_k)$. Two adjacent such cones, $\mathcal{Z}_P(p_{k-1}, p_k)$ and $\mathcal{Z}_P(p_k, p_{k+1})$, share a bounding ray from p_{k+1} to p_k. Thus their union is $\mathcal{Z}_P(p_{k-1}, p_{k+1})$, so by induction, the union over all k is $\mathcal{Z}_P(p_i, p_j)$. Therefore this Zorro is precisely the region of points whose right tangent query answers one of $p_{i+1}, \ldots, p_j$. □

We also establish a few basic facts that will be useful later:

Fact 6.4. *Given a point q guaranteed to be exterior to an upper convex chain P, we can test whether q is in the Zorro $\mathcal{Z}_P(p_i, p_j)$ in $O(1)$ time.*

Proof. Though the Zorro's boundary is potentially complicated, if q is known to be outside the polygon, it suffices to test the side of q relative to the lines $p_{i+1}p_i$, $p_j p_{j+1}$, and $p_i p_j$ (the dashed lines in Figure 6-1). □

Fact 6.5. *For any upper convex chain P and any indices $1 \leq i < j < k < m$, we have the decomposition: $\mathcal{Z}_P(p_i, p_k) = \mathcal{Z}_P(p_i, p_j) \cup \mathcal{Z}_P(p_j, p_k)$ and $\mathcal{Z}_P(p_i, p_j) \cap \mathcal{Z}_P(p_j, p_k) = \emptyset$.*

Proof. Disjointness follows from Fact 6.3. The union property follows from taking the union of adjacent Zorro cones as argued in the proof of Fact 6.3. □

Because Zorros describe the structure of our search problem, we want to define a quantitative measure that allows us to make the information progress argument outlined above. It turns out that information progress is only need (and, actually, only true) for a region of the Zorro. We define the *left slab* of $\mathcal{Z}_P(p_i, p_j)$ as the vertical slab between $x = 0$ and $x = x(p_{i+1})$. The *left vertical extent* $\mathcal{L}(\mathcal{Z}_P(p_i, p_j))$ is the length of the subsegment of the vertical line $x = 0$ intersected by $\mathcal{Z}_P(p_i, p_j)$. The *right vertical extent* $\mathcal{R}(\mathcal{Z}_P(p_i, p_j))$ is the length of the subsegment of the vertical line $x = x(p_{i+1})$ intersected by the Zorro.

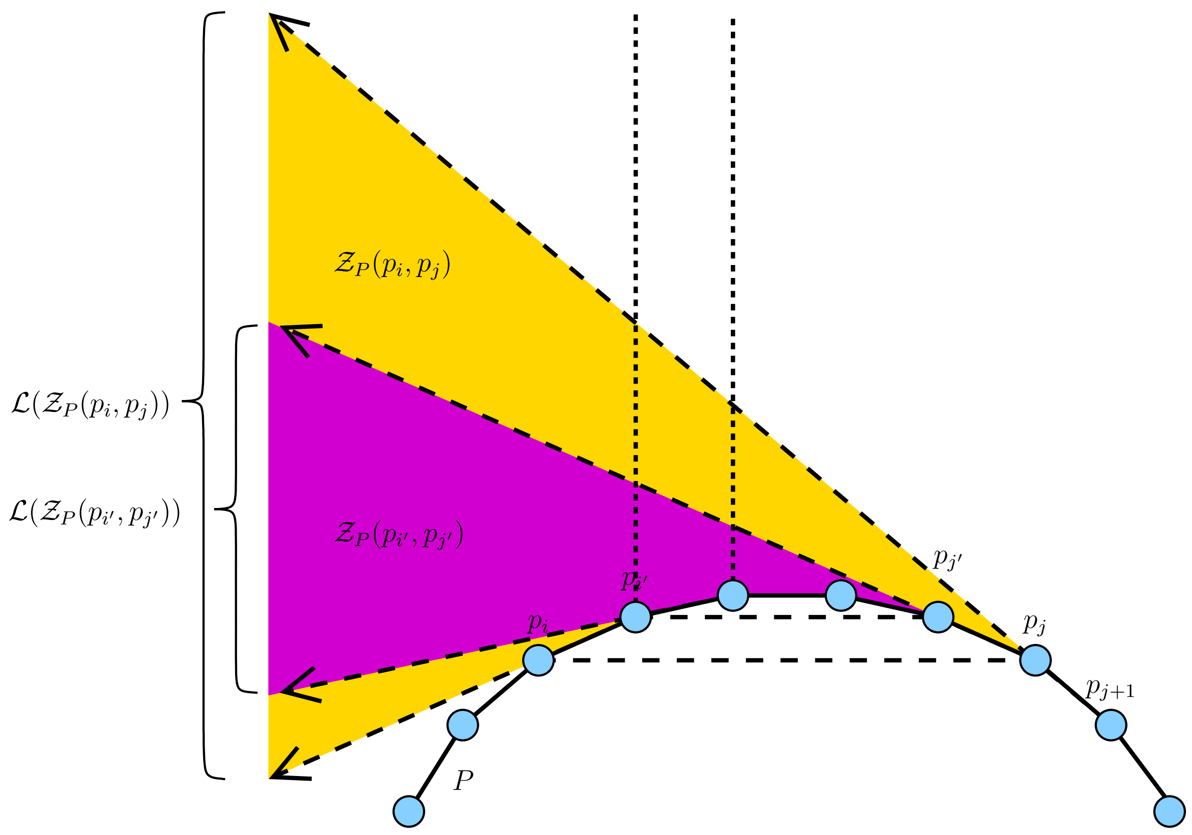
$\mathcal{Z}_P(p_i, p_j)$
$\mathcal{L}(\mathcal{Z}_P(p_i, p_j))$
$\mathcal{L}(\mathcal{Z}_P(p_{i'}, p_{j'}))$
$\mathcal{Z}_P(p_{i'}, p_{j'})$
$p_{j'}$
$p_{i'}$
p_i
p_j
p_{j+1}
P

Figure 6-2: The Zorro $\mathcal{Z}_P(p_{i'}, p_{j'})$ is contained in $\mathcal{Z}_P(p_i, p_j)$.

Definition 6.6. *The* entropy *of a Zorro $\mathcal{Z}_P(p_i, p_j)$ is:*

$$H(\mathcal{Z}_P(p_i, p_j)) = \lg \mathcal{L}(\mathcal{Z}_P(p_i, p_j)) + \lg \mathcal{R}(\mathcal{Z}_P(p_i, p_j)).$$

We now establish the following monotonicity property about Zorros "contained" in other Zorros:

Fact 6.7. *For an upper convex chain P and indices $1 \le i \le i' < j' \le j < m$, we have $\mathcal{Z}_P(p_{i'}, p_{j'}) \subseteq \mathcal{Z}_P(p_i, p_j)$ and $H(\mathcal{Z}_P(p_{i'}, p_{j'})) \le H(\mathcal{Z}_P(p_i, p_j))$.*

Proof. Refer to Figure 6-2. Fact 6.5 immediately implies that $\mathcal{Z}_P(p_{i'}, p_{j'}) \subseteq \mathcal{Z}_P(p_i, p_j)$. This geometric containment implies $\mathcal{L}(\mathcal{Z}_P(p_{i'}, p_{j'})) \le \mathcal{L}(\mathcal{Z}_P(p_i, p_j))$, because $\mathcal{Z}_P(p_{i'}, p_{j'}) \cap \{x = 0\} \subseteq \mathcal{Z}_P(p_i, p_j) \cap \{x = 0\}$. The segments at the intersection with $x = x(p_{i+1})$ are similarly contained. Furthermore, $x(p_{i'+1}) \ge x(p_{i+1})$. Because we are working with an upper chain, moving to the right can only decrease vertical extents, so $\mathcal{R}(\mathcal{Z}_P(p_{i'}, p_{j'})) \le \mathcal{R}(\mathcal{Z}_P(p_i, p_j))$. $\square$

Finally, we need to analyze Zorros with respect to a subset of the original chain, because in one step, we plan to analyze only B points out of the hull. The following fact follows from monotonicity of slopes along a convex chain, similarly to previous facts:

Fact 6.8. *For an upper convex chain S and a subsequence $P \subseteq S$ of m vertices, and for indices $1 < i < j < m$, we have:*

$$\mathcal{Z}_P(p_i, p_{j-1}) \cap \text{exterior}(S) \subseteq \mathcal{Z}_S(p_i, p_j) \subseteq \mathcal{Z}_P(p_{i-1}, p_j),$$
$$H(\mathcal{Z}_P(p_i, p_{j-1})) \le H(\mathcal{Z}_S(p_i, p_j)) \le H(\mathcal{Z}_P(p_{i-1}, p_j)).$$

6.4 The Data Structure

We first reinterpret the results of Lemma 6.2 in the language of Zorros. The reason we insist to relate the query time to the entropy of $\mathcal{Z}_P(p_{k-1}, p_{k+2})$, instead of $\mathcal{Z}_P(p_k, p_{k+1})$ is that we will need some slack when switching between Zorros with respect to P, and Zorros with respect to the whole convex hull (see Fact 6.8).

