Solving Geometric Problems in Space-Conscious Models

by

Yu Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

Abstract

When dealing with massive data sets, standard algorithms may easily "run out of memory". In this thesis, we design efficient algorithms in space-conscious models. In particular, in-place algorithms, multi-pass algorithms, read-only algorithms, and stream-sort algorithms are studied, and the focus is on fundamental geometric problems, such as 2D convex hulls, 3D convex hulls, Voronoi diagrams and nearest neighbor queries, Klee's measure problem, and low-dimensional linear programming.

In-place algorithms only use $O(1)$ extra space besides the input array. We present a data structure for 2D nearest neighbor queries and algorithms for Klee's measure problem in this model.

Algorithms in the multi-pass model only make read-only sequential access to the input, and use sublinear working space and small (usually a constant) number of passes on the input. We present algorithms and lower bounds for many problems, including low-dimensional linear programming and convex hulls, in this model.

Algorithms in the read-only model only make read-only random access to the input array, and use sublinear working space. We present algorithms for Klee's measure problem and 2D convex hulls in this model.

Algorithms in the stream-sort model use sorting as a primitive operation. Each pass can either sort the data or make sequential access to the data. As in the multi-pass model, these algorithms can only use sublinear working space and a small (usually a constant) number of passes on the data. We present algorithms for constructing convex hulls and polygon triangulation in this model.

Acknowledgements

These 6 years in my graduate study are unforgettable. Timothy Chan, my supervisor, guided me through such an interesting but challenging research. Sometimes, he acts as a mentor, leading me into new topics. Sometimes, he acts as a commander, setting challenging and reachable goals to achieve. Sometimes, he acts as a co-worker, sharing all successful and failed ideas. Sometimes, he acts as an outsider, leaving space for me to try different ideas. Sometimes, he even acts as an attacker, questioning all possible flaws in my proofs. But always, he is a great supervisor, teaching me how to clarify and simplify complicated problems and make progress.

I wish to thank my committee members, Michiel Smid, Chaitanya Swamy, Anna Lubiw, and Ian Munro. They spent so much time in the Christmas break to go through my thesis thoroughly and provided many helpful comments. Without their help, this work is impossible.

I also would like to thank Mark Keil. He taught me many entry-level algorithm courses and opened the door for such an exciting research area, computational geometry.

Finally, I must appreciate the encouragement and trust from my wife and parents. They shared all the exciting times and hard times with me in these 6 years.

Dedication

For my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, the study of algorithms mainly focuses on running-time performance. When handling massive datasets, those "good" algorithms can easily "run out of memory". One way researchers have dealt with this issue is to consider algorithms and data structures in external memory. Here, we examine this issue from another direction—designing algorithms and data structures that consume little processing space. To define the space constraints formally, different space-conscious computational models have been proposed, for example, in-place algorithms, the multi-pass model, and the streaming model.

Massive datasets arise in different fields, such as in geographic information systems, sensor networks, internet applications, bioinformatics, robotics, and electronic circuit designs. Many problems in these applications may be abstracted as geometric problems. In this thesis, we will study fundamental geometric problems in these space-conscious models.

1.1 Computational Geometry

In computational geometry, we focus on finding asymptotically fast algorithms and data structures for solving geometric problems. This research topic was established in the 70's [61], when some fundamental geometric problems were investigated, for example, convex hulls, Voronoi diagrams, triangulation, and point location.

1.1.1 Convex Hulls

For a set of points, its convex hull is the smallest convex polytope containing all points. In 2D, the convex hull is a convex polygon; in 3D, it is a convex polyhedron. Convex hulls are also discussed in *dual* space. By this transformation, in 2D, each point (a, b) in *primal* space is mapped to a line $y = ax - b$ in dual space, and the convex hull of a set of points corresponds to the lower envelope of a set of lines. In

3D, a point is mapped to a plane in dual space, and the convex hull corresponds to the lower envelope of a set of planes. In higher dimensions, a point is mapped to a hyperplane, and the convex hull corresponds to the lower envelope of a set of hyperplanes.

Constructing convex hulls can be viewed as an extension of the sorting problem to higher dimensions. Some well-known convex hull algorithms use a sorting algorithm as a subroutine, such as Graham's scan [61]. Some other algorithms share similar divide-and-conquer strategies with mergesort or quicksort. It is also known that the sorting problem can be reduced to the convex hull problem in 2D. Indeed, in comparison-based models, where fixed-degree algebraic predicates over the input points can be evaluated in $O(1)$ time, constructing the convex hull of n points in both 2D and 3D requires $\Theta(n \log n)$ time [40, 87].

However, we can still aim for convex hull algorithms with better performance by taking the size of the output into account. The concept of *output-sensitive* algorithms has been proposed for this reason. These algorithms only take $O(n \log h)$ time [20, 71], where h is the number of points on the convex hull.

In computational geometry, the convex hull plays an important role. It has connections to many geometric problems, for example, the *linear programming problem*, where the goal is to optimize a linear objective function, subject to a set of linear inequality constraints. Geometrically, each linear constraint can be viewed as a hyperplane. Solving the linear programming problem is equivalent to finding the extreme point on the lower envelope of a set of hyperplanes in a direction determined by the objective function.

1.1.2 Voronoi Diagrams

The Voronoi diagram is related to the "post office" problem. Given a set of n points on the plane, we can divide the plane into n regions, such that all points in the same region share the same nearest post office, measured in Euclidean distance. This partition is known as the *Voronoi diagram*. These post offices are called *sites*. Each site corresponds to exactly one region (called the *Voronoi cell*) in the partition. One edge of the Voronoi cell is defined by two sites, which is a part of the bisector line of these two sites.

The *Delaunay triangulation* is another well-known geometric structure. In the Delaunay triangulation of a set of 2D points, the circumcircle of any triangle does not contain any points in the point set, other than the three vertices of the triangle.

For a set of points, its Voronoi diagram and Delaunay triangulation can be transformed into one another in linear time. Each vertex in the Delaunay triangulation maps to a Voronoi cell; each edge in the Delaunay triangulation maps to a Voronoi edge. Therefore, the Delaunay triangulation is referred to as the dual of the Voronoi diagram.

2D Voronoi diagrams/Delaunay triangulations can be constructed in $\Theta(n \log n)$ time [40, 80, 87]. They can be constructed directly, for example, by Fortune's sweeping algorithm [40], or by randomized incremental algorithms [80]. It can also be reduced to a 3D convex hull problem and be constructed by a 3D convex-hull algorithm. In this reduction, each 2D point (x, y) is lifted to a 3D point $(x, y, x^2 + y^2)$. The projection of the 3D convex hull of the generated point set in the xy plane is the Delaunay triangulation of the original point set [87].

One main application of Voronoi diagrams is to answer nearest neighbor queries. After the Voronoi diagram is obtained, answering nearest neighbor queries becomes locating the Voronoi cell containing the query point—a point location query in a planar subdivision.

1.1.3 Point Location and Other Planar Subdivisions

Given a planar subdivision and a query point, the point location problem is to locate the region containing the query point. With additional data structures, a query can be answered in $O(\log n)$ time, where n is the number of vertices in the subdivision. Several well-known structures have been proposed since the late 80's, such as the Kirkpatrick's hierarchical method [87], the persistent search tree [91], the chain method [48], and a randomized incremental method [80].

Besides Voronoi diagrams and Delaunay triangulations, other geometric structures are also widely used, for example, the *trapezoidal decomposition* [40]: given a set of n disjoint line segments in 2D, we divide the plane into a set of trapezoids. For every trapezoid, its upper and lower edges are parts of the given line segments. The left and right walls pass through the endpoints of the line segments. None of the trapezoids intersects any of the line segments. The trapezoidal decomposition of a set of n disjoint line segments can be computed in $O(n \log n)$ time. By answering a point location query in this planar subdivision, we can locate the line segment right above or below the query point.

Trapezoidal decompositions can also be used to support other queries. By considering a simple polygon as a set of disjoint line segments and answering point location queries on its trapezoidal decomposition, we can answer a membership query—deciding whether a query point is contained in the given polygon. Alternatively, we can also answer membership queries by point location in a triangulation of the polygon. Since it is also known that the trapezoidal decomposition and the triangulation of a set of line segments can be transformed into one another in $O(n)$ time [40], we can build both the trapezoidal decomposition and the triangulation of a polygon in $O(n \log n)$ time, by computing the trapezoidal decomposition of a set of line segments. In fact, for a simple polygon, both of these structures can be computed faster in $O(n)$ time [29].

All geometric problems mentioned are usually studied in traditional computational models: the working space can be randomly accessed, and the size of the working space is unlimited. The main measurement of algorithms and data structures is their running time. In this proposed research, we will consider space constraints. In the rest of this chapter, we will introduce various space-conscious models, and also summarize the new results in the thesis.

1.2 Space-Conscious Models

In this section, we introduce several space-conscious model. For in-place algorithms and in the read-only model, input data are loaded in memory, and the extra working space is sublinear. In the multi-pass model and the stream-sort model, input data are in some special device with restrictive access, and the working memory is sublinear.

1.2.1 In-Place Algorithms and Implicit Data Structures

For *in-place* algorithms, besides the space storing the given input (as an array of data elements), algorithms and data structures can only take $O(1)$ extra working space.[1] For example, heapsort is in-place.

Most other well-known optimal sorting algorithms need $\Omega(n)$ extra space. In the 80's and 90's, stable sorting algorithms [39] were studied and several in-place merging algorithms were developed [45, 76]. Based on those results, mergesort can also be made to run in $O(n \log n)$ time with $O(1)$ extra space. Later, two research groups independently found $O(n \log n)$-time stable mergesort algorithms taking $O(\sqrt{n})$ extra space [44, 95]. An in-place stable mergesort algorithm was finally discovered in 1995 [104].

In the same period, stable partitioning algorithms have also been studied. With those partitioning algorithms, Munro et al. gave a stable quicksort running in $O(n \operatorname{polylog} n)$ time with $O(1)$ extra space [85]. Later, Katajainen and Passanen gave a linear-time stable partitioning algorithm using $O(1)$ extra space, resulting in an optimal stable quicksort [69]. Recently, in-place sorting algorithms with $O(n \log n)$ comparisons and $O(n)$ moves have been studied [55, 54]. In 2007, an optimal stable in-place sorting algorithm with $O(n)$ moves has been obtained [54] by Franceschini.

Researchers also studied the related selection problem. Lai and Wood gave a solution to select the k-th largest element running in $O(n)$ time with $O(1)$ words or $O(\log n)$ bits of extra space [73].

We can also consider data structures taking sublinear extra space. In *implicit* data structures, all data are kept in an array of n units, and only $O(\operatorname{polylog} n)$ extra space is used. Implicit dictionary structures supporting dynamic updates have been studied since the 80's. Munro [81] proposed an efficient dynamic structure

[1] In this thesis, the space is counted in the number of words or data units, unless specified otherwise

supporting updates and predecessor search in $O(\mathrm{polylog}\, n)$ time with $O(\mathrm{polylog}\, n)$ extra space. In this paper, an interesting bit-encoding trick is introduced. Given a pair of elements, if the first element is smaller than the second one, 0 is encoded, otherwise 1 is encoded. Based on this idea, pointers and other additional information can be stored without using extra space. For example, by grouping $O(\log n)$ pairs of data together, a pointer of size $O(\log n)$ can be encoded. Roughly speaking, this is why we only need a very small amount of extra space but have an extra $O(\log n)$ factor in the running time. After a sequence of improvement [56, 58] (over almost 20 years), Franceschini and Grossi finally proposed a structure obtaining $O(\log n)$ running time for both predecessor queries and updates, with only $O(1)$ extra space [57].

In computational geometry. In the early stage, some implicit data structures for database queries have been proposed, which can also be considered as data structures for geometric problems [51]. After 2000, research on in-place geometric algorithms and data structures became active [15, 17, 18, 19, 27, 33, 34, 105, 106]. Convex hulls in 2D can be constructed in the optimal $O(n \log n)$ time with only $O(1)$ extra space [19]. If the input is represented as a polygonal chain with no self intersections, the 2D convex hull can be constructed in $O(n)$ time with $O(1)$ space [17]. With Brönnimann and Chan [18, 31], we gave an in-place algorithm to construct 3D convex hulls in $O(n \log^3 n)$ time, which implies that 2D Voronoi diagrams/Delaunay triangulations can also be constructed in the same time and space complexity if the output is written to a write-only stream. In the same paper, we also showed that off-line nearest neighbor queries can be done in $O(n \log^2 n)$ time, with $O(\log^2 n)$ extra space. These results appeared in the author's Master's thesis. Meanwhile, computing the closest pair in a set of 2D points has been solved by Bose et al. in optimal $O(n \log n)$ time with $O(1)$ extra space [15]. Intersections between axis-aligned line segments can be reported in $O(k + n \log n)$ time with $O(1)$ extra space [15]. Intersections between general line segments can be reported in $O((n+k) \log^2 n)$ time, with $O(\log^2 n)$ extra space [33], which has recently been improved to $O(k + n \log^2 n)$ time with $O(1)$ extra space by Vahrenhold [106]. Implicit geometric data structures have also been studied, such as kd-trees and partition trees [18].

From the view of methodology, these papers mainly follow two trends. The first trend is by extending sorting algorithms. The general strategy is by divide-and-conquer. In the combination step, an in-place (or nearly in-place) merging procedure is designed for each problem or adapted from in-place merging algorithm. The results in [15, 18, 19, 106] belong to this category. The other trend is by building complicated implicit data structures. Sweeping is a well-known technique in computational geometry, which simulates a vertical line sweeping through the plane [40]. This technique is normally supported by maintaining a 1D dynamic searching structure and a priority queue simultaneously. With implicit data structures, we can combine a priority queue and a searching structure together. In Munro's implicit structure [81], data are stored in blocks, and an AVL tree on blocks is built. We can build a second AVL tree on the same blocks. In the second tree, the blocks are indexed by the order of priority values. Each block maintains a pointer to the data with the highest priority value. Whenever the priority value is updated, the corresponding block and the second AVL tree are updated. Several recent results are based on this type of data structures [31, 33, 34].

We normally see an $O(\log n)$ slow-down factor in the algorithms using implicit data structures [33, 34, 105], since all extra pointers are encoded. However, this encoding technique does not always introduce the $O(\log n)$ factor. Indeed, the 2D convex hull of a simple polygon is computed using implicit data structures in the optimal $O(n)$ time with $O(1)$ extra space [17]. The reason is that only a doubly linked list, which is much simpler than a tree, is encoded in that algorithm.

New results. We have two main results for in-place algorithms and implicit data structures in this thesis. One is an improved static implicit structure for 2D nearest neighbor queries. Previously, we have obtained a data structure to answer 2D nearest neighbor queries with no extra space [18]. In the initial version of that paper, the query time is $O(n^\varepsilon)$ for any fixed constant $\varepsilon > 0$; in the full version of the paper, the query time is lowered to $O(\log^2 n)$. Due to the encoding/decoding operations for implicit data structures, the extra $O(\log n)$ slow down factor was introduced. That data structure can be viewed as a modification of standard data structures for 2D nearest neighbor queries, described in section 1.1.3.

The new implicit data structure presented in this thesis is different from any well-known data structures for nearest neighbor queries and does not store the Voronoi diagrams explicitly. It may be viewed as an extension of the van Emde Boas layout [41, 56, 92] to 2D. With such a highly recursive data structure, we can control the size of pointers better, and reduce the query time to $O(\log^{\log_{3/2} 2} n \log\log n)$ (roughly $O(\log^{1.71} n)$). This data structure will be shown in section 2.3.

The other main result is about Klee's measure problem in 2D: given a set of axis-aligned rectangles, the goal is to compute the area/perimeter of the union of the rectangles. In an earlier paper [34], we solved it in $O(n^{3/2} \log n)$ time with $O(\sqrt{n})$ extra space. That was improved to $O(n^{3/2} \log n)$ time with $O(1)$ space by Vahrenhold [105]. In this thesis, we give a further improved algorithm that runs in $O(n \log^2 n)$ time with $O(1)$ space, if the coordinates are $O(\log n)$-bit integers. These algorithms will be shown in section 2.4.

1.2.2 Multi-Pass Model

In the *multi-pass* model, input data are stored in a device, for example, on tapes. An algorithm can only sequentially access the input by several passes. The input must remain unchanged after each pass. Depending on the problem, the answer may be sent to a write-only output stream or kept in memory. The goal is to minimize the amount of working space (measured in words), while keeping the number of passes small. Munro and Paterson defined this model in the early 80's and studied the classical sorting and selection problems [82]. They gave a selection algorithm that requires only $\lceil 1/\delta \rceil$ passes and $O(n^\delta \log^{2-2\delta} n)$ space for any fixed constant $\delta > 0$ and provided an almost matching lower bound in comparison-based models.

To prove the lower bound, they presented an adversary argument. Given that only b units can be kept in memory, any program must discard one unit of the data, before it can access the $(b+1)$-st datum. Therefore, any program cannot know the relationship between the discarded datum and the newly arrived datum. Based on this observation, we can divide the data into roughly n/b groups. In one pass, no algorithm can tell the difference between any two data in the same group. By this adversary strategy, they can force any selection algorithm to make many comparisons [82].

Some variations of the multi-pass model have been proposed recently. For example, a so-called W-stream model is proposed by Ruhl [94], where intermediate temporary streams of size $O(n)$ between passes are allowed. Demetrescu et al. have studied several graph problems in this model [42]. They have shown that single-source shortest paths in directed graphs can be solved in $O((n \log^{3/2} n)/\sqrt{s})$ passes and the connected components in undirected graphs can be found in $O((n \log n)/s)$ passes, where s is the number of bits kept in memory. They have also proved an $\Omega(n/s)$ lower bound on the number of passes for both of these problems.

The original streaming model is more restrictive, where only a single pass on the data is allowed. Many approximation algorithms have been proposed in this model, such as for the problems of selection, clustering [64], and constructing histograms [52]. A more comprehensive survey of the work in this model can be found in [86].

The stream-sort model is another variation of the multi-pass model, which will be discussed in more detail later in this section.

In contrast to in-place algorithms, there has not been much work on multi-pass algorithms in computational geometry. In fact, this thesis contains the first set of results along this direction.

New results. In the multi-pass model, we obtain a variety of results for some fundamental geometric problems. For example, we show that low-dimensional linear programming problems can be solved with $O(n^\varepsilon)$ space in a constant number of passes, and we also provide nearly matching lower bounds [26]; with $O(s)$ space, convex hulls in 2D can be computed in $O(n/s)$ passes, which also has a matching lower bound. These results will be shown in chapter 3.

1.2.3 Read-Only Model

In the *read-only* model, all data are given in a read-only array (not by a particular permutation defined by the program), and the algorithm can only take a small amount of working space besides the input. This model is "between" the in-place model and the multi-pass model, since the read-only array can still be randomly accessed. For example, the standard sorting problem has been considered in this model. Here, the goal is to minimize the time-space product. Frederickson gave an algorithm, where the time-space product is $O(n^2 \log n)$ [59] (space is counted in bits). An $\Omega(n^2)$ lower bound was later shown by Beame [12]. Then Pagter and Rauhe proposed an algorithm achieving this time-space lower bound [90]. Their algorithm can be viewed as an extension of heapsort. They divide the input array into blocks and build a heap-like navigational structure on it. In this new heap, the number of branches depends on the depth in the heap. At the bottom level, extra information is maintained for each block. When running in $O(n \log n)$ time, this algorithm only needs $O(n/\log n)$ bits of extra space.

Selection is another well-studied problem in the model. Frederickson gave a selection algorithm taking $O((n \log n)/\log s + n \log^* s)$, where s is the number of registers and s is $\Omega(\log^2 n)$ [59]. With less space, Munro and Raman's algorithm takes $O(2^s s! n^{1+1/s})$ time with $1 \leq s \leq \sqrt{\log n/\log\log n}$ extra space [83]. Further improvement has been made by Raman and Ramnath, if s is $\Omega(\log n)$ and $o(\log^2 n)$ [107].

Along with the multi-pass results, we are also the first to study geometric problems in the read-only model.

New results. In contrast with our multi-pass results, we show that we can construct 2D convex hulls for sorted point sets and solve the low-dimensional linear programming problems with less extra space [26]. For example, with $O((1/\delta)n^\delta)$ space, we can compute the convex hull from sorted points in linear time, which beats the lower bound of this problem in the multi-pass model. This result will be shown in sections 4.2 and 4.3.

We also propose an algorithm solving Klee's measure problem in $O(n^3/s^2)$ time, where s is the memory space counted in words. This algorithm will be shown in section 4.4.

1.2.4 Stream-Sort Model

Another model proposed in the literature is Ruhl et al.'s *stream-sort* model [94]. Since sorting is a well-studied problem, it is fully optimized in many systems. In this model, the input data are still given as a stream, but now we have two ways to access them. One is by a scan pass: input stream is scanned sequentially, and one output stream is generated. The other is by a sorting pass: the data in the input stream are sorted based on a user-defined comparator, and the data are placed in the sorted order in the output stream. The generated output stream is called an *intermediate* stream. In the next pass, this intermediate stream becomes the input stream. All intermediate streams must be of size $O(n)$. At the end, we count how many passes are used in total, and how much working space is used in the scan passes.

Much of the back-end data processing in Google is based on a so-called *map-reduce* framework. The stream-sort model is highly related to the abstraction of this framework. In practice, this framework has some advantages. For example, applications implemented in the model can be easily distributed or parallelized; the load balancing of tasks can be easily achieved in distributed systems.

In computational geometry. Ruhl et al. [5, 94] showed how to compute intersections between two disjoint sets of 2D line segments, with $O(1)$ space and $O(\text{polylog}\, n)$ passes. Lammersen and Sohler [74] solved some other geometric problems, such as computing 2D convex hulls with $O(1)$ space and $O(\text{polylog}\, n)$ passes.

Note that as Ruhl et al. [5, 94] have shown, many parallel circuits, and consequently many parallel algorithms, can be simulated in the streaming model augmented with a sorting primitive. Several known geometric algorithms [4] can be simulated in $O(\text{polylog}\, n)$ passes directly. Therefore, we mainly focus on results where only $O(1)$ passes are allowed.

New results. We solve two fundamental geometric problems with only $O(1)$ passes. One is the convex hull problem in 2D and 3D. The other is polygon triangulation in 2D. With $O(n^{\varepsilon})$ memory space, we can solve all these problems in a constant number of passes. These results will be shown in section 5.3, 5.4, and 5.5.

