Output-Size Sensitive Algorithms for Constructive Problems in Computational Geometry

Raimund Seidel
(Ph.D. Thesis)

TR 86-784

September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

OUTPUT-SIZE SENSITIVE ALGORITHMS FOR CONSTRUCTIVE PROBLEMS IN COMPUTATIONAL GEOMETRY

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Raimund Seidel

January 1987

OUTPUT-SIZE SENSITIVE ALGORITHMS FOR CONSTRUCTIVE PROBLEMS IN COMPUTATIONAL GEOMETRY

Raimund Seidel, Ph.D.

Cornell University 1987

In computer science the efficiency of algorithms is usually measured in terms of the size of the input. The output size, on the other hand, has been used for this purpose rather infrequently, except in certain enumerative query problems.

This thesis deals with several *constructive* (in contrast to *query*) problems in computational geometry and presents algorithms whose running time depends non-trivially on the output size.

We present an algorithm that finds the convex hull on n points in the plane in worst case time $\mathbf{O}(n\log H)$, where H is the number of points that turn out to be vertices of the convex hull.

We examine the d-dimensional maximal vector problem and show that as long as V, the number of maximal vectors in a set, is not too large, these maximal vectors can be found in time $\mathbf{O}(n\log V)$.

We present an algorithm for solving the planar convex subdivision overlay problem in time proportional to the combined input and output size.

Finally we show that, after some preprocessing in the form of linear programs, d-dimensional convex hulls can be constructed at logarithmic cost per face.

Biographical Sketch

born 1957 in Graz, Austria, and named Raimund G. Seidel;

primary and secondary education in Graz;

1973/74 at an American highschool in Hudson, Wisconsin, as exchange student under the auspices of the American Field Service (AFS);

entered the Technical University of Graz in 1975 to study Technical Mathematics and obtained degree of Diplom Ingenieur in 1982;

1979 to 1981 in Vancouver, B.C., Canada; study of computer science at the University of British Columbia leading to an M.Sc. degree;

1982 to 1986 graduate student in computer science at Cornell University;

after receiving M.S. and Ph.D. degree denied further graduate student status.

Zum Gedenken an meine Mutter

Elfriede Seidel, geb. Riemelmoser

Acknowledgements

Little did I know about computer science when I attended my first class in North America seven years ago at UBC. But even less did I know what influence David Kirkpatrick, who taught that very first class of mine, would have on me as teacher, mentor, collaborator, and friend. Thank you, David!!

Here at Cornell, John Gilbert has been a perfect advisor for me. Thank you, John, for giving me so much freedom, thank you for your criticism, and thanks for not making me fulfill all the promises I made, of things I would do, or would not do.

The Cornell Computer Science Department has been a very interesting environment. I want to thank the faculty for trying to make it more than just a place of teaching and research. Special thanks to Bengt Aspvall and Vijay Vazirani. And special thanks to Tim Teitelbaum for being on my committee after Alan Demers had to drop out.

The professors of the Operations Research department have been very kind to me. I want to thank Mike Todd for serving on my committee, Louis Billera for teaching me about polytopes, and Jack Edmonds for teaching me about all kinds of things.

During the last years I have had the privilege to get to know, learn from, and work with many researchers. I am grateful to Joe O'Rourke, Mike McKenna, and the whole Johns Hopkins geometry group; David Avis and Godfried Toussaint with their group at McGill; Leo Guibas, Mark Overmars,

Hossam El-Gindy, Jim Little, Jorge Stolfi, Micha Sharir, Emo Welzl, and last but most, Herbert Edelsbrunner.

My fellow students have been wonderful. I will especially remember Bennett Battaille, Rogerio Drummond, Van Nguyen, Jim Hook, Roger Hoover, David Rossiter, "Nat" Natarajan, and the departmental basketball and volleyball squads.

Life was certainly made easy for me through the generous financial support that I received from the graduate school and the college of engineering. I also would like to acknowledge that I did parts of the work that went into this thesis while being supported by the DEC Systems Research Center.

It was wonderful to have the same housemates, Roger, Thomas, and Tommy, for almost three years. Finally, I want to say "danke sehr" to Bea and the whole gang on Bryant Avenue for making my stay in Ithaca much more enjoyable than it would have been without them.

Table of Contents

List of Figures

1. Introduction

> *. . . il faut vous résoudre à composer un gros livre, et à expliquer tout si amplement, si clairement et si distinctement que ces Messieurs, qui n'étudient qu'en baillant et qui ne peuvent se peiner l'imagination pour entendre une proposition de Geometrie, ni tourner les feuillets pour regarder les lettres d'une figure, ne trouvent rien en vostre discours, qui leur semble plus malaisé à comprendre qu'est la description d'un palais enchanté dans un roman.*
>
> Descartes *in a letter to* Desargues, *June 19, 1639.*

When analyzing the efficiency of algorithms and data structures one usually measures their complexity in terms of the size of the input. For instance, heapsort is said to sort a sequence of n numbers in worst case time $\mathbf{O}(n\log n)$. In a dictionary of n words that is stored in a balanced tree a query word can be located in worst case time $\mathbf{O}(\log n)$. While this kind of measurement is certainly very natural and appropriate in many cases, it becomes almost meaningless in others. Consider for instance the following problem: Store a set S of n numbers in a data structure such that for any pair $a \leq b$ all $x \in S$ that satisfy $a \leq x \leq b$ can be reported quickly. The worst case query time for any method that solves this problem is at best $\mathbf{O}(n)$, since all members of S might lie in a query range and would need to be reported.

Of course such an $\mathbf{O}(n)$ bound is not particularly useful. A more meaningful estimate can be obtained by expressing the query time as a function of not only the input size n but also the output size K, the number of elements of S found to satisfy a query. This way one can say the query time of an algorithm

is $\mathbf{O}(K\log n)$, or it is $\mathbf{O}(\log n + K)$, which is certainly more succinct than claiming it is $\mathbf{O}(n\log n)$ in the worse case, or $\mathbf{O}(n)$.

This idea of measuring the complexity of an algorithm in terms of input size as well as output size has been pursued most extensively in the design of methods for solving all kinds of query problems ([E82],[O],[C-G-L] and many others). In this thesis this idea is applied to so-called constructive problems in computational geometry. Maybe the best example for what we mean by a "constructive problem" is the planar convex hull problem: Given a set S of n points in the plane, find its convex hull, i.e. determine the smallest convex polygon that contains S. The input size for this problem is obviously n. Its output size is the number of vertices (i.e. corners) the convex hull of S turns out to have. Note that this output size, call it H, can vary considerably. The convex hull of S might be just a triangle, but it is also possible that every point in S turns out to be a vertex of the convex hull. Also note that it is impossible to tell H in advance.

Worst case optimal, but output-size insensitive $\mathbf{O}(n\log n)$ algorithms for the planar convex hull problem have been known since 1972 [Gra]. Two output-size sensitive algorithms have also been published. However, their $\mathbf{O}(nH)$ running time is somewhat wanting, since it is far from optimal for most values of the parameter H.

In chapter 2 of this thesis we present an algorithm for the planar convex hull problem with worst case running time $\mathbf{O}(n\log H)$. This bound

simultaneously improves upon the $\mathbf{O}(n\log n)$ and the $\mathbf{O}(nH)$ bound. Moreover, we also show that, at least in the asymptotic sense, this bound is optimal even if the complexity of the problem is measured in terms of both n and H.

Our algorithm exploits an interesting variation of the divide-and-conquer paradigm which one might call *marriage-before-conquest.* Typically, divide-and-conquer algorithms proceed by splitting a problem into subproblems (divide), recursively solving these subproblems (conquer), and finally combining the subsolutions into the global solution (marry). Our algorithm reverses the last two stages. It determines how subsolutions will combine before actually computing them.

Chapter 3 of this thesis deals with the so-called maximal vector problem. For a given set S of n points in $\mathbf{R}^d$ one is to find the set of its maximal vectors, i.e. the points in S that are not dominated by any other point in S. Here x *is dominated* by y means $x \neq y$ and no coordinate of x exceeds the corresponding coordinate of y. Whereas for $d=1$ every set has exactly one maximal vector, the number V of maximal vectors of a set S in $\mathbf{R}^d$, $d \geq 2$, can be anywhere between 1 and n.

This possible variation in the output size was not reflected in the running time of the first efficient algorithms for the maximal vector problem. Kung et al. [K-L-P] achieved $\mathbf{O}(n\log n)$ worst case bounds for $d=2$ and $d=3$ and an $\mathbf{O}(n\log^{d-2} n)$ bound for $d \geq 4$. Much later Gabow et al. [G-B-T] managed to replace one of the $\log n$ factors of the second bound by a $\log\log n$ term.

Ramanan [R] was the first to publish efficient algorithms whose running time also depends on the output size V. He obtined the bounds $\mathbf{O}(n\log V)$ for $d=2$ and $\mathbf{O}(n\log n\log^{d-3}V)$ for $d\geq 3$.

We present algorithms whose worst case running time is $\mathbf{O}(n\log V)$ for $d=2,3$, and $\mathbf{O}(n\log^{d-2}V)$ for $d\geq 4$. Moreover, we show that when V is sufficiently small compared to n, the exponent of the logarithmic term can be decreased. In particular, for $d\geq 4$ and $n\geq V^{2+\alpha}$ with $\alpha>0$, the maximal vector problem can be solved in time $\mathbf{O}(n\log V)$. We also prove an $\Omega(n\log V)$ lower bound for the maximal vector problem for a rather strong model of computation.

Our algorithms exploit two techniques. One might be dubbed "lazy sorting:" Only sort as much as you need to; incomplete order information might already be sufficient. The second technique is yet another variation on the divide-and-conquer paradigm: When "dividing" be judicious about the choice of the number of subproblems.