Lemma 6.9. *Given an upper convex chain $P = \langle p_1, p_2, \dots, p_B \rangle$, in time $B^{O(1)}$ we can build a data structure that answers queries of the following form: given indices $1 < i < j < B-1$, and given a point q guaranteed to be within the Zorro $\mathcal{Z}_P(p_i, p_j)$* and *its left slab, find an index $i < k < j$ such that q is in the Zorro $\mathcal{Z}_P(p_k, p_{k+1})$. The running time of the query is: $t = O\big(1 + \frac{B}{w}\big(H(\mathcal{Z}_P(p_i, p_j)) - H(\mathcal{Z}_P(p_{k-1}, p_{k+2}))\big)\big)$.*

Proof. We build the structure for every choice of i and j, incurring an $O(B^2)$-factor penalty in construction time. For some fixed i and j, we need to solve a point location problem in the left slab of $\mathcal{Z}_P(p_i, p_j)$, with the segments given by the intersection of the slab with the rays $p_{i+1}p_i, p_{i+2}p_{i+1}, \dots, p_{j+1}p_j$. A Zorro $\mathcal{Z}_P(p_k, p_{k+1})$ is actually the wedge between two consecutive rays.

Denote by $\ell_i, \dots, \ell_j$ the y coordinate of the intersections of these rays with $x = 0$ (the left boundary of the slab). Similarly denote by $r_i, \dots, r_j$ the intersections with the right boundary of the slab. Note that ℓ_i, r_i are rational.

We will now use the adaptive point location data structure of Lemma 6.2. If the answer is k, the query time will be asymptotically:

$$\begin{aligned} t &\approx 1 + \frac{B}{w}\left(\lg \frac{\ell_j - \ell_i}{\ell_{k+2} - \ell_{k-1}} + \lg \frac{r_j - r_i}{r_{k+2} - r_{k-1}}\right) \\ &= 1 + \frac{B}{w}\big(\lg \mathcal{L}(\mathcal{Z}_P(\ell_i, \ell_j)) + \lg \mathcal{R}(\mathcal{Z}_P(\ell_i, \ell_j)) - \lg \mathcal{L}(\mathcal{Z}_P(\ell_{k-1}, \ell_{k+2})) - \lg(r_{k+2} - r_{k-1})\big) \\ &\le 1 + \frac{B}{w}\big(H(\mathcal{Z}_P(\ell_i, \ell_j)) - H(\mathcal{Z}_P(p_{k-1}, p_{k+2}))\big) \end{aligned}$$

The last inequality follows from $x(p_k) \ge x(p_{i+1})$, using the familiar observation that moving to the right reduces vertical extents. □

The general structure in which we will be performing queries is a B-tree representation of an upper convex chain. For a node v of such a B-tree, let S_v be the set of points in v's subtree, and P_v the set of at most B points stored in the node v. For the sake of queries, each node is augmented with the following information:

- an atomic heap [FW94] for the x coordinates of the points in P_v.
- the structure of Lemma 6.9 for the convex chain given by P_v.

6.4.1 Query Invariants

A tangent query proceeds down a root-to-leaf path of the B-tree, spending $O(1)$ time at each node but also applying Lemma 6.9 at some of the nodes. Therefore the time required by a query is $O(\log_B n)$ plus the total time spent in Lemma 6.9.

For readability, we will write $\mathcal{Z}_v(a,b)$ for $\mathcal{Z}_{S_v}(a,b)$. At each recursive step of the query, we have $q \in \mathcal{Z}_v(a,b)$ where v is the current node and $a, b \in S_v$. We write $\mathrm{succ}_v(b)$ for the point in S_v which follows b to the right on the upper convex chain. We also assume $\mathrm{succ}_v(b) \in S_v$, to make the Zorro well-defined. In addition, we maintain the invariant that a, b, and $\mathrm{succ}_v(b)$ are nodes of the global upper convex hull C as well. Thus $\mathcal{Z}_v(a,b) = \mathcal{Z}_C(a,b)$.

At the next level of recursion, the Zorro will be some $\mathcal{Z}_{v'}(a',b')$ where v' is a child of v and $a', b', \mathrm{succ}_{v'}(b') \in S_{v'} \cap C$. Furthermore, we will guarantee that $x(a) \le x(a') < x(b') \le x(b)$. Hence, by Fact 6.7, $Z_{v'}(a',b') = \mathcal{Z}_C(a',b') \subseteq \mathcal{Z}_C(a,b)$, and $H(\mathcal{Z}_C(a',b')) \le H(\mathcal{Z}_C(a,b))$.

The query may apply Lemma 6.9 at this recursive step to a Zorro $\mathcal{Z}_{P_v}(p_i,p_j)$, locating q in a Zorro $\mathcal{Z}_{P_v}(p_k,p_{k+1})$. In this case, we guarantee further that

$$\mathcal{Z}_v(a,b) \supseteq \mathcal{Z}_{P_v}(p_i,p_j) \cap \mathrm{exterior}(C) \supseteq \mathcal{Z}_{P_v}(p_{k-1},p_{k+2}) \cap \mathrm{exterior}(C) \supseteq \mathcal{Z}_{v'}(a',b'),$$
$$H(\mathcal{Z}_v(a,b)) \ge H(\mathcal{Z}_{P_v}(p_i,p_j)) \ge H(\mathcal{Z}_{P_v}(p_{k-1},p_{k+2})) \ge H(\mathcal{Z}_{v'}(a',b')).$$

Now we bound the total cost incurred by Lemma 6.9. By the invariants stated above, $H(Z)$ never increases as we shrink our Zorro Z known to contain q. Furthermore, when we apply Lemma 6.9, if we spend t time, we guarantee that $H(Z)$ decreases by $\frac{w}{B}(\Omega(t)-1)$. Hence the total cost incurred by Lemma 6.9 is at most the maximum total range of $H(\cdot)$, divided by $\frac{w}{b}$. Because the points are on a $u \times u$ grid, any nonzero vertical extent, measured at an x coordinate of the grid, between two lines drawn between grid points, is between $1/u$ and u. Thus, $-2\lg u \le H(\cdot) \le 2\lg u$. Because $w \ge \lg u$, the total cost incurred by Lemma 6.9 is $O(B)$. Therefore the total query time is $O(\log_B n + B) = O(\lg n/\lg\lg n)$, by choosing $B = \lg^{\varepsilon} n$, for any constant $\varepsilon > 0$.

6.4.2 Querying a Hull-Fusion Node

We now describe how to implement a query using $O(1)$ time at each node plus possibly one application of Lemma 6.9, while satisfying all of the invariants described above. First we apply the following lemma:

Lemma 6.10. *Given a node v with $P_v = \langle p_1, p_2, \ldots, p_m\rangle$, given two points a and b on v's hull where $x(a) < x(b)$, and given a query point $q \in \mathcal{Z}_v(a,b) \cap \mathrm{exterior}(C)$ we can find in $O(1)$ time one of the following outcomes:*

1. *An index $1 \le k < m-1$ such that q is in the Zorro $\mathcal{Z}_v(p_k,p_{k+1})$.*
2. *Indices $1 < i < j < m-1$ such that $q \in \mathcal{Z}_{P_v}(p_i,p_j)$ and $x(a) \le x(p_i) < x(p_{j+1}) \le x(b)$.*

Proof. We round a to the clockwise next bridge point $p_i \in P_v$, and we round b to the counterclockwise next bridge point $p_j \in P_v$. The implementation of this depends on the representation of points, so we defer a discussion of this step until later. We will be able to support this step in constant time.