We also introduce a new model. Since the sorting pass is the most expensive part, we define a simpler model in which the sorting pass is not allowed, but we can only choose a direction to scan the stream (forward or backward). This model is weaker than the stream-sort model. It can also be viewed as relaxation of the multi-pass model. This model is interesting, because we can construct the 2D convex hull from a set of sorted points in $O(1)$ passes with $o(\sqrt{n})$ space, which surpasses the lower bound in the multi-pass model. This algorithm will be shown in section 5.6.

1.2.5 Other Models

There are other space-conscious models studied in the literature which will not be considered in this thesis. For instance:

- Cache-oblivious algorithms: similar to the external memory model, data are stored in a hierarchical memory structure. The CPU only has direct access to data stored in cache. When the required data are missed, we have to fetch them from the storage in the next level. Unlike in the external memory model, the page size is unknown to the algorithm. In the cache oblivious model, the running time of algorithms automatically adapts to different page sizes. A more comprehensive survey can be found in [41].

- The streaming model: Algorithms can only make a single pass over the input and work with a small amount of space. This is a more restrictive model than the multi-pass model studied in this thesis. A more comprehensive survey can be found in [86].

- Succinct data structures: Some fundamental topological structures can be stored in only $O(n)$ bits. With these data structures, we can perform traversal operations on trees and graphs [68]. For example, given the pointer to a vertex, we can move the pointer to its parent or children in trees, or move the pointer to an adjacent vertex in graphs. Succinct data structures have also been studied for string matching, for example, suffix trees [35]. In computational geometry, triangulations have been studied [6, 7, 8]. Given a finger to a triangle in a triangulation, we can move the finger to its adjacent triangles. This is different from in-place algorithms, where only O(1) extra words are allowed.

Chapter 2

In-Place Algorithms and Implicit Data Structures

2.1 Our Results

In this chapter, the following in-place algorithms and implicit data structures will be presented.

- Given a set of 2D points, a static data structure is built with $O(1)$ extra space, such that the nearest-neighbor queries can be answered in $O(\log^{\log_{3/2} 2} n \log\log n) = O(\log^{1.71} n)$ time with $O(1)$ space. This result has been published in SODA'08 [27].
- Given a set of axis-aligned rectangles, the area/perimeter of the union of the rectangles (Klee's measure problem in 2D) can be computed in $O(n^{3/2} \log n)$ time with $O(\sqrt{n})$ space. This result has been published in CCCG'05 [34].
- If the coordinates of rectangles are $O(\log n)$-bit integers, we can improve our result on Klee's measure problem to $O(n \log^2 n)$ time with $O(1)$ space.
- Given a set of axis-aligned rectangles, a point covered by the largest number of rectangles (a.k.a. the *depth* problem) can be found in $O(n \log^2 n)$ time with $O(1)$ space.

2.2 The Permutation+Bits Model

To simplify the presentation of our implicit data structures and in-place algorithms, we will work mostly in an intermediate model. Here, in addition to the input array holding a permutation of the given elements, we also have an extra array of bits. The content of this extra space can only be accessed through bit probes, with each bit access costing unit time. For lack of a better name, we call this the *permutation+bits* model.

Curiously, even if we are allowed to use a huge number of extra bits (quadratic or worse), many problems remain nontrivial in this model. For example, consider one of the most naive solutions to nearest neighbor queries: the *slab method* [91]. We draw vertical lines at every vertex of the Voronoi diagram, and store the line segments of the subdivision at each of the $O(n)$ vertical slabs in sorted y-order. To answer a query, we first perform a binary search in x to find the slab containing the query point q, and perform another binary search in y to find segment immediately above q. This method uses near-quadratic space but has asymptotically the best query time possible, $O(\log n)$, in the standard model. However, to support binary search, we need to store each of the $O(n)$ sorted lists in subarrays. In the permutation+bits model, one list can be represented within the input array itself, but with multiple lists sharing common elements, in general one needs to tap into the extra array of bits. Each occurrence of a point beyond its first can only be represented implicitly through pointers. (We assume that points are indivisible data types—think of them as infinite-precision real numbers.) With the bit probe restriction, each access to a point would require logarithmic cost, so the binary search would now cost $O(\log^2 n)$!

Nevertheless, by storing the data set in a carefully arranged order, we can adapt the results in this model to in-place algorithms and implicit data structures, if the number of extra bits is $O(n)$, by a well-known encoding trick [81].

2.3 In-Place 2D Nearest Neighbor Search

In this section, we describe a new data structure for 2D exact nearest neighbor queries that uses no extra storage other than the input array. Our query time is $O(\log^\gamma n \log\log n)$, where $\gamma = \log_{3/2} 2 < 1.71$. This new algorithm for point location in the Voronoi diagram is based on using planar graph separators in a recursive fashion. Although this is hardly the first time separators are used in computational geometry (or in point location algorithms for that matter), the way recursion is used is different and, in our opinion, quite fascinating (as one might suspect from the relative unusualness of our time bound).

The techniques underlying our new method also bear connections with known data structuring techniques: our organization of the input array can be viewed as a more sophisticated extension of the "van Emde Boas layout" [41, 92], popular in the context of cache-oblivious data structures. We perform n^β-way divide-and-conquer, for some constant β, to ensure that the logarithm of the input size—hence, the size of pointers—decreases exponentially, thus mitigating the effect of the logarithmic-factor slow-down.

The rest of the section has a simple organization. After some preliminaries in the next subsection, the data structure and query algorithm are described in subsection 2.3.2 (with 3 interdependent subroutines), which are then analyzed in subsection 2.3.3.

2.3.1 A Permutation from Voronoi Diagrams and Separators

We now specify the order for the input array. For this, we apply a standard, multiple-clusters version of the planar separator theorem [75] to the dual of the Voronoi diagram, where the removal of a set of separator Voronoi cells produces $O(b)$ clusters of $O(n/b)$ Voronoi cells. Let $\mathrm{Vor}(P)$ denote the Voronoi diagram of a point set P and $\mathrm{Vor}(p, P)$ denote the Voronoi cell (a convex polygon) of the point $p \in P$ in $\mathrm{Vor}(P)$. The theorem implies:

Lemma 2.3.1 (Separator Theorem) *Given a set P of n points in the plane and a parameter $1 \leq b \leq n$, we can partition P into a subset P_S (the* separator*) of $O(\sqrt{bn})$ points and $O(b)$ subsets $P_1, P_2 \ldots$ (the* clusters*) each of $O(n/b)$ points, so that:*

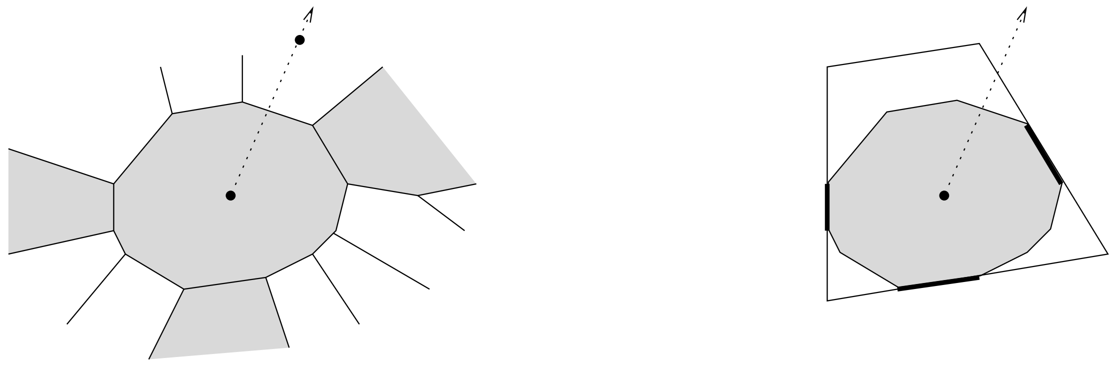
For every two points $p, p' \in P - P_S$, if $\mathrm{Vor}(p, P)$ is adjacent to $\mathrm{Vor}(p', P)$, then p and p' belong to the same cluster.

To generate the permutation for P, we simply generate the permutations for the subsets P_S, P_1, P_2,... recursively and concatenate these permutations. The choice of the parameter b will be specified later.

Applying this recursive procedure to the input point set P_0, we implicitly obtain a tree T of subsets, where the root holds the global set P_0, and each generated subset P resides in a contiguous subarray of the input array. It is important to note that we are working not with one global Voronoi diagram but with Voronoi diagrams of subsets. Understanding the relationship between Voronoi diagrams of different subsets will be vital (see Observations 2.3.2 and 2.3.3 to come). We will make use of the fortunate fact that nearest neighbor search is a "decomposable" problem—the nearest neighbor of a point in a union of two subsets can be obtained from the nearest neighbor in each of the two subsets trivially.

Once the permutation is fixed, our subsequent data structures cannot change the order of the input array but can only append to the extra array of bits. Note the resemblance to the van Emde Boas layout [41] (P_S is the "top" structure and $P_1, P_2, \ldots$ are the "bottom" structures—in a query, the top structure helps us determine which bottom structure to recurse in).

In this and the next subsection, we use N to denote the size of the global point set P_0, and reserve the symbol n for the size of an arbitrary subset P in the tree T.



(Left) A cell $\mathrm{Vor}(p, P)$ ($p \in P_S$) drawn with a ray $\overrightarrow{pq}$; the separator cells are shaded. (Right) The same cell $\mathrm{Vor}(p, P)$ drawn inside the larger cell $\mathrm{Vor}(p, P_S)$.

Figure 2.1: Ray shooting in Voronoi diagrams

2.3.2 The Query Algorithm

Nearest neighbor queries. With the tree structure T defined in subsection 2.3.1, the following query algorithm is natural to think of. To find the nearest neighbor of q in a given subset P in T, we first find the nearest neighbor p of q in the separator subset P_S recursively. If the actual nearest neighbor is not in P_S, we deduce which cluster P_i it belongs to and make a second recursive call to find the nearest neighbor of q in P_i.

To determine the right cluster to recurse in, we find the edge of $\mathrm{Vor}(p, P)$ hit by the ray $\overrightarrow{pq}$ and let P_i be the cluster containing the point in $P - \{p\}$ that defines this edge. See figure 2.1 (left). Finding this edge is simply a "one-dimensional" problem along the boundary of the convex polygon $\mathrm{Vor}(p, P)$ and is doable by binary search.

To see correctness, observe that all points on the line segment $\overline{pq}$ have the same nearest neighbor in P_S, namely p, by convexity of $\mathrm{Vor}(p, P_S)$. Thus, all cells $\mathrm{Vor}(p', P)$ hit by $\overline{pq}$, with the exception of $\mathrm{Vor}(p, P)$, must have $p' \notin P_S$. So, all such points p', including the nearest neighbor of q in P, must belong to the same cluster.

Unfortunately, the above approach has one major flaw: the binary search is trivially implementable in the traditional model, but not so in the permutation+bits model. To form the one-dimensional lists for all the convex polygons $\mathrm{Vor}(p, P)$, we need pointers with logarithmic number of bits, but this would slow down search to $O(\log^2 n)$ time, for the same reason mentioned in section 2.2. (If all convex polygons have constant size, this issue goes away, but there could be many points with many Voronoi neighbors.) Secondly, the number of extra bits used would be slightly superlinear.

At least the second issue can be resolved, by the following observation: to find the cluster P_i, it suffices to search in $\mathrm{Vor}(p, P_S)$ instead of $\mathrm{Vor}(p, P)$. This observation is helpful, as $|P_S|$ is much smaller than $|P|$ (though not small enough to make the problem trivial, as $|P_S| \gg \sqrt{|P|}$ in the separator theorem). Throughout the section, the $\widetilde{O}$ notation hides polylogarithmic factor.

Observation 2.3.2 *We can preprocess a subset P in T into a data structure with $\widetilde{O}(|P_S|)$ bits so that:*

Given a query ray $\overrightarrow{pq}$ with $p \in P_S$ and given the point $p' \in P_S - \{p\}$ that defines the edge e of $\mathrm{Vor}(p, P_S)$ *hit by $\overrightarrow{pq}$, we can determine the cluster P_i containing the point in $P - \{p\}$ that defines the edge of* $\mathrm{Vor}(p, P)$ *hit by $\overrightarrow{pq}$ in $O(\log |P|)$ time.*

Proof: Consider the boundary of the convex polygon $\mathrm{Vor}(p, P)$. Blacken those edges that are adjacent to separator cells $\mathrm{Vor}(p', P)$ with $p' \in P_S - \{p\}$; color the remaining subchains white. Since the points in $P - \{p\}$ that define edges in the same white subchain lie in the same cluster, it suffices to determine which white subchain is hit by the query ray.

Consider the relationship between the two convex polygons $\mathrm{Vor}(p, P_S)$ and $\mathrm{Vor}(p, P)$. The former contains the latter. Furthermore, the black edges of $\mathrm{Vor}(p, P)$ remain on the Voronoi diagram of P_S and thus lie on the boundary of $\mathrm{Vor}(p, P_S)$. See figure 2.1 (right). Determining which white subchain of $\mathrm{Vor}(p, P)$ is hit by a ray reduces to determining which *cap*—i.e., connected component of $\mathrm{Vor}(p, P_S) - \mathrm{Vor}(p, P)$—is hit by the ray.

If we know the edge e of $\mathrm{Vor}(p, P_S)$ hit by the ray, there are one or two caps that involve e; these are the only caps that may be hit by the ray. If there are two, we can deduce which of the two is the answer by comparing the ray with an arbitrary vertex v on $e \cap \mathrm{Vor}(p, P)$.

During preprocessing, we simply precompute all answers in a table: for each pair p and p' defining a Voronoi edge e of $\mathrm{Vor}(P_S)$, we store the cluster index for each of the at most two caps, and also the vertex v if necessary. Since the number of edges in a 2D Voronoi diagram is linear, the table can be stored within the array of extra bits of size $\widetilde{O}(|P_S|)$ by using perfect hashing [60]; the parameters associated with the required hash functions take O(1) space, which can be stored as well. The normal $O(1)$-time worst-case query algorithm for perfect hashing now takes $O(\log |P|)$ time, since each entry of the table (and parameter of the hash functions) requires logarithmic number of bits. $\square$

To summarize, the main bottleneck is the following subproblem:

The Ray Problem. Preprocess a subset P in the tree T so that given a query ray $\overrightarrow{pq}$ originating from a point $p \in P$, we can quickly find the point $p' \in P - \{p\}$ that defines the edge of $\mathrm{Vor}(p, P)$ hit by $\overrightarrow{pq}$. Equivalently, we want to find the first line $\ell(p, p')$ hit by $\overrightarrow{pq}$ over all $p' \in P - \{p\}$, where $\ell(p, p')$ denotes the bisector between p and p'.

If there is an efficient solution to this ray problem, called Ray-Query(P, $\overrightarrow{pq}$), then we can answer nearest neighbor queries by the following pseudocode NN-Query() (omitting obvious base cases).

```
Algorithm NN-Query($P$, $q$):
1.  $p$ = NN-Query($P_S$, $q$)
2.  $p'$ = Ray-Query($P_S$, $\overrightarrow{pq}$)
3.  determine $i$ from $p$ and $p'$ by Observation 2.3.2
4.  $p''$ = NN-Query($P_i$, $q$)
5.  if $q$ is closer to $p''$ than $p$ then return $p''$
    else return $p$
```

A remark on implementation: We assume that the sizes of P_S and the P_i's have been stored as part of the data structure (which requires only $\widetilde{O}(b)$ extra space, which is smaller than $|P_S| = O(\sqrt{bn})$). A point is represented by its location in the current input subarray. We have ignored issues surrounding standard address calculations in both the input array and the extra bit array, which are doable via offsets. These calculations are essential to ensure pointer costs ($O(\log n)$) indeed decrease as the number of points n in the subarray decreases.

Let $S(n)$ and $S_{\text{ray}}(n)$ denote the number of bits required by a data structure for the nearest neighbor and the ray problem for $|P| = n$ respectively. Let $Q(n)$ and $Q_{\text{ray}}(n)$ denote the query time for these two problems. Then we have the recurrences:

$$S(n) = S(n_S) + \textstyle\sum_i S(n_i) + S_{\text{ray}}(n_S) + \widetilde{O}(n_S) \tag{2.1}$$

$$Q(n) = Q(n_S) + \max_i Q(n_i) + Q_{\text{ray}}(n_S) + O(\log n), \tag{2.2}$$

where $n_S = |P_S| = O(\sqrt{bn})$, $n_i = |P_i| = O(n/b)$, and $n_S + \sum_i n_i = n$.

Ray queries. Although the ray problem appears easier (more "one-dimensional") than the original problem, for some time we were not able to come up with an efficient algorithm under the permutation+bits model, until we gave up on the binary search approach altogether. Our new idea is simple: why not exploit the 2D tree structure T that we already have to solve this one-dimensional problem as well? The query time will not be logarithmic (but will be $o(\log^2 n)$), and we will need significantly more than linear space, but fortunately the ray problem is only required for small (separator) subsets anyway (as we have used Observation 2.3.2).

Again we are unable to solve the problem directly but need a subroutine for another subproblem, a bichromatic version of the ray subproblem. This version has two main differences: the source of a ray is not from the same given subset P but is from another subset Q, and the Voronoi diagram of interest is not Vor(P) but Vor($P \cup \{p\}$) for a query point $p \in Q$.

The Bichromatic Ray Problem. Preprocess two disjoint subsets P (the "red" points) and Q (the "blue" points) in the tree T so that given a query ray $\overrightarrow{pq}$ originating from a point $p \in Q$, we can quickly find the point $p' \in P$ that defines the edges of $\mathrm{Vor}(p, P \cup \{p\})$ hit by $\overrightarrow{pq}$. Equivalently, we want to find the first bisector line $\ell(p, p')$ hit by $\overrightarrow{pq}$ over all $p' \in P$.

If there is an efficient solution to the above subproblem, called Bichromatic-Ray-Query(P, Q, $\overrightarrow{pq}$), then we can solve the ray problem by the following pseudocode:

```
Algorithm Ray-Query($P$, $\overrightarrow{pq}$), where $p \in P$:
1. if $p \in P_S$ then
2.     $p' =$ Ray-Query($P_S$, $\overrightarrow{pq}$)
3.     determine $i$ from $p$ and $p'$ by Observation 2.3.2
4.     $p'' =$ Bichromatic-Ray-Query($P_i$, $P_S$, $\overrightarrow{pq}$)
5. else
6.     determine $i$ with $p \in P_i$
7.     $p' =$ Bichromatic-Ray-Query($P_S$, $P_i$, $\overrightarrow{pq}$)
8.     $p'' =$ Ray-Query($P_i$, $\overrightarrow{pq}$)
9. if $\overrightarrow{pq}$ hits $\ell(p, p'')$ before $\ell(p, p')$ then return $p''$
   else return $p'$
```

The correctness immediately follows from the same Observation 2.3.2 and the "decomposability" of the ray problem. Note that line 6 takes $O(1)$ time.

Let $S_{\mathrm{bi}}(n, m)$ and $Q_{\mathrm{bi}}(n, m)$ denote the number of bits and query time required by a data structure for the bichromatic ray problem for $|P| = n$ and $|Q| = m$ respectively. Then we have the recurrences:

$$S_{\mathrm{ray}}(n) = S_{\mathrm{ray}}(n_S) + \textstyle\sum_i S_{\mathrm{ray}}(n_i) + \sum_i (S_{\mathrm{bi}}(n_i, n_S) + S_{\mathrm{bi}}(n_S, n_i)) \; + \; \widetilde{O}(n_S) \quad (2.3)$$

$$Q_{\mathrm{ray}}(n) = max\,\{Q_{\mathrm{ray}}(n_S) + \max_i Q_{\mathrm{bi}}(n_i, n_S), \max_i (Q_{\mathrm{ray}}(n_i) + Q_{\mathrm{bi}}(n_S, n_i))\} + O(\log n). \quad (2.4)$$

Bichromatic ray queries. To complete the solution, we need to present an algorithm for the bichromatic ray problem. For this, we use the same recursive approach for a third (and final!) time. This time, we need a variant of Observation 2.3.2 given below, where the space usage might at first appear too large but turns out not to be a problem if parameters are chosen carefully enough.

Observation 2.3.3 *We can preprocess two subsets P and Q of T into a data structure with $\widetilde{O}(|P_S| \cdot |Q|)$ bits so that:*

Given a ray $\overrightarrow{pq}$ with $p \in Q$ and a point $p' \in P_S$ that defines the edge of $\mathrm{Vor}(p, P_S \cup \{p\})$ *hit by $\overrightarrow{pq}$, we can determine the index i of the cluster containing the point in P that defines the edge of* $\mathrm{Vor}(p, P \cup \{p\})$ *hit by $\overrightarrow{pq}$ in $O(\log |P|)$ time.*

Proof: We modify the proof of Observation 2.3.2, considering $\mathrm{Vor}(p, P \cup \{p\})$ instead of $\mathrm{Vor}(p, P)$. We blacken those edges that are adjacent to $\mathrm{Vor}(p', P \cup \{p\})$ with $p' \in P_S$. We add one easy fact: for any two points $p_1, p_2 \in P$, if $\mathrm{Vor}(p_1, P \cup \{p\})$ and $\mathrm{Vor}(p_2, P \cup \{p\})$ are adjacent, then $\mathrm{Vor}(p_1, P)$ and $\mathrm{Vor}(p_2, P)$ are also adjacent. Thus, points in P that define edges in the same white subchain still lie in the same cluster.