Whereas in the problems of chapters 2 and 3 the size of the output can never exceed the size of the input, in the problems of chapters 4 and 5 the output size can become quite large. In chapter 4 we consider the planar convex subdivision overlay problem. Loosely speaking, a planar convex subdivision is a partition of the plane into a finite number of convex polygons (some of which may be unbounded). The overlay **X** of two convex subdivision **B** and **G** is the subdivision consisting of all convex polygons formed by intersections between polygons from **B** and polygons from **G**. Intuitively, draw **B** on white paper,

draw **G** on transparent paper, put the transparent paper over the white paper, and what you see is the overlay **X** of the two subdivisions **B** and **G**. Of course the convex subdivision overlay problem asks to contruct the description of **X** from descriptions of **B** and **G**. Let us denote the sizes of **B**, **G**, and **X** by m, n, and K, respectively. Loosely speaking, the size of a planar convex subdivision is the number of its polygons. Obviously m and n measure input size, whereas K measures output size. Again K is not known in advance and can vary considerably: It can be as small as $m+n$ or as large as mn.

The plane sweep algorithm of Bentley and Ottmann [B-O] can be used to solve the overlay problem in time $\mathbf{O}((m+n+K)\log(m+n))$. Nievergelt and Preparata [N-P] improved upon this bound and presented an algorithm with running time $\mathbf{O}((m+n)\log(m+n)+K)$. In chapter 4 of this thesis we show that it is possible to do away with the log term altogether, and present an algorithm with running time $\mathbf{O}(m+n+K)$, which is clearly optimal. We also discuss an application. We show that using the new algorithm the Minkowski sum of two 3-polytopes can be constructed in optimal time.

As the main tool in the new algorithm we use the idea of a topological sweep line. Plane sweep has been a traditional algorithmic paradigm in computational geometry: Move a straight vertical line from left to right over a set of geometric objects in the plane and maintain the invariant that all computations involving objects that lie entirely to the left of the sweep line have been correctly completed. Typically such an algorithm has to maintain a priority

queue to discover which objects will intersect the sweep line next and which one will be the next to lie entirely to the left of the sweep line. We can dispense with this priority queue by forgoing the condition that the sweep line be straight. We allow it to "bulge" to a certain extent and thus manage to save a logarithmic factor in the running time of the algorithm.

Finally, chapter 5 of this thesis deals with the higher dimensional convex hull problem. The notion of convex hulls can be defined for point sets in any dimension, not just for the plane. The convex hull of a set S in $\mathbf{R}^d$ is just the smallest convex set containing S. For finite S in $\mathbf{R}^2$ we already mentioned that its convex hull is a convex polygon. For finite S in $\mathbf{R}^3$ the convex hull is always a polyhedron, i.e. a convex body with straight "sides." Generally, the convex hull of a finite set S in $\mathbf{R}^d$ is a convex d-dimensional body with straight "sides" and is called a *polytope*. Technically, these "sides" are called *faces* and occur with different dimensionalities. The corners of the polytopes are the 0-dimensional faces which are usually called vertices. The 1-dimensional faces are called edges, and the highest-dimensional faces are usually referred to as facets. The number of faces of a polytope is always finite. Thus they provide a discrete description of a polytope. For $d \leq 3$ the number of faces of the convex hull of a set S of m points in $\mathbf{R}^d$ cannot be larger than $\mathbf{O}(m)$. However, for $d > 3$ this is not true any more. In the worst case the number of faces can be as large as $\mathbf{O}(m^{\lfloor d/2 \rfloor})$. But in the best case this number can also be as small as $\mathbf{O}(1)$. It is thus desirable to obtain convex hull algorithms whose running time

depends on the number of faces the constructed convex hull is found to have.

In this thesis we consider two versions of the convex hull problem. The first one is called the facet enumeration problem. For a set S of m points in $\mathbf{R}^d$ that are in non-degenerate position (this roughly means that no $d+1$ points lie in a common plane) enumerate all facets of its convex hull. The non-degeneracy assumption implies that each facet can be uniquely identified by a subset of d points in S. Thus enumerating facets amounts to enumerating certains subsets of S of cardinality d.

The second version of the convex hull problem that we consider is the facial graph problem. The faces of a polytope form a partial order which is called the face lattice of the polytope. For an arbitrary finite set S in $\mathbf{R}^d$ we want to construct the facial graph of the convex hull of S, i.e. a directed acyclic graph that forms a minimal representation (Hasse diagram) of the face lattice of the convex hull of S.

Two types of algorithms have been published for solving the convex hull problem. The first type proceeds incrementally by constructing the convex hull of S from the hull of $S-\{p\}$ for some $p \in S$ ([Kal],[R-W],[S81]). For these algorithms it is impossible to bound their worst case running time in terms of the output size. The best bound in terms of the input size was obtained by Seidel [S81]. The algorithm presented there solves the facial graph problem and the facet enumeration problem in worst case time $\mathbf{O}(m^{\lfloor (d+1)/2 \rfloor})$. The other type of algorithms constructs the convex hull facet by facet ([C-K],[Sw],[B]). Swart

[Sw] gives the most thorough analysis of such algorithms. The facet enumeration problem is solved in time $\mathbf{O}(Fm)$, where F is the number of facets found, and the facial graph problem is solved in time $\mathbf{O}(K_1 m + K_2 \log m)$, where K_1 is the number of arcs in the facial graph, and K_2 the number of paths of length two. In all these bounds d is considered a fixed constant greater than 3.

In chapter 5 we present algorithm that solve the facet enumeration problem in time $\mathbf{O}(F \log m)$ and the facial graph problem in time $\mathbf{O}(K_2 \log m)$. In addition $\mathbf{O}(m)$ linear programs with at most $m-1$ constraints in at most $d-1$ variables need to be solved. These algorithms follow the paradigm of constructing a convex hull facet by facet. However, in contrast to the previous algorithms they are very judicious about the order in which the facets are computed. They exploit the notion of shelling a polytope, which has been a notion of central importance in the theory of polytopes.

Chapters 2 to 5 can be read independently. They are parts or extensions of various papers that I have been involved in during my years at Cornell ([K-S86],[K-S85],[G-Se],[S86]). With joy I acknowledge collaboration with David Kirkpatrick on the contents of chapters 2 and 3. Chapter 4 grew out of collaboration with Leo Guibas. Of course any errors in this thesis are to be blamed solely on me.

2. Computing Planar Convex Hulls

2.1 Introduction

The *convex hull* of a finite point set S in the plane is the smallest convex polygon containing the set. The vertices (corners) of this polygon must be points of S. Thus in order to compute the convex hull of a set S it is necessary to find those points of S which are vertices of the hull. For the purposes of constructing upper bounds we define the *planar convex hull problem* as the problem of constructing the ordered sequence of points of S which constitute the sequence of vertices around the hull.

The convex hull problem was one of the first problems in the field of computational geometry to have been studied from the point of view of computational complexity. In fact, efficient algorithmic solutions were proposed even before the term "computational geometry" was coined. This, along with its very extensive analysis in recent years, reflects both the theoretical and practical importance of the problem.

Of the convex hull algorithms proposed so far several have $\mathbf{O}(n\log n)$ worst case time bounds [A-E-S,B-S,Gra,Pre,P-H,Sh], where n is the size of the input point set. Shamos [Sh] even argued that $\mathbf{O}(n\log n)$ time bound is worst case optimal. He observed that a set S of n real numbers could be sorted by finding the convex hull of the planar set $S' = \{ (x,x^2) \mid x \in S \}$. But sorting, of course, has an $\Omega(n\log n)$ lower bound on a wide range of computational models. Yao [Y] and on weaker computational models Avis [A], van Emde Boas [vE], and

Preparata and Hong [P-H] proved the $\Omega(n\log n)$ bound for a less demanding version of the convex hull problem: just the vertices of the convex hull are to be identified, irrespective of their sequence.

A special feature of the planar convex hull problem is the possible variation in output size. The convex hull of $n>2$ points in the plane can be any convex H-gon, with H ranging between 3 and n (in fact, in extremely degenerate cases $H=1$ or $H=2$ is possible as well). It is notable that all of the lower bound arguments mentioned in the previous paragraph are insensitive to H in that they assume that some fixed fraction of the data points are vertices of the convex hull. This leaves open the intuitively appealing possibility that the convex hull problem may be "easier" for smaller values of H. Indeed, at least two planar convex hull algorithms are known [Ed,J] that exhibit such behaviour in that their running time is $\mathbf{O}(nH)$. For small values of H these algorithms seem superior to the $\mathbf{O}(n\log n)$ methods. However, with increasing H their attractiveness vanishes rather quickly.

In this paper we present a convex hull algorithm with worst case time complexity $\mathbf{O}(n\log H)$. Thus its running time is not only sensitive to both n and H, but it is also worst case optimal in the traditional sense when the running time is measured as a function of n only. However, we also show that our algorithm is asymptotically worst case optimal even if the complexity of the problem is measured as a function of both n and H.

The algorithm is based on a variation of the divide-and-conquer paradigm that appears to be interesting in its own right. Traditional divide-and-conquer algorithms adhere to the following strategy: First break the problem into subproblems (*divide*), then recursively solve the subproblems (*conquer*), and finally combine the subsolutions to form the global solution (*marry*). Our algorithm reverses the last two steps. After dividing the problem it first determines how the solutions of the subproblems will combine (without actually computing them!) and then proceeds to solve the subproblems recursively. We thus call this approach the "marriage-before-conquest" principle. Its advantage lies in the fact that it allows to remove parts of the subproblems that upon merging (or marrying) turn out to be redundant. Thus it reduces the sizes of the subproblems that are to be solved recursively. It remains to be seen whether this principle has other applications.

Section 2 and 3 of this chapter describe the new algorithm. Section 4 deals with a randomized version of the new algorithm. Finally in section 5 we prove an $\Omega(n\log H)$ lower bound for a rather general model of computation and thus show that the new algorithm is asymptotically worst case optimal even if the complexity of the planar convex hull problem is measured in terms of both n and H.

Throughout this chapter, unless stated otherwise, we deal with sets of points in the plane. For a point p, $x(p)$ and $y(p)$ denote its standard cartesian coordinates. We will feel free to use loose but descriptive geometric terminol-

ogy such as "vertical line", "a point lies above a line", "a point lies to the left of a vertical line", etc.