If $i > j$, then $q \in \mathcal{Z}_v(a, b) \subseteq \mathcal{Z}_v(p_j, p_{j+1})$ (Case 1). Otherwise, $1 < i \leq j < m$. By Fact 6.5, $\mathcal{Z}_v(a, b) = \mathcal{Z}_v(a, p_i) \cup \mathcal{Z}_v(p_i, p_{j-1}) \cup \mathcal{Z}_v(p_{j-1}, p_j) \cup \mathcal{Z}_v(p_j, b)$. In $O(1)$ time, we can determine which of these Zorros contains q. If it is the first Zorro, $q \in \mathcal{Z}_v(a, p_i) \subseteq \mathcal{Z}_v(p_{i-1}, p_i)$ (Case 1). If it is the second Zorro, $q \in \mathcal{Z}_v(p_i, p_{j-1})$ (Case 2). If it is the third Zorro, $q \in \mathcal{Z}_v(p_{j-1}, p_j)$ (Case 1). If it is the fourth Zorro, $q \in \mathcal{Z}_v(p_j, b) \subseteq \mathcal{Z}_v(p_j, p_{j+1})$ (Case 1). □

Now, if we are in Case 1, say $q \in \mathcal{Z}_v(p_k, p_{k+1})$, then we know to recurse into the recursive subchain between p_k and p_{k+1}, corresponding to some child v'. In this case, we want to recurse with $a' = \max\{a, p_k\}$ and $b' = \min\{b, \operatorname{pred}_{v'}(p_{k+1})\}$ (where max and min are with respect to x coordinates). Thus $x(a) \leq x(a') < x(b') \leq x(b)$, satisfying the guarantee above. Before recursing, however, we check in $O(1)$ time whether $q \in \mathcal{Z}_{v'}(a', b')$; if not, we determine the answer to the right-tangent query to be p_{k+1}.

If we are in Case 2, say $q \in \mathcal{Z}_{P_v}(p_i, p_j)$ where $x(a) \leq x(p_i) < x(p_{j+1}) \leq x(b)$, then there are two subcases. If q is not in the left slab of the Zorro $\mathcal{Z}_{P_v}(p_i, p_j)$, then we perform a successor query $x(q)$ among the x coordinates of the bridge points P_v to find an index i', $i < i' \leq j$, such that $x(p_{i'-1}) \leq x(q) < x(p_{i'})$. Next we test in $O(1)$ time whether q is in the Zorro $\mathcal{Z}_{P_v}(p_{i'}, p_j)$. If not, we know to recurse in the recursive chain between $p_{i'-1}$ and $p_{i'}$, and we proceed as in Case 1. Otherwise, we determine that q is in the left slab of $\mathcal{Z}_{P_v}(p_{i'}, p_j)$, so we replace i with i' to obtain the other subcase.

So now suppose q is in the left slab of the Zorro $\mathcal{Z}_{P_v}(p_i, p_j)$ where $x(a) \leq x(p_i) < x(p_{j-1}) \leq x(b)$. We can apply Lemma 6.9 to obtain a Zorro $\mathcal{Z}_{P_v}(p_k, p_{k+1})$ containing q. By Fact 6.8, $\mathcal{Z}_{P_v}(p_k, p_{k+1}) \cap \operatorname{exterior}(C) \subseteq \mathcal{Z}_v(p_k, p_{k+2})$. By Fact 6.5, $\mathcal{Z}_v(p_k, p_{k+2}) = \mathcal{Z}_v(p_k, p_{k+1}) \cup \mathcal{Z}_v(p_{k+1}, p_{k+2})$. In $O(1)$ time, we can determine which of these three Zorros contains q, and recurse in the corresponding child v' as in Case 1 with Zorro $\mathcal{Z}_{v'}(a', b')$.

Finally we prove the guarantees about Zorro containment and entropy monotonicity. By Fact 6.8, we have:

$$\begin{aligned} \mathcal{Z}_{P_v}(p_i, p_j) \cap \operatorname{exterior}(C) &\subseteq \mathcal{Z}_v(p_i, p_{j+1}), \\ H(\mathcal{Z}_{P_v}(p_i, p_j)) &\leq H(\mathcal{Z}_v(p_i, p_{j+1})). \end{aligned}$$

Because $x(a) \leq x(p_i) < x(p_{j+1}) \leq x(b)$, $\mathcal{Z}_v(p_i, p_{j+1}) \subseteq \mathcal{Z}_v(a, b)$, and by Fact 6.7, $H(\mathcal{Z}_v(p_i, p_{j+1})) \leq H(\mathcal{Z}_v(a, b))$. Thus, $H(\mathcal{Z}_{P_v}(p_i, p_j)) \leq H(\mathcal{Z}_v(a, b))$ as desired. On the other hand, by Fact 6.8:

$$\begin{aligned} \mathcal{Z}_v(p_k, p_{k+2}) &\subseteq \mathcal{Z}_{P_v}(p_{k-1}, p_{k+2}), \\ H(\mathcal{Z}_v(p_k, p_{k+2})) &\leq H(\mathcal{Z}_{P_v}(p_{k-1}, p_{k+2})). \end{aligned}$$

By Fact 6.7 and because $x(p_k) \leq x(a') < x(b') \leq x(p_{k+2})$, we have the desired result:

$$\begin{aligned} \mathcal{Z}_{v'}(a', b') &\subseteq \mathcal{Z}_v(p_k, p_{k+2}) \subseteq \mathcal{Z}_{P_v}(p_{k-1}, p_{k+2}), \\ H(\mathcal{Z}_{v'}(a', b')) &\leq H(\mathcal{Z}_v(p_k, p_{k+2})) \leq H(\mathcal{Z}_{P_v}(p_{k-1}, p_{k+2})) \end{aligned}$$

6.4.3 Updates

It remains to describe how we maintain a B-tree with the upper convex hull, as used by the query. A straightforward approach is to only modify the representation of the convex hulls at each node of the Overmars-van Leeuwen structure, storing these as persistent catenable B-trees. Because we do not use parent pointers, we can use standard persistence techniques [DSST89]. Unfortunately, however, a catenable B-tree rebuilds $O(\log_B n)$ nodes per update. Rebuilding a node takes $B^{O(1)}$ time, because we must rebuild the associated data structure of Lemma 6.9. Finally, the Overmars-van Leeuwen structure performs $O(\lg n)$ splits and joins, so the total update time is $O(\lg^2 n \frac{B}{\lg B}) = O(\lg^{2+\varepsilon} n)$.

To reduce updates to $O(\lg^2 n)$, we abandon persistence, and build the query B-tree in close parallel to the Overmars-van Leeuwen tree. This has a similar flavor to the original approach of Overmars and van Leeuwen [OvL81], which was developed before persistence was known.

Each node of the Overmars-van Leeuwen tree discovers one bridge (because we are only dealing with the upper hull), and two *bridge points* that define it. We compress the bridge points from all nodes on $\lg B - 2$ consecutive levels of the Overmars-van Leeuwen tree into one node of our B-tree. This means a B-tree node will store $2 \cdot (2 \cdot 2^{\lg B - 2} - 1) = B - 2 < B$ points. The depth will be $O(\lg n / \lg B)$. Note, however, that this "B-tree" is not necessarily balanced with respect to the values it stores (the nodes on the hull), but is balanced with respect to the original set of points, closely following the balance of the Overmars-van Leeuwen tree.

An update can only change bridge points on a root-to-leaf path in the Overmars-van Leeuwen tree. This means that only $O(\log_B n)$ nodes of the B-tree are changed, and we can afford to rebuild the associated structures for all of them. This takes time $\log_B n \cdot B^{O(1)} = \lg^{1+\varepsilon} n$, which is a lower-order term.