By the same reasoning as before, for each pair $p \in Q$ and $p' \in P_S$, we can generate at most two possible answers. We store them all in a table as before, except that this time we don't need hashing. The size of the two-dimensional table is trivially $\widetilde{O}(|P_S| \cdot |Q|)$. Note that the query time $O(\log |P|)$ does not depend on Q, since each entry of the table has $O(\log |P|)$ bits. $\square$

The pseudocode for bichromatic ray queries is as follows:

```
Algorithm Bichromatic-Ray-Query($P$, $Q$, $\overrightarrow{pq}$),
where $p \in Q$:
1.  $p'$ = Bichromatic-Ray-Query($P_S$, $Q$, $\overrightarrow{pq}$)
2.  determine $i$ from $p$ and $p'$ by Observation 2.3.3
3.  $p''$ = Bichromatic-Ray-Query($P_i$, $Q$, $\overrightarrow{pq}$)
4.  if $\overrightarrow{pq}$ hits $\ell(p, p'')$ before $\ell(p, p')$ then return $p''$
    else return $p'$
```

Correctness is self-evident. We have these recurrences:

$$S_{\mathrm{bi}}(n, m) = S_{\mathrm{bi}}(n_S, m) + \textstyle\sum_i S_{\mathrm{bi}}(n_i, m) + \widetilde{O}(n_S m) \tag{2.5}$$

$$Q_{\mathrm{bi}}(n, m) = Q_{\mathrm{bi}}(n_S, m) + Q_{\mathrm{bi}}(n_i, m) + O(\log n). \tag{2.6}$$

2.3.3 Analysis

First version. For the analysis, we first describe a simple choice of b that leads to an $o(\log^2 n)$ query time bound with linear space: namely, we set $b = n^{0.22}$ for each subset P of size n in the tree T. Here, $n_S = O(\sqrt{bn}) = O(n^{0.61})$ and $n_i = O(n/b) = O(n^{0.78})$. The solutions to the recurrences, while technically straightforward, are delicate and rely on two numerical facts: $0.61^{1.95} + 0.78^{1.95} < 1$ and $0.61 \cdot 1.61 < 1$. Below, c is some constant and $\varepsilon > 0$ is any sufficiently small constant.

$$\begin{aligned}
\text{by (2.6): } Q_{\text{bi}}(n,m) &= Q_{\text{bi}}(cn^{0.61},m)+Q_{\text{bi}}(cn^{0.78},m)+O(\log n)\\
\Longrightarrow\ Q_{\text{bi}}(n,m) &= O(\log^{1.95-\varepsilon} n)\\
\text{by (2.4): } Q_{\text{ray}}(n) &= Q_{\text{ray}}(cn^{0.78})+O(\log^{1.95-\varepsilon} n)\\
\Longrightarrow\ Q_{\text{ray}}(n) &= O(\log^{1.95} n)\\
\text{by (2.2): } Q(n) &= Q(cn^{0.61})+Q(cn^{0.78})+O(\log^{1.95-\varepsilon} n)\\
\Longrightarrow\ Q(n) &= O(\log^{1.95} n)\\
\text{by (2.5): } S_{\text{bi}}(n,m) &= \widetilde{O}(nm)\\
\text{by (2.3): } S_{\text{ray}}(n) &= \widetilde{O}(n^{1.61})\\
\text{by (2.1): } S(n) &= S(n_S)+\textstyle\sum_i S(n_i)+\widetilde{O}((n^{0.61})^{1.61})\\
\Longrightarrow\ S(n) &= O(n).
\end{aligned}$$

Second version. Now, to improve upon the $O(\log^{1.95} n)$ query bound, we use a more clever choice of b, switching between two cases. We first mark subsets in the tree T as *ordinary* or *special* according to the following rules:

> The global set P_0 is ordinary. If P is ordinary, then P_S is special but $P_1, P_2, \ldots$ are ordinary. If P is special, then $P_S, P_1, P_2, \ldots$ are all special.

For each special subset P of size n, we set $b = n^{1/3}$, so that $n_S = O(\sqrt{bn}) = O(n^{2/3})$ and $n_i = O(n/b) = O(n^{2/3})$. For each ordinary subset P of size n, we set $b = n^{2\varepsilon}$, so that $n_S = O(n^{1/2+\varepsilon})$ and $n_i = O(n^{1-2\varepsilon})$.

The solution to the recurrences becomes as follows, where $\gamma := \log_{3/2} 2 < 1.71$, and functions $Q(\cdot), S(\cdot), \ldots$ marked with superscripts * represent the specified time/space requirement for special subsets P, while functions without superscripts are for ordinary subsets P:

$$\begin{aligned}
\text{by (2.6): } Q^*_{\text{bi}}(n,m) &= 2Q^*_{\text{bi}}(cn^{2/3},m)+O(\log n)\\
\Longrightarrow\ Q^*_{\text{bi}}(n,m) &= O(\log^{\gamma} n)\\
\text{by (2.4): } Q^*_{\text{ray}}(n) &= Q^*_{\text{ray}}(cn^{2/3})+O(\log^{\gamma} n)\\
\Longrightarrow\ Q^*_{\text{ray}}(n) &= O(\log^{\gamma} n)\\
\text{by (2.2): } Q^*(n) &= 2Q^*(cn^{2/3})+O(\log^{\gamma} n)\\
\Longrightarrow\ Q^*(n) &= O(\log^{\gamma} n\log\log n)\\
Q(n) &= Q^*(cn^{1/2+\varepsilon})+Q(cn^{1-2\varepsilon})+O(\log^{\gamma} n)\\
&= Q(cn^{1-2\varepsilon})+O(\log^{\gamma} n\log\log n)\\
\Longrightarrow\ Q(n) &= O(\log^{\gamma} n\log\log n)\\
\text{by (2.5): } S^*_{\text{bi}}(n,m) &= \widetilde{O}(nm)\\
\text{by (2.3): } S^*_{\text{ray}}(n) &= \widetilde{O}(n^{5/3})\\
\text{by (2.1): } S^*(n) &= \widetilde{O}((n^{2/3})^{5/3}) = \widetilde{O}(n^{10/9})\\
S(n) &= S^*(cn^{1/2+\varepsilon})+\textstyle\sum_i S(n_i)+\widetilde{O}((n^{1/2+\varepsilon})^{5/3})\\
&= \textstyle\sum_i S(n_i)+o(n^{1-\varepsilon})\\
\Longrightarrow\ S(n) &= O(n).
\end{aligned}$$

We can in fact make the space bound at most δN for an arbitrarily small constant $\delta > 0$: just terminate the recursion when $|P| = n$ drops below a sufficiently large constant. The solution to the $S(n)$ recurrence now yields at most δn. All three types of queries can be handled trivially in $O(1)$ time in the base cases when the subset P has constant size, and so the overall query time is still $O(\log^\gamma N \log\log N)$.

2.3.4 In-Place Version

Finally, we convert our data structure in the permutation+bits model into an in-place data structure with no extra space at all.

We apply the following well-known bit-encoding trick, used in many previous work on in-place algorithms (e.g., [81]): Divide the input array into consecutive pairs. For each pair (p, q), we compare p and q lexicographically, and permute the pair in the array to encode a bit so that if the bit is 0, the smaller point appears first, else the larger point appears first. (We may assume no two points are identical, by removing duplicates.) As long as $\delta < 1/2$, this scheme can encode the entire data structure within the array.

We argue that our data structure remains valid after such pairs are permuted. First, in the application of the separator theorem, assuming the given set has even size, we may ensure that all subsets have even size. Indeed, if a cluster P_i has odd size, we can move a point over to P_S; the separator size $|P_S|$ would still be $O(\sqrt{bn} + b) = O(\sqrt{bn})$. Thus, subsets in our tree T still reside in contiguous input subarrays. When the data structure refers to a point p, the actual location i of p may now be off by 1. We use a different representation of p that is unaffected by permutations of pairs: namely, we take the index $\lfloor i/2 \rfloor$ together with an extra bit indicating whether p is the smaller or the larger point of the $\lfloor i/2 \rfloor$-th pair in the input array. Given the index and the bit, we can recover the actual location of p.

We conclude:

Theorem 2.3.4 *Given an array of N points in the plane, we can permute the array without using extra storage so that given any query point, its nearest neighbor can be found in $O(\log^{1.71} N)$ time.*

We have omitted preprocessing cost in the analysis, but it is possible to show that our data structure can be built in $O(n \operatorname{polylog} n)$ deterministic time by a non-in-place algorithm (using a known efficient deterministic hashing scheme such as [65]).

2.4 Klee's Measure Problem

In this section, we give in-place algorithms for three related problems. Klee's measure problem in 2D is to compute the area/perimeter of the union of a set of axis-aligned rectangles. In the traditional model, it can be solved in $O(n \log n)$ time with $O(n)$ space [91]. Given a set of n axis-aligned rectangles in an input array, we can solve Klee's measure problem in $O(n^{3/2}\log n)$ time using $O(\sqrt{n})$ extra space. If coordinates are $O(\log n)$-bit integers, we improve the time bound to $O(n\log^2 n)$ and the space bound to $O(1)$. Given a set of axis-aligned rectangles, the point covered by the maximum number of rectangles can be located in $O(n\log^2 n)$ time with $O(1)$ space. Given a set of n points in the plane, we can find the axis-aligned unit square that covers the maximum number of points in $O(n\log^2 n)$ time with $O(1)$ extra space. In this section, if not specified, all rectangles are axis-aligned.

2.4.1 The Sweep-Line Algorithm

We follow the traditional sweep-line idea to solve Klee's problem [91]. To perform the sweep over the plane from left to right, we need a priority queue Q to record the events to process, and also need a structure T to maintain the rectangles intersecting the vertical sweep line. The structure T should be able to maintain the total length *cover(T)* of the portion of the sweep line covered by intersecting rectangles and should also support insertion and deletion of rectangles. With Q and T, we can solve Klee's problem as follows, where $x(e)$ returns the x-coordinate of an event e:

```
**Algorithm** Klee($R$), where $R$ is the set of given rectangles:
Set $Q$ and $T$ to empty
For each left $x$-coordinate of the rectangle
    Add a "left-end" event into $Q$
For each right $x$-coordinate of the rectangle
    Add a "right-end" event into $Q$
Set $area = 0$
Set $x_{\text{pre}} = -\infty$
While $Q$ is not empty
    $\text{area} = \text{area} + \text{cover}(T) \times (x(e) - x_{\text{pre}})$    (*)
    Case: left-end event
        Add the corresponding rectangle into $T$
    Case: right-end event
        Remove the corresponding rectangle from $T$
    $x_{\text{pre}} = x(e)$
Return *area*
```

The perimeter of the rectangles can be computed in a similar way, by replacing line (*) in the algorithm with the proper calculation.

2.4.2 An $O(n^{3/2}\log n)$ Solution

In this subsection, we present an algorithm running in $O(n^{3/2}\log n)$ time with $O(\sqrt{n})$ space.

Kd-trees are widely used to index points in high dimensions [40]. We use kd-trees in a new way. For each rectangle $(x_1, x_2) \times (y_1, y_2)$, we map it to a 2D point (y_1, y_2). By using this idea, we can maintain both Q and T within $O(\sqrt{n})$ extra space as follows.

Our kd-tree Q_{kd} can be stored in the input array without using any extra space, as noted in [18]. When we initialize the structure, the median is picked by an in-place selection algorithm [73].

For a given horizontal *slab* $I = (-\infty, +\infty) \times (y_b, y_t)$, we say an interval (y_1, y_2) *spans* over I if it contains I, and say that it *crosses* I if it intersects I but does not span over I. We also say a rectangle *spans* over I if its y-interval spans over I, and say a rectangle *crosses* I if its y-interval crosses I.

To report all rectangles crossing (y_b, y_t), we query Q_{kd} to find all rectangles crossing (y_1, y_2) such that $y_b \le y_1 \le y_t$ or $y_b \le y_2 \le y_t$, and report all qualified rectangles. We can query Q_{kd} in $O(\sqrt{n} + m)$ time [40], where n is the number of rectangles and m is the number of rectangles crossing (y_b, y_t).

Before we initialize Q_{kd} in the array, we horizontally divide the plane into $O(\sqrt{n})$ strips, such that each strip contains $\sqrt{n}$ y-coordinates of rectangles. With $\sqrt{n}$ extra space, this can be done by $O(\sqrt{n})$ linear scans through the array. For each slab σ, we store three pieces of information:

- the boundary of the slab $r_\sigma = (y_b, y_t)$;
- the length l_σ of the sweep line covered by rectangles crossing r_σ;
- the number c_σ of rectangles spanning over r_σ.

All this additional information about the slabs are stored in a structure called T_{v1} using $O(\sqrt{n})$ space, sorted by the the boundaries of slabs top-down. When a new rectangle is added into or removed from T_{v1}, we (i) recalculate l_σ of at most two slabs σ crossed by this new rectangle; (ii) update c_σ of the strips over which this new rectangles spans; and (iii) add up the total covered length from all strips. Step (i) takes $O(\sqrt{n} \log n)$ time with $O(\sqrt{n})$ space, by doing a bottom-to-top scan over all rectangles crossing this slab. Step (ii) takes $O(\sqrt{n})$ time with $O(1)$ space, by checking each slab σ spanned by the new rectangle and updating each c_σ. Step (iii) takes $O(\sqrt{n})$ time with $O(1)$ space.

To maintain the priority queue, knowing the value of current event, we find the next $\sqrt{n}$ events by scanning the input array once, and store them using $O(\sqrt{n})$ space in a sorted list, called Q_{v1}. The amortized time cost for each event is $O(\sqrt{n} \log n)$.

To perform the sweep-line algorithm in subsection 2.4.1, we use T_{v1} to replace T, and use Q_{v1} to replace Q. Because one update in T_{v1} takes $O(\sqrt{n} \log n)$ time, and there are $O(n)$ events, the resulting algorithm takes $O(n^{3/2} \log n)$ time with only $O(\sqrt{n})$ extra space.

Therefore, we have

Theorem 2.4.1 *Given an array of n axis-aligned rectangles in $\mathbb{R}^2$, the area/perimeter of the union of rectangles can be computed in $O(n^{3/2} \log n)$ time with $O(\sqrt{n})$ extra space.*

2.4.3 An $O(n \log^2 n)$ Solution

We now describe an improved algorithm, which only takes $O(n \log^2 n)$ time with $O(1)$ extra space. Similar to the implicit data structure for nearest neighbor queries, we will first show an algorithm using $O(n)$ bits in the permutation+bits model, and then make it in-place. In this subsection, we assume all coordinates are $O(\log n)$-bit integers.

Solving Klee's measure problem in a slab. We consider an extension of the problem: given a set of rectangles and a horizontal slab, computing the area/perimeter of the region covered by the union of rectangles inside the slab.

Given a set of n rectangles spanning over a slab, computing the area/perimeter covered by the rectangles may be viewed as a 1D version Klee's measure problem. Each rectangle may be viewed as a line segment in the x-direction. With Klee's algorithm [72], we can compute the length of the union of the line segments. As suggested by Vahrenhold [105], this problem can be solved in-place by modifying the heapsort algorithm in $O(n \log n)$ time. Consequently, we can compute the area/perimeter covered by the rectangles.

The area/perimeter covered by the rectangles crossing the slab can be computed by the sweep-line algorithm presented in subsection 2.4.1. Here, we only describe the data structure of T and Q in detail.

We maintain T using a segment tree T_P and a priority queue Q using an AVL tree Q_P. These two data structures are stored as extra bits in the permutation+bits model.

The segment tree is represented as an AVL tree [40]. Each node represents a slab, and the root node represents the given slab in the problem. At each internal node, its slab is divided into two sub-slabs, and the sub-slabs are stored in its child nodes. To support the "cover" operation of T_P, in each node, we store (1) the number of segments spanning over this slab, and (2) the length covered by the segments crossing this slab. This information can be represented with $O(\log n)$ bits, because the coordinates are $O(\log n)$-bit integers. The boundaries and the splitting point of the slab are represented with pointers to the rectangles defining them, which takes $O(\log n)$ bits in each node.

As the sweep line intersects a rectangle, the corresponding line segment is inserted in the segment tree. We update the information in all nodes along the path to the leaf, create a new node at the leaf level, and then balance the tree. Similarly, as the sweep line leaves a rectangle, the corresponding line segment is removed from the tree.

The priority queue is maintained in an AVL tree. Each node stores a pointer of $O(\log n)$ bits, pointing to the rectangle attached to it. The priority value of a node can be computed from x-coordinates of the rectangle and the current position of the sweep line in $O(1)$ time with $O(1)$ space. The query and update operations on the AVL tree are standard.

Given a set R_S of rectangles spanning over the slab S and a set R_C of rectangles crossing S, we now compute the area/perimeter covered by $R_S \cup R_C$. We first partition the array into two chunks, one for R_S and one for R_C. This step takes $O(n)$ time with $O(1)$ space. Then we combine the previous two sweeping algorithms together. As the sweep line goes from left to right, we maintain the portion of the sweep line covered by R_S and R_I separately. As a side of a rectangle is touched (an event occurs), we update the corresponding data structures, compute the area/perimeter covered by rectangles between the current position and the previous position of the sweep line, and then update the position of the sweep line.

In the permutation+bits model, when an event caused by a rectangle in R_S, updating the corresponding data structures takes $O(\log^2 n)$ time; when an event caused by a rectangle in R_C, it takes $O(\log n)$ time. We also note that the size of the extra space is only related to the number of rectangles crossing the slab. Therefore, we have:

Lemma 2.4.2 *Given a set of n axis-aligned rectangles and a slab in $\mathbb{R}^2$, the area/perimeter of the union of the rectangles in the slab can be computed in $O(n_S \log n + n_C \log^2 n)$ time with $O(n_C \log n)$ bits of extra space in the permutation+bits model, where n_S is the number of rectangles spanning over the slab and n_C is the number of rectangles crossing the slab.*

Solving Klee's measure problem in the permutation+bits model. We divide the plane into $O(\log n)$ horizontal slabs such that each slab is crossed by $O(n/\log n)$ rectangles, and then run the above procedure on each slab.

The boundary of the i-th slab can be identified by solving a selection problem over the y-coordinates of the rectangles. Naively, by modifying the heapsort algorithm, this takes in $O(n \log n)$ time with $O(1)$ space. By lemma 2.4.2, Klee's problem can be solved in $O(n \log n)$ time with $O(n)$ bits of space in the permutation+bits model, since $n_S = O(n/\log n)$ and $n_I = O(n)$.

By processing all $O(\log n)$ slabs one by one, we have:

Lemma 2.4.3 *Given a set of n axis-aligned rectangles in $\mathbb{R}^2$, the area/perimeter of the union of rectangles can be computed in $O(n \log^2 n)$ time with $O(n)$ bits of extra space in the permutation+bits model.*

Solving Klee's measure problem in-place. Since the sorting algorithm is already in-place, the only data structures maintained in the extra bits are T_P and Q_P for the rectangles crossing the slab.

Again, we apply the bit-encoding technique: we group rectangles into pairs, and encode one bit per pair. We must ensure the data structure is still valid after permuting each pair of rectangles.

Recall that the rectangles have been partitioned into two groups. We assume the rectangles crossing the slab is stored in a prefix of the array, and all other rectangles are stored the second part. Before encoding data structures, we put each pair of the data in its lexicographical order. For example, we can insist that the upper side of the first rectangle is higher than the upper side of the second rectangle in each pair. We also ensure the number of rectangles crossing the slab is even, so that all pairs can be permuted.

We first build the heap for the second part of the array. In the heap, each pair of rectangles is considered as a single data unit. With the position of the sweep line and the slab, we can identify the coordinate defining the priority of the pair. The heap is only a rearrangement of pairs in the second part of the array. We still can encode bits in every pair of rectangles.

Then, we build the priority queue Q_P and the segment tree T_P for the rectangles crossing the slab. By adjusting the number of rectangles crossing a slab, we ensure the total number of extra bits is less than $n/2$ bits, then encode both data structure into the array.

Recall that in both data structures, the indices of rectangles are stored, and the array position of all rectangles crossing the slab should remain unchanged during the sweeping procedure. After the data structure is encoded into the array, the indices can be changed. We thus store the indices before encoding. When retrieving a rectangle, we locate the pair, restore them in the lexicographical order, and retrieve the rectangle by the index.

As we perform the sweeping algorithm, in the second part of the array, a pair of rectangles may change its position in the heap. This may also affect the encoded data structures. When two pairs are swapped in the heap, we can ensure the bit encoded in that array position is unchanged, which can be preserved by re-permuting each pair of rectangles in its new array position. This adjustment only brings a constant number of extra steps per swap in the heap. This will not change the running time asymptotically. Therefore, we have:

Theorem 2.4.4 *Given a set of n axis-aligned rectangles in $\mathbb{R}^2$, the area/perimeter of the union of the rectangles can be computed in $O(n \log^2 n)$ time with $O(1)$ extra space, if the coordinates are $O(\log n)$-bit integers.*

2.4.4 The Depth Problem

Given a set of n rectangles in the plane, the *depth* of a point is the number of rectangles covering it and the *depth* problem is to find a point with the maximum depth. This problem is related to Klee's measure problem, in the sense that both problems can be solved by similar techniques.

To solve this problem, we modify the information stored in the segment tree. Considering the segments corresponding to the rectangles intersecting the sweep line, we store three pieces of information in each node: (1) the maximum depth from segments crossing the slab; (2) the index of the rectangle defining the point with the above depth; (3) the number of segments spanning over this slab but not the slab in its parent node. As segments are inserted or deleted, the above information in the segment tree is updated. Since the number of extra bits in each node is still $O(\log n)$ bits, the size of the segment tree is still $O(n)$. By adjusting the size of the slab, we ensure the total size of additional data structures is no more than $n/2$ bits, then encode them in the input array as before. Therefore, we have:

Corollary 2.4.5 *Given a set of n axis-aligned rectangles in $\mathbb{R}^2$, the point covered by the maximum number of rectangles can be found in $O(n \log^2 n)$ time with $O(1)$ extra space.*

Another variation is the *unit-square covering* problem: given a set of points, locate an axis-aligned unit square covering the maximum number of points. By transforming each point to a unit square, the unit-square covering problem can be viewed as the depth problem on a set of unit squares. Therefore, we have:

Corollary 2.4.6 *Given a set of n points in $\mathbb{R}^2$, the unit-square covering the maximum number of points can be found in $O(n \log^2 n)$ time with $O(1)$ extra space.*

2.5 Final Remarks

In-place data structure for nearest neighbor queries. Our algorithm in section 2.3 uses only limited properties about Euclidean Voronoi diagrams and so is likely generalizable to solve other problems. For instance, by the same approach, we can answer point location queries in the xy-projection of the 3D lower envelope of n planes tangent to a given convex algebraic surface of constant degree. The 2D Euclidean nearest neighbor problem corresponds to the special case where the surface is the unit paraboloid [40]. In the dual, we can answer extreme point queries (finding the minimal point along a given direction) for a 3D lower hull where all vertices lie on a convex surface of constant complexity. It is unclear at the moment how to obtain $o(\log^2 n)$ query time for certain related problems, such as point location in the 2D Delaunay triangulation (or projection of a 3D lower hull), as these problems are not directly decomposable.

The most obvious open problem is to improve the $O(\log^{1.71} n)$ bound for 2D nearest neighbor search. This might be difficult and require significant new ideas, as we have already attained the seemingly best recurrence one can hope for with the separator approach ($Q(n) \approx 2Q(n^{2/3})$).