2.2 The Main Algorithm

In this section we show how the "marriage-before-conquest" principle can be used for an improved convex hull algorithm. We construct the convex hull in two pieces, the upper hull and the lower hull (see Figure 2.2.1). It should be clear that if the two chains forming the upper and lower hull are given, they can be concatenated in constant time (at most two vertical edges may need to be inserted) to yield the sequence of vertices around the hull. Also observe that an algorithm for constructing the upper hull could easily be modified to construct the lower hull also. Therefore we concentrate at first on constructing an algorithm for finding the sequence of vertices on the upper hull.

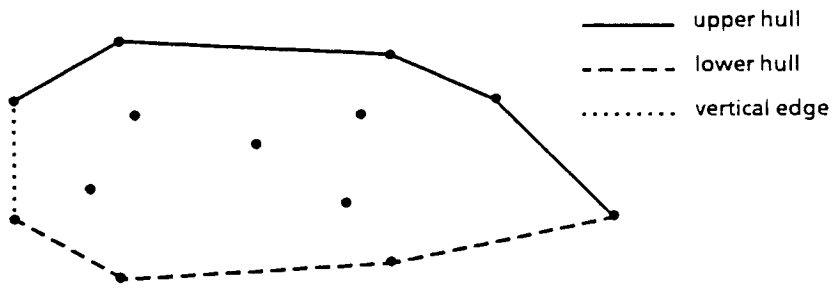


Figure 2.2.1: Upper and lower hull

Exploiting the "marriage-before-conquest" principle, our convex hull algorithm should do something like the following: First find a vertical line that divides the given point set in two approximately equal sized parts. Next determine the "bridge" crossing this line, i.e. the edge of the upper hull that intersects this line. Eliminate the points that lie underneath the bridge, and finally apply the algorithm recursively to the two sets of the remaining points on the left and right side of the vertical line.

The only difficult part in such an algorithm appears to be the construction of the bridge. We show a linear time solution to this problem in Section 2.3.

The following PIDGIN-ALGOL routine presents our convex hull algorithm in some detail. It takes as input a set $S = \{p_1, \ldots, p_n\}$ of n points in the plane and prints the sequence of indices of the vertices on the upper hull of S. It uses the function BRIDGE specified in section 2.3, which given a set $S \subset \mathbf{R}^2$ and a real a returns the indices of the left and right endpoint of the edge of the upper hull that intersects the vertical line $L = \{(x,y) | \ x = a \}$.[1]

[1]In the case that two edges of the upper hull, (p_i,p_j) and (p_j,p_k), intersect L, i.e vertex p_j lies on L, BRIDGE will return (j,k).

Algorithm 2.1:

Procedure UPPER-HULL(S)

1. Initialization

Let min and max be the indices of two points in S that form the left and right endpoint of the upper hull of S respectively, i.e.

$$x(p_{min}) \leq x(p_i) \leq x(p_{max}) \text{ and}$$
$$y(p_{min}) \geq y(p_i) \quad \text{if } x(p_{min}) = x(p_i),$$
$$y(p_{max}) \geq y(p_i) \quad \text{if } x(p_{max}) = x(p_i) \quad \text{for } i=1,...,n.$$

If $min = max$ then print min and stop.

Let $T := \{p_{min}, p_{max}\} \cup \{ p \in S \mid x(p_{min}) < x(p) < x(p_{max}) \}$.

2. CONNECT(min,max,T)
where CONNECT(k,m,S) is
begin

2.1 Find a real number a such that
$x(p_i) \leq a$ for $\lceil |S|/2 \rceil$ points in S and
$x(p_i) \geq a$ for $\lfloor |S|/2 \rfloor$ points in S.

2.2 Find the "bridge" over the vertical line $L = \{(x,y) \mid x = a\}$, i.e
$(i,j) :=$ BRIDGE(S,a).

2.3[2] Let $S_{left} := \{p_i\} \cup \{p \in S \mid x(p) < x(p_i)\}$.
Let $S_{right} := \{p_j\} \cup \{p \in S \mid x(p) > x(p_j)\}$.

2.4 If $i = k$ then print(i)
else CONNECT(k,i,S_{left}).
If $j = m$ then print(j)
else CONNECT(j,m,S_{right}).

end.

Theorem 2.2.1:

Algorithm UPPER-HULL correctly determines the sequence of vertices on the upper hull of S in $\mathbf{O}(n)$ space and $\mathbf{O}(n \log H_u)$ time, where H_u is the number of edges on the upper hull of S.

---

[2] S_{left} contains p_i and the points of S to the left of the vertical line through p_i. M. McQueen from McGill University has pointed out that S_{left} could be restricted to contain p_k,p_i and all the points of S above the straight line through p_k and p_i. S_{right} can be restricted analogously.

Proof:

If the upper hull of S consists if only one vertex (i.e. all of S lies on one vertical line) then the algorithm is trivially correct and reports that vertex in linear time in step 1.

Otherwise the correctness of the algorithm follows from an inductive argument. A call CONNECT(k,m,S) discovers a previously unknown edge (p_i,p_j) on the upper hull. If p_i turns out to be the leftmost vertex of the upper hull its index will be printed, otherwise the recursive call CONNECT(k,i,S_{left}) will cause the sequence of vertices of the upper hull from p_k up to p_i to be printed. Similarly, if p_j is the rightmost vertex of the upper hull its index will be printed, otherwise the call CONNECT(j,m,S_{right}) will cause the portion of the upper hull from p_j up to p_m to be printed.

For the complexity bounds first observe that step 1 of the algorithm can easily be implemented to run in linear time. Thus it remains to show that the procedure CONNECT takes no more than $\mathbf{O}(n\log H_u)$ time. Note that using the median finding algorithm of Blum et al. [A-H-U, p.99] and using our bridge finding algorithm of section 2.3 steps 2.1 to 2.3 can be implemented to run in linear time. Thus the running time of CONNECT is determined by $f(|S|,H_u)$ where the function f must satisfy the recurrence relation

$$f(n,h) \leq \begin{cases} cn & \text{if } h=2 \\ cn + \max\limits_{h_l+h_r=h} \{ f(\frac{n}{2},h_l) + f(\frac{n}{2},h_r) \} & \text{if } h>2, \end{cases}$$

where c is some positive constant and $n \geq h > 1$.

We claim that $f(n,h) = \mathbf{O}(n\log h)$. To prove this we show that $f(n,h) = cn\log h$ satisfies the above recurrence relation. This is trivially true for the base case $h=2$. For $h>2$ note that

$$f(n,h) \leq cn + \max_{h_l + h_r = h} \{ c\frac{n}{2}\log h_l + c\frac{n}{2}\log h_r \}$$

$$= cn + \frac{1}{2}cn \max_{h_l + h_r = h} \{ \log(h_l h_r) \}.$$

Using elementary calculus it is easy to verify that the maximum is realized when $h_l = h_r = \frac{h}{2}$. Thus

$$f(n,h) \leq cn + \frac{1}{2}cn\log\left[\frac{h}{2}\right]^2 = cn + cn\log\left[\frac{h}{2}\right]$$

$$= cn + cn\log h - cn = cn\log h.$$

The linear space bound is trivial.

Q.E.D.

Corollary:

The convex hull of a set of n points in the plane can be found in time $\mathbf{O}(n\log H)$ using $\mathbf{O}(n)$ space, where H is the number of vertices found to be on the hull.

2.3 Finding the Bridge

We are given a set S of n points in the plane and a vertical line L which has points of S to its left and right. We are to find the edge of the upper hull of S that intersects L. If two edges intersect L, i.e. L contains a vertex v of the upper hull, we want to identify the edge for which v is the left endpoint. Call this edge the *bridge* and its endpoints *bridge points* (see Figure 2.3.1). Let us define a *supporting line* of S to be a non-vertical straight line which contains at least one point of S but has no points of S above it. Obviously the bridge must be contained in some supporting line. Call this supporting line b and let s_b be the slope of b.

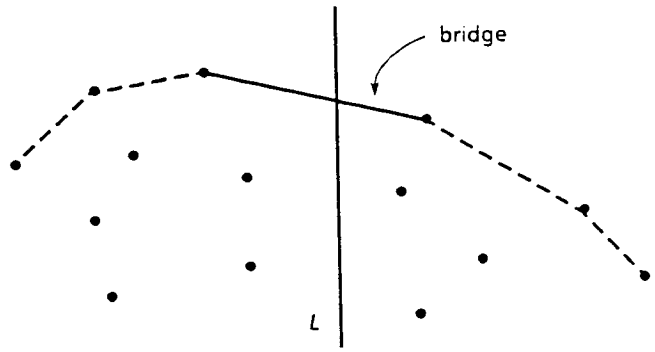


Figure 2.3.1: Bridge over separating line L

For our purposes, finding the bridge means identifying the two bridge points. One possible way of achieving this is to successively eliminate points from S as candidates for bridge points. For this purpose we pair up the points of S into $\lfloor n/2 \rfloor$ couples. The following two lemmas show how forming pairs of points facilitates the elimination of candidates for bridge points.

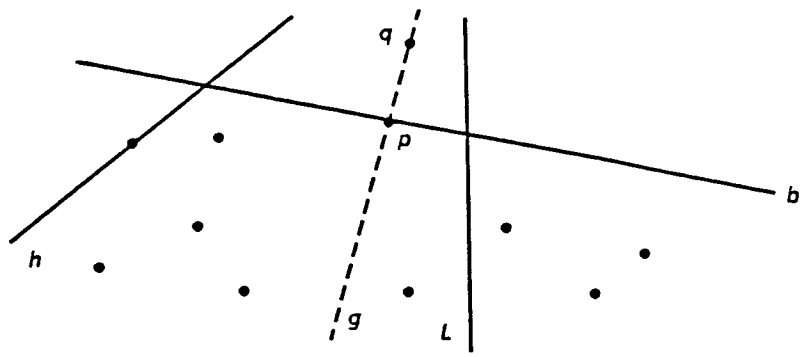


Figure 2.3.2: p cannot be bridge point

These two lemmas can be used to eliminate a bridgepoint candidate from every one of the $\lfloor n/2 \rfloor$ pairs. However, it is not clear at first how a condition like $s_{pq} > s_b$ can be tested without explicitly knowing s_b, the slope of b, and hence knowing the bridge, which after all is the entity that we want to compute. The solution to this problem is suggested by the following lemma:

Lemma 2.3.3:

Let h be the supporting line of S with slope s_h.