Finally, we must augment each node to support the constant-time operation we have assumed: given a point on the node's hull, round it to the next bridge point. Since the upper hull is sorted by x coordinates, it suffices to store an atomic heap [FW94] for the x coordinates of the bridge points. Maintaining the atomic heap is a lower order term compared to reconstructing our hull fusion data structure at each node.

Chapter 7

Open Problems

We now turn to a collection of fundamental problems in computational geometry that our results bring to light. We believe further examination of these problems from the point of view of geometric information could reveal additional structure on the problem in particular and geometric information in general. We divide the open problems into three major categories: *algorithms* (compute an entire structure such as the Voronoi diagram, or bulk offline queries), *static data structures* (preprocess the geometry to support fast queries), and *dynamic data structures* (maintain geometry subject to fast updates and queries). Computational geometry is ripe with relations between such categories of problems: many data structures have been invented solely because they resulted in an optimal geometric algorithm. But under this new vantage point, we need to reconsider whether such reductions still produce optimal solutions, or whether additional structure and separations result.

7.1 Open Problems in Algorithms

Offline point location. Our current best solution to this problem runs in $O(m) + n \cdot 2^{O(\sqrt{\lg \lg m})}$ time, given m segments and n points. Intuitively, the bound does not "look" optimal, but we have so far failed to improve it.

Open Problem 1. *Can offline planar point location be solved with running time $n(\lg \lg n)^{O(1)}$? Does randomization help?*

A faster randomized algorithm would be particularly interesting, as it would be the first improvement through randomization of a search strategy in geometry. There is ample precedent for randomization offering simpler optimal solutions to geometric problems, such as randomized incremental construction, but randomization also plays an important role in the current state-of-the-art in integer sorting [HT02].

On the other hand, it seems unlikely that planar point location can be solved in linear time. So far, sorting has been essentially the only search problem with complexity between $\omega(n)$ and $o(n \lg n)$. Having point location as a second example with radically different properties might shed some light into the area. One can hope for lower bounds under a natural restriction on the model, perhaps obtained by examining the common features between sorting and planar point location.

Voronoi diagrams. For computing the Voronoi diagram in the plane, or more generally the convex hull in 3-d, we obtain a randomized reduction to offline point location. Although frequently used in tandem with point location, Voronoi diagrams and 3-d convex hulls are important structures in their own right, and it is natural to wonder whether they are in fact easier problems than offline planar point location. The additional structure offered by Voronoi diagrams may make the problem easier to solve. Alternatively, and perhaps more likely, one may be able to find a reverse reduction, converting any offline planar point location problem into an equivalent 2-d Voronoi or 3-d convex hull problem.

Open Problem 2. *Is computing a Voronoi diagram as hard as offline planar point location?*

Another direction to pursue is whether the random-sampling reduction from 2-d Voronoi and 3-d convex hull to offline planar point location can be made deterministic via derandomization. Unless we find a faster solution to offline planar point location that exploits randomization, it is unsatisfying to require randomization just for this reduction. Instead of derandomizing the reduction directly, an alternative is to attempt to apply our point-location techniques directly to constructing Voronoi diagrams, and therefore obtain a deterministic time bound.

Open Problem 3. *Can Voronoi diagrams be computed in $o(n \lg n)$ time without randomization?*

Line-segment intersection. The segment intersection problem comes in three flavors: *reporting* (list all intersections), *counting* (count intersections) and *existential* (does there exist an intersection?). It is reasonable to assume the existential and reporting problems behave the same way, up to an additive cost in the output size. For these problems, we may ask the same questions as for constructing Voronoi diagrams: Can we beat offline point location? Can we obtain $o(n \lg n)$ bounds without randomization?

The general counting problem is suspected to have complexity n^α, for $\alpha > 1$, making it uninteresting from the point of view of our study. Consider, however, *red-blue intersection counting*: given n line segments, each marked red or blue, count the number of intersections between red and blue. It is guaranteed that red, respectively blue, segments do not intersect among themselves. This problem can be solved in $O(n \lg n)$ time, and it is the only "natural" problem of this complexity for which our techniques do not offer any improvement.

Note, however, that in this case we cannot hope to achieve bounds similar to offline point location. This problem is at least as hard as counting inversions in a sequence of integers, for which it is a long-standing open question to obtain a running time better than $O(n \lg n / \lg \lg n)$. Unfortunately, we do not know how to mimic even this running time for the geometric problem.

Open Problem 4. *Is there a $o(n \lg n)$-time algorithm for red-blue line-segment intersection counting?*

Note that the same problem can be easily solved in the orthogonal case, by using an optimal partial-sums data structure [PD06].

7.2 Open Problems in Static Data Structures

Next we turn to data structural open problems where the data is static but queries come online. Because we need an instant answer for each query, we cannot expect the same kind of speedup as for offline queries. Indeed, here there are many more interesting questions about lower bounds, building on the field of 1-d data structural lower bounds.

Planar point location. Perhaps the central open problem suggested by our work is:

Open Problem 5. *How fast can point-location queries be supported by a linear-space data structure on n points?*

Improving the present results would require the development of new techniques in finite-precision geometric algorithms. On the other hand, the present bounds are fairly natural, and may well turn out to be the correct answer. In this case, we would aim to find a matching lower bound. Such a lower bound would be particularly interesting because it would be fundamentally geometric, being the first lower bound for static data structures that is not an immediate consequence of known one-dimensional lower bounds. Specifically, the point of comparison here is the predecessor problem (the one-dimensional analog of point location). We [PT06] recently established the optimal query bound for a linear-space predecessor data structure to be roughly $\Theta(\min\{\lg n/\lg w, \lg w\})$. This bound differs substantially from our point-location bound in the second term, roughly a square root versus a logarithm.

A somewhat easier question in this context is whether one can prove that standard techniques for the predecessor problem, such as the sketching approach of fusion trees, cannot generalize to the multidimensional setting (in some model). For example, it seems that there is no data structure supporting planar point location queries among w^ε segments in constant time, thus making impossible the black box of a fusion-tree node.

Nearest neighbor. In traditional computational geometry and the algebraic computation tree model, this problem requires $\Omega(\lg n)$ time in the worst case, so it is optimal to reduce it to planar point location. But from the finite-precision perspective, we may be able to circumvent any lower bounds for planar point location in the special case of finding nearest neighbors. Of course, the lower bounds of the predecessor problem still apply here, but it is possible that the difficulty of nearest neighbor falls strictly in between predecessor and planar point location.

Open Problem 6. *How fast can nearest-neighbor queries be supported by a linear-space data structure on n points?*

Alternatively, we may establish lower bounds for nearest neighbor stronger than predecessor. For example, if we can establish such strong lower bounds for planar point location, the argument may be transferable to the special case of nearest neighbors. Such lower bounds would give the first formal sense in which *approximate* nearest neighbor, which reduces to a one-dimensional predecessor problem [Cha02], is asymptotically easier to solve than exact nearest neighbors.

Connectivity. A subtle variation on point location is *planar point connectivity*: preprocess a planar map subject to queries for whether two given points are in the same face of the map. Obviously, this problem reduces to planar point location: determine the face containing the two points and test for equality. But it is possible that planar point connectivity can be solved faster than not only planar point location, but possibly even the predecessor problem. The analogy here is to one-dimensional range reporting (is there a point in a given interval?), for which bounds substantially better than predecessor search are known [MPP05].

Open Problem 7. *Can static data structures for planar point connectivity outperform those for planar point location?*

This problem arises in a natural setting studied by Thorup (personal communication). Specifically, an efficient data structure for connectivity is essentially what is needed to obtain efficient approximate distance oracles. Here we would like to know the approximate number of faces (dual vertices) along the best path between two query points in a static planar map.

Polygons. On the simpler side, we can think of queries about just a single convex polygon in the plane. As noted in Corollary 5.1, tangent queries, and the dual line-stabbing queries, can be solved through point location. However, the special convex structure may make the complexity of the problem fall strictly in between point location and predecessor search.