Some of the most intriguing open questions for us concern the permutation+bits model. As noted in section 2.2, even if we allow potentially huge amount of extra space, it is not clear how one can answer nearest neighbor queries in $O(\log n)$ time here. It would be exciting if a superlogarithmic lower bound could be proved in this model, for 2D nearest neighbor search or some other natural problem. The model is appealing in that it contains features of the bit-probe model but is still "compatible" with traditional computational geometry thinking (that input points form an abstract data type accessible only through comparisons of certain constant-degree algebraic functions).

Finally, while we do not claim that our in-place data structure is directly practical (constant factors might be large due to the repeated usage of planar graph separators), some of the ideas here could still have an impact on implementations. For instance, without using the bit-encoding trick, our method can still be implemented using sublinear extra space. Also, our data structure is automatically cache-oblivious, with $O(\log_B^{1.71} n)$ query cost for cache size B (although better cache-oblivious, non-in-place results were known).

Klee's measure problem. Compared to the previous results [34, 105], our algorithm for Klee's problem is the first in-place algorithm running in $O(n \operatorname{polylog} n)$ time. If the coordinates are $O(\log n)$-bit integers, in [34], we could only give an algorithm running in $O(n \log^2 n)$ time with $O(\log^2 n)$ extra space if we are allowed to destroy the input. Even without destroying the input, our new algorithm can solve Klee's measure problem taking the same running time but only using $O(1)$ extra space.

In [34], we could only solve the unit square covering problem in $O(n \log^3 n)$ time with $O(\log^2 n)$ extra space, which can be viewed as the depth problem on a set of axis-aligned unit squares. Our new algorithm can solve the depth problem on a set of axis-aligned rectangles, and it only takes $O(n \log^2 n)$ time with $O(1)$ extra space.

Chapter 3

Multi-Pass Algorithms

In this chapter, we examine the multi-pass model, where algorithms are allowed to make multiple passes over the input. Multi-pass algorithms have only been briefly touched upon in some of the previous streaming papers in geometry (e.g., [2, 103]), but have gained more attention recently for other problems, such as clustering and graph problems statistic (e.g., [11, 43, 49, 50, 67]). The history of multi-pass algorithms can be traced back to much earlier times (in compiler design and automata theory). Munro and Paterson's seminal paper [82] in 1980 defined the model and studied the classical sorting and selection problems. The results in this chapter can be seen as a (belated) continuation of Munro and Paterson's work for sorting and searching problems in dimension beyond one.

3.1 Our Results

All results in this section have been announced in SoCG'05 [26]. We present:

- an $O(n)$-time randomized algorithm for linear programming (or any LP-type problem) in fixed dimensions, using a constant number of passes and $O(n^\delta)$ space for any fixed constant $\delta > 0$;
- an $O(n)$-time algorithm for computing the convex hull of n sorted points in 2D, using a constant number of passes and $O(n^{1/2+\delta})$ space;
- an $O(n \log n)$-time algorithm for computing the convex hull of n arbitrary points in 2D, using a constant number of passes and $O(hn^\delta)$ space, where h denotes the output size;
- a lower bound proof to show that any $\lfloor 1/\delta \rfloor$-pass algorithm that solves the linear programming problem in $\mathbb{R}^2$ must require a storage of $\Omega(n^\delta)$ points;
- a lower bound proof to show that any $O(1)$-pass algorithm that can generate (and print out) the vertices of the upper hull (in any order), given n points in

$\mathbb{R}^2$ sorted from left to right, must require $\Omega(\sqrt{n/\log n})$ units of storage, where a "unit" refers to a point or $\log n$ bits of information. Here, it is assumed that to print a point to the output stream, the point must currently reside in memory[1].

3.2 2D Convex Hulls

We warm up with the most basic geometric problem, 2D convex hulls. As Munro and Paterson [82] observed, there is a simple algorithm for sorting n numbers using $O(s)$ space, $O(n/s)$ passes, and $O(n(\log n + n/s))$ total number of comparisons: the algorithm simply finds the next s smallest elements in each pass. These bounds were shown to be optimal asymptotically. Since 1D sorting reduces to 2D convex hulls, we cannot hope for a better result for 2D convex hulls. Still, we show that the same result can be attained in 2D by a similarly simple algorithm. It suffices to consider just the upper part of the convex hull between the leftmost and the rightmost vertices (known as the *upper hull*):

Theorem 3.2.1 *Given n points in $\mathbb{R}^2$ and a parameter s, we can generate the vertices of the upper hull from left to right (and print to an output stream), by an $O(n/s)$-pass algorithm that uses $O(s)$ space and runs in $O(n(\log n + n/s))$ time.*

Proof: We first present the pseudocode as a normal algorithm and afterwards consider its efficiency as a multi-pass algorithm.

```
Algorithm CH2D:
 0.  $v$ = the leftmost point
 1.  while $v \neq$ the rightmost point:
 2.      find a vertical slab $\sigma$ containing $s$ points (if possible)
             with its left wall through $v$
 3.      let $\langle q_0, \ldots, q_j \rangle$ be the upper hull of the points inside $\sigma$
 4.      for each point $p$ to the right of $\sigma$:
 5.          while $p$ is above $\overleftrightarrow{q_{j-1}q_j}$, $j = j - 1$
 6.          set $j = j + 1$ and $q_j = p$
 7.      print out $q_0, \ldots, q_j$ and set $v = q_j$
```

We prove correctness of the pseudocode. In each iteration of the main loop (lines 1–7), we start with a known hull vertex v and compute the portion of the upper hull from v up to and including the bridge (hull edge) at the right wall of σ. This is accomplished by imitating Graham's scan [91] (lines 4–6), except that points are processed not necessarily in sorted order (although points inside σ are

[1]Note that a tighter bound, $\Omega(\sqrt{n})$, has been proved by Guha and McGregor [63] recently.

examined in line 3 before points to the right of σ). It is straightforward to verify (for example, by induction) that at the end of the for loop, $q_{j-1}q_j$ is indeed the bridge at the right wall of σ. Thus, after at most $\lceil n/s \rceil$ iterations, the entire upper hull is computed.

We now analyze the pseudocode as a multi-pass algorithm. Line 2 reduces to finding the s leftmost points among those points to the right of v, and can be carried out in one pass, since we can easily maintain the s smallest elements of a stream with $O(s)$ space. The running time is linear if we maintain the s smallest elements by the following method: read in the next s elements, insert them to the current buffer (but don't sort the buffer), select the median in the buffer in $O(s)$ time, remove the s largest elements from the buffer, and repeat. Line 3 takes $O(s \log s)$ time and $O(s)$ space in main memory. Lines 4–6 require an additional pass and takes $O(n)$ time. The whole algorithm thus can be implemented with at most $2\lceil n/s \rceil$ passes, $O(s)$ space, and $O((n/s) \cdot (n + s \log s))$ running time. $\square$

Even in the more relaxed read-only model, no algorithm with substantially better time-space product is possible, due to known lower bounds for sorting [13]. The situation is unfortunate, as many geometric problems (e.g., closest pair and diameter) are at least as hard as sorting-like problems (e.g., element uniqueness) and probably do not admit space-efficient algorithms with few passes. Still, we will identify some geometric problems not threatened by the sorting lower bound in the next sections.

3.3 Fixed-Dimensional Linear Programming

We consider linear programming in fixed dimensions, where a number of (deterministic and randomized) linear-time algorithms were known in the traditional model of computation. We show that many of these algorithms can be modified to work with little space ($O(n^\delta)$) using a constant number of passes, although some are more time-efficient than others. Some of the modifications are not too difficult, while some require new ideas. We also prove a nontrivial lower bound. (In the bounds below, we suppress dependence on d but not δ.)

3.3.1 Prune-and-Search in 2D

We begin in 2D with the first published linear-time algorithm by Megiddo [77] and Dyer [46], based on prune-and-search. There are two difficulties in turning this algorithm into a multi-pass algorithm: First, since the original algorithm removes a constant fraction of the input at each iteration and consequently requires at least a logarithmic number of passes, we need to remove a larger fraction. This is accomplished by changing some parameters (instead of pairing, we use grouping of a larger size). Second, since we cannot explicitly remove any element from the input in the multi-pass model, we need to encode the current subset of surviving elements using a small amount of information and be able to retrieve this subset in a single pass whenever required. This task is accomplished by describing the current subset as the outcome of a series of processes ("filters") and retrieving the subset using a nontrivial "pipelining" technique.

Theorem 3.3.1 *Given n halfplanes in $\mathbb{R}^2$, we can compute the lowest point in their intersection by an $O(1/\delta)$-pass deterministic algorithm that uses $O((1/\delta^2)n^\delta)$ space and runs in $O((1/\delta)n^{1+\delta})$ time.*

Proof: We first define two subroutines: given a stream of halfplanes H, a parameter r, and a vertical slab σ, $\textsc{List}_{r,\sigma}(H)$ outputs a stream of vertical lines, and $\textsc{Filter}_{r,\sigma}(H)$ outputs a subset of halfplanes. (Throughout this chapter, "slabs" are implicitly vertical slabs.)

```
Algorithm LIST_{r,σ}(H):
 0. while there are remaining halfplanes in $H$:
 1.     read in the next $r$ halfplanes $h_1, \ldots, h_r$
 2.     compute the intersection $I = h_1 \cap \cdots \cap h_r$
 3.     print out the vertical lines through the vertices of $I$ inside $\sigma$
```

```
Algorithm FILTER_{r,σ}(H):
 0. while there are remaining halfplanes in $H$:
 1.     read in the next $r$ halfplanes $h_1, \ldots, h_r$
 2.     compute the intersection $I = h_1 \cap \cdots \cap h_r$
 3.     print out the halfplanes that are involved in the boundary of $I \cap \sigma$
```

For an input stream of size n, both subroutines are one-pass algorithms and require $O(r)$ space and $O((n/r) \cdot r \log r) = O(n \log r)$ time.

Given a set H of n halfplanes, our algorithm follows the outline below, where $r_0, r_1, \ldots$ is a sequence to be determined later. Here, we maintain the invariant that at each iteration i, the solution lies in a vertical slab σ_i and is defined only by halfplanes in a subset H_i.

```
Algorithm LP2D:
 0. let $\sigma_0$ be the whole plane and $H_0 = H$
 1. for $i = 0, 1, \ldots$:
 2.     if $|H_i|$ is below a constant then return solution for $H_i$ directly
 3.     divide the slab $\sigma_i$ into $r_i$ subslabs so that each subslab contains $O(1/r_i)$-th
           of the lines from $\textsc{List}_{r_i,\sigma_i}(H_i)$
 4.     decide which subslab contains the solution, and let this subslab be $\sigma_{i+1}$
 5.     $H_{i+1} = \textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$
```

We first prove correctness. Assume that the invariant holds at iteration i. After iteration i, since we know that the solution lies in σ_{i+1}, by design of Filter, only halfplanes in $H_{i+1} = \textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$ can indeed affect the solution.

Let $n_i = |H_i|$. We now show that n_i decreases rapidly. Since $\textsc{List}_{r_i,\sigma_i}(H_i)$ generates at most n_i lines, at most $O(n_i/r_i)$ of these lines are inside σ_{i+1}. Observe that if $h_1 \cap \cdots \cap h_{r_i}$ has j vertices inside σ_{i+1}, then at most $j+2$ halfplanes are involved in the boundary of $h_1 \cap \cdots \cap h_{r_i} \cap \sigma_{i+1}$. Thus, $\textsc{Filter}_{r_i,\sigma_{i+1}}(H_i)$ generates at most $J + 2\lceil n_i/r_i \rceil$ halfplanes, where J is the total number of vertices inside σ_{i+1}. As $J = O(n_i/r_i)$, we have $n_{i+1} = O(n_i/r_i)$.

One issue needs to be addressed: in the multi-pass setting, we do not have space to store the subset H_i explicitly. Instead, during each iteration, we need to re-generate H_i from the original input H, by re-running the Filter operations from scratch. In other words, we work with the following modified pseudocode:

```
Algorithm LP2D-Multi-Pass:
0′. let $\sigma_0$ be the whole plane
1′. for $i = 0, 1, \ldots$:
2′.     if the size of $\textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$ is below a constant
            return solution for $\textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$ directly
3′.     divide the slab $\sigma_i$ into $r_i$ subslabs so that each subslab contains
            $O(1/r_i)$-th of the lines from
            $\textsc{List}_{r_i,\sigma_i}(\textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$
4′.     decide which subslab contains the solution, and let this subslab be $\sigma_{i+1}$
```

We briefly observe how in general one may combine a series of one-pass processes $P_1, \ldots, P_i$ (a *pipeline*) into a single one-pass algorithm in a space-efficient manner. Each process P_i ($i > 1$) has a buffer that P_i reads from (and that P_{i-1} writes into); for theoretical purposes, we may take the buffer size to be 1. During each cycle, if none of the buffers are full, we can execute the next step of P_1. Otherwise, we identify the process P_j with the largest j whose buffer is full and execute the next step of P_j. (In this step, P_j may read from its buffer or write to P_{j+1}'s buffer, but we know that P_j's buffer is not empty and P_{j+1}'s buffer is not full.) The space required for the pipeline is thus bounded by the sum of the space required for the individual processes.

We can now analyze the cost of iteration i of our algorithm. The Filter pipeline requires $O(r_0 + \cdots + r_{i-1})$ space and $O(n_0 \log r_0 + \cdots + n_{i-1} \log r_{i-1})$ time. For line 3′, we can pipe this output stream through List and then through an algorithm for finding r_i *approximate quantiles* in a stream of n_i elements—i.e., finding elements of ranks within an additive error $O(n_i/r_i)$ from n_i/r_i, $2n_i/r_i$, $3n_i/r_i$, ... Munro and Paterson [82] provided a simple tree-based algorithm for this task that requires one pass and $O(r_i \log^2 n_i)$ space; it can be checked that the running time is $O(n_i \log(r_i \log n_i))$. (Munro and Paterson did not explicitly state the approximate quantiles problem in their paper but this subroutine was used as part of their exact, multi-pass selection algorithm; see [62] for another algorithm.)

For line $4'$, recall [46, 77] that deciding whether the solution is to the left or right of a line ℓ reduces to computing the intersection of the halfplanes at ℓ, which reduces to computing the minimum or maximum of a 1D set. Thus, we can decide which of the $O(r_i)$ subslabs contains the solution in one pass, by maintaining $O(r_i)$ minima/maxima simultaneously, with $O(r_i)$ space and $O(nr_i)$ time.

The simplest choice of parameters is $r_0 = r_1 = \cdots = r$, with $O(\log_r n)$ iterations. The algorithm thus makes $O(\log_r n)$ passes, uses $O((1/\delta)r + r\log^2 n)$ space, and runs in $O(nr\log_r n)$ time. The theorem follows, for example, by setting $r = n^{\delta/2}$ (and noting that $\log n = O((1/\delta)n^{\delta/4})$). $\square$

We can speed up the algorithm to run in *almost* linear time: this theorem gives our best deterministic result. (A faster randomized algorithm will be given later.)

Theorem 3.3.2 *The running time in theorem 3.3.1 can be improved to* $O((1/\delta)n\log^{(c)} n)$*, where* c *is any fixed integer constant and* $\log^{(c)}$ *denotes logarithm iterated* c *times.*

Proof: The algorithm is the same, but line $3'$ (approximate quantiles) and line $4'$ (the decision step) need to be done more efficiently, and the choice of parameters is different.

For line $3'$, we use a variant of Munro and Paterson's algorithm where the tree has degree b instead of 2. It is straightforward to check (see [82]) that the space bound becomes $O(br_i\log_b^2 n_i)$ and the time bound becomes $O(n_i\log(r_i\log_b n_i))$. We set $b = n^{\delta/2}$.

For line $4'$, recall that this step reduces to computing the intersection of n halfplanes at each of $O(r_i)$ vertical lines. We speed up the naive $O(nr_i)$-time approach by the following method: read in the next r_i halfplanes, compute their intersection I in $O(r_i\log r_i)$ time [91], compute the intersection of I with the $O(r_i)$ vertical lines in an additional $O(r_i\log r_i)$ time (by binary searches), update the current answers at the $O(r_i)$ vertical lines, and repeat. The space bound is still $O(r_i)$, but the running time is improved to $O((n/r_i)\cdot r_i\log r_i) = O(n\log r_i)$.

This time bound can be further improved by applying the decision step only to halfplanes in $H_i = \textsc{Filter}_{r_{i-1},\sigma_i}(\cdots(\textsc{Filter}_{r_0,\sigma_1}(H))\cdots))$. Thus, line $4'$ takes only $O(n_i\log r_i)$ time, in addition to the cost of re-executing the $\textsc{Filter}$ pipeline in a new pass.

We choose $r_0 = \log^{(c-1)} n$, $r_1 = \log^{(c-2)} n$, $\ldots$, $r_{c-2} = \log n$, and $r_{c-1} = r_c = \cdots = n^{\delta/2}$, with at most $c - 1 + \lceil 2/\delta\rceil$ iterations. Since $n_{j+1} = O(n_j/r_j)$ for all j, the running time of iteration i is now

$$O(n_0\log r_0 + \cdots + n_i\log r_i) \;=\; O\left(n\log r_0 + \sum_{i=1}^{c-1}\frac{n}{r_{i-1}}\log r_i\right) \;=\; O(n\log^{(c)} n).$$

The upper bounds on the number of passes and space are unchanged. □

We will need the following generalization for later applications:

Theorem 3.3.3 *Given n halfplanes in $\mathbb{R}^2$ and q directions, we can compute the q extreme points along the given directions in the intersection of the halfplanes by an $O(1/\delta)$-pass deterministic algorithm that uses $O((1/\delta^2)qn^\delta)$ space and runs in $O((1/\delta)(n\log^{(c)} n + n\log q))$ time.*

Proof: We modify the algorithm so that σ_i is not a single slab but the union of up to q (vertical) slabs. In line $3'$, σ_i is divided into up to $q + r_i$ subslabs. In line $4'$, σ_{i+1} is set to be the union of the subslabs containing the q solutions. In the analysis, we now have $n_{i+1} = O(qn_i/r_i)$, so we need to increase all the r_i's by a factor of q to keep the same number of iterations. □

3.3.2 Prune-and-Search in Higher Dimensions

Megiddo [78] extended his algorithm to higher dimensions, but the original algorithm seems difficult to work with in the multi-pass setting (among other things, it requires pairing of *nonadjacent* input elements). Nevertheless, with more modern tools, namely, *cuttings*, we can obtain a multi-pass variant of the prune-and-search algorithm. In fact, this version is more straightforward, because the current subset of surviving elements can be encoded by just a simplex and can be retrieved easily without any pipelining tricks. One helpful technique, of running several multi-pass algorithms "simultaneously", is illustrated.

Theorem 3.3.4 *Given n halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an $O(1/\delta^{d-1})$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O((1/\delta^{O(1)})n^{1+\delta})$ time* w.h.p.*—i.e., with probability at least $1 - 1/n^c$ for any fixed constant c.*

Proof: Let H be the set of n bounding hyperplanes. The outline of the algorithm is simple:

```
Algorithm LP-Megiddo-Multi-Pass:
0.  let $\Delta$ be the whole space
1.  repeat:
2.      if $|\{h \in H \mid h \text{ intersects } \Delta\}| \le r\log n$ then
            return the solution for $\{h \in H \mid h \text{ intersects } \Delta\}$ directly
3.      take a random sample $R$ of expected size $r\log n$
            from $\{h \in H \mid h \text{ intersects } \Delta\}$
4.      compute a triangulation $T$ of the arrangement of $R$ restricted inside $\Delta$
5.      decide which simplex in $T$ contains the solution,
            and set $\Delta$ to be this simplex
```

Correctness is self-evident, since the solution is always contained in Δ and is defined only by hyperplanes that intersect Δ.

Let Δ_i be the simplex Δ at the beginning of the i-th iteration, let $H_i = \{h \in H \mid h \text{ intersects } \Delta_i\}$, and let $n_i = |H_i|$. It is well-known [36, 66, 80] that for a random sample $R \subseteq H_i$ of size $r \log n$, w.h.p. a triangulation of the arrangement of R forms an $O(1/r)$-*cutting* of H_i—i.e., a partition of $\mathbb{R}^d$ into simplices such that each simplex intersects at most $O(n_i/r)$ hyperplanes of H_i. Since all hyperplanes intersecting Δ_{i+1} must intersect Δ_i, we have $n_{i+1} = O(n_i/r)$ w.h.p. The number of iterations is thus $O(\log_r n)$ w.h.p.

We can compute n_i (line 2) easily in one pass. Line 3 can also be easily done in another pass by Bernoulli sampling (for each hyperplane, if it intersects Δ_i, put it into R with probability $(r \log n)/n_i$). In line 4, a triangulation of size $O((r \log n)^d)$ can be found in $O((r \log n)^d)$ time and space in main memory.

For line 5, recall [78] that deciding which side of a given hyperplane h contains a solution reduces to solving a linear program restricted in h. Thus, line 5 can be done by solving $O((r \log n)^d)$ subproblems in $d-1$ dimensions. We handle these subproblems by making $O((r \log n)^d)$ calls to a $(d-1)$-dimensional algorithm not in series but in parallel. In other words, in every pass, we execute one pass of all invocations of the algorithms simultaneously; the space usage is multiplied by $(r \log n)^{O(1)}$ but not the number of passes.

Let $P_d(n)$, $S_d(n)$, and $T_d(n)$ be the number of passes, the space requirement, and the running time of the d-dimensional algorithm. Then

$$\begin{aligned}
P_d(n) &= O((\log_r n) P_{d-1}(n)), P_1(n) = 1 \\
&\Longrightarrow P_d(n) = O(\log_r^{d-1} n) \\
S_d(n) &= O((r \log n)^{O(1)} S_{d-1}(n)), S_1(n) = O(1) \\
&\Longrightarrow S_d(n) = O((r \log n)^{O(1)}) \\
T_d(n) &= O(((r \log n)^{O(1)} (\log_r n) T_{d-1}(n)), T_1(n) = O(n) \\
&\Longrightarrow T_d(n) = O(n r^{O(1)} \log^{O(1)} n).
\end{aligned}$$

The theorem follows by setting $r = n^{\Theta(\delta)}$ (and readjusting δ by a constant factor).
□

We remark that some of the polylogarithmic factors may be improved, although these are hidden under the n^δ factors. For example, the size of the triangulation T can be reduced to $O((r \log n)^{\lfloor d/2 \rfloor})$, since only one cell of the arrangement needs to be triangulated. Also, the extra $\log n$ factor in the sample size can be avoided via known techniques on cuttings [30, 80], if we abandon high-probability bounds for expected bounds. More significantly, the number of calls to the $(d-1)$-dimensional algorithm in line 5 could be greatly reduced while increasing the number of passes by only a constant factor, by following an approach from the prune-and-search linear programming algorithm by Dyer and Frieze [47].