(1) $s_h < s_b$ iff h contains only points of S that are strictly to the right of L.

(2) $s_h = s_b$ iff h contains a point of S that is strictly to the right of L and a point of S that is to the left of or on L.

(3) $s_h > s_b$ iff h contains only points of S that are to the left of or on L.

Proof: trivial.

Thus to test whether $s_{pq} > s_h$ it suffices to find the supporting line h of S with slope s_{pq} and to determine whether h contains points of S to the right or to the left of L. Of course, finding this supporting line h requires linear time which is clearly too expensive to be done for every one of the $\lfloor n/2 \rfloor$ pairs individually. However, this problem can be overcome by judiciously choosing a slope s_h with the property that if $s_h > s_b$ then $s_{pq} > s_h$ (and hence $s_{pq} > s_b$) for a large number of pairs p,q and, if $s_h < s_b$ then $s_{pq} < s_h$ (and hence $s_{pq} < s_b$) for a large number of pairs p,q. A natural choice for an s_h with this property is the median of the slopes of the lines defined by the $\lfloor n/2 \rfloor$ pairs of points.

Now we are ready to give a more detailed PIDGIN-ALGOL description of our bridge finding algorithm. The function BRIDGE(S,a) takes as parameters a set $S = \{p_1,...,p_n\}$ of $n > 1$ points and a real number a representing the vertical line $L = \{(x,y) | x = a\}$. It is assumed that the point p_{min} in S with minimum x-coordinate is unique and that $x(p_{min}) \leq a$. Similarly, the point p_{max} in S with maximum x-coordinate is assumed to be unique and with $x(p_{max}) > a$. BRIDGE(S,a) returns as its value a pair (i,j), where p_i and p_j are the left and right bridge point respectively.

Algorithm 2.3.1

Function BRIDGE(S,a)

0. $C := \emptyset$ (* C denotes the set of candidates for bridgepoints *)
1. If $|S|=2$ then return((i,j)), where $S=\{p_i,p_j\}$ and $x(p_i)<x(p_j)$.
2. Choose $\lfloor |S|/2 \rfloor$ disjoint sets of size 2 from S.
   If a point of S remains, then insert it into C.
   Arrange each subset to be an ordered pair (p_i,p_j), such that $x(p_i)\leq x(p_j)$.
   Let $PAIRS$ be the set of these ordered pairs.
3. Determine the slopes of the straight lines defined by the pairs. In case the slope does not exist for some pair, apply Lemma 3.1, i.e:
   For all (p_i,p_j) in $PAIRS$ do
   if $x(p_i)=x(p_j)$ then delete (p_i,p_j) from $PAIRS$
   if $y(p_i)>y(p_j)$ then insert p_i into C
   else insert p_j into C
   else let $k(p_i,p_j) := \dfrac{y(p_i)-y(p_j)}{x(p_i)-x(p_j)}$.
4. Determine K, the median of $\{ k(p_i,p_j) \mid (p_i,p_j) \in PAIRS \}$.
5. Let $SMALL := \{ (p_i,p_j) \in PAIRS \mid k(p_i,p_j) < K \}$.
   Let $EQUAL := \{ (p_i,p_j) \in PAIRS \mid k(p_i,p_j) = K \}$.
   Let $LARGE := \{ (p_i,p_j) \in PAIRS \mid k(p_i,p_j) > K \}$.
6. Find the set of points of S which lie on the supporting line h with slope K, i.e.:
   Let MAX be the set of points $p_i \in S$, s.t. $y(p_i)-K^*x(p_i)$ is maximum.
   Let p_k be the point in MAX with minimum x-coordinate.
   Let p_m be the point in MAX with maximum x-coordinate.
7. Determine if h contains the bridge, i.e.:
   if $x(p_k) \leq a$ and $x(p_m) > a$ then return((k,m)).
8. h contains only points to the left of or on L:
   if $x(p_m) \leq a$ then
   for all $(p_i,p_j) \in LARGE \cup EQUAL$ insert p_j into C.
   for all $(p_i,p_j) \in SMALL$ insert p_i and p_j into C.
9. h contains only points to the right of L:
   if $x(p_k) > a$ then
   for all $(p_i,p_j) \in SMALL \cup EQUAL$ insert p_i into C.
   for all $(p_i,p_j) \in LARGE$ insert p_i and p_j into C.
10. return(BRIDGE(C,a)).

Theorem 3.1:

The function BRIDGE correctly determines the left and right bridge point in $\mathbf{O}(n)$ worst case time and space.

Proof:

The algorithm is trivially correct if S contains only two points. As long as S contains more than two points, BRIDGE either finds the bridge in step 7 or discards redundant points of S applying the rules of Lemma 2.3.1 and 2.3.2 (steps 3,8,9) and calls itself recursively with a smaller pointset.

Using the linear time median algorithm of Blum et al. [A-H-U, p.99], the body of BRIDGE without the recursive call can be executed in linear time and space. Furthermore, at least one quarter of the points of S are eliminated and not contained in C. Thus the worst case time and space requirements for the algorithm are bounded by

$$f(n) = \begin{cases} \mathbf{O}(1) & n = 2 \\ f(\frac{3n}{4}) + \mathbf{O}(n) & n > 2. \end{cases}$$

But it is well known that such a recursive function is $\mathbf{O}(n)$ [A-H-U,p.64].

Q.E.D.

At this point we want to mention that the bridge finding algorithm was inspired by the linear time two variable linear programming algorithms of M. Dyer [D] and N. Megiddo [M83]. A closer look even shows that the bridge problem can be formulated as a linear programming problem. However, for the

sake of simplicity and completeness it seems worthwhile to spell out the bridge finding algorithm explicitly.

2.4 The Expected Time Case

The divide-and-conquer algorithms in the two preceding sections are not terribly complicated. At first sight it even seems possible to actually implement these algorithms in some high level programming language in an hour's time, or so. However, one quickly discovers that the major obstacle to doing so is the median find algorithm. Thus quite naturally the question arises whether it is possible to do without it.

The median find algorithm is used in our algorithms to find a vertical line that divides a given point set evenly. What happens if we follow the example of Quicksort and choose a separating line at random? Ample experimental results have shown that Quicksort is one of the fastest sorting algorithms and these results have been supported by a careful theoretical analysis of the algorithm [K,Sk]. As it turns out the method of choosing a separator at random can also be successfully applied to our algorithms, thus changing the worst case time complexity to $\mathbf{O}(n^2)$ but retaining the $\mathbf{O}(n \log H)$ expected case time complexity.

Theorem 2.4.1:

If step 2.1 in Algorithm 2.2.1 is replaced by

"Let $a = x(p_i)$, where p_i is randomly chosen from $S-\{p_m\}$ such that the choice of every point in $S-\{p_m\}$ is equally likely."

then the modified algorithm has $\mathbf{O}(n\log H_u)$ expected case time complexity.

Proof:

The expected case running time of the modified algorithm can be bounded by the function g that must satisfy the following recurrence relation:

$$g(n,h) \le \begin{cases} bn & \text{if } n \ge h = 2 \\ bn + \dfrac{1}{n-1} \displaystyle\sum_{1 \le i < n} \max_{h_l + h_r = h} \{ g(i,h_l) + g(n-i,h_r) \} & \text{if } n \ge h > 2, \end{cases}$$

where b is some positive constant.

We claim that $g(n,h) = \mathbf{O}(n\log h)$, i.e. there is positive real constant c, such that for all $n \ge h \ge 2$, $g(n,h) \le cn\log h$ [3]. We prove our claim by induction.

The claim is trivially true for all n if $h=2$ and for all $n \le 5$ otherwise. Now we want to show the claim for some $n>5$ and $h<n$ on the assumption that $g(n',h') \le cn'\log h'$ for all $n'<n$ and $h'<h$. By definition of g and our inductive assumption we thus have

$$g(n,h) \le bn + \frac{1}{n-1} \sum_{1 \le i < n} \max_{h_l + h_r = h} \{ci\log h_l + c(n-i)\log h_r\}.$$

Using elementary calculus it is easy to show that for every i the maximum is

[3] In this proof we use w.l.o.g. the natural logarithm.

realized when $h_l = i\frac{h}{n}$ and $h_r = (n-i)\frac{h}{n}$. Therefore

$$g(n,h) \leq bn + \frac{c}{n-1}\sum_{1\leq i<n}\left[i\log i\frac{h}{n} + (n-i)\log(n-i)\frac{h}{n}\right]$$

$$= bn + \frac{2c}{(n-1)}\sum_{1\leq i<n} i\log i\frac{h}{n}$$

$$= bn + \frac{2c}{(n-1)}\log\frac{h}{n}\sum_{1\leq i<n} i + \frac{2c}{(n-1)}\sum_{1\leq i<n} i\log i.$$

As $\sum_{1\leq i<n} i\log i \leq \frac{1}{2}n^2\log n - \frac{1}{4}n^2$ (see [A-H-U, p.94])

and $\sum_{1\leq i<n} i = \frac{1}{2}n(n-1)$ we have

$$g(n,h) \leq bn + cn\log h - cn\log n + c\frac{n}{n-1}n\log n - \frac{c}{2}\frac{n^2}{n-1}$$

$$\leq bn - \frac{c}{2}n + cn\frac{\log n}{n-1} + cn\log h.$$

As $\frac{\log n}{n-1} < \frac{1}{2}$ for all integers $n>5$, there exists a real constant $c>0$ such that

$bn - \frac{c}{2}n + cn\frac{\log n}{n-1} < 0$ for all $n>5$ and hence $g(n,h) \leq cn\log h$.

Q.E.D.

The median find algorithm is used on one more occasion in our algorithms: in the bridge finding procedure. Again we can dispense with the median find algorithm and use random choice instead. The worst case complexity of such a modified bridge finding procedure is $\mathbf{O}(n^2)$, however the expected case running time is still $\mathbf{O}(n)$.