Open Problem 8. *How fast can tangent queries be supported by a static data structure on a convex n-gon?*

7.3 Open Problems in Dynamic Data Structures

Our final collection of open problems is also the most challenging: maintain geometric data subject to both queries and updates. The techniques for static data structures do not generally apply, requiring the development of new techniques.

Convex hull. Our lower bound for dynamic convex hull queries (Appendix B) is $\Omega(\lg n/\lg w)$, given polylog(n) update time. While we achieve this query time for some of the simpler queries (e.g. linear programming), for the hardest queries (e.g. tangent queries) the running time is $O(\lg n/\lg\lg n)$ regardless of precision. To the best of our knowledge, this running time independent of w is rather unique. If this distinction is indeed real, it separates convex hull queries, which would be interesting. Furthermore, this would separate dynamic geometric search (tangent queries) from the related 1-d search problems, which have $O(\log_w n)$ upper bounds.

Open Problem 9. *What is the optimal complexity of tangent queries, given update time* polylog(n)*?*

Of course, another natural question is whether the updates can be improved to logarithmic or even sublogarithmic while maintaining fast queries. In principle, we do not know of any lower bounds that prevent, say, an $O(\lg\lg n)$ update time, although this seems unlikely. Resolving this issue will likely require the development of new lower bound techniques. Achieving an upper bound of $o(\lg^2 n)$ may be a rather difficult question, given the sublogarithmic queries seem to prohibit the single known idea for breaking the $O(\lg^2 n)$ barrier.

Open Problem 10. *What is the optimal update time for dynamic convex hull while supporting $O(\lg n/\lg\lg n)$ queries?*

Point location. Achieving $O(\lg n)$ bounds for dynamic point location is a well-known open problem. Current bounds revolve around $O(\lg n \lg\lg n)$. However, given our bounded-precision perspective, it is not even clear that $O(\lg n)$ is the right goal.

Open Problem 11. *Can dynamic planar point location be solved with $o(\lg n)$ time per operation?*

Plane graphs. Part of dynamic planar point location is the maintenance of a planar map subject to insertion and deletion of points and edges. Two other natural queries on such dynamic planar maps are (a) are two given edges on the same face? and (b) are two vertices in the same connected component of the graph? The latter query is a natural analog of dynamic connectivity in graphs. It is known that dynamic connectivity requires $\Omega(\lg n)$ time per operation, even when the graph is planar. However, the lower bound does not preserve the planar embedding of the graph; it can change drastically in each step. When the user must explicitly maintain the embedding by modifying the points that define vertices, the problem may be substantially easier.

Open Problem 12. *Can dynamic connectivity in plane graphs be solved in sublogarithmic time per operation?*

If the answer to this question is positive, we would have a nice contrast where the two-dimensional problem is actually easier than the motivating data structural problem: geometry helps. Even if dynamic connectivity is not possible in sublogarithmic time, the simper query of testing whether two edges bound the same face may be possible, cutting a finer line through the problem space.

Appendix A

Results in Higher Dimensions

A.1 Online Point Location

The sublogarithmic point location algorithm from Chapter 2 can be generalized to any constant dimension d. The main observation is very similar to Observation 2.3:

Observation A.1. *Let S be a set of n disjoint $(d-1)$-dimensional simplices in $\Re^d$, 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 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 2.3, 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 $x_d = 0$. 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 2.3, 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 2.3, by performing the required arithmetic operations on $O(h)$-bit integers in parallel, using a constant number of arithmetic operations on w-bit integers.

Proposition A.2. *Given a sorted list of n disjoint $(d-1)$-simplices spanning a vertical prism in $\Re^d$ with $O(w)$-bit rational coordinates, we can build a data structure in $O(n)$ time and space, so that point location queries can be answered in $t(n) := O(\lg n/\lg\lg n)$ time.*

A.2 Offline Point Location

Observation A.1 works equally well with the offline algorithm from Section 3.2. By contrast, the variation in Section 3.3 does not seem to generalize to higher dimensions, because there is no equivalent of a central slab. Thus, the trick that allowed us to use only standard operation in 2-d does not help us in higher dimensions.

Remember that the offline algorithm needs to apply the projective transform in parallel to points packed in a word. It does not seem possible to implement this in constant time using standard word RAM operations (since, according to the formula for projective transform, this operation requires multiple divisions where the divisors are all different). We will instead simulate the special operation in slightly superconstant time. We can use the circuit simulation results of Brodnik et al. [BMM97] to reduce the operation to $\lg w \cdot (\lg\lg w)^{O(1)}$ standard operations. Since precision is generally polylogarithmic ($w \leq \text{polylog}(n)$), we tend to think of this slowdown per operation as very small. We obtain:

Proposition A.3. *Given a sorted list of m disjoint $(d-1)$-simplices spanning a vertical prism in $\Re^d$, and n points in the prism, we can sort the points relative to the simplices in time $O(m) + n \cdot 2^{O(\sqrt{\lg\lg m})} \lg^{1+o(1)} w$.*

A.3 Applications in Higher Dimensions

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 A.4. *Let $t(n)$ be as in Proposition A.2, and assume d is fixed.*

(a) *We can solve the point location problem for any subdivision of $\Re^d$ into polyhedral cells, with $n^{O(1)}$ space and preprocessing time, and $O(t(n))$ query time.*

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

(c) *We can preprocess a fixed polyhedral robot and a polyhedral environment with n facets in $\Re^d$ with $O(w)$-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 $O(t(n))$ time.*

(d) *Given an arrangement of n semialgebraic sets of the form $\{x \in \Re^d \mid p_i(x) \geq 0\}$ 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 $O(t(n))$ time.*

(e) *(Point location in 2-d among curved segments.) Given n disjoint x-monotone curve segments 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 $O(t(n))$ time.*

(f) *Part* (b) *also holds under the ℓ_p metric for any constant integer $p > 2$.*

(g) *We can preprocess a polyhedral environment with n facets in $\Re^d$ with $O(w)$-bit rational coordinates, in $n^{O(1)}$ time and space, so that ray shooting queries (finding the first facet hit by a ray) can be answered in $O(t(n))$ time.*

(h) *We can preprocess a convex polytope with n facets in $\Re^d$ with $O(w)$-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 $O(t(n))$ time.*

Proof. (a) For a naïve solution, we project all $(d-2)$-faces vertically to $\Re^{d-1}$, triangulate the resulting arrangement in $\Re^{d-1}$, lift each cell to form a vertical prism, and build the data structure from Proposition A.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 $O(t(n))$ for any fixed d.

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

(c) This reduces to point location in the arrangement formed by the Minkowski difference [dBSvKO00, O'R98] of the environment with the robot.

(d) 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.

(e) This is just a 2-d special case of (d).

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

(g) Parametrize the query ray $\{x + ty \mid t \geq 0\}$ with $2d$ variables $x, y \in \Re^d$. Suppose that the facets are defined by hyperplanes $\{x \in \Re^d \mid a_i \cdot x = 1\}$. The "time" the ray hits such a 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 $\{(x, y) \in \Re^{2d} \mid (1 - a_i \cdot x)/(a_i \cdot y) \leq (1 - a_j \cdot x)/(a_j \cdot y)\}$ over all i, j and $\{(x, y) \in \Re^{2d} \mid a_i \cdot x \leq 1\}$ 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.

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

□

We remark that by proceeding exactly as in Section 2.4, we can also obtain a w-sensitive version of all the bounds, with query time $t(n,w) := O\left(\min\left\{\lg n/\lg\lg n,\ \sqrt{w/\lg w}\right\}\right)$.

The preprocessing time and space in Corollary A.4 can be reduced by applying random sampling techniques [Cla88, CS89] like in Section 4.3. 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.

The offline result has a few applications as well, to offline nearest neighbor search in higher dimensions and curve-segment intersection in 2-d. The former follows from (b) above. For the latter, we combine (d) with the same technique as for regular segment intersection, in Section 5.2. Then, intersecting curved segments has the same bound as for straight segments, up to the factor $\lg^{1+o(1)} w$.