The above algorithm can be derandomized, luckily, because of a recent result on derandomization for data streams. The running time is not as good as our earlier 2D result, however:

Theorem 3.3.5 *The algorithm in theorem 3.3.4 can be made deterministic with the same performance.*

Proof: We replace the random sample R (line 3) with a $(1/r)$-*approximation* of H_i in a suitable range space [100]. Bagchi et al. [9] have recently described a tree-based algorithm for computing a $(1/r)$-approximation of size $O(r^2 \log r)$ that requires one pass, uses $O(r^{O(1)} \log^{O(1)} n)$ space, and runs in $O(nr^{O(1)} \log^{O(1)} n)$ time.

The triangulation T (line 4) still forms an $O(1/r)$-cutting of H_i, although its size is now $O((r^2 \log r)^d)$. (Alternatively, we can replace T with an $O(1/r)$-cutting of R, which has size $O(r^d)$ and can be computed by an internal-memory algorithm.) Again we can set $r = n^{\Theta(\delta)}$. □

The same approach can be applied to the *ham-sandwich cut* problem in the 2D separable case [79]. (The decision step here requires an exact multi-pass algorithm for selection, which was provided by Munro and Paterson [82].)

Theorem 3.3.6 *Given two n-point sets A and B that are separable by a line in $\mathbb{R}^2$, we can find a line ℓ such that each side of ℓ contains $\lfloor n/2 \rfloor$ points of A and $\lfloor n/2 \rfloor$ points of B, by an $O(1/\delta^2)$-pass algorithm using $O((1/\delta)^{O(1)} n^\delta)$ space.*

3.3.3 Clarkson's Algorithm

Another linear programming algorithm that can be made to work in the multi-pass model is Clarkson's randomized algorithm [37]. His approach, without any major modification, already yields a result with few passes and sublinear space.

Theorem 3.3.7 *Given n halfspaces in $\mathbb{R}^d$, we can compute the lowest point in the intersection by a $(d+1)$-pass Las Vegas algorithm that uses $O(\sqrt{n \log n})$ space and runs in $O(n)$ time w.h.p.*

Proof: Let H be the given set of halfspaces. The nonrecursive version of Clarkson's algorithm can be paraphrased as follows:

```
Algorithm LP-Clarkson:
0.  take a random sample $R$ of expected size $r \log n$
1.  for $i = 0, \ldots, d$:
2.      let $v_i$ be the solution for $\{h \in H \mid h \in R$ or $\exists j < i,\ v_j$ violates $h\}$
3.  return $v_d$
```

Let $H_i = \{h \in H \mid h \in R \text{ or } \exists j < i,\ v_j \text{ violates } h\}$. It can be shown [37] that at least i of the halfspaces defining the optimal solution is in H_i, and thus v_d is indeed the optimal solution.

It is well-known [80] that w.h.p., a random sample R of size $r \log n$ is an $O(1/r)$-*net*—a subset $R \subseteq H$ with the property that any point violating no halfspaces in R violates at most $O(n/r)$ halfspaces in H. Thus, $|H_i| = r \log n + O(n/r)$ w.h.p.

Line 0 can be easily done in one pass. In each iteration of line 1, the subset H_i can be found in one pass, and v_i can be computed by an internal-memory linear programming algorithm in $O(|H_i|)$ time and space. Note that iteration 0 does not require a new pass (since $H_0 = R$). The theorem follows by setting $r = \sqrt{n/\log n}$. Note that if the problem could be infeasible, an additional pass is required to verify the solution. $\square$

We can combine Clarkson's algorithm with our previous prune-and-search method and thereby improve the expected running time of theorem 3.3.4 to linear:

Theorem 3.3.8 *Given n halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an $O(1/\delta^{d-1})$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O((1/\delta^{O(1)})n)$ time w.h.p.*

Proof: We use the same algorithm from the proof of theorem 3.3.7, except that line 2 is now done by feeding H_i into the algorithm in theorem 3.3.4. In other words, in every pass, we only read in a halfspace if it is in R or is violated by v_j for some $j < i$. (Note that H_i cannot be explicitly stored.) Compared to theorem 3.3.4, the number of passes is increased by a $d+1$ factor, and the space requirement is increased by an $O(r \log n)$ term. The expected running time becomes $O((1/\delta^{d-1})n + (1/\delta^{O(1)})(r \log n + n/r)^{1+\delta})$. The theorem follows by setting $r = n^\delta$. $\square$

We can also obtain a multi-pass algorithm purely from Clarkson's recursive algorithm. Here, the intermediate subsets are a little harder to describe but still do not require pipelining.

Theorem 3.3.9 *Given n halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by a $2^{O(1/\delta)}$-pass Las Vegas algorithm that uses $O((1/\delta^{O(1)})n^\delta)$ space and runs in $O(2^{O(1/\delta)}n)$ time w.h.p.*

Proof: The recursive version of Clarkson's algorithm can be rewritten as follows, where the argument $\mathcal{C}$ is a small collection of pairs of the form (R, V) with a subset $R \subseteq H$ and a set V of at most $d+1$ points. (In the initial call, $\mathcal{C} = \emptyset$.)

```
Algorithm Clarkson($\mathcal{C}$):
0.  if $|\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}| \le 2r\log n$ then
        return the solution for
        $\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$
1.  take a random sample $R'$ of expected size $r\log n$ from
        $\{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$
2.  for $i = 0, \ldots, d$:
3.      $v_i = \text{Clarkson}(\mathcal{C} \cup \{(R', \{v_0, \ldots, v_{i-1}\})\})$
4.  return $v_d$
```

Let $H_{\mathcal{C}} = \{h \in H \mid \forall (R,V) \in \mathcal{C},\ (h \in R \text{ or } \exists v \in V,\ v \text{ violates } h)\}$. The same analysis shows that $|H_{\mathcal{C} \cup \{(R', \{v_0,\ldots,v_{i-1}\})\}}| \le r\log n + O(|H_{\mathcal{C}}|/r)$ w.h.p. Thus, the depth of the recursion is $O(\log_r n)$ and the number of recursive calls is $(d+1)^{O(\log_r n)}$ w.h.p., since $(d+1)^{O(\log_r n)}$ is polynomial in n. In particular, $|\mathcal{C}| = O(\log_r n)$.

We can compute $|H_{\mathcal{C}}|$ (line 0) easily in one pass in $O(n|\mathcal{C}|)$ time. Line 1 can also be done in another pass within the same time bound. The algorithm thus takes $(d+1)^{O(\log_r n)}$ passes, requires $O(r\log n \log_r n)$ space, and runs in $O((d+1)^{O(\log_r n)} n \log_r n)$ time w.h.p. Setting $r = n^{\Theta(\delta)}$ (and readjusting δ) yields the theorem. □

The algorithm can be derandomized with $O(2^{O(1/\delta)} n^{1+\delta})$ running time, like in theorem 3.3.5 (because $(1/r)$-approximations are special cases of $(1/r)$-nets).

The above theorem appears weaker than theorem 3.3.8 in terms of the dependence on δ, but Clarkson's recursive algorithm has a few advantages. First, the number of passes is polynomial in d for a fixed δ. Second, the randomized version can be applied to the entire class of *LP-type* problems [101] (including convex programming [1]).

3.3.4 A Lower Bound

We now establish a lower bound for linear programming in 2D (and consequently in higher dimensions). We show that the algorithm in theorem 3.3.1 is nearly optimal (up to a constant factor in the number of passes, and ignoring the $1/\delta^{O(1)}$ factor in the space). Our proof is based on an adversary argument by Munro and Paterson [82] that established a similar lower bound for selection. Our proof is not a straightforward adaptation, however, because (i) linear programming is different from selection (needless to say), and (ii) Munro and Paterson's proof assumes a comparison-based model of computation where the only allowable operations on the input elements are comparisons of two elements. This model is not sufficient to solve geometric problems.

Our proof works under a very general decision-tree computation model: input coefficients are real numbers of unlimited precision, and the only allowable operations on the input halfplanes are testing the sign of a function evaluated at the coefficients of a subset of halfplanes currently in memory. The test function can be any continuous function (typically but not necessarily multi-variate polynomials).

Theorem 3.3.10 *Any $\lfloor 1/\delta \rfloor$-pass algorithm that can find the lowest point in the intersection of n upper halfplanes in $\mathbb{R}^2$ must require a storage of $\Omega(n^\delta)$ points.*

For the proof of the above theorem, it is more convenient to work in dual space, where the problem is as follows: given two n-point sets P_1 and P_2 separated by the y-axis, find the *bridge*, i.e., the edge of the upper hull of $P_1 \cup P_2$ at the y-axis. We need the lemma below, which roughly states that after one pass, a problem on $2n$ points ($P_1 \cup P_2$) remains as difficult as a problem of about $2n/s$ points ($X_1 \cup X_2$):

Lemma 3.3.11 *Given two open disks $D_1, D_2 \subset \mathbb{R}^2$ separated by the y-axis, and an algorithm that can store less than s points, there exists a sequence P_j of n points inside D_j, a subset $X_j \subseteq P_j$, and an open disk $D''_j \subset D_j$, for each $j \in \{1, 2\}$, such that after we run the first pass of the algorithm on the concatenation of P_1 and P_2,*

(i) *no point of $X_1 \cup X_2$ is in memory;*

(ii) *the result of the pass would be identical if we move the points of X_1 to arbitrary points in D''_1 and the points of X_2 to arbitrary points in D''_2;*

(iii) *the bridge for $P_1 \cup P_2$ is equal to the bridge for $X_1 \cup X_2$ at the y-axis, even if we move the points of X_1 and X_2 as in* (ii)*;*

(iv) *$|X_1| = |X_2| = \lceil n/s \rceil - 1$.*

Proof: The adversary builds the first half of the input P_1 entirely using copies of s points $p_1, \ldots, p_s$, whose coordinates $(x_1, y_1, \ldots, x_s, y_s)$ are chosen from an open set $U \subset \mathbb{R}^{2s}$ to be specified later. At every step, the adversary identifies a point p_i that is currently not in memory and chooses as the next input point a new copy of p_i. When a point p_k is about to be chosen for the $\lceil n/s \rceil$-th time, the adversary stops this process, chooses copies of any point other than p_k to fill in the rest of P_1, and sets X_1 to contain all $\lceil n/s \rceil - 1$ existing copies of p_k. Observe that no two copies of p_k can reside in memory at any time, and at the end of the pass over P_1, no copy of p_k is in memory.

Initially, we set U to contain all tuples $(x_1, y_1, \ldots, x_s, y_s)$ such that the points $p_1 = (x_1, y_1), \ldots, p_s = (x_s, y_s)$ form a strictly concave chain inside D_1, and for each i, there is a tangent line ℓ_i that touches the chain only at p_i and intersects D_2. This set U is indeed nonempty and open. Whenever the algorithm performs a test, we consider the sign of the test function for each choice $(x_1, y_1, \ldots, x_s, y_s) \in U$; if not all choices yield the same sign, we refine U to a smaller open subset in which they do. At the end of the pass over P_1, since U is open, we can fix the coordinates of $p_1, \ldots, p_s$ and find a neighborhood $\hat{D}_1$ of p_k so that moving p_k to any point in

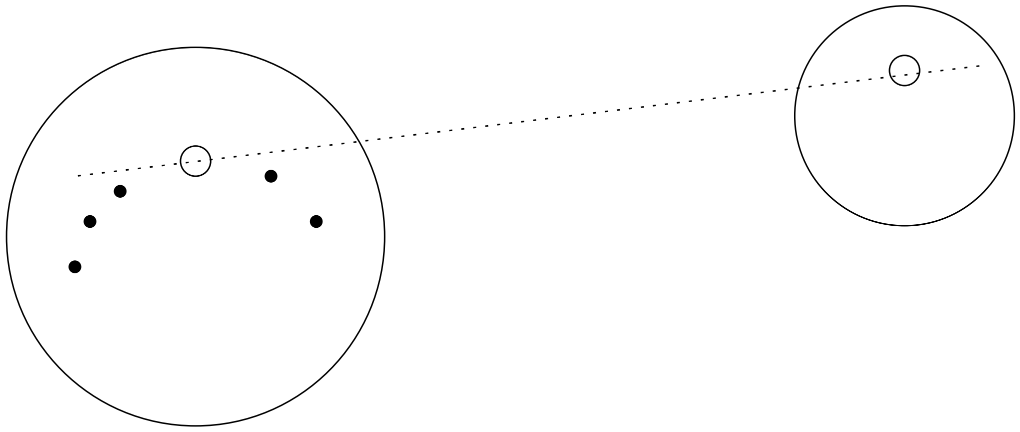


Figure 3.1: Proof of lemma 3.3.11

$\hat{D}_1$ would produce the same outcome. In fact, each copy of p_k can be moved to a different point in $\hat{D}_1$, since no two copies of p_k can participate in the same test.

We now take a point $q \in D_2$ on a tangent line ℓ_k at p_k. We can find a sufficiently small neighborhood $D'_1 \subset \hat{D}_1$ at p_k and a sufficiently small neighborhood $D'_2 \subset D_2$ at q, such that any line intersecting D'_1 and D'_2 is above every p_i $(i \neq k)$. (See figure 3.1.) Thus, the bridge between P_1 and any point set inside D'_2 is defined by a point in X_1, even if each copy of p_k is moved to an arbitrary point in D'_1.

In a similar fashion, the adversary proceeds to build the second half of the input P_2 and the subset X_2 during the remainder of the first pass, using copies of s points that form a concave chain inside D'_2, with tangents intersecting D'_1. The disks D'_1 and D'_2 are similarly refined to D''_1 and D''_2, so that the bridge between P_1 and P_2 is defined by a point in X_1 and a point in X_2, even if the points in X_1 and X_2 are moved to arbitrary points in D''_1 and D''_2 respectively. The construction is complete. □

Proof of theorem 3.3.10: We apply the lemma to obtain the sequences P_1 and P_2, together with subsets X_1 and X_2 (of size about n/s). The points of $P_1 - X_1$ and $P_2 - X_2$ are considered fixed, while the points of X_1 and X_2 will later be perturbed; such a perturbation will not change the execution of the first pass by (ii). By (iii), the answer is not yet determined after the first pass, if $|X_1| = |X_2| \geq 2$. In the second pass, we apply the lemma again, to obtain perturbations of the subsequences of the points in X_1 and X_2 only (skipping over all other points); this process yields its own versions of the subsets X_1 and X_2 (of size about n/s^2) and fixes more points. We can repeat the same process for about $\log_s n$ passes, until $|X_1| = |X_2|$ drops below a constant and the answer is determined. The theorem follows by setting $s \approx n^{\delta}$. □

3.4 2D Convex Hulls for Sorted Point Sets

We next examine special cases of the 2D convex hull problem where more efficient multi-pass algorithms are possible. In this section, we consider the case when the input points are given in sorted order according to their x-coordinates. Here, the best traditional algorithm is Graham's scan, which takes linear time [91]; however, this algorithm requires $\Omega(n)$ space in the worst case. We give different strategies with sublinear space. (Similar strategies might work also for input that is preprocessed and stored in other "natural" orders.)

We first introduce some terminology: a *partial hull* H of a point set refers to a subset of the edges of the upper hull; a *gap* of H refers to a maximal closed (vertical) slab whose interior does not intersect any edges of H; the *gap size* of H refers to the maximum number of points in a gap.

3.4.1 Two Algorithms

Our first algorithm is simple but not time-optimal.

Theorem 3.4.1 *Given n points in $\mathbb{R}^2$ sorted from left to right, we can generate the vertices of the upper hull from left to right by an $O(1/\delta)$-pass algorithm that uses $O((1/\delta^2)n^{1/2+\delta})$ space and runs in $O((1/\delta)n\log n)$ time.*

Proof: We first build a partial hull H with gap size at most $\sqrt{n}$, as follows: just divide the plane into $\sqrt{n}$ (vertical) slabs each with $\sqrt{n}$ points, and take the bridges at the walls of these slabs. Computing these $\sqrt{n}$ bridges reduces to solving $q = \sqrt{n}$ linear programs on a common set of constraints in 2D, by duality, and can be done using theorem 3.3.3.

Once H is constructed, we can finish in a single pass by examining each gap σ of H from left to right and computing the portion of the hull within σ, by applying an internal-memory algorithm (Graham's scan) to the points within σ. With $O(\sqrt{n})$ space, the final pass takes linear time. □

We now present a more complicated linear-time method by adapting the standard "merge-hull" algorithm [91] based on bottom-up divide-and-conquer. To minimize the number of passes, we implement the divide-and-conquer in a breadth-first rather than depth-first manner, and we replace binary merges with r-way merges. We also need to generalize merging of convex hulls to merging of partial hulls, accomplished by the following lemma:

Lemma 3.4.2 *Given two vertically separated point sets in $\mathbb{R}^2$, and given their partial hulls with gap size g stored in main memory, we can find the bridge between the two point sets by a 2-pass algorithm that uses $O(g)$ extra space and runs in time linear in the number of points. Moreover, at the end of each pass, the extra space usage is reduced to $O(1)$.*

Proof: We apply the binary-search algorithm by Overmars and van Leeuwen [89, 91] to the two partial hulls H_A and H_B of the two point sets A and B. In each iteration of this algorithm, we examine the median edges of H_A and H_B, perform some constant-time operations, and throw away half of one of the two partial hulls. After $O(\log n)$ iterations, one of the partial hulls, say H_A, has no edge remaining. In this case, we have identified a gap σ_A of H_A that contains a vertex defining the solution. In one pass, we read in the $O(g)$ points of A inside σ_A, compute the hull H'_A of these points, and apply the binary-search algorithm to H'_A and H_B. After $O(\log n)$ iterations, H_B has no edge remaining and we have identified a gap σ_B of H_B that contains the other vertex defining the solution. We release H'_A from memory. In a second pass, we read in the $O(g)$ points of A in σ_A and the $O(g)$ points of B in σ_B and return their bridge. □

Theorem 3.4.3 *The running time in theorem 3.4.1 can be improved to $O((1/\delta)n)$.*

Proof: We use a different method based on bottom-up merging, outlined below, to compute a partial hull H of the input point set with gap size at most $\sqrt{n}$. As in the proof of theorem 3.4.1, once H has been computed, we can fill in the gaps in a final pass in linear time.

```
Algorithm CH2D-Sorted-Points:
 0. divide the plane into $\sqrt{n}$ vertical slabs each with $\sqrt{n}$ points
 1. for each slab $\sigma$, create a partial hull of the points inside $\sigma$ with 0 edges
 2. while there is more than one partial hull in existence:
 3.     divide the current partial hulls into groups of $r$ members each,
            from left to right
 4.     for each group $G$:
 5.         compute the bridge between every pair of partial hulls in $G$
 6.         merge the $r$ partial hulls in $G$ into a single partial hull
```

All partial hulls here have gap size at most $\sqrt{n}$ and they have total size $O(\sqrt{n})$ at every iteration. After $O(\log_r n)$ iterations, we arrive at one partial hull of the whole point set, as desired. Line 5 takes $O(r^2)$ calls per group to the subroutine in lemma 3.4.2. The key idea is to perform these calls simultaneously over all groups: the number of passes is still two, the extra space usage is $O(r^2\sqrt{n})$, and the running time is $O(rn)$. (It should be noted that during a pass, as we go from one group to another, we only need to retain $O(r^2)$ amount of information per group due to the last part of lemma 3.4.2.) Once the bridges have been identified in line 5, line 6 can be done easily by using the highest bridge at each of the $r-1$ vertical separating lines. The algorithm thus makes a total of $O(\log_r n)$ passes, uses $O(r^2\sqrt{n})$ space, and runs in $O(rn\log_r n)$ time. Setting $r = n^{\delta/2}$ yields the theorem. □

3.4.2 A Lower Bound

We now show that the near-$\sqrt{n}$ space complexity is actually close to optimal. Our lower-bound proof this time is information-theoretic rather than adversary-based.

Theorem 3.4.4 *Any $O(1)$-pass algorithm that can generate (and print out) the vertices of the upper hull (in any order), given n points in $\mathbb{R}^2$ sorted from left to right, must require $\Omega(\sqrt{n/\log n})$ units of storage, where a "unit" refers to a point or $\log n$ bits of information. Here, it is assumed that to print a point to the output stream, the point must currently reside in memory.*

Proof: Set $r = 2\sqrt{n \log n}$. Create a point set P as follows: take r uniformly spaced arcs of length $\varepsilon\varepsilon'$ on the upper part of the unit circle and place a group G_i of n/r points on the i-th arc (indexed from left to right); also place an extra point at the leftmost point of the circle. Let v_i be the rightmost point in G_i. Given a string $z \in \{0,1\}^r$, define a modified point set $P(z)$, obtained by moving v_i upward by a distance of ε whenever the i-th bit of z is 1. By making ε and ε' sufficiently small, $P(z)$ obeys the following property: if the i-th bit of z is 0, all points in G_i are extreme; otherwise, no point in G_i is extreme except for v_i.

Suppose the algorithm makes p passes and at any time stores at most s units of space. Assume that $ps < \sqrt{n/\log n}$. For a given input, define the *exit configuration* to be the concatenation of the memory content after each of the p passes. (Here, for a point, its "content" comprises its index, together with an extra bit indicating whether it has been moved upward.) The number of different exit configurations is $2^{ps(\log n + O(1))} < 2^r$. By the pigeonhole principle, there exist two strings $z, z' \in \{0,1\}^r$ such that $P(z)$ and $P(z')$ have the same exit configuration.

Suppose that z and z' agree in the first $i-1$ bits, but the i-th bit of z is 0 and the i-th bit of z' is 1. Consider each pass made by the algorithm on the two inputs $P(z)$ and $P(z')$. Since the memory content at the end of the previous pass is identical and the inputs are identical up to v_i, the algorithm behaves identically before v_i is read during the pass; so, no points in G_i can be printed until v_i is read. At most s points are stored at this time; so at most s points in G_i can be printed in the rest of the pass. On input $P(z)$, the algorithm is supposed to print all n/r points in G_i in p passes. Thus, $ps \geq n/r = \Omega(\sqrt{n/\log n})$. $\square$

We comment that the assumption made in the above theorem seems reasonable, but a stronger lower bound without the assumption would be desirable. Our proof does not work, for example, if we can print just the indices (relative to the sorted input order) of the points on the hull, or if we just want to count the overall number of hull vertices.