Theorem 2.4.2:

If step 4 of Algorithm 2.3.1 is replaced by

"Randomly choose an element (p_i,p_j) from $PAIRS$ such that the choice of every element is equally likely, and let $K := k(p_i,p_j)$,"

then the modified algorithm has expected case time complexity $\mathbf{O}(n)$.

Proof:

In the worst case no points are eliminated in step 3 of the modified function BRIDGE, and all the slopes $k(p_i,p_j)$ generated in that step are distinct. By the random choice of K, the cardinalities of $SMALL$ and $LARGE$ are uniformly distributed between 0 and $N-1$, where $N = \lfloor |S|/2 \rfloor$, the cardinality of $PAIRS$.

Assume pessimistically that whenever $|SMALL| \leq N/2$, the supporting line h contains only points to the right of L, and by step 9 only $|SMALL|+1$ points are eliminated. Symmetrically, assume that if $|LARGE| < N/2$, h contains only points to the left or on L, and step 8 is applied.

With these pessimistic assumptions the expected case running time of the modified algorithm is bounded from above by the function f, where for some positive constant b

$$f(n) = \begin{cases} bn & \text{if } n \leq 2 \\ bn + \frac{4}{n} \sum_{1 \leq i \leq n/4} f(n-i) & \text{if } n > 2. \end{cases}$$

It is an easy exercise in induction to show that $f(n) = \mathbf{O}(n)$.

Q.E.D.

2.5 Lower Bounds

The results of this section demonstrate that our $\mathbf{O}(n\log H)$ upper bound for the convex hull problem is the best possible on a quite general model of computation. Specifically, we prove an $\Omega(n\log H)$ lower bound for this problem on d-th order algebraic decision trees, for any fixed d.

There exist at least four variants of the convex hull problem characterized by increasingly stringent conditions on the form of the output. Let $S = \{p_1, \ldots, p_n\}$ be a set of points in $\mathbf{R}^2$, and let $ext(S)$ denote the set of vertices of the convex hull of S. The *convex hull sequence problem* asks for the elements of $ext(S)$ in consecutive cyclic order. The *convex hull set problem* asks for the elements of $ext(S)$ in arbitrary order. The *convex hull multiset problem* asks for a listing, in arbitrary order, of elements of S that coincide with elements of $ext(S)$. (This differs from the set problem only if S is a multiset). Finally, the *convex hull size problem* asks for the cardinality of $ext(S)$ (i.e. H).

It should be clear that the algorithm outlined in section 2.3 can be adapted to solve all of these problem variants in worst case time $\mathbf{O}(n\log H)$. Furthermore, since the sequence variant is at least as hard as the set variant, which in turn is at least as hard as the size variant, it will suffice to demonstrate a lower bound on the convex hull size problem, preferably using input point sets with no multiplicities. In fact we establish a lower bound on the even weaker *convex hull size verification problem*: given S and H, confirm that $|ext(S)| = H$. We show that any d-th order algebraic decision tree algorithm for this verification

problem must take $\Omega(n\log H)$ steps in the worst case, even if it can be assumed that all input points are distinct.

We follow Steele and Yao [S-Y] and Ben-Or [BO] in adopting algebraic decision trees as our model of computation. This model is rather strong. It allows evaluation of multivariate polynomials of bounded degree at unit cost. A *d-th order algebraic decision tree algorithm* (hereafter a *tree algorithm*) T for testing membership in a set $W \subset \mathbf{R}^n$ is a rooted tree whose internal nodes are labelled by polynomials in n variables of degree at most d, and whose leaves are labelled either YES or NO. Each internal node has out-degree three; the edges are labelled $-$,0, and $+$. Every input $\vec{z} \in \mathbf{R}^n$ determines a unique root to leaf path in T that can be constructively described as follows: Start at the root; as long as the current node is not a leaf evaluate the associated polynomial at $\vec{z}$ and follow the out-edge labelled with the sign of the outcome. We say that T decides membership in W iff, for every $\vec{z} \in \mathbf{R}^n$, $\vec{z}$ leads to a YES leaf of T if and only if $\vec{z} \in W$.

Yao [Y] establishes an $\Omega(n\log n)$ worst case lower bound for the convex hull set problem on algebraic decision trees of order two. This result is generalized by Ben-Or [BO], who demonstrates the same $\Omega(n\log n)$ lower bound for the convex hull size problem on algebraic decision trees of any fixed order d. Ben-Or's result is just one of a number of applications of the following general theorem concerning tree algorithms.

Theorem 2.5.1: [BO, Theorem 8]

Let $W \subset \mathbf{R}^n$ be any set and let T be any d-th order algebraic decision tree that solves the membership problem for W.

If W has N disjoint connected components, then T must have height (and hence worst case complexity) $\Omega(\log N - n)$.

We use the following generalization of the element distinctness problem [BO] to establish our lower bound. The *multiset size verification problem* asks to confirm, given a multiset $Z = \{z_1, \ldots, z_n\} \subset \mathbf{R}$ and an integer k, that Z has k distinct elements.

Corollary 2.5.1:

The multiset size verification problem requires $\Omega(n \log k)$ steps in the worst case, with any d-th order decision tree algorithm.

Proof:

It suffices to prove that the set $M_k = \left\{ (z_1, \ldots, z_n) \in \mathbf{R}^n \;\middle|\; |\{z_1, \ldots, z_n\}| = k \right\}$ has at least $k!k^{n-k}$ disjoint connected components.

Consider all tuples $(z_1, \ldots, z_n)$ with $z_1, \ldots, z_k$ set to distinct integers between 1 and k, and $z_{k+1}, \ldots, z_n$ set to arbitrary integers in that range. There are $k!k^{n-k}$ such tuples and each of them must lie in a different connected component of M_k.

Q.E.D.

We are now prepared to demonstrate our lower bound.

Theorem 2.5.2:

The convex hull size verification problem requires $\Omega(n\log H)$ steps, in the worst case, with any d-th order decision tree algorithm.

Proof:

We reduce the multiset size verification problem to the convex hull size verification problem in the following obvious way:

Let $Z = \{z_1, \ldots, z_n\}$ and k be an instance of the multiset size verification problem. Define $S = \{p_1, \ldots, p_n\} \subset \mathbf{R}^2$ by $p_i = (z_i, z_i^2)$. Then the set $ext(S)$ has exactly k elements if and only if Z has exactly k distinct elements.

Q.E.D.

The proof of the above theorem is somewhat disappointing in that the convex hull problem formed in the reduction has multiplicities on the convex hull. This straightforward reduction leaves open the possibility that there exists an algorithm solving the convex hull size verification problem (or any of the other variants) in $o(n\log H)$ steps for point sets that are known a priori to contain no duplicates. Fortunately, it is possible to strengthen the lower bound to include tree algorithms based on this rather dubious assumption as well. One can show that a convex hull algorithm that is only guarantueed to be correct when the input points are distinct could be used to solve a certain perturbed convex hull problem without input restrictions. An algorithm for this perturbed problem in

turn can yield a solution for the multiset size problem. This two stage reduction is given in [K-S86]. Here we just state the resulting theorem without proof.

Theorem 2.5.3:

The convex hull size verification problem requires $\Omega(n\log H)$ steps, in the worst case, with any d-th order decision tree algorithm, even if the input points may be assumed to be distinct.

3. Finding Maximal Vectors

3.1. Introduction, Background, and Notation

Let S be any finite set of vectors in $\mathbf{R}^d$ (real d-dimensional space). The familiar total order $\leq$ defined on $\mathbf{R}$ extends naturally to two different orders, which we denote by $\leq_L$ and $\leq_M$ on $\mathbf{R}^d$. Formally if $v = (v_1,...,v_d)$ and $w = (w_1,...,w_d)$ then $v \leq_L w$ if and only if $v = w$ or there exists a k, $1 \leq k \leq d$, such that $v_i = w_i$, $1 \leq i < k$, and $v_k < w_k$. ($\leq_L$ is the familiar *lexicographic order* on $\mathbf{R}^d$.) On the other hand $v \leq_M w$ (we say v is *dominated* by w) if and only if $v_i \leq w_i$, for $1 \leq i \leq d$. If $v \leq_M w$ and $v \neq w$ then we say that v is *strictly* dominated by w.

A vector $v \in T$ is said to be a *maximal element* in T if for all $w \in T$, $v \leq_M w$ implies $v = w$, that is, v is not strictly dominated by any element of T. We denote by $M(T)$ the set of maximal vectors in T. The *(d-dimensional) maximal vector problem* is defined as follows:

INPUT: A finite set $T \subset \mathbf{R}^d$.

OUTPUT: The set $M(T)$.

We are interested in characterizing the worst-case complexity of the maximal vector problem.

Earlier work on the maximal vector [K-L-P] characterized its inherent complexity in terms of the parameters d and n (where $n = |T|$). $C_d(n)$ denotes the

familiar min (over all algorithms) max (over all inputs) complexity of finding maximal vectors in sets of size n in $\mathbf{R}^d$. Kung et al. [K-L-P] establish the following bounds on $C_d(n)$:

$$C_d(n) \leq \mathbf{O}(n \log n), \qquad d = 2,3; \tag{1}$$

$$C_d(n) \leq \mathbf{O}(n(\log n)^{d-2}), \qquad d \geq 4; \tag{2}$$

$$C_d(n) \geq \mathbf{\Omega}(n \log n), \qquad d \geq 2. \tag{3}$$

The lower bound (3) is based on a binary comparison tree model of computation. Thus, in terms of n, the asymptotic complexity of the maximal vector problem is completely determined in this model, for $d = 2$ and 3. More recently the upper bound of [K-L-P] was improved to $C_d \leq \mathbf{O}(n(\log n)^{d-3} \log\log n)$ for $d \geq 4$ [G-B-T]. The maximal vector problem has also been investigated for its average case complexity. In [BKST] an algorithm is given that runs in linear *expected* time for certain classes of geometric distributions of the input points.

In the spirit of this thesis, our objective is to characterize more tightly the worst case complexity of the maximal vector problem by identifying its dependence on the output size $v = |V|$ (which could, of course, be anywhere in the range 1 to n). We use the notation $C_d(v,n)$ to describe the worst case complexity of maximal vector problems with input size n and output size v. Our algorithms (and hence our upper bounds) do not assume that v is known in advance.