Appendix B

A Careful Look at Dynamic Queries

In Chapter 6, we discussed the tangent query in the dynamic convex hull problem, and obtained $O(\frac{\lg n}{\lg \lg n})$ query time, with $O(\lg^2 n)$ update time. This is just one of the many queries that are typically considered in this problem, but it turns out that these results extend to all queries. However, for some queries we can do better, as we describe here. In addition, we describe (almost) tight lower bounds for the query times.

In Section B.1, we discuss and classify the commonly considered queries.

In Section B.3, we prove lower bounds of $\Omega(\lg n / \lg w)$ for the query time of almost all queries, assuming updates (point insertion and deletion) run in polylogarithmic time. The sole exception is the gift-wrapping query (walking the hull), which requires only $\Theta(1)$ time. Our lower bounds are based on a reduction from the classic marked ancestor problem [AHR98]. This result holds in the all-powerful cell-probe model, which just measures the number of memory accesses required per operation, without worrying about any computation costs. Of course, such lower bounds apply to the word RAM, and any computer architecture commonly used today.

This lower bound matches the results of Chapter 6, if the precision is polylogarithmic, and if we are not optimizing the precise update time. In Section B.2, we show how to support a certain subset of the queries in time exactly $O(\lg n / \lg w)$, while also supporting updates in just $O(\lg n \lg \lg n)$ time. This result shows that in essence, some queries are one-dimensional, and can be implemented by a careful combination of results in 1-d search. In particular, this result holds for linear programming (extreme-point queries), but not for tangent queries.

This upper bound starts from the decomposable-search approach introduced by Chan [Cha01a] and refined by Brodal and Jacob [BJ00]. In this structure, it seems impossible to support the most difficult decomposable query, tangent, in the optimal time bound $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$. Essentially, the trade-off we could make in Chapter 6 between a node's query cost and the information it reveals relies on an essentially explicit representation of the convex hull as in the Overmars-van Leeuwen structure. Representing the convex hull as the hull of $O\left(\frac{\lg n}{\lg \lg n}\right)$ overlapping convex hulls, as in the Brodal-Jacob structure, restricts us to optimal implementation of linear-programming queries, which can be viewed as tangent queries for points at infinity. So although the update time is better in this case, the techniques required for optimal query bounds actually become less interesting.

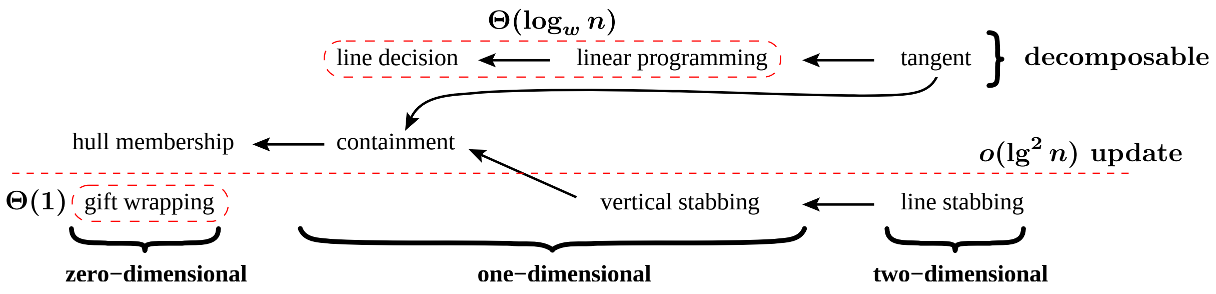

Figure B-1: Dynamic convex hull queries. The queries in the top row are all decomposable; the rest are not. Arrows indicate reducibility between queries: generalization → specialization. Vertically aligned queries are also dual to each other.

Our results therefore illustrate a refined sense of the difficulty of various queries about dynamic planar convex hulls. The challenge with tangent queries is that the input has two geometric degrees of freedom (the coordinates of the query point); thus we call the query *two-dimensional*. On the other hand, linear programming is essentially *one-dimensional*, the input being defined by a single directional coordinate. This distinction is what makes both linear-programming and tangent queries possible in sublogarithmic query time in the explicit structure, but only linear-programming queries possible in the decomposable structure. Our information-theoretic lens therefore highlight an even stronger contrast between the original Overmars-van Leeuwen structure and the more modern structures based on decomposable search: the latter structures are not informationally efficient. It thus remains open to break the $\Theta(\lg^2 n)$ barrier (again) while achieving informationally efficient two-dimensional queries.

B.1 Query Taxonomy

Formally, the *dynamic planar convex hull* problem is to maintain a dynamic set of points, S, subject to insertion and deletion of points and a number of potential queries summarized in Figure B-1. We assume for simplicity of exposition that all x coordinates in S are distinct, as are all y coordinates in S. We classify queries by the number of (continuous) degrees of freedom in their input:

- Zero-dimensional queries, where the input is the discrete set S:

  Gift wrapping: Given a vertex of the convex hull, report the two adjacent vertices of the hull. This is the one query that can be supported in $\Theta(1)$ time, as a direct consequence of applying standard tree-threading and persistence techniques to the Overmars-van Leeuwen structure.

  Hull membership: Test whether a point is on the convex hull.

- One-dimensional queries, which have only one degree of freedom:

  Linear programming: Report the extreme point of the set S in a given direction.

Line decision: Given a line ℓ, test whether it intersects the convex hull. Although this query might seem two-dimensional, in fact it is a decision version of linear-programming queries: it tests whether the extreme points in the direction perpendicular to ℓ are on opposite sides of ℓ.

Vertical line stabbing: Given a vertical line that intersects the convex hull, report the two edges it cuts.

Containment: Report whether a point q is contained in the interior of the convex hull. This query is a decision version of vertical line stabbing, because we only need to test that q is between the two edges that intersect the vertical line. This query is more general than hull membership: applying this query to a perturbation (away from the center of mass) determines whether the point is on the hull.

- Two-dimensional queries:

Tangent: Given a point q outside the convex hull, report the two tangents of the hull that pass through q. This query is more general than linear programming, because linear programming can be reduced to tangents of points at infinity. This query is also more general than containment: we can assume that the point is outside the hull, find its tangents, and then verify that the tangents are correct (by running linear-programming queries perpendicular to the tangents).

Line stabbing: (a.k.a. bridge finding) Given a line that intersects the convex hull, report the two edges that it cuts.

The original data structure by Overmars and van Leeuwen [OvL81] supported all queries in $O(\lg n)$ time, with $\Theta(\lg^2 n)$ update time. The later data structures with $o(\lg^2 n)$ update time [Cha01a, BJ00, BJ02] use techniques from decomposable search problems, and thus are limited to decomposable queries. Though containment and hull membership are not decomposable, the trick of reducing containment to tangent means that these queries also can be supported with $o(\lg^2 n)$ update time. Referring to Figure B-1, the horizontal dashed line marks the class of queries for which $o(\lg^2 n)$ updates and $O(\lg n)$ queries are known.

For our lower bound, it suffices to consider the sinks of this reduction graph: hull membership and line decision. All other queries are harder than at least one of them, so we obtain an $\Omega(\log_w n)$ lower bound for all queries but gift wrapping.

Our data structure from Chapter 6 supports all queries in $O(\frac{\lg n}{\lg \lg n})$ query time, and $O(\lg^2 n)$ update time. The data structure from Section B.2 below has update time $O(\lg n \lg \lg n)$, and supports one-dimensional, decomposable queries in $O(\log_w n)$ — that is, it supports linear programming, and, by reduction, line decision queries. Like all previous structures with $o(\lg^2 n)$ update times, ours cannot support nondecomposable queries like line stabbing. However, there is an additional discrepancy: because we cannot handle the two-dimensional tangent queries in this data structure, we cannot support the nondecomposable hull-membership or containment queries through the trick of reducing them to tangent. Thus our information-theoretic lens highlights an even stronger contrast between decomposable and nondecomposable queries.

Static problems. To situate these queries in context, we remind the reader about their behavior on a static convex polygon. Zero-dimensional queries trivially take constant time. For the other queries, we can only consider half of them, because duality makes vertically aligned queries identical. (Note that this is different from the dynamic setting, in which duals are expensive to maintain.)