3.5 2D Output-Sensitive Convex Hulls

For unsorted point sets, results better than theorem 3.2.1 are still possible if the output size h is small. For example, for points distributed uniformly in a square, it is known that the expected value of h is logarithmic [91] (although for this special case there is actually a simple one-pass algorithm).

We prove the following theorem by mimicking Kirkpatrick and Seidel's output-sensitive algorithm [71] based on top-down divide-and-conquer. Again, the divide-and-conquer is executed breadth-first and with a larger branching factor. (It is interesting to note that Kirkpatrick and Seidel's algorithm was originally designed to make the running time output-sensitive, not the space; other output-sensitive convex hull algorithms do not seem to work as well in the multi-pass model.)

Theorem 3.5.1 *Given n points in $\mathbb{R}^2$, we can generate the h vertices of the upper hull from left to right, by an $O(1/\delta^2)$-pass algorithm that uses $O((1/\delta^2)hn^\delta)$ space and runs in $O((1/\delta^2)n \log n)$ time.*

Proof: The "unfolded" version of Kirkpatrick and Seidel's algorithm is outlined below. Here, we repeatedly insert edges to a partial hull H until it becomes the complete hull:

Algorithm CH2D-Output-Sensitive:
```
0.  $H = \emptyset$
1.  for $i = 0, 1 \ldots$:
2.      if the gap size of $H$ is below a constant then
            compute and print the hull within each gap and return
3.      for each gap $\sigma$ of $H$:
4.          divide $\sigma$ into $r_i$ subslabs each containing $O(1/r_i)$-th of the points
5.          compute the bridges at the walls of these subslabs and insert them to $H$
```

The simplest choice of parameters is $r_0 = r_1 = \cdots = r$, with $O(\log_r n)$ iterations of the outer loop. Line 4 can be carried out by an approximate quantiles algorithm [62, 82]. Line 5 requires solving r linear programs and can be carried out by theorem 3.3.3 in $O(1/\delta)$ passes, $O((1/\delta)rm^\delta)$ space, and at most $O((1/\delta)m \log m)$ time, where m is the number of points in the gap σ.

In each iteration of the outer loop, there are at most $h+1$ gaps. The key idea is to handle all $O(h)$ iterations of the inner loop (lines 3–5) simultaneously. Each point belongs to only one gap of H, which can be identified in $O(\log h)$ time. So, the number of passes for lines 3–5 remains $O(1/\delta)$, the space usage is $O((1/\delta)hrn^\delta)$, and the running time is at most $O((1/\delta)n \log n)$. The algorithm thus makes $O((1/\delta) \log_r n)$ passes, uses $O((1/\delta)hrn^\delta)$ space, and runs in $O((1/\delta)n \log n \log_r n)$ time. The theorem follows by setting $r = n^\delta$ (and readjusting δ). $\square$

We can make the running time output-sensitive as well, provided that an upper bound on h is given. (Note that a standard trick [20] of "guessing" the output size cannot be applied, because it would increase the number of passes by a nonconstant factor.)

Theorem 3.5.2 *The running time in theorem 3.5.1 can be improved to* $O((1/\delta^2)(n \log^{(c)} n + n \log \bar{h}))$ *time, where* $\bar{h}$ *is a known upper bound on* h.

Proof: The algorithm is the same, but we provide a more careful analysis using a different choice of parameters.

For line 4, we use a variant of Munro and Paterson's approximate quantiles algorithm, as in the proof of theorem 3.3.3. For line 5, we use theorem 3.3.3. Then in iteration i of the outer loop, the running time becomes $O(n \log h + (1/\delta)(n_i \log^{(c)} n_i + n_i \log r_i))$, where n_i is the number of points inside the gaps of H at the beginning of the iteration.

We choose $r_0 = \bar{h} \log^{(c-1)} n$, $r_1 = \bar{h} \log^{(c-2)} n$, $\ldots$, $r_{c-2} = \bar{h} \log n$, and $r_{c-1} = r_c = \cdots = \bar{h} n^{\delta/2}$, with at most $c - 1 + \lceil 2/\delta \rceil$ iterations. Since $n_i = O(\bar{h}n/r_{i-1})$, we have

$$O(n_0 \log r_0) = O(n \log^{(c)} n + n \log \bar{h}),$$

and,

$$O(n_i \log r_i) = O\left(\frac{\bar{h}n}{r_{i-1}} \log r_i\right) = O(n \log \bar{h}) \quad (i \geq 1).$$

So the running time of each iteration is at most $O((1/\delta)(n \log^{(c)} n + n \log \bar{h}))$. The upper bounds on the number of passes and space are unchanged. □

We can also obtain a tradeoff result analogous to theorem 3.2.1. (Due to the sorting lower bound, our result is near optimal except for the n^{δ} and logarithmic factors.)

Theorem 3.5.3 *Given* n *points in* $\mathbb{R}^2$ *and a parameter* s*, we can generate the* h *vertices of the upper hull from left to right, by an* $O((1/\delta^2)\lceil h/s \rceil)$*-pass algorithm that uses* $O((1/\delta^2)sn^{\delta})$ *space and runs in* $O((1/\delta^2)\lceil h/s \rceil n \log n)$ *time.*

Proof: It is straightforward to modify the algorithm in theorem 3.5.1 to output the leftmost s hull edges with $O((1/\delta^2)sn^{\delta})$ space (by only keeping the leftmost s edges of H in every iteration). Repeating this process $\lceil h/s \rceil$ times yields the theorem. □

3.6 3D Convex Hulls

Finally, we consider the 3D convex hull problem. As before, due to sorting lower bounds, we cannot hope for a nontrivial result with a constant number of passes in general. Still, we show that a result analogous to theorem 3.2.1 is possible. This time, the algorithm is more involved but relies on a standard divide-and-conquer approach in dual space (similar to the proof of theorem 3.3.4 as well as an algorithm by Clarkson and Shor [38]).

Theorem 3.6.1 *Given n points in $\mathbb{R}^3$ and $s \geq n^{1/c}$ for a fixed constant c, we can generate the facets of the upper hull by an $O((n/s)\operatorname{polylog} n)$-pass algorithm that uses $O(s)$ space and runs in $O((n^2/s)\operatorname{polylog} n)$ time w.h.p.*

Proof: We solve the dual problem of computing the vertices of the lower envelope of a set H of planes in $\mathbb{R}^3$ by the following recursive procedure (initially, we call $\textsc{Env}_0(\mathbb{R}^3)$):

```
Algorithm $\textsc{Env}_i(\Delta)$:
 0.  if $i = c$ then
         compute and print the vertices of the lower envelope of
         $\{h \in H \mid h$ intersects $\Delta\}$ inside $\Delta$ and return
 1.  take a random sample $R$ of expected size $r \log n$ of $\{h \in H \mid h$ intersects $\Delta\}$
 2.  compute a triangulation $T$ of the lower envelope of $R$ restricted inside $\Delta$
 3.  for each simplex $\Delta' \in T$ do $\textsc{Env}_{i+1}(\Delta')$
```

Let $H_\Delta = \{h \in H \mid h \text{ intersects } \Delta\}$. As in the analysis of theorem 3.3.4, we have $|H_{\Delta'}| = O(|H_\Delta|/r)$ for all $\Delta' \in T$ w.h.p. So, at each leaf (on the c-th level) of the recursion, we have $|H_\Delta| = O(n/r^c)$ w.h.p. Since lower envelopes have linear complexity in $\mathbb{R}^3$, we can find a triangulation T of size $O(r \log n)$. So, there are $O((r \log n)^c)$ leaves in the recursion.

Line 0 requires a single pass with $O(n/r^c)$ space and $O(n + (n/r^c)\log(n/r^c))$ time w.h.p. Line 1 can be easily done in one pass. Line 2 takes $O(|R| \log |R|) = O(r \log^2 n)$ time [80, 91] in main memory. The overall number of passes is thus $O((r \log n)^c)$, the space usage is $O(n/r^c + r \log n)$, and the running time is $O((r \log n)^c \cdot [n + (n/r^c)\log(n/r^c) + r \log^2 n])$ w.h.p. The theorem follows by setting $r = (n/s)^{1/c}$ (and noting that $(n/s)^{1/c} = o(s)$ for $s \geq n^{1/c}$). □

By the standard lifting transformation, we immediately obtain the same result for Voronoi diagrams of 2D point sets.

3.7 Final Remarks

We have given new algorithms and lower bounds for some basic geometric problems under the multi-pass model. We hope that the techniques here (like pipelining) may be applicable to solve other geometric problems and may inspire further work. Although one-pass streaming algorithms admittedly are more desirable than multi-pass algorithms in many settings involving massive data sets, certain hardware systems such as the graphics card [2] do favor streaming computation with multiple passes. The multi-pass model can also be viewed as a simpler form of external-memory algorithms. In some ways, one might regard streaming algorithms, which process input sequentially, as the opposite of parallel algorithms. Curiously though, many of our algorithms (e.g., see theorem 3.3.4) are naturally parallelizable, as each pass is parallelizable. Apparently, designing algorithms that minimize the number of passes or rounds share some similarities to designing parallel algorithms that minimize processor time.

Chapter 4

Algorithms in the Read-Only Model

In this chapter, we consider the *read-only* model, where we are allowed read-only random access to the input, and we are concerned no longer with the number of passes, but just the running time (as measured in the standard RAM) and the amount of extra working space. The first two results are announced with the results in chapter 3 in SoCG'05.

4.1 Our Results

In this chapter, we present:

- a read-only $O(n)$-time randomized algorithm for linear programming in fixed dimensions, using $O(\log n)$ space.
- a read-only $O(n)$-time algorithm for finding the convex hull of n sorted points in $\mathbb{R}^2$, using $O(n^\delta)$ space for any fixed $\delta > 0$.
- a read-only $O(n^3/s^2)$-time deterministic algorithm for Klee's measure problem in 2D, using $O(s)$ space, where $s < n/\log n$.

4.2 Fixed-Dimensional Linear Programming

The read-only model is less restrictive than the multi-pass model. Here, we start with another well-known linear programming algorithm, Seidel's randomized incremental algorithm [97], which we have not examined in chapter 3. We apply this algorithm in a recursive fashion, in a similar way described by Chan [21] (in the context of solving "implicit" linear programs).

Theorem 4.2.1 *Given a read-only array of n halfspaces in $\mathbb{R}^d$, we can compute the lowest point in their intersection by an algorithm that runs in $O(n)$ expected time and uses $O(\log n)$ space.*

Proof: We describe a recursive algorithm, assuming that the input resides in the union of at most d subarrays each of size at most n: First divide the input into at most dr blocks of size n/r. The idea is to treat each block as a single constraint. More precisely, each block represents a convex object (the intersection of the halfspaces in the block), and the problem is to find the lowest point inside the intersection of these r convex objects—a convex programming problem. It is important to note that these objects are not explicitly constructed but are rather viewed abstractly.

The convex program (or more generally, LP-type problems) can by solved directly by an algorithm proposed by Sharir and Welzl [101]. We use a variation of Seidel's randomized incremental algorithm [97] for linear programming. This algorithm mainly performs a set of "violation tests"—deciding whether a point is outside a given object—and a set of of "basis evaluations"—computing the optimal solution for a given subset of objects. For $O(r)$ constraints, this algorithm requires an expected $O(r)$ number of violation tests and an expected $O(\log^d r)$ number of basis evaluations for a given set of d objects.

For our convex objects, each violation test can be done in $O(n/r)$ time and $O(1)$ space by scanning the n/r halfspaces in a block; a basis evaluation can be done by making a recursive call for the union of d subarrays. We therefore obtain the following recurrence for the running time for some constant c:

$$T(n) \leq (c \log^d r) T(n/r) + O(r \cdot n/r).$$

Setting r to be a sufficiently large constant (depending only on d) yields $T(n) = O(n)$. The space requirement is $O(r)$ times the depth of the recursion $O(\log_r n)$. $\square$

4.3 2D Convex Hull for Sorted Point Sets

Recall the lower bound of constructing the convex hull of a set of sorted points in the multi-pass model, where $\Omega(\sqrt{n/\log n})$ space is required for any algorithm taking $O(1)$ number of passes. Here, we present an algorithm which only uses $O(n^\delta)$ extra space and $O(n)$ time, in the read-only model.

Theorem 4.3.1 *Given a read-only array of n points in $\mathbb{R}^2$ sorted from left to right, we can generate the vertices of the upper hull from left to right by an algorithm that uses $O((1/\delta)n^\delta)$ extra space and runs in $O((1/\delta)n)$ expected time.*

Proof: We describe a recursive algorithm: First divide the input into r blocks each containing n/r points, from left to right. As in the proof of theorem 4.2.1, it is helpful to view each block abstractly as a single convex object. Consider the following variant of Graham's scan [61] to compute all tangents of the upper hull of r vertically separated convex objects $\gamma_1, \ldots, \gamma_r$, ordered from left to right:

```
Algorithm CH2D-Read-Only:
0.  $e_1$ = tangent between $\gamma_1$ and $\gamma_2$, $i_0 = 1$, $i_1 = 2$, $j = 1$
1.  for $i = 3, \ldots, r$:
2.      while $j > 0$ and $\gamma_i$ is not entirely below the line through $e_j$, $j = j - 1$
3.      $e_{j+1}$ = tangent between $\gamma_{i_j}$ and $\gamma_i$, $i_{j+1} = i$
4.      $j = j + 1$
5.  return $\langle e_1, \ldots, e_j \rangle$
```

The above method performs $O(r)$ "primitive operations"—testing whether an object is entirely below a line (line 2), and finding the tangent (bridge) between two objects (lines 0 and 3). In our case, each object is the upper hull of the points in a block. The first type of operation can be carried out in $O(n/r)$ time and $O(1)$ space by scanning the points in the block. The second type can be handled by theorem 4.2.1 in $O(n/r)$ expected time and $O(\log n)$ space. The method then gives us a partial hull with gap size at most n/r.

After this process, we take each gap of this partial hull and recursively output the upper hull of the points within each gap. The expected running time of the whole algorithm satisfies the recurrence

$$T(n) = rT(n/r) + O(r \cdot n/r),$$

which solves to $T(n) = O(n \log_r n)$. The space bound is $O(r \log_r n)$. Setting $r = n^\delta$ yields the theorem. □

We can avoid randomization if we do not mind a larger constant factor:

Theorem 4.3.2 *The algorithm in theorem 4.3.1 can be made deterministic if the running time is increased to $O(2^{O(1/\delta)} n)$.*

Proof: In the second type of operation, we want to find a bridge of the upper hull of $2n/r$ points. Instead of using theorem 4.2.1, we handle this operation by a recursive call to our upper-hull algorithm. (From the output of the algorithm, we can skip over all edges but keep only the bridge we want.) As a result, we obtain a new recurrence:

$$T(n) = O(r)T(2n/r) + O(r \cdot n/r),$$

which solves to $T(n) = O(2^{O(\log_r n)} n)$. The space bound is the same. Setting $r = n^\delta$ yields the theorem. □

4.4 Klee's Measure Problem

Unlike the problems studied in the previous two sections, there were no efficient algorithms to solve Klee's measure problem in the multi-pass model. The algorithm we present here can be viewed as a variation of the in-place algorithm described in subsection 2.4.3. Instead of dividing the plane into $O(\log n)$ slabs, we divide the plane into n/s slabs, such that each slab is crossed by s rectangles. When we solve Klee's measure problem in a slab, we maintain the segment tree structure explicitly in $O(s)$ extra space.

We combine the priority queues for intersecting rectangles and spanning rectangles together. First observe that we can implement a priority queue with $O(\log s + n/s)$ update time and $O(s)$ space as follows. We divide the rectangles in the input array into $O(s)$ blocks, and each block has $O(n/s)$ rectangles. In one block, we identify the next rectangle to process in $O(n/s)$ time. With $O(s)$ extra words of space, we maintain a priority queue of blocks explicitly. Therefore, reporting all coordinates in order takes $O(n \log s + n^2/s)$ total time in one slab[1].

With only $O(1)$ time per rectangle, we can identify whether the rectangle crosses, spans over, or is disjoint from the slab. In the sweeping algorithm, maintaining the segment tree structure explicitly takes $O(s \log s)$ total time and $O(s)$ extra space. If $s < n/\log n$, $O(n^2/s)$ dominates, then we have:

Lemma 4.4.1 *Given a set of n axis-aligned rectangles and a slab in $\mathbb{R}^2$ and $s < n/\log n$, the area/perimeter of the region covered by the rectangles in the slab can be computed in $O(n^2/s)$ time with $O(s)$ data units, in the read-only model, where n is the number of rectangles and s is the number of rectangles intersecting the slab.*

To determine the boundaries of the slabs, we can maintain another priority queue of $O(s)$ extra units based on the y-coordinates of rectangles. When a slab is determined, we solve Klee's measure problem in that slab by lemma 4.4.1. Then we proceed to the next slab. Since determining all slabs takes $O(n^2/s)$ time and there are n/s slabs, we have:

Theorem 4.4.2 *Given a set of n axis-aligned rectangles in $\mathbb{R}^2$ and $s < n/\log n$, the area/perimeter of the region covered by the rectangles can be computed in $O(n^3/s^2)$ time, with $O(s)$ data units, in the read-only model.*

Similarly, we can solve the depth problem in the read-only model. We can maintain the segment tree for the depth problem explicitly in memory, which takes $O(s \log n)$ bits. Thus, we can obtain an analog of theorem 4.4.2 for the depth problem. In fact, we can improve the time bound slightly: by using the priority queue in [90], we can have a priority queue using $O(s)$ bits and $O(n^2/s)$ total running time. By readjusting s, we have:

[1]This time bound improves slightly to $O(n^2/(s \log n))$ time, if coordinates are $O(\log n)$-bit integers, by using the heap-like priority queue proposed in [90]

Corollary 4.4.3 *Given a set of n axis-aligned rectangles in $\mathbb{R}^2$ and $s < n/\log^2 n$, the point covered by the maximum number of rectangles can be found in $O(n^3/(s^2 \log n))$ time with $O(s)$ data units in the read-only model.*

Corollary 4.4.4 *Given a set of n points in $\mathbb{R}^2$ and $s < n/\log^2 n$, the unit-square covering the maximum number of points can be found in in $O(n^3/(s^2 \log n))$ time with $O(s)$ data units in the read-only model.*

Chapter 5

Stream-Sort Algorithms

In the stream-sort model, the input data are given in one data stream. There are two ways to access these data. One is the scan pass. The input stream is scanned sequentially, and one output stream is generated. The other is the sorting pass. The input stream is sorted based on a user-defined comparator, and data are sorted in the output stream based on that order. The generated output stream of data is called an *intermediate* stream. In the next scan, this intermediate stream becomes the input stream, and another output stream is generated. All intermediate streams must be of size $O(n)$. At the end, we count how many passes are used in total, and how much space is used in memory by the program in the scan passes.

5.1 Our Results

All results in this chapter are presented in ISAAC'07 [32].

- We can construct 2D convex hulls in $O(1)$ passes with $O(n^{\varepsilon})$ extra space.
- We can construct 3D convex hulls in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space.
- We can construct a triangulation of a simple polygon in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space, where n is the number of vertices on the polygon.
- We can report all k intersections of a set of 2D line segments in $O(1)$ passes with $O(n^{\varepsilon})$ extra space, if an intermediate stream of size $O(n + k)$ is allowed.
- If we are only allowed to choose the direction to scan the data but not for sorting passes, we still can construct 2D convex hull in $O(1)$ passes with $O(n^{\varepsilon})$ extra space.

5.2 Preliminaries

Divide-and-conquer is a general strategy commonly used under the stream-sort model. With this strategy, the data set of the problem is divided into data sets for several subproblems. We describe a technique that can help us solve all subproblems simultaneously in one pass.

This technique can be described as follows. Given a stream containing multiple independent data sets and one data set per subproblem, if the elements of each data set are grouped together in the stream, we can process these data sets one after another in a single pass. In the scan pass, after scanning over the data set for one subproblem, we reset memory for the data set of the next subproblem. As defined in subsection 1.2.4, in intermediate streams, we can write an extra field for each element to identify which data set it belongs to. Storing this extra field only lengthens the stream by $O(n)$. Then, with a sorting pass, we can group the elements for the same data set together.

Now we treat the recursive calls in a divide-and-conquer algorithm as a tree structure. Each node represents one subproblem we need to solve. We can solve all problems in the same level of the recursion tree in the same pass. Therefore, we can bound the number of passes by the number of levels in the recursion tree.

Since the sorting pass is the most expensive part, we define a simpler and self-contained model in which the sorting pass is not allowed. Instead, we can only choose a direction to scan the stream (forward or backward). We call this model the *direction-flexible* model. This model is weaker than the stream-sort model.

5.3 2D Convex Hulls

We begin with a very simple example, so that we can get familiar with this model. In this section we give an algorithm that constructs the convex hull of a set of 2D points in $O(1)$ passes with $O(n^\varepsilon)$ extra space under the stream-sort model. This simple result will also be used as a sub-routine in solving later problem. Constructing the convex hull of a set of 2D points in primal space is equivalent to constructing the lower envelope of a set of halfplanes in dual space [40]. In the pseudocode below, the input H is the set of halfplanes and the output L is the lower envelope of H. The parameter B in the pseudocode below will be determined later in this section.

```
**Algorithm** 2D-Envelope($H$):
    If $|H| \leq B$
        Solve the problem directly in memory and return $L$
    Divide $H$ in $B$ disjoint sets of equal size
    For each subset $H_i$
        $L_i$ = 2D_Envelopes($H_i$)
    Merge $B$ lower envelopes $L_i$ to obtain the lower envelope $L$ and return $L$
```

Merging B lower envelopes can be done in a constant number of passes in the stream-sort model by a sweep from left to right as follows. We maintain all $O(B)$ edges intersecting the sweep line and a list of edges appearing in the merged lower envelope. Whenever the sweep line touches an intersection of two edges in the merged lower envelope, we add the appearing edge to the list. We sort all edges by the x-coordinates of their left endpoints in a sorting pass. We only keep the B edges intersecting the sweep line in memory in the scan pass. The intersections between edges are computed in memory, so the space needed is $O(B)$.

We view the recursive calls of our algorithm as a tree structure. Applying the technique presented in section 5.2, we perform all merges at the same level of the recursion tree in one round. In a sorting pass, we use the group identifier of each element as the primary key and the left x-coordinates of edges in the same group as the secondary key. In the scan pass, we simply merge each group of lower envelopes, one by one.