The results can be summarized as follows:

$$C_d(v,n) \leq \mathbf{O}(n \log v) \qquad d=2,3 \qquad (1')$$

$$C_d(v,n) \leq \mathbf{O}(n(\log v)^{d-2}) \qquad d \geq 4 \qquad (2')$$

$$C_d(v,n) \leq \mathbf{O}(n(\log v)^{d-3}) \qquad d \geq 4,\ n \geq v^2 \qquad (3')$$

$$C_d(v,n) \leq \mathbf{O}(n \log v) \qquad d \geq 4,\ n \geq v^{2+\alpha},\ \alpha > 0 \qquad (4')$$

$$C_d(v,n) \geq \Omega(n \log v) \qquad d \geq 2 \qquad (5')$$

The O-notation suppresses constants which may be rather fast growing functions of d. Furthermore, the $\log v$ terms should be read as $\max\{1,\log v\}$. The lower bound (5') holds for arbitrary fixed-order algebraic decision tree algorithms. Thus, in this very general computational setting, we can demonstrate asymptotically optimal algorithms for many instances of the maximum vector problem (notably, when $d < 4$ or when $d \geq 4$ and $n \geq v^{2+\alpha}$), in terms of *both* size parameters n and v.

In independent work P. Ramanan [R] obtained weaker bounds for $C_d(v,n)$, namely $C_2(v,n) \leq \mathbf{O}(n \log v)$, $C_d(v,n) \leq \mathbf{O}(n \log n (\log v)^{d-3})$ for $d \geq 4$, and $C_d(v,n) \geq \Omega(n \log v)$ for the linear decision tree model of computation.

It will be helpful in subsequent sections to generalize and supplement some of the notation introduced earlier. Specifically, if S and R are finite subsets of $\mathbf{R}^d$, we denote by $S \leq_L R$ the assertion that $x \leq_L y$, for all $x \in S$ and for all $y \in R$. Similarly (but *note* the different quantification), $S \leq_M R$ denotes the assertion that for all $x \in S$, there exists a $y \in R$, such that $x \leq_M y$.

If $x = (x_1,...,x_d) \in \mathbf{R}^d$, we denote by x' the vector $(x_2,...,x_d) \in \mathbf{R}^{d-1}$. S' denotes the set $\{x' \mid x \in S\}$.

S/R denotes the set $\{x \in S \mid x \not\leq_M R\}$, that is the vectors in S not dominated by vectors in R. We call the problem of obtaining S/R from S and R the *screening* problem. Following [K-L-P], we denote by $F_d(r,s)$ the worst case complexity of constructing S/R, given $S,R \subset \mathbf{R}^d$, satisfying $|S| = s$, and $|R| = r$. The screening problem is a fundamental subproblem of the maximal vector problem and thus much of our effort concerns finding tight bounds for $F_d(r,s)$.

The following results about the 1- and 2-dimensional screening problem are taken from [K-L-P] Theorem 3.1.

Lemma 3.1.1:

(i) $F_1(r,s) \leq \mathbf{O}(r+s)$.

(ii) $F_2(r,s) \leq \mathbf{O}((r+s)\log r)$.

(iii) A finite set $R \subset \mathbf{R}^2$ can be represented in a data structure such that S/R can be found in $\mathbf{O}(|S| \log|R|)$ time for any $S \subset \mathbf{R}^2$.

Moreover, this data structure can be constructed incrementally (i.e. by a sequence of independent single point insertions) in time $\mathbf{O}(|R| \log|R|)$.

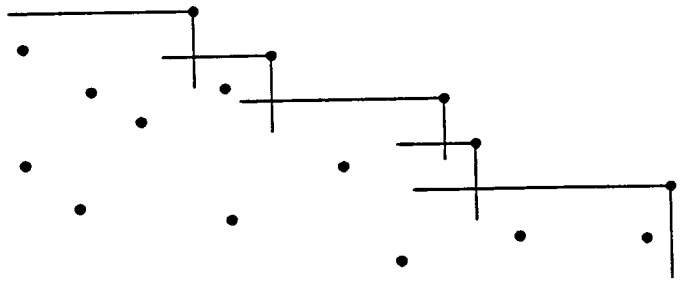
Proof:

First note that in computing S/R one removes and element from S if and only

if it is dominated by some *maximal* element of R, in other words, $S/R = S/M(R)$.

With this observation (i) becomes trivial because any non-empty $R \subset \mathbf{R}^1$ has exactly one maximal element and that can be found in $\mathbf{O}(|R|)$ time.

For the proof of (ii) and (iii) observe that the maximal elements of a set $R \subset \mathbf{R}^2$ induce in a natural way a "staircase" enveloping the set R (see the figure below).



Thus determining whether an element x is dominated by any point in R amounts to testing whether x lies "beneath" this staircase formed by $M(R)$. If $M(R)$ is stored in a balanced search tree such a test can be performed in logarithmic time. Moreover, when a new point is added to R the staircase of maximal elements of R can be updated in logarithmic amortized time.

Q.E.D.

3.2 The Two- and Three-Dimensional Cases

In this section we present algorithms for the maximal vector problem in $\mathbf{R}^2$ and $\mathbf{R}^3$ whose worst case running time is $\mathbf{O}(n\log v)$. In section 3.4 we prove a matching lower bound. Thus these algorithms are asymptotically worst case optimal even if the complexity of the problem is measured in terms of n, the input size, and v, the number of maximal vectors found, i.e. the output size.

The algorithms in this section and, as a matte of fact, all algorithms in this chapter rely heavily on the fact formalized in the following lemma whose simple proof is omitted.

Lemma 3.2.1:

Let $T \subset \mathbf{R}^d$, $d \geq 2$, be the union of two disjoint sets S and R, with $S \leq_L R$.

(i) $y \in R$ is maximal in T iff y is maximal in R.

(ii) $x \in S$ is maximal in T iff x is contained in and maximal in S/R.

The following divide-and-conquer type algorithm exploits Lemma 3.2.1. It takes as input a set T of n points in $\mathbf{R}^2$ and outputs $M(T)$.

```
Algorithm max1(T)
 if |T| ≤ 1 then return T
            else partition T into two sets S ≤_L R, with |S|, |R| ≤ ⌈n/2⌉;
                 M(R) ← max1(R);
                 construct Q = S/M(R);
                 M(Q) ← max1(Q);
                 return M(R) ∪ M(Q).
```

The partitioning step is easily implemented with the linear time median algorithm of Blum et al. [A-H-U p.99]. Thus ignoring the recursive calls this algorithm can be implemented in time $F_{d-1}(r,n/2) + \mathbf{O}(n)$, where $r = |M(R)|$. Thus we have:

Lemma 3.2.2:

There exists r, $0 < r < v$ and $r \leq n/2$, such that

$$C_d(v,n) \leq C_d(r,n/2) + F_{d-1}(r,n/2) + C_d(v-r,n/2) + \mathbf{O}(n).$$

Corollary 3.2.1:

$$C_2(v,n) \leq \mathbf{O}(n \log v).$$

Proof:

From Lemma 3.1.1 (i) and the conditions on r we get $F_1(r,n/2) = \mathbf{O}(n)$. Thus for $d=2$ the recurrence for C_d is essentially the same as the recurrence for the function f in Theorem 2.2.1 and the analysis presented there yields the desired result.

Q.E.D.

Unfortunately algorithm max1 does not lead to the optimal $\mathbf{O}(n \log v)$ time bound for $C_3(v,n)$. In the following we present an algorithm which can achieve this time bound for both $d=2$ and $d=3$. Instead of the divide-and-conquer paradigm, this algorithm is based on the sweeping paradigm. A naive sweeping algorithm for the maximal vector problem is stated below. It is essentially

Algorithm 3.1 of [K-L-P]. It constructs a set V containing the maximal vectors of a set T.

Algorithm max2(T)
1. sort T in lexicographically decreasing order;
2. **Let** $V \leftarrow \emptyset$;
   For each $x \in T$ (in decreasing order) **do**
   if $x' \not\leq_M V'$ **then** $V \leftarrow V \cup \{x\}$;
3. return V.

We analyze the running time of algorithm max2 for the case $d=3$. Recall that x' is the $(d-1)$-vector obtained from the d-vector x by omitting the first component, and $V'=\{x'|x \in V\}$. In this case $\{x'\}$ and V' are subsets of $\mathbf{R}^2$. Hence by Lemma 3.1.1 (iii) the set V, or to be precise, its projection V', can be organized in a data structure such that the test $x' \not\leq_M V'$ can be performed in $\mathbf{O}(\log v)$ time. Furthermore, all the insertions into this data structure do not take more than $\mathbf{O}(v \log v)$ time. Thus the time required for step 2 of algorithm max2 is $\mathbf{O}((n+v)\log v)$, i.e. $\mathbf{O}(n \log v)$. However, the overall running time of the entire algorithm is dominated by the sorting in step 1, which requires $\mathbf{O}(n \log n)$ time.

If the entire running time of this algorithm is to be reduced to $\mathbf{O}(n \log v)$, clearly something has to be done about the sorting in step 1. Removing it altogether seems impossible if the algorithm is to follow the sweeping paradigm. Hence the only sensible option seems to be to be lazy about the sorting. Sort "just a little bit" so that not too much time is spent and then use some modified sweeping algorithm on the partially sorted point set. A major obstacle to this

approach is the fact that one does not know how much time one can afford to spend on partial sorting since v, the number of maximal vectors to be found, is not known in advance. Let us ignore this obstacle for the time being and let us see what happens if we are willing to spend $\mathbf{O}(n\log k)$ time on sorting for $1 \leq k \leq n$. Consider the following algorithm:

Algorithm max3(T)
1. Partition T into k sets $T_1 \leq_L T_2 \leq_L \cdots \leq_L T_k$,
   such that $|T_i| \leq \lceil n/k \rceil$ for all i.
2. $V \leftarrow \emptyset$;
   For $i = k$ **down to** 1 **do**
2.1 $T_i \leftarrow T_i/V$;
2.2 **while** $T_i \neq \emptyset$ **do**
2.2.1 $z \leftarrow lexicographic\_max(T_i)$;
2.2.2 $T_i \leftarrow T_i/z$;
2.2.3 $V \leftarrow V \cup \{z\}$.