One-dimensional queries can be solved through predecessor search. Chazelle [Cha99] provides a matching lower bound. Two-dimensional queries are reducible to point location, and our Corollary 5.1 provides the first sublogarithmic bound. It is conceivable that these queries are in fact easier than point location.

B.2 Fast Queries with Near-Logarithmic Update Time

Brodal and Jacob [BJ00] prove a general reduction from a dynamic convex hull data structure that, on $O(\lg^4 n)$ points, supports queries in $Q(n)$ time and updates in $U(n)$ time, into a dynamic convex hull data structure that, on n points, supports queries in $Q(n) \cdot \frac{\lg n}{\lg \lg n}$, updates in $U(n) \cdot \frac{\lg n}{\lg \lg n}$ and deletes in $O(\lg n \lg \lg n)$ time. The reduction works for decomposable queries. We will show how to build the polylogarithmic structure that supports linear-programming queries in $O(1)$ time and updates in $O((\lg \lg n)^2)$ time, which results in a dynamic convex hull data structure that supports linear-programming queries in $O(\lg n / \lg \lg n)$ time and updates in $O(\lg n \lg \lg n)$ time.

The data structure. Our data structure maintains the upper convex hull of $k = O(\log^4 n)$ points using four components:

1. A (binary) Overmars-van Leeuwen structure [OvL81], where each node represents the upper convex hull of its descendant points, as the concatenation of a subchain from the convex hull of each child, and two bridges.
2. One atomic heap [FW94] storing all slopes that appear in the hulls of the nodes of the Overmars-van Leeuwen structure. There are $k' = O(k)$ such slopes. An atomic heap supports insertions, deletions, and predecessor/successor queries on $\lg^{O(1)} n$ values in $O(1)$ time per operation.
3. A list labeling structure [DS87, BCD+02] maintaining $O(\sqrt{\lg n})$-bit labels for each such slope such that label order matches slope order. Unlike standard list labeling, our labels must be explicit, without the ability to simultaneously update pieces of several labels via indirection. Fortunately, our label space $2^{O(\sqrt{\lg n})}$ is much larger than our object space $k' = O(\lg^4 n)$. When the label space is polynomially larger than the object space, we can maintain explicit labels in $O(1)$ time per update, e.g., using the root-to-node labels in a weight-balanced search tree structure (BB[α] trees [NR73] weight-balanced B-trees [AV03]); see [BCD+02].

The lists representing the convex hull in each node have a nontrivial implementation. First of all, they represent the edges of the hull (in particular, their slopes), rather than the vertices. Second, they are organized as persistent, catenable B-trees with branching factor

$B = O(\sqrt{\lg n})$, and thus, height $O(1)$. Each slope is replaced by its label of $O(\sqrt{\lg n})$ bits, which means that a node has $O(\lg n)$ bits of information. We pack each node in one word. We refer to this representation of the convex hulls as *label trees*.

Observe that slopes are sorted on the hull, so a label tree is actually a search tree. Using standard parallel comparisons, we can locate the predecessor/successor labels of a query label in a label tree in constant time.

Updates. When we insert or delete a point, it affects the hulls of $O(\lg k)$ nodes in the Overmars-van Leeuwen structure. In each of these hulls, we may create $O(1)$ new slopes and/or delete $O(1)$ old slopes. We can compute these $O(\lg k)$ changes in $O(\lg^2 k)$ time using the standard Overmars-van Leeuwen data structure. The atomic heap and the list labeling structures can support these changes in $O(\lg k)$ total time. The labeling structure may update $O(\lg k)$ labels in total, and each label appears in $O(\lg k)$ label trees. Because we can search for a label in a label tree in constant time, we can update the label trees in $O(\lg^2 k)$ total time.

Finally, as we propagate the changes in the node hulls according to Overmars-van Leeuwen, we update the corresponding label trees using persistence, splits, and concatenations. The key property here is that a node can be split at an arbitrary point, or two nodes can be concatenated, in constant time because a node fits in a word. The total update time is therefore $O(\lg^2 k) = O((\lg\lg n)^2)$.

Linear-programming query. Given a query direction d, we take the slope normal to d. We search for the two adjacent slopes in the global atomic heap. Then we find the label assigned to these slopes in the list labeling structure, and average these two labels together (which is possible if we double the label space). Finally we search for the nearest two labels in the label tree of the root. Thus we obtain the two edges of the overall (root) convex hull whose slopes are nearest to the query slope, so the common endpoint of these two edges is the answer to the query.

B.3 Lower Bound

Our lower bound is based on a reduction from the *marked-ancestor problem* [AHR98]. In this problem, we have a static tree, say, a perfect binary tree with n leaves. Each node can be either *marked* or *unmarked*. Initially all nodes are unmarked. The two update operations are marking and unmarking a node. The (leaf decision) query is to decide whether a given leaf has a marked ancestor. We can also assume that every leaf has at most one marked ancestor, and trivially that the root is unmarked. Under these conditions, the marked-ancestor problem has the following (tight) lower bound: if updates run in t_u time, queries require $\Omega\left(\frac{\lg n}{\lg w + \lg t_u}\right)$ time.

For dynamic convex hull, we obtain a lower bound in the following form, implying a lower bound for all queries as in Figure B-1:

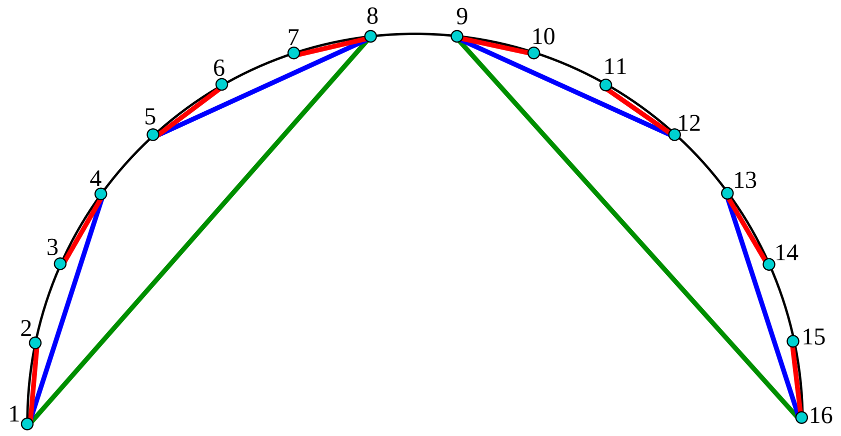

Figure B-2: The lower bound construction for a perfect binary tree on 8 leaves. Lines are the duals of points in the convex hull problem, and they correspond to nodes in the tree.

Theorem B.1. *Any data structure for maintaining a convex hull subject to insertion and deletion of points in amortized time t_u and either hull-membership or line-decision queries, requires $\Omega\left(\frac{\lg n}{\lg w + \lg t_u}\right)$ time per query in the cell-probe model.*

Proof. While we can prove a lower bound for dynamic convex hull directly, it turns out that it is considerably simpler to construct the lower bound in the dual space. In the dual problem, each point becomes a line, and maintaining the (upper chain of the) convex hull morphs into maintaining the lower envelope of the lines.

Hull membership dualizes to the question whether a given line from the set is part of the lower envelope. Line decision dualizes to containment: given a point, we ask whether it is below or above the lower envelope.

To reduce from marked ancestor, we consider a convex semicircle, and $2n$ points at uniform angular distances. Refer to Figure B-2. To leaf number i ($i \in \{1, \dots, n\}$) in the in-order traversal of the complete binary tree, we associate points number $2i - 1$ and $2i$ on the semicircle.

Now consider a node whose subtree spans leaves i through j. We associate this node with the line from point $2i - 1$ to point $2j$. For a leaf, this is a line between consecutive points on the circle.

At all times, the set of lines in the dynamic convex hull problem contains all lines associated with leaves, plus lines associated with marked nodes. Thus, marking or unmarking a node translates into adding or removing a line.