By setting $B = O(n^{\varepsilon})$, we ensure that there are $O(1)$ levels of the recursion tree. Thus we have:

Theorem 5.3.1 *The convex hull of a set of n 2D points can be constructed in $O(1)$ passes with $O(n^{\varepsilon})$ extra space, for any fixed $\varepsilon > 0$.*

Remark: If we are also concerned with the time taken in the scan passes, the above algorithm runs in $O(n^{1+\varepsilon})$ time in each pass. We can reduce the time bound to $O(n \operatorname{polylog} n)$ as follows. In the merging procedure, the intersections between the sweep line and envelopes are a set of points moving vertically with constant speeds. Merging B envelopes is equivalent to keeping track of the lowest point among the set of moving points. This can be maintained by a *kinetic heap* with $O(B)$ space in memory [16]. Whenever a new edge is touched by the sweep line, the velocity of the corresponding vertex is changed.

5.4 3D Convex Hulls

In this section, we highlight another useful technique, random sampling, for this model. Sampling is a well-known technique in computational geometry, but to our knowledge, it has not been applied in the stream-sort model before. A similar sampling idea is also used for constructing 3D convex hulls in the multi-pass model in section 3.6.

Given a set of 3D points, we show how to construct its 3D convex hull in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space in the stream-sort model. We follow a random sampling approach [38, 80].

We transform the problem into dual space. Each point in primal space maps to a halfspace in dual space. Constructing the convex hull of a set of 3D points is equivalent to constructing the lower envelope of a set of 3D halfspaces [40]. We first describe our algorithm in the traditional memory model, and then modify it to fit in the stream-sort model. The notations are defined as follows. The cell Δ_f defined by a triangle f refers to the vertical prism underneath f. The set of planes intersecting Δ_f is denoted by H_f. In the algorithm, the input H is a set of n 3D halfspaces, and the output E is the set of faces in the lower envelope. (See [80] for the definition of the *canonical triangulation*.)

```
**Algorithm** 3D-Envelope($H$):
Initialize an empty set $R$
If $|H| \leq B$
    Solve the problem directly in memory and return the answer
Repeat
    Sample a random subset $R$ of size $B$ in $H$
    Find the lower envelope $E_R$ of $R$
    Build the canonical triangulation $T$ for $E_R$
Until $\sum_{f \in T} |H_f| = O(n)$ and $\max_{f \in T} |H_f| = O((n/B) \log B)$
For each $f \in T$
    $E_f$ = 3D_Envelope($H_f$)
Merge all the $E_f$'s to form $E$ and return $E$
```

The set R is a randomly selected subset, which can be selected in three passes. In the first scan pass, we compute the size of the data set. When the end of the data set is reached, we add B random numbers to the output stream. With a sorting pass, we move the generated numbers to the beginning the data set. In the third scan pass, we first read these numbers in memory, then select data unit with corresponding indices, as the data set is scanned.

By the analysis from Clarkson and Shor [38] for randomly selected samples, it is known that the expected value of $\sum_{f \in T} |H_f|$ is $O(n)$. Therefore, we have $\sum_{f \in T} |H_f| \leq cn$ with probability greater than a constant, for a sufficiently large constant c. It is also known [80] that $\max_{f \in T} |H_f| \leq c'(n/B) \log B$ with probability greater than a constant, for a sufficiently large constant c'. Therefore, the expected number of iterations before both conditions are satisfied is constant.

We modify this algorithm to fit in the stream-sort model. We set B to n^{ε}. Thus, R and E_R takes $O(n^{\varepsilon})$ space in memory. The operation for choosing the set R can be done in one scan pass. Constructing E_R can be done in memory using $O(n^{\varepsilon})$ extra space.

By keeping T in memory, verifying the conditions to terminate the loop can be done by one scan pass. One iteration of the first loop can be done in $O(1)$ passes with $O(n^{\varepsilon})$ extra space.

For the second loop structure, we proceed as follows. We create a copy of h for each cell Δ_f intersected by the halfspace h. We also attach a label to each copy to identify the corresponding cell. This operation can be done in one scan pass. In the following sorting pass, we use the attached label as the key. All halfspaces are grouped together in the data stream. With the technique introduced in section 5.2, all subproblems in the same level of the recursion tree are solved simultaneously in one scan pass.

Because the total number of intersections between halfspaces and cells is $O(n)$, the duplication of halfspaces only lengthens the size of the intermediate stream by a constant factor times. The size of any subproblem is $O(n^{1-\varepsilon}\log n)$, since $B = n^{\varepsilon}$. The number of levels of the recursion tree is constant, since the size of the intermediate stream increases by a constant factor every round. Therefore, all intermediate streams are of size $O(n)$.

Because the first loop in the above algorithm terminates in a constant expected number of iterations, this algorithm takes $O(1)$ expected number of passes in one round. Therefore, the expected total number of passes is also $O(1)$.

The merging step at the end of the algorithm can be done by a sorting pass and a scan pass. In the sorting pass, we use the planes containing facets as the key and group all facets in the same plane together.

Theorem 5.4.1 *Given a set of n 3D points, its convex hull can be constructed in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space, for any fixed $\varepsilon > 0$.*

Remark: The algorithm can also be derandomized in the same bounds. Instead of making the set R random, we can use a $(1/B)$-*net* to make the set, where $B = O(n^{\varepsilon})$. A streaming algorithm by Bagchi et al. [9] can deterministically compute this $(1/B)$-net in one pass with $O(\text{polylog}\, n)$ space.

5.5 Triangulation of Simple Polygons

In this section, we describe how to triangulate a simple polygon. This algorithm is not only based on the general techniques discussed in previous sections, but also based on new interesting properties of a type of special polygons proposed in section 5.5.4.

More generally, our algorithm can triangulate an arbitrary set of disjoint line segments. Given a set L of n disjoint line segments in a plane, we show how to construct a triangulation of L (covering the convex hull of the endpoints of L and not using extra vertices) in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space, for any fixed $\varepsilon > 0$.

Before we describe our algorithm, we define some terms. By a *unimonotone* polygon, we mean an x-monotone polygon with one edge connecting its leftmost and rightmost vertex. We call this edge the *long* edge of the polygon. See fig. 5.1.

Our algorithm consists of four major phases. In section 5.5.1, we explain the construction of a trapezoidal decomposition of the line segments. In section 5.5.2, we describe the transformation of the trapezoidal decomposition to a decomposition

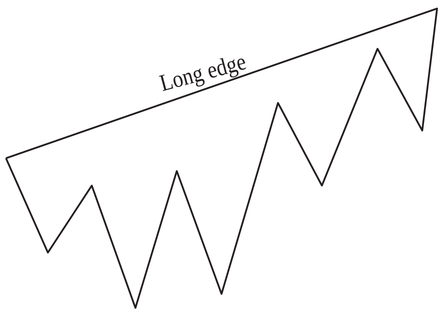


Figure 5.1: A unimonotone polygon

of unimonotone polygons. In section 5.5.3, we describe the decomposition of each unimontone polygons to a set of *special* polygons. In section 5.5.4, we show how to triangulate a special polygon. In section 5.5.5, we put these four phases together to obtain the overall algorithm.

5.5.1 Trapezoidal Decomposition of Line Segments

Recall the trapezoidal decomposition defined in subsection 1.1.3. In this subsection, we present a recursive algorithm constructing the trapezoidal decomposition of a set of disjoint line segments. The input L is a set of n disjoint line segments. The output T is the trapezoidal decomposition for T. We denote the set of line segments intersecting a trapezoid t by L_t. The parameter B will be determined later in this section. This algorithm uses a well-known random sampling approach [38, 80].

```
**Algorithm** Trap-Decomp($L$):
If $|L| \leq B$
    Solve directly in memory
Repeat
    Randomly select a subset $R$ of size $B$ from $L$
    Build the trapezoidal decomposition $T_R$ of $R$
Until $\sum_{t \in T_R} |L_t| = O(n)$ and $\max_{t \in T_R} |L_t| = O((n/B) \log B)$
For each $t \in T_R$
    $T_t$ = Trap_Decomp($L_t$)
Merge all the $T_t$'s together to form $T$ and return $T$
```

The set R is a randomly selected subset. The analysis is similar to theorem 5.4.1. Therefore, we have $\sum_{t \in T_R} |L_t| \leq cn$ with probability greater than a constant, for a sufficiently large constant c, and $\max_{t \in T_R} |L_t| \leq c'(n/B) \log B$ with probability greater than a constant, for a sufficiently large constant c'. Therefore, the first loop iterates only a constant expected number of times.

In the stream-sort model, we keep the set R and T_R in memory. By setting $B = n^\varepsilon$, these two structures only take $O(n^\varepsilon)$ space in memory. With T_R in memory, we can check the conditions to terminate the first loop in $O(1)$ passes.

We use the same duplication idea used in section 5.4. To prepare for the recursive calls in the second loop, in a scan pass, we create one copy of the segment for each trapezoid intersected and attach a label to each copy to identify the intersected trapezoid. In the sorting pass, we use the attached label as the key and group segments by the trapezoids intersected. Because $\max_{t \in T_R} |L_t| = O(n^{1-\varepsilon}) \log n$, the number of levels of recursions is $O(1)$. Because $\sum_{t \in T_R} |L_t| = O(n)$, the intermediate stream for one level contains $O(n)$ line segments. Since there are only $O(1)$ levels of recursive calls, any intermediate stream contains $O(n)$ line segments.

The merging step can be done with one sorting pass and one scan pass. Each trapezoid is bounded by an upper edge, a lower edge, and two walls. In the sorting pass, we use the line segment of the upper edge as the primary key and the left-to-right order as the secondary key to sort all trapezoids. In the scan pass, trapezoids bounded by the same walls are all merged together.

We simultaneously solve subproblems in the same level in the same pass using the technique described in section 5.2. Our algorithm can build the trapezoidal decomposition in $O(1)$ rounds. Since the expected number of pass to obtain a valid trapezoid is $O(1)$, our algorithm takes $O(1)$ expected number of passes in total.

Thus we have:

Lemma 5.5.1 *For any fixed $\varepsilon > 0$, the trapezoidal decomposition of a set of n disjoint 2D line segments can be constructed in $O(1)$ expected number of passes with $O(n^\varepsilon)$ extra space.*

5.5.2 Decomposition of Line Segments

In this subsection, we show how to construct a decomposition of line segments into unimonotone polygons. This is a well-known algorithm described in [40]. We only adapt it in the stream-sort model. The input T is a set of trapezoids from the trapezoidal decomposition described in the previous subsection. The output M is the set of unimonotone polygons from this decomposition.

```
Algorithm Monotone-Decomposition($T$):
Initialize $S = \emptyset$
/* step 1: split trapezoids */
Check for each $t \in T$
    /* These three cases are shown in fig. 5.2 */
    Case 1: the upper or lower edge of $t$ is a whole segment
        Put $t$ into $S$
```

(Case 1) **(Case 2)** **(Case 3)** **(Case 4)**

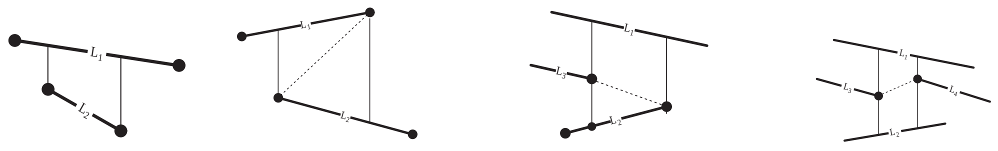


Figure 5.2: Different cases to split a trapezoid

```
    Case 2: two vertices are endpoints of line segments,
            but they are not from the same edge
        Draw the diagonal between the two vertices
        Split $t$ into $t_\uparrow$ and $t_\downarrow$ and put $t_\uparrow$ and $t_\downarrow$ into $S$
    Case 3: one vertex $p$ is an endpoint of a line segment
    and another endpoint $q$ of a line segment is on a vertical edge
        Draw the edge $\overline{pq}$
        Split $t$ into a triangle $\Delta_t$ and trapezoid $Q_t$ and put $\Delta_t$ and $Q_t$ into $S$
    Case 4: no vertices are endpoints of line segments
    and two endpoints, $p_1$ and $p_2$ are on the vertical edges
        Draw the edge $\overline{p_1p_2}$
        Split $t$ into two trapezoids $T_1$ and $T_2$ and put $T_1$ and $T_2$ into $S$

/*step 2: merge polygons in $S$ to unimonotone polygons */
For each line segment $l$
    Sort all polygons using $l$ as the upper edge from left to right
    Merge all these polygons to form a unimonotone polygon $m$ and put $m$ into $M$
    Sort all polygons using $l$ as the lower edge from left to right
    Merge all these polygons to form a unimonotone polygon $m$ and put $m$ into $M$
Return $M$
```

Now we modify both steps of the algorithm for the stream-sort model. Step 1 can be simply done by a scan pass. Instead of writing the results to a data structure S, we write them into the output stream. For step 2, in a sorting pass, we use the segment of the upper edge as the primary key and the left-to-right order as the secondary key to group and sort polygons. In a scan pass, we merge the polygons, whose upper edges are of the same segment, to a unimonotone polygon and write the polygon to the output stream. We do the same for the polygons whose lower edges are of the same segment. Both of these two steps take $O(1)$ passes with $O(1)$ extra space.

5.5.3 Decomposition of a Unimonotone Polygon

It is not obvious how to triangulate unimonotone polygons directly in the stream-sort model. In this section and the next, we introduce nontrivial new ideas that differ from the approaches in previous (sequential or parallel) polygon triangulation algorithms. The following definition is the key:

Definition Given a direction d, a unimonotone polygon is a *special* polygon at direction d, if its chain of edges is monotone in direction d and both vertices of the long edge are higher than any other vertices in the direction perpendicular to d.

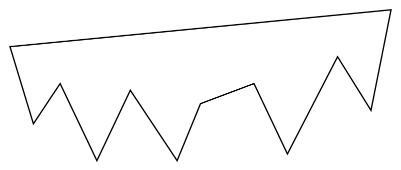


Figure 5.3: A special polygon

The decomposition is built recursively. Given the unimonotone polygon P, we divide the monotone chain into B parts with equal size, and build the upper hull for each part. Below the upper hulls, the decomposition will be built recursively. Above the upper hulls, we obtain a new unimonotone polygon, whose monotone chain Q is formed by at most B concave chains. See fig. 5.4. The following top-down sweeping algorithm decomposes Q into a set of special polygons. Interestingly, this part is inspired by a well-known algorithm for constructing the 2D maxima of a set of points [91]. The input Q is an x-monotone chain formed by B concave chains. The output C is the set of edges of the special polygons. Without loss of generality, we assume the right end is higher than the left end.

Algorithm Special-Decomp(Q):
```
Initialize $v$ to the right end of the long edge and $C$ empty
Put all vertices along the $y$-direction in a sorted list $L$ in decreasing order
For each vertex $v_i$ in decreasing $y$-order
    If $v_i$ is left of $v$
        Add edge $(v_i, v)$ to $C$
        $v = v_i$
Return $C$
```

By this algorithm, we connect the right endpoint of the long edge to the left end with an xy-monotone chain, and all regions below this chain are special polygons along the x-direction, as one can easily see. Consider the region Q' above the xy-monotone chain. (See fig. 5.4.) This is also a special polygon, where, this time, the

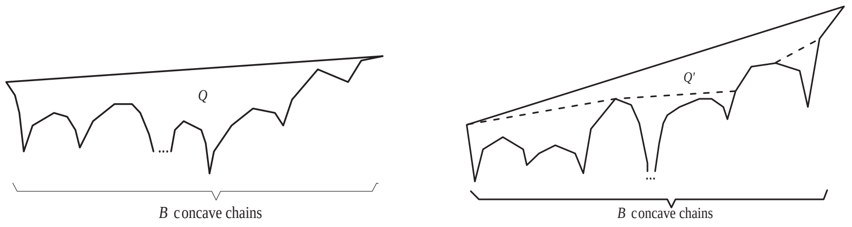


Figure 5.4: Decomposition of Q into special polygons

direction d is perpendicular to the long edge. In the scan pass, we only need to keep v and v_i in memory. Any created edge in C is written into the output stream as scan proceeds. With a sorting pass, we sort all edges using the x-coordinate of the first point as the primary key and the x-coordinate of the second point as the secondary key. Then, with another scan pass, we can obtain the xy-monotone chain in order. Therefore, in the stream-sort model, the above algorithm takes $O(1)$ passes with $O(1)$ space in memory.

By setting B to n^{ε}, the number of levels of the recursion is $O(1)$. Then we simultaneously solve all problems in the same level of the recursion tree in one round. Because any edge we write to the output stream is an edge in the final triangulation and the size of the triangulation is $O(n)$, the size of all intermediate streams is $O(n)$. Since the upper hulls are computed by our algorithm in section 5.3, it takes $O(B)$ space in memory with $O(1)$ passes.

5.5.4 Triangulation of a Special Polygon

We build a triangulation for a special polygon, say in the direction of the x-axis, by a top-down sweeping procedure. In this algorithm, we maintain a tree structure *bridges* in memory. It stores a set of pairs *(left, right).* Each pair corresponds to one portion of the chain intersecting the sweep line. They are ordered left to right in direction y. For p, which is a query point or a pair in *bridges*, its predecessor and successor in *bridges* refer to the bridge immediately left of and right of p, denoted as p^- and p^+, respectively. There are two types of events: 1. the sweep line touches a vertex which belongs to an edge already intersecting the sweep line and 2. the sweep line touches a vertex which does not belong to an edge already intersecting the sweep line. The input is a special polygon whose monotone chain is monotone in x direction without loss of generality. For any other special polygon in other directions, we can rotate the coordinate plane, so that the polygon is a special polygon in direction of y-axis. The triangulated region constructed by the algorithm is illustrated in fig. 5.5.

Algorithm Special-Triangulate(P):

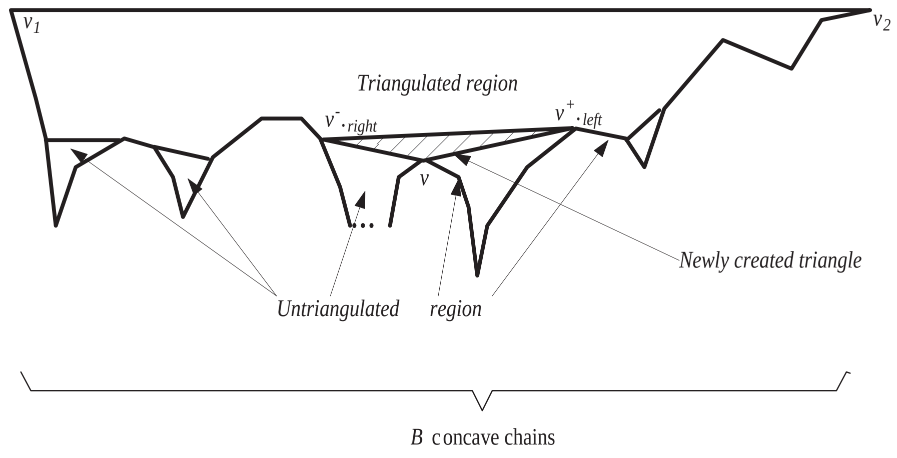


Figure 5.5: Triangulating a special polygon

```
Put all vertices on the monotone chain top-down in direction $y$ in a sorted list $L$
Let the left end point of the long edge be $E_l$ and the right one be $E_r$
Initialize *bridges* with two pairs $(-\infty, E_l)$ and $(E_r, \infty)$
While $L$ is not empty
    Pick the next vertex $v$ from $L$
    Add triangle $(v^-.\text{right}, v, v^+.\text{left})$ to $T$
    Case 1: $v$ causes an event of the type 1 // see fig. 5.6 (left)
        If the left side of $v$ along the sweep line is inside $P$
            $v^+.\text{left} = v$
        Else if the right side of $v$ along the sweep line is inside $P$
            $v^-.\text{right} = v$
        Else
            Delete $v^-$ and $v^+$ and insert $(v^-.\text{left}, v^+.\text{right})$
    Case 2: $v$ causes an event of the type 2 // see fig. 5.6 (right)
        Add $(v, v)$ to *bridges*
    End of cases
End of loop
```

In the stream-sort model, we use a sorting pass to prepare L. Then we perform the loop part of the algorithm and write all triangles into the output stream in one scan pass, and store only *bridges* in memory. Therefore the extra space used in memory is linear to the maximum number of edges intersecting the sweep line.

Lemma 5.5.2 *Given a special polygon at direction d, it can be triangulated in $O(1)$ passes with $O(m)$ extra space, where m is the maximum number of edges of the polygon intersected by a line in direction d.*

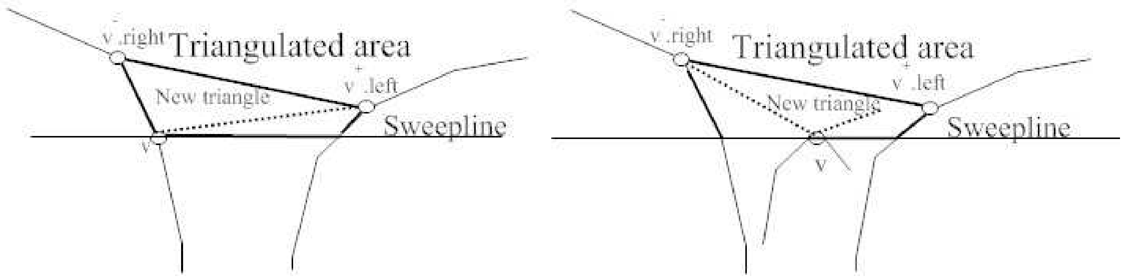


Figure 5.6: Adding a new triangle into the triangulated area

Proof: We prove the following invariant: at any time, the portion of the polygon above the bridges is triangulated and lies above the sweep line. Initially, after the first two vertices v_1 and v_2 are read, the invariant holds. From the definition of special polygons, we know that v_1 and v_2 must be vertices of the long edge, so $v_1 v_2$ lies above the sweep line.