For proof of correctness of the above algorithm observe that $lexicographic\_max(S) \in M(S)$. This along with Lemma 3.2.1 ensures that the following invariant is maintained for the loop 2.2:

> V contains exactly all maximal elements z of T with $T_i \leq_L z$, and no $x \in T_i$ is dominated by any $z \in V$.

This invariant implies that at termination $V = M(T)$.

Next let us analyze the running time of algorithm max3 for $d=3$. Step 1 takes $\mathbf{O}(n\log k)$ time by recursive application of the linear time median algorithm of Blum et al. [A-H-U p.99]. Step 2.1 involves a total of at most $k\left\lceil \frac{n}{k} \right\rceil$

tests of the form $x' \leq_M V'$ each of which takes $\mathbf{O}(\log v)$ time by Lemma 3.1.1 (iii). The body of loop 2.2 is executed exactly v times as each iteration produces one new element of $M(T)$. Steps 2.2.1 and 2.2.2 can be performed in $O(\frac{n}{k})$ time and hence take $\mathbf{O}(\frac{v}{k}n)$ time overall. Step 2.2.3 takes $\mathbf{O}(v \log v)$ time overall by Lemma 3,1.1 (iii). We have thus shown the following:

Lemma 3.2.3:

For $1 \leq k \leq n$,

$$C_3(v,n) \leq \mathbf{O}(n \log k + n \log v + \frac{v}{k}n).$$

The main difficulty with this algorithm is the choice of k. If it is chosen too small, the $\mathbf{O}(\frac{v}{k}n)$ term contributed by step 2 dominates the running time. If k is chosen too large, the $\mathbf{O}(n \log k)$ contribution of the partitioning step 1 is too expensive. However, when $k = \mathbf{O}(v)$, the running time of the entire algorithm is $\mathbf{O}(n \log v)$ which would be optimal.

The solution to this dilemma is to interleave steps 1 and 2: run part of step 1 and create a coarse partition $T_1, \ldots, T_k$ of T for some small k; run step 2 on this coarse partition until either the algorithm terminates or more than k maximal vectors have been found; at this point return to step 1 and refine the partition of T by median-splitting each of the existing T_i's; resume step 2 and continue until the algorithm terminates or more than $2k$ maximal vectors have been found; in the latter case interleave another phase of step 1 to refine the

existing partition of T by a factor of 2 and carry on step 2 until termination or discovery of more than $4k$ maximal vectors, and so on. We give a more detailed description of this algorithm below.

Algorithm max4(T)

This algorithm takes as input a set T of n point in $\mathbf{R}^d$, $d \geq 2$ and constructs a set V containing the maximal vectors of T.

Q denotes a list of disjoint subsets of T.
Q is always sorted in the sense that for any two consecutive sets S_1, S_2 in Q, $S_1 \leq_L S_2$ holds.
Let *FIRST* always denote the first element in Q.

1. $V \leftarrow \emptyset$; $k \leftarrow 1$; $Q \leftarrow \{T\}$;
2. **while** $Q \neq \emptyset$ **do**
2.1. $FIRST \leftarrow FIRST/V$;
2.2. **while** $FIRST \neq \emptyset$ **do**
2.2.1 $z \leftarrow lexicographic\_max(FIRST)$;
2.2.2 $FIRST \leftarrow FIRST/z$;
2.2.3 $V \leftarrow V \cup \{z\}$;
2.2.4 **if** $|V| = 2^k - 1$ **then**
2.2.4.1 **For** each $S \in Q$ with $|S| \geq n/2^k$ **do**
partition S into two sets $S_1 \leq_L S_2$ of size at most $\lceil n/2^k \rceil$;
replace S in Q by S_1 and S_2 ;
2.2.4.2 $k \leftarrow k + 1$;
2.3 remove *FIRST* from Q.

The correctness of algorithm max4 follows from the same arguments as the correctness of max3. To analyze its complexity observe the following. For the loop 2.2 two invariants hold:

(1) $2^{k-1} \leq |V|+1 < 2^k$, and

(2) for all S in Q (in particular $S = FIRST$), $|S| \leq \lceil n2^{1-k} \rceil$.

Condition (1) implies that on termination $k \leq K = \lceil \log_2(v+1) \rceil$. Thus the condition in step 2.2.4 is true at most K times. As each execution of 2.2.4.1 takes linear time, step 2.2.4 takes $\mathbf{O}(n \log v)$ time overall.

For each k, $1 \leq k \leq K$, steps 2.2.1 and 2.2.2 are executed at most 2^{k-1} times. Because of invariant (2) each of these requires $\mathbf{O}(n2^{1-k})$ time. Summing over all k gives an $\mathbf{O}(n \log v)$ overall time bound for these two steps.

Step 2.2.3 is executed v times. It remains to bound the cost of step 2.1. Suppose step 2.1 is executed t times. Let $|FIRST| = n_i$ and $|V| = v_i$ on the i-th execution. Then step 2.1 takes $\sum_{i=1}^{t} F_{d-1}(v_i, n_i)$ time in total. Note that $\sum_{i=1}^{t} n_i = \mathbf{O}(n)$. This follows because those sets $FIRST$ which are not split in loop 2.2 must be mutually disjoint (and hence have total cardinality at most $n = |T|$). The sizes of the remaining sets $FIRST$ form a geometrically decreasing series (by invariant (2)) whose sum is $\mathbf{O}(n)$.

From the analysis of the running time of this algorithm we get the following general result.

Lemma 3.2.4:

There exist integer t, and n_i, v_i, $1 \le i \le t$, satisfying

$\sum_{i=1}^{t} n_i = \mathbf{O}(n)$, $n_i \ge n/v_i$, and $v_i \le v$, such that

$$C_d(v,n) \le \mathbf{O}(n \log v) + \sum_{i=1}^{t} F_{d-1}(v_i, n_i)$$

For the case $d=3$ the analysis of algorithm max4 can be carried further. As a consequence of Lemma 3.1.1 (iii) the i-th execution of step 2.1 takes time $\mathbf{O}(n_i \log v)$. As $\sum_{i=1}^{t} n_i = \mathbf{O}(n)$ this step takes therefore time $\mathbf{O}(n \log v)$ overall. Again because of Lemma 3.1.1 (iii) the v insertions of step 2.2.3 can be performed in time $\mathbf{O}(n \log v)$. Thus the overall running time of algorithm max4 for the case $d=3$ is $\mathbf{O}(n \log v)$.

Using this fact and citing again Corollary 2.1 we can state the main results of this section.

Theorem 2.2.1:

(i) $C_2(v,n) \le \mathbf{O}(n \log v)$.

(ii) $C_3(v,n) \le \mathbf{O}(n \log v)$.

3.3 The Higher Dimensional Case

In the previous section we saw that the maximal vectors of point sets in $\mathbf{R}^2$ and $\mathbf{R}^3$ can be found in optimal $\mathbf{O}(n \log v)$ time by using a method that can be

dubbed "lazy sorting". In this section we apply this method in the higher dimensional setting. It turns out that the optimal $\mathbf{O}(n\log v)$ time bound can still be achieved if the number of maximal vectors is sufficiently small. To be precise, $v = \mathbf{O}(n^{\frac{1}{2}})$ is required for point sets in $\mathbf{R}^d$, $d=4$, and for $d \geq 5$ it must be the case that $v = \mathbf{O}(n^{\frac{1}{2}-\varepsilon})$. The reason for this sparsity requirement is the fact that for $d=2,3$ one can exploit data structures to solve the "screening" subproblems that arise during dimension reduction, whereas for $d \geq 4$ we know of no efficiently maintainable data structure for this purpose. Before addressing the maximal vector problem itself we therefore present and analyze an algorithm for the d-dimensional screening problem.

Before giving the algorithm we restate the problem: For two finite sets $R,S \subset \mathbf{R}^d$ we want to determine S/R, i.e. we want to remove from S all points that are dominated by some element of R. Also recall that we denote the worst case complexity of this problem by $F_d(r,s)$.

Algorithm SCREEN is intended for non-empty sets $R,S \subset \mathbf{R}^d$, $d \geq 3$. It is a recursive divide-and-conquer algorithm with the special feature that its divisiveness can be specified via an integer parameter $t>1$. The running time of the algorithm depends crucially on t and thus can be minimized by a prudent choice of t as a function of the sizes of R and S.

The geometric intuition behind this algorithm is roughly as follows: Using $t-1$ hyperplanes that are normal to the first coordinate axis cut d-space into t parts $P_1, \ldots, P_t$, indexed such that all points in P_i have smaller first coordi-

nate than points in P_{i+1}. Moreover, choose those hyperplanes such that the cardinalities of $R_i = R \cap P_i$ are approximately the same for all i. For each i let $S_i = S \cap P_i$.

Note that a point x in S_j can only be dominated by points in R_j or by points in R_i with $i>j$. In the first case x can be removed from S_j by the (recursive) screening operation S_j/R_j. In the second case x can be removed by the screening operation $S_j/(\bigcup_{i>j} R_j)$. However, in this case the first coordinates of all points involved are irrelevant since by construction the first coordinate of any point in S_j is smaller than the first coordinate of any point in $\bigcup_{i>j} R_j$. Thus the $(d-1)$-dimensional (recursive) screening operation $S_j'/(\bigcup_{i>j} R_j')$ can be used instead of the d-dimensional one.