We now argue that a marked ancestor query to leaf i is equivalent to a hull membership query to the line associated with the leaf. Because the lower envelope is convex, and the center of the semicircle is always inside it, it follows that the line associated with leaf i is on the lower envelope iff the segments from the center of the semicircle to points $2i - 1$ and $2i$ do not intersect any other line in the set. Since these segments cannot intersect lines of other leaves, we only need to consider lines associated with marked nodes. One of the two segments intersects a line associated with the marked node iff the left endpoint defining the line is at point $\leq 2i - 1$ and the right endpoint is at a point $\geq 2i$ on the semicircle. This happens iff the node is an ancestor of leaf i.

To prove a lower bound for containment also, note that it can be used to test if the line between point $2i - 1$ and point $2i$ is on the lower envelope. Indeed, we can take the midpoint of this segment, and perturb it slightly towards the center of the semicircle. Then, the segment is on the envelope iff the point is inside the lower envelope.

Finally, we need to discuss issues of precision, since we are proving a lower bound for finite precision models. Note that even though we cannot make the points be perfectly on the circle, our proof only requires than for any triple $i < j < k$, point j is strictly above the line from i to k. Since we have $2n$ points at uniform distances on the semicircle, the minimum distance between such a point j and the line from i to j is $\Omega(n)$. Thus, rounding points in the dual space to a grid of $[cn]^2$, where c is a sufficiently large constant, will preserve the needed property.

Once dual points are integral, the lines are described by rational parameters, so points in the primal space are rational, with a constant-factor increase in precision. Therefore, our lower bound holds for precision $w = \Omega(\lg n)$. $\square$

Bibliography

[ABR01] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[AD96] Lyudmil Aleksandrov and Hristo Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal on Discrete Mathematics*, 9(1):129–150, 1996.

[AEIS01] Arnon Amir, Alon Efrat, Piotr Indyk, and Hanan Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30(2):164–187, 2001. See also FOCS'99.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998. See also STOC'95.

[AHR98] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–543, 1998.

[AMM01a] Sunil Arya, Theocharis Malamatos, and David M. Mount. Entropy-preserving cuttings and space-efficient planar point location. In *Proc. 20th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 256–261, 2001.

[AMM01b] Sunil Arya, Theocharis Malamatos, and David M. Mount. A simple entropy-based algorithm for planar point location. In *Proc. 20th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 262–268, 2001.

[AMT99] Arne Andersson, Peter Bro Miltersen, and Mikkel Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215(1-2):337–344, 1999.

[And96] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–141, 1996.

[AT02] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *CoRR cs.DS/0210006*. See also FOCS'96, STOC'00, 2002.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[AV03] Lars Arge and Jeffrey S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003. See also FOCS'96.

[BCD$^+$02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[BDP05] Ilya Baran, Erik D. Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3sum. In *Proc. 9th Workshop on Algorithms and Data Structures (WADS)*, pages 409–421, 2005.

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. See also STOC'99.

[BJ00] Gerth S. Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log\log n)$ update time. In *Proc. 7th Workshop on Algorithms and Data Structures (WADS)*, pages 57–70, 2000.

[BJ02] Gerth S. Brodal and Riko Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002.

[BKRS92] Marshall W. Bern, Howard J. Karloff, Prabhakar Raghavan, and Baruch Schieber. Fast geometric approximation techniques and geometric embedding problems. *Theoretical Computer Science*, 106(2):265–281, 1992. See also SoCG'89.

[BMM97] Andrej Brodnik, Peter Bro Miltersen, and J. Ian Munro. Trans-dichotomous algorithms without multiplication—some upper and lower bounds. In *Proc. 5th Workshop on Algorithms and Data Structures (WADS)*, pages 426–439, 1997.

[CF97] L. Paul Chew and Steven Fortune. Sorting helps for Voronoi diagrams. *Algorithmica*, 18(2):217–228, 1997.

[Cha91] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991. See also FOCS'90.

[Cha96] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[Cha99] Bernard Chazelle. Geometric searching over the rationals. In *Proc. 7th European Symposium on Algorithms (ESA)*, pages 354–365, 1999.

[Cha00] Timothy M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM Journal on Computing*, 30(2):561–575, 2000. See also FOCS'98.

[Cha01a] Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *Journal of the ACM*, 48(1):1–12, 2001. See also FOCS'99.

[Cha01b] Timothy M. Chan. On enumerating and selecting distances. *International Journal of Computational Geometry and Applications*, 11(3):291–304, 2001. See also SoCG'98.

[Cha02] Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 472–473, 2002.

[Cha04] Timothy M. Chan. Faster core-set constructions and data stream algorithms in fixed dimensions. In *Proc. 20th ACM Symposium on Computational Geometry (SoCG)*, pages 152–159, 2004.

[Cha06] Timothy M. Chan. Point location in $o(\log n)$ time, Voronoi diagrams in $o(n \log n)$ time, and other transdichotomous results in computational geometry. In *Proc. 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 333–344, 2006.

[Cla88] Kenneth L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, 1988.

[Col86] Richard Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–220, 1986.

[CP07] Timothy M. Chan and Mihai Pătraşcu. Voronoi diagrams in $n \cdot 2^{O(\sqrt{\lg \lg n})}$ time. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, 2007.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4:387–421, 1989.

[DadH90] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc 17th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 6–19, 1990.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *Proc. ACM SIGCOMM*, pages 3–14, 1997.

[dBSvKO00] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[dBvKS95] Mark de Berg, Marc J. van Kreveld, and Jack Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18(2):256–277, 1995. See also SWAT'92.

[DP07] Erik D. Demaine and Mihai Pătraşcu. Tight bounds for dynamic convex hull queries (again). In *Proc. 23rd ACM Symposium on Computational Geometry (SoCG)*, 2007.

[DS87] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. See also STOC'86.

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[EGS86] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. See also FOCS'82.

[FM84] Alain Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153–174, 1984.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994. See also FOCS'90.

[Goo95] Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995. See also STOC'92.

[Han01] Yijie Han. Improved fast integer sorting in linear space. *Information and Computation*, 170(1):81–94, 2001. See also SODA'01.

[Han04] Yijie Han. Deterministic sorting in $o(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. See also STOC'02.

[HPM05] Sariel Har-Peled and Soham Mazumdar. Fast algorithms for computing the smallest k-enclosing circle. *Algorithmica*, 41(3):147–157, 2005. See also ESA'03.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in $0(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144, 2002.

[Iac04] John Iacono. Expected asymptotically optimal planar point location. *Computational Geometry*, 29(1):19–22, 2004. See also SODA'01.

[IL00] John Iacono and Stefan Langerman. Dynamic point location in fat hyperrectangles with integer coordinates. In *Proc. 12th Canadian Conference on Computational Geometry (CCCG)*, 2000.

[Kir83] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[LT80] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980. See also FOCS'77.

[MPP05] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005.

[Mul90] Ketan Mulmuley. A fast planar partition algorithm. *Journal of Symbolic Computation*, 10(3/4):253–280, 1990. See also FOCS'88.

[Mul00] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice-Hall, 2000.

[NR73] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973. See also STOC'72.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 2nd edition, 1998.

[OvL81] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. See also STOC'80.

[Păt06] Mihai Pătraşcu. Planar point location in sublogarithmic time. In *Proc. 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 325–332, 2006.

[PD06] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. See also SODA'04 and STOC'04.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *Proc. 18th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564, 2007.

[Rab76] Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–30. Academic Press, 1976.

[Ruž07] Milan Ružić. Making deterministic signatures quickly. In *Proc. 18th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 900–909, 2007.

[SA00] Raimund Seidel and Udo Adamy. On the exact worst case query complexity of planar point location. *Journal of Algorithms*, 37(1):189–217, 2000. See also SODA'98.

[Sno04] Jack Snoeyink. Point location. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd edition)*, pages 767–785. Chapman & Hall/CRC, 2004.

[ST86] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[Tho02a] Mikkel Thorup. Equivalence between priority queues and sorting. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 125–134, 2002.

[Tho02b] Mikkel Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *Journal of Algorithms*, 42(2):205–230, 2002. See also SODA'97.

[Tur36] Alan M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. Announced by van Emde Boas alone at FOCS'75.

[Wil00] Dan E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000. See also SODA'92.