As the sweeping algorithm proceeds, the newly added vertices can be connected to the two adjacent vertices in the bridges without intersecting polygon edges (see fig. 5.6). Therefore, the invariant holds after the new vertex is added. □

5.5.5 Triangulation of Line Segments

By solving multiple subproblems in one round, we simultaneously construct the triangulation of all special polygons decomposed from the same level of the recursive calls in the monotone decomposition algorithm described in section 5.5.3. Recall that the special polygons are decomposed from a unimonotone polygon whose monotone chain is composed of n^ε upper hulls. We conclude that the special polygon Q' above the xy-monotone chain and all obtained special polygons below the xy-monotone chain (see fig. 5.4) have $O(n^\varepsilon)$ edges intersecting its sweep line. The extra space used to triangulate one special polygon is $O(n^\varepsilon)$. Note that all of the four phases take $O(1)$ passes with $O(n^\varepsilon)$ space. Thus, we have:

Theorem 5.5.3 *A triangulation of a set of n disjoint line segments can be constructed in $O(1)$ expected number of passes with $O(n^\varepsilon)$ extra space, for any fixed $\varepsilon > 0$.*

Remark: The merging procedure used in sections 5.5.1 and 5.5.2 can be done in the same sorting pass, to minimize the number of expensive sorting passes in practice. This algorithm can also be derandomized by using the same $(1/B)$-net algorithm [9] mentioned in section 5.4.

5.5.6 Intersection Reporting for Line Segments

As another application of the trapezoidal decomposition algorithm, we can report all k intersections of a set of line segments with intersections in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space, if intermediate streams of size $O(n+k)$ are allowed, where k is the number of intersections.

We adapt the trapezoidal decomposition algorithm (section 5.5.1) to report intersections for a set of line segments with intersections. We modify the random sampling verification step used in the algorithm in section 5.5.1. Here we use $\sum_{t \in T_R} |L_t| \leq c(n + kB/n)$ [38], where k is the number of intersections detected so far. If this is a valid random sample, we build the trapezoidal decomposition on this sample, break the segment intersecting the boundary of the trapezoidal decomposition into two segments, one above the boundary and one below the boundary.

Corollary 5.5.4 *For any fixed constant $\varepsilon > 0$, all intersections of a set of n 2D line segments can be reported in $O(1)$ expected number of passes with $O(n^{\varepsilon})$ extra space, if size $O(n+k)$ intermediate streams are allowed, where k is the number of intersections between line segments.*

5.6 2D Convex Hulls of Sorted Points Sets in a Weaker Model

Since the sorting pass is the most expensive part, we define a simpler and self-contained model in which the sorting pass is not allowed. Instead, we can only choose a direction to scan the stream (forward or backward). We call this model the *direction-flexible* model. This model is weaker than the stream-sort model. For example, it is impossible to sort data in order in a constant number of passes in the direction-flexible model, if the extra space allowed is sublinear.

This result is interesting to us, because it contrasts with the near-$\sqrt{n}$ lower bound result for the same problem shown as theorem 3.4.4 in the original multi-pass stream model in chapter 3.

We only describe how to construct the upper hull, because constructing the lower hull is symmetric. Computing the lower hull simultaneously at most doubles the size of the intermediate stream.

Given a set P of 2D points and a vertical line h, we define the *bridge* at h as the edge of the upper hull of P intersecting h. To compute the bridge at h, we transform the problem in dual space. In dual space, each point becomes a halfplane and a bridge corresponds to an extreme point in the intersection of halfplanes. This point can be computed using linear programming. With $O(n^{\varepsilon})$ extra space in memory, this linear programming problem can be solved under the multi-pass model in $O(1)$, as shown in chapter 3.

We find all points not on the upper hull by divide-and-conquer. The input is a set P of 2D points sorted in x-order. In the output, all points not on the upper hull are marked. In the final pass, we scan this output, and report all unmarked points in x-order, which form the upper hull of P.

```
Algorithm MarkUH($P$):
If $|P| = O(Bn^{\varepsilon})$
    Solve the problem directly in memory
Else
    Divide $P$ into $B$ groups: $P_1, P_2, ..., P_B$
    Find the set $H$ of vertical lines between any two adjacent groups
    For $P$, compute the bridges at each $h \in H$.
    Mark any $p \in P$ if $p$ is below one of these bridges
    For each $P_i$
        MarkUH($P_i$)
```

Again, we solve all subproblems in one level simultaneously. For each subproblem, we need to compute $B-1$ bridges. The computation for each bridge corresponds one linear-programming problem. We use the multi-pass linear-programming algorithm mentioned above. This algorithm scans an array of data sequentially multiple times without modifying it, and the order of the scan does not affect the algorithm. However, between scans, additional information must be kept by the algorithm in memory.

In each subproblem, we compute all $B-1$ bridges by simulating this multi-pass linear programming algorithm. The memory contents of these simulations are kept in memory. After all data of one subproblem are scanned, we write all memory contents to the output stream after the data of that subproblem. Then the memory is reset for the next subproblem in the stream. Thus, in intermediate streams, data and memory contents of the multi-pass algorithm are interleaved. For the next scan, this output stream becomes the input stream. We start from the other end of the stream, so that we can reload the memory content of each subproblem back in memory, before any data point of that subproblem is accessed. After the final pass, all bridges are determined. They are written into the stream after the corresponding data set. To mark points under a bridge, we start from the other end. All bridges are loaded into memory, before the corresponding data set is accessed. This step takes $O(1)$ passes. By setting $B = O(n^{\varepsilon})$, the height of the tree is constant. Therefore, the total number of passes is $O(1)$ and the memory required is $O(n^{2\varepsilon})$.

Besides one mark for each item, all extra information written into the stream consists of the memory contents produced by the multi-pass algorithm. At each level, the total number of bridges to compute is at most $O(\frac{n}{Bn^{\varepsilon}})$. The total extra space is $O(n/B) = o(n)$.

By setting $2\varepsilon = \delta$, we have:

Theorem 5.6.1 *Given a set of n x-ordered 2D points, its convex hull can be constructed in $O(1)$ passes with $O(n^{\delta})$ extra space in the direction-flexible streaming model, for any fixed $\delta > 0$.*

Chapter 6

Conclusion

In this thesis, we have studied some fundamental geometric problems in the in-place model, the multi-pass model, the read-only model, and the stream-sort model.

Most of our results are summarized in table 6. At the end of the thesis, we would like to note some open problems in the models we have studied.

In-place algorithms and implicit data structures.

- Given a set of n 3D points, can its convex hull be constructed in $O(n \log n)$ time with $O(1)$ space? At the end, all points on the convex hull should be stored in a prefix of the input array, and all edges and facets should be outputted to a write-only stream.
- In the permutation+bits model, if we are allowed to have unlimited number of extra bits, can we build a data structure to answer nearest neighbor queries in $O(\log n)$ time? Or can we prove a lower bound $\omega(\log n)$ on this problem in this model?
- Given a set of n points in 2D, can we build a data structure of size $n + O(1)$, such that point-location queries in the Delaunay triangulation can be answered in $o(\log^2 n)$ time with $O(1)$ space?
- Given a set of n axis-aligned boxes in 3D, can we solve Klee's measure problem in $O(n^{3/2} \operatorname{polylog} n)$ time with $O(\operatorname{polylog} n)$ space? In the traditional model, this problem can be solved in $O(n^{3/2} \log n)$ time with $O(n)$ space [88][1].
- Given an arbitrary planar subdivision defined by n points in the plane, can we build a data structure by a permutation of these points, which can support point-location queries in $O(\log n)$ time with $O(1)$ space? We recently have proposed a data structure supporting queries in $O(\log^2 n)$ time with $O(1)$ space [14].

[1]Recently, Chan has proposed a slightly faster algorithm in the traditional model [25].

In the multi-pass model.

- In our algorithms for linear programming (theorems 3.3.2, 3.3.4, 3.3.5, and 3.3.8), the number of passes depends on the fixed constant δ. Can we make this constant smaller? Formally, given $\delta > 0$, can we compute the lowest point in the intersection of n halfplanes in $\mathbb{R}^d$, using $o(1/\delta^{d-1})$ passes and $O(n^\delta)$ space?
- Our algorithm for linear programming (theorem 3.3.7) takes $d + 1$ passes. Can we prove a lower bound of $d+1$ passes for any sublinear space algorithm solving the fixed-dimensional linear programming problem?

In the read-only model.

- Is there an algorithm running in $O(n \log n + n^2/s)$ time with $O(s)$ units of space to solve Klee's measure problem in 2D?

In the stream-sort model.

- Does there exist a lower bound of $\Omega(\operatorname{polylog} n)$ passes for constructing convex hulls with $O(1)$ words of space?
- Can we compute the 2D convex hull in $O(1)$ number of sorting passes, and $O(\operatorname{polylog} n)$ number of scan passes with $O(\operatorname{polylog} n)$ space?

In-place Algorithms and Implicit Data Structures			
Problem	Space	Time	Reference
2D Nearest Neighbor Query	$O(1)$	$O(\log^{1.71} n)$	Thm. 2.3.4
Klee's Measure	$O(\sqrt{n})$	$O(n^{3/2} \log n)$	Thm. 2.4.1
With integer coordinates	$O(1)$	$O(n \log^2 n)$	Thm. 2.4.4
Depth problem	$O(1)$	$O(n \log^2 n)$	Cor. 2.4.5 and 2.4.6
Multi-Pass Algorithms			
Problem	Space	# Passes	Reference
Fixed-D LP	$O(\sqrt{n \log n})$	$d+1$	Thm. 3.3.7
	$O(n^{\delta})$	$O(1)$	Thm. 3.3.2, 3.3.4, 3.3.5, 3.3.8
	$\Omega(n^{\delta})$	$O(1)$	Thm. 3.3.10
2D convex hull	$O(s)$	$O(n/s)$	Thm. 3.2.1
	$\Omega(s)$	$O(n/s)$	by [82]
	$O(hn^{\delta})$	$O(1)$	Thm. 3.5.1 and 3.5.2
Sorted case	$O(n^{1/2+\delta})$	$O(1)$	Thm. 3.4.1 and 3.4.3
	$\Omega(\sqrt{n/\log n})$	$O(1)$	Thm. 3.4.4
3D convex hull	$O(s)$	$O((n/s) \operatorname{polylog} n)$	Thm. 3.6.1
The Read-Only Model			
Problem	Space	Time	Reference
Fixed-D LP	$O(\log n)$	$O(n)$ randomized	Thm. 4.2.1
2D convex hull, sorted case	$O(n^{\delta})$	$O(n)$	Thm.4.3.1 and 4.3.2
Klee's Measure	$O(s)$, $s < n/\log n$	$O(n^3/s^2)$	Thm. 4.4.2
Depth problem	$O(s)$, $s < n/\log^2 n$	$O(n^3/(s^2 \log n))$	Cor. 4.4.3 and 4.4.4
The Stream-Sort Model			
Problem	Space	# Passes	Reference
2D Convex hull	$O(n^{\varepsilon})$	$O(1)$	Thm. 5.3.1
3D Convex hull	$O(n^{\varepsilon})$	$O(1)$	Thm. 5.4.1
Triangulation of Polygons	$O(n^{\varepsilon})$	$O(1)$	Thm. 5.5.3
The Direction Flexible Model			
Problem	Space	# Passes	Reference
2D convex hull: sorted case	$O(n^{\varepsilon})$	$O(1)$	Thm. 5.6.1

Table 6.1: Summary of results

References

[1] I. Adler and R. Shamir. A randomization scheme for speeding up algorithms for linear and convex quadratic programming problems with a high constraints-tovariables ratio. *Math. Programming*, 61:39–52, 1993. 38

[2] P. Agarwal, S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *Proc. of 11th European Sympos. Algorithms (ESA), LNCS 2832*, pages 544–555, 2003. 28, 47

[3] P. Agarwal and H. Yu. A space-optimal data-stream algorithm for coresets in the plane. In *Proc. of the 23rd ACM Sympos. Computational Geometry (SoCG)*, pages 1–10, 2007.

[4] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, 1988. 9

[5] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proc. of the 45th IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 540–549, 2004. 9

[6] L. C. Aleardi, O. Devillers, and G. Schaeffer. Dynamic updates of succinct trianglations. In *Proc. of the 17th Canadian Conference on Computational Geometry (CCCG)*, pages 135–138, 2005. 10

[7] L. C. Aleardi, O. Devillers, and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Workshop on Algorithms and Data Structures, (WADS), LNCS 3608*, pages 134–145, 2005. 10

[8] L. C. Aleardi, O. Devillers, and G. Schaeffer. Optimal succinct representations of planar maps. In *Proc. of the 22nd ACM Sympos. Computational Geometry (SoCG)*, pages 309–318, 2006. 10

[9] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. In *Proc. of 20th ACM Sympos. on Computational Geometry (SoCG)*, pages 144–151, 2004. 36, 57, 64

[10] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proc. of the 11th ACM Sympos. on Computational Geometry (SoCG)*, pages 211–219, 1995.

[11] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proc. of the 43rd IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 209–218, 2002. 28

[12] P. Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. 8

[13] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11:287–297, 1982. 30

[14] P. Bose, E. Y. Chen, M. He, A. Maheshwari, and P. Morin. Succinct geometric indexes supporting point location queries. In *Proc. of the 20th ACM-SIAM Sympos. on Discrete Algorithms (SODA), to appear*, 2009. 67

[15] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory and Applications*, 37(3):209–227, 2007. 5

[16] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. of the 43rd IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 617–626, 2002. 55

[17] H. Brönnimann and T. M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Computational Geometry: Theory and Applications*, 34:75–82, 2006. 5, 6

[18] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. of the 20th ACM Sympos. on Computational Geometry (SoCG)*, pages 239–246, 2004. 5, 6, 22

[19] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient plannar convex hull algorithms. *Theoretical Computer Science*, 312(1):25–40, 2004. 5

[20] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996. 2, 45

[21] T. M. Chan. An optimal randomized algorithm for maximum Tukey depth. In *Proc. of the 15th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 423–429, 2004. 48

[22] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proc. of the 17th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 1196–1202, 2006.

[23] T. M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Computational Geometry: Theory and Applications*, 35:20–35, 2006.

[24] T. M. Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. Manuscript, 2006, http://www.cs.uwaterloo.ca/˜tmchan/pub.html#sss.

[25] T. M. Chan. A (slightly) faster algorithm for Klee's measure problem. In *Proc. of the 24th ACM Sympos. on Computational Geometry (SoCG)*, pages 94–100, 2008. 67

[26] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37:79–102, 2007. 7, 8, 28

[27] T. M. Chan and E. Y. Chen. In-place 2-d nearest neighbor search. In *Proc. the 19th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 904–911, 2008. 5, 11

[28] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: point location in sublogarithmic time. *SIAM J. Comput.*, to appear; preliminary versions in FOCS 2006.

[29] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991. 3

[30] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10:229–249, 1990. 35

[31] E. Y. Chen. Towards in-place algorithms in computational geometry. Master's thesis, University of Waterloo, 2004. 5, 6

[32] E. Y. Chen. Geometric streaming algorithms with a sorting primitive. *International Journal of Computational Geometry and Applications, to appear, preliminary version in ISAAC 07*, 2008. 53

[33] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *Proc. of 15th Canadian Conference on Computational Geometry (CCCG)*, pages 68–71, 2003. 5, 6

[34] E. Y. Chen and T. M. Chan. Space-efficient algorithms for Klee's measure problem. In *Proc. of the 17th Canadian Conference on Computational Geometry (CCCG)*, pages 38–41, 2005. 5, 6, 11, 27

[35] D. R. Clark and J. I.Munro. Efficient suffix trees on secondary storage. In *Proc. of the 7th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 383 – 391, 1996. 10

[36] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987. 35

[37] K. L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *J. ACM*, 42:488–499, 1995. 36, 37

[38] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4(1):387 – 421, 1989. 46, 55, 56, 58, 65

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2001. 4

[40] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer, 1997. 2, 3, 5, 22, 23, 26, 54, 55, 59

[41] E. D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes Comput. Sci. (LNCS). 2002. 6, 9, 12, 13

[42] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proc. of 17th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 714–723, 2006. 7

[43] P. Drineas and R. Kannan. Pass efficient algorithms for approximating large matrices. In *Proc. of 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 223–232, 2003. 28

[44] S. Dvorak and B. Durian. Stable linear time sublinear space merging. *The Computer Journal*, pages 372–375, 1987. 4

[45] S. Dvorak and B. Durian. Unstable linear time $O(1)$ space merging. *The Computer Journal*, 31(3):279–282, 1988. 4

[46] M. E. Dyer. Linear time algorithms for two- and three-variable linear programs. *SIAM J. Comput.*, 13:31–45, 1984. 30, 33

[47] M. E. Dyer and A. M. Frieze. A randomized algorithm for fixed-dimensional linear programming. *Math. Programming*, 44:203–212, 1989. 35

[48] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. on Comput.*, 15(2):317–340, 1986. 3

[49] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proc. 31st Int. Colloq. Automata Languages and Programming, LNCS 3142*, pages 521–543, 2004. 28

[50] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proc. of the 16th ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 745–754, 2005. 28

[51] A Fiat, J. I. Munro, M. Naor, A. A. Schäffer, J. P. Schmidt, and A. Siegel. An implicit data structure for searching a multikey table in logorithmic time. *J. Comput. Syst. Sci.*, 43(3), 1991. 5

[52] P. Flajolet and G. Nigel Martin. Probabilistic counting. In *Proc. of the 24th IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 76–82, 1983. 7

[53] S.J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[54] G. Franceschini. Sorting stably, in place, with $O(n \log n)$ comparisons and $O(n)$ moves. *Theory Comput. Systems*, 40:327–353, 2007. 4

[55] G. Franceschini and V. Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. *J. ACM*, 52(4):515–537, 2005. 4

[56] G. Franceschini and R. Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log \log n)$ time. In *Proc. 14th of the ACM-SIAM Sympos. on Discrete Algorithms (SODA)*, pages 670–678, 2003. 5, 6

[57] G. Franceschini and R. Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Workshop on Algorithms and Data Structures (WADS), LNCS 2748*, pages 114–126, 2003. 5

[58] G. Franceschini, R. Grossi, J. I. Munro, and L. Pagli. Implicit B-trees: new results for the dictionary problem. In *Proc. of the 43rd IEEE Sympos. Foundation of Computer Science*, pages 145–154, 2002. 5

[59] G. N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987. 8

[60] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31:538–544, 1984. 15

[61] R. L. Graham. An efficient algorithm for determining the convex hull of a finite plannar set. *Information Processing Letters*, 1:132–133, 1972. 1, 2, 50

[62] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD conference*, pages 58–66, 2001. 33, 44

[63] S. Guha and A. McGregor. Tight multi-pass stream lower bounds via pass elimination. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP) Part I, LNCS 5125*, pages 760–772, 2008. 29

[64] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. of the 41st IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 359–366, 2000. 7

[65] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41:69–85, 2001. 20

[66] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987. 35

[67] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical report, Digital Systems Research Center, Palo Alto, CA, 1998. 28

[68] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th IEEE Sympos. on Foundation of Computer Science*, pages 549–554, 1989. 10

[69] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992. 4

[70] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[71] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithms? *SIAM J. Comput.*, 15(1):287–299, 1986. 2, 44

[72] V. Klee. Can the measure of $\bigcup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *Amer. Math. Monthly*, 84(4):284–285, 1977. 23

[73] T. W. Lai and D. Wood. Implicit selection. In *1st Scandinavian Workshop on Algorithm Theory (SWAT), LNCS 318*, pages 14–23. Springer-Verlag, 1988. 4, 22

[74] C. Lammersen and C. Sohler. Strsort algorithms for geometric problems. In *European Woarkshop on Computatonal Geometry*, pages 69–72, 2007. 9

[75] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979. 13

[76] H. Mannila and E. Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18:203–208, 1984. 4

[77] N. Megiddo. Linear time algorithms for linear programming in R^3 and related problems. *SIAM J. Comput.*, 12:759–776, 1983. 30, 33

[78] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM*, 31:114–127, 1984. 34, 35

[79] N. Megiddo. Partitioning with two lines in the plane. *J. Algorithms*, 6:430–433, 1985. 36

[80] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, 1994. 3, 35, 37, 46, 55, 56, 58

[81] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Sys. Sci.*, 33:66–74, 1986. 4, 5, 12, 20

[82] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980. 7, 28, 29, 32, 33, 36, 38, 44, 69

[83] J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theor. Comput. Sci.*, 165(2):311–323, 1996. 8

[84] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. of the 38th IEEE Sympos. on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[85] J. I. Munro, V. Raman, and J. Salowe. Stable in situ sorting and minimum data movement. *BIT*, 30:220–234, 1990. 4

[86] S. Muthukrishnan. *Data streams: algorithms and applications.* Now Publishers Inc, 2005. 7, 10

[87] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994. 2, 3

[88] M. Overmars and C.-K. Yap. New upper bounds in Klee's measure problem. *SIAM J. Comput.*, 20:1034–1045, 1991. 67

[89] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.*, 23:166–204, 1981. 42

[90] J. Pagter and T.Rauhe. Optimal time-space trade-offs for sorting. In *Proc. of 39th IEEE Sympos. Foundations Computer Science (FOCS)*, pages 264–268, 1998. 8, 51

[91] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985. 3, 12, 21, 29, 33, 41, 42, 44, 46, 61

[92] H. Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999. 6, 12

[93] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. *Lecture Notes in Computer Science (LNCS)*, 2125:426–437, 2000.

[94] J. M. Ruhl. *Efficient Algorithms for New Computational Models.* PhD thesis, Massachusetts Institute of Technology, 2003. 7, 8, 9

[95] J. Salowe and W. Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8:557–571, 1987. 4

[96] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[97] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991. 48, 49

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

[99] R. Seidel and D. Hoey. Closest-point problems. In *Proc. of the 16th IEEE Sympos. on Foundations of Computer Science*, pages 151–162, 1975.

[100] J. Matoušek. *Derandomization in computational geometry*, pages 559–595. Elsevier, Amsterdam, 2000. 36

[101] M. Sharir and E. Welzl. A combinatorial bound for linear programming and related problems. In *Proc. of the 9th Sympos. Theoret. Aspects Comput. Sci., LNCS 577*, pages 569–579, 1992. 38, 49

[102] J. Snoeyink. *Point location*, chapter 30, pages 559–574. Handbook of Discrete and Computational Geometry, CRC Press LLC, 1997.

[103] S. Suri, C. D. Tóth, and Y. Zhou. Range counting over multidimensional data streams. *Discrete Comput. Geom.*, 36(4):633–655, 2006. 28

[104] A. Symvonis. Optimal stable merging. *The Computer Journal*, 38(8):681–690, 1995. 4

[105] J. Vahrenhold. An in-place algorithm for Klee's measure problem in two dimensions. *Information Processing Letters*, 102(4):169–174, 2007. 5, 6, 23, 27

[106] J. Vahrenhold. Line-segment intersection made in-place. *Computational Geometry: Theory and Applications*, 38(3):213–230, 2007. 5

[107] S. Ramnath V.Raman. Improved upper bounds for time-space trade-offs for selection. *Nord. J. Comput.*, 6(2):162–180, 1999. 8