SCREEN(R,S,d,t)

1. case 1: $|R| = 1$
   remove from S all points dominated by the only element of R
2. case 2: $d = 2$
   apply Lemma 3.1.1 (iii)
3. case 3: $|R| > 1$ and $d > 2$

3.1 **if** $d = 3$ **then** $k \leftarrow |R|$
else $k \leftarrow \min(t,|R|)$

3.2 partition R into k disjoint sets
$R_1 \leq_L \cdots \leq_L R_k$, each of size at most $\lceil n/k \rceil$

3.3 partition S into k disjoint sets $S_1 \leq_L \cdots \leq_L S_k$, such that
for $1 \leq j \leq k$, $R_{j-1} \leq_L S_j \leq_L R_{j+1}$, where $R_0 = R_{k+1} = \emptyset$

3.4 SCREEN(R_k,S_k,d,t)
$RR' \leftarrow R_k'$ (* if $d=3$ use the data structure mentioned in Lemma 1.1 (iii) to maintain RR'. *)

3.5 **For** $j = k-1$ **downto** 1 **do**

3.5.1 SCREEN($RR', S_j', d-1, t$) (* Here it is assumed that removing a point x' from S_j' also deletes the corresponding x from S_j. *)

3.5.2 SCREEN(R_j, S_j, d, t)

3.5.3 $RR' \leftarrow RR' \cup R_j'$

Algorithm SCREEN can easily be proven correct using the results of Lemma 3.2.1. Let $G_d(r,s,t)$ denote the worst case running time of Algorithm SCREEN.

Lemma 3.3.1:

(i) $G_d(1,s,t) = \mathbf{O}(s)$

(ii) There exists a constant $c > 0$ such that for $d \geq 3$ and $t \geq 2$

$$G_d(r,s,t) \leq c(d-2)\left[s + r(t-1)^{d-3}\right] \log_2 r (\max(1, \log_t r))^{d-3} .$$

Proof:

Assertion (i) is clearly true since if $|R| = 1$ only step 1 of Algorithm SCREEN is executed which takes only $\mathbf{O}(s)$ time.

For assertion (ii) we consider the case $d=3$ separately. In this case k is set to $|R|$ and the sets R_j formed in step 3.2 are all singleton sets. Steps 3.2 and 3.3 together therefore take $\mathbf{O}((s+r)\log r)$ time. Furthermore, steps 3.4 and 3.5.2 take $\mathbf{O}(s)$ time overall because of assertion (i). The set RR' is maintained in a data structure as mentioned in Lemma 3.1.1 (iii). Thus all executions of step 3.5.3 take $\mathbf{O}(r \log r)$ time and all the executions of step 3.5.1 take $\mathbf{O}(s \log r)$ time. Hence for $d=3$ the running time of the algorithm is $\mathbf{O}((s+r)\log r)$ as claimed.

It remains to prove that G_d satisfies the bound in (ii) when $r>1$ and $d>3$

First we consider the case $t \geq r$. In this case step 3.1 of the algorithm sets k to r. Thus step 3.2 and 3.3 take time $\mathbf{O}((s+r)\log r)$ and all the executions of step 3.4 and 3.5.2 together take time $\mathbf{O}(s+r)$ since $|R_j| = 1$ for all j. It follows that in this case G_d must satisfy the recurrence

$$G_d(r,s,t) \leq c(s+r)\log_2 r + \sum_{j=1}^{r-1} G_{d-1}(r,s_j,t) ,$$

where c is a constant and s_j is the cardinality of set S_j formed in step 3.3. Using the base case $G_3(r,s,t) \leq c(s+r)\log_2 r$ it is an easy exercise in induction on d to show that

$$G_d(r,s,t) \leq c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 r$$

as desired.

Next we consider the case $t < r < t^2$. Here k is set to t in step 3.1. Thus G_d must satisfy the recurrence

$$G_d(r,s,t) \leq c(s+r)\log_2 t + \sum_{j=1}^{t} G_d(\frac{r}{t},s_j,t) + \sum_{j=1}^{t-1} G_{d-1}(r,s_j,t) .$$

As by assumption $\frac{r}{t} \leq t$, the first sum is at most

$$c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 \frac{r}{t}.$$

Again noting that $\frac{r}{t} \leq t$, G_d must therefore satisfy the recurrence

$$G_d(r,s,t) \leq c(s+r)\log_2 t + c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 \frac{r}{t} + \sum_{j+1}^{t-1} G_{d-1}(r,s_j,t) .$$

Assuming inductively on d that

$G_{d-1}(r,s_j,t) \leq c(d-3)\left[s_j + r(t-1)^{d-4}\right]\log_2 r(\log_t r)^{d-4}$ yields

$$\begin{aligned} G_d(r,s,t) &\leq c(s+r)\log_2 t \\ &\quad + c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 \frac{r}{t} \\ &\quad + c(d-3)\left[s + r(t-1)^{d-3}\right]\log_2 r(\log_t r)^{d-4} \\ &\leq c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 \frac{r}{t} \\ &\quad + c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 r(\log_t r)^{d-4} \\ &\leq c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 \frac{r}{t}(\log_t r)^{d-4} \\ &\quad + c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 r(\log_t r)^{d-4} . \end{aligned}$$

It is not difficult to check that $\log_2 \frac{r}{t} + \log_2 r \leq \log_2 r \log_t r$ and therefore we conclude that for $t < r \leq t^2$

$$G_d(r,s,t) \leq c(d-2)\left[s + r(t-1)^{d-3}\right]\log_2 r(\log_t r)^{d-3} ,$$

as desired.

Finally we consider the case $r > t^2$. Now G_d must satisfy the recurrence

$$\begin{aligned} G_d(r,s,t) &\leq c(s+r)\log_2 t \\ &\quad + \sum_{j=1}^{t} G_d(\frac{r}{t},s_j,t) + \sum_{j+1}^{t-1} G_{d-1}(r,s_j,t) . \end{aligned}$$

We proceed by double induction on d and r and obtain

$$
\begin{aligned}
G_d(r,s,t) \;\le\; & c(s+r)\log_2 t \\
& + \sum_{j=1}^{t} c(d-2)\left[s_j + \frac{r}{t}(t-1)^{d-3} \right] \log_2 \frac{r}{t} \,(\log_t \frac{r}{t})^{d-3} \\
& + \sum_{j=1}^{t-1} c(d-3)\left[s_j + r(t-1)^{d-3} \right] \log_2 r\,(\log_t r)^{d-4} \\
\\
\le\; & c(s+r)\log_2 t \\
& + c(d-2)\left[s + r(t-1)^{d-3} \right] \log_2 \frac{r}{t} \,(\log_t r)^{d-3} \\
& + c(d-3)\left[s + r(t-1)^{d-3} \right] \log_2 r\,(\log_t r)^{d-4} \\
\\
\le\; & c(d-2)\left[s + r(t-1)^{d-3} \right] \log_2 r\,(\log_t r)^{d-3} \\
& - c(d-2)\left[s + r(t-1)^{d-3} \right] \log_2 t\,(\log_t r)^{d-3} \\
& + c(d-2)\left[s + r(t-1)^{d-3} \right] \log_2 r\,(\log_t r)^{d-4} .
\end{aligned}
$$

But this proves the desired result since the second and third line cancel each other because $\log_t r = \dfrac{\log_2 r}{\log_2 t}$.

Q.E.D.

Lemma 3.3.1 immediately implies bounds for $F_d(r,s)$, the worst case complexity of the screening problem.

Corollary 3.3.1: For $d \ge 3$

$$F_d(r,s) \;=\; \mathbf{O}((d-2)(r+s)(\log r)^{d-2}) .$$

Proof: Use Lemma 3.3.1 (ii) with $t = 2$.

Corollary 3.3.2: For $d \geq 3$ and $s \geq r^{d-2}$

$$F_d(r,s) = \mathbf{O}((d-2)s\log r).$$

Proof: Use Lemma 3.3.1 (ii) with $t = r$.

Corollary 3.3.3: For $d > 3$ and $r < s \leq r^{d-2}$

$$F_d(r,s) = \mathbf{O}\left((d-2)s\log r(\ \max(1, \frac{d-3}{\log_r s - 1})\ )^{d-3}\right)$$

Proof: Use Lemma 3.3.1 (ii) with $t = (\frac{s+r}{r})^{\frac{1}{d-3}}$.

Maybe the most surprising consequence is the following:

Corollary 3.3.4:

Let $d \geq 3$ be fixed and let ε be a positive constant.

For the class of d-dimensional screening problems with $r^{1+\varepsilon} < s$

$$F_d(r,s) = \mathbf{O}(s\log r).$$

This concludes our investigation of the complexity of the "screening" problem. We have now everything ready to address the higher dimensional maximal vector problem.

Theorem 3.3.1:

For all $d \geq 4$, there exists a constant c such that

$$C_d(v,n) \leq cn(d-3+\log v)^{d-2}.$$

Proof:

We proceed by induction on v. When $v = 1$ the bound is immediate, so assume $v > 1$. Lemma 3.2.2 of section 3.2 gives rise to the recurrence

$$C_d(v,n) \leq C_d(v_1,n/2) + C_d(v-v_1,n/2) + F_{d-1}(v_1,n/2) + \mathbf{O}(n).$$

By corollary 3.3.1 and our inductive assumption we have

$$\begin{aligned} C_d(v,n) &\leq c\, n/2(d-3+\log v_1)^{d-2} \\ &\quad + c\, n/2(d-3+\log(v-v_1))^{d-2} + \mathbf{O}(d\, n(\log v_1)^{d-3}) \\ &\leq c\, n(d-3+\log(v/2))^{d-2} + \mathbf{O}(dn(\log v)^{d-3}) \qquad (*) \\ &\leq c\, n(d-3+\log v)^{d-3}((d-3+\log v)-1) + \mathbf{O}(dn(\log v)^{d-3}) \\ &\leq c\, n(d-3+\log v)^{d-2}. \end{aligned}$$

Step (*) exploits the fact that the function $H(v) = (d-3+\log v)^{d-2}$ has a negative second derivative for all $v \geq 1$, and hence $H(v_1) + H(v-v_1)$ achieves its maximum, for $0 < v_1 < v$, when $v_1 = \frac{v}{2}$.

Q.E.D.

Thus in any fixed dimension $d \geq 3$, we have established that $C_d(v,n) \leq \mathbf{O}(n(\log v)^{d-2})$. This should be compared with the bound $C_d(n) \leq \mathbf{O}(n(\log n)^{d-2})$ established by Kung et al [K-L-P]. A comparison of the bounds on $F_d(r,s)$, for $d \geq 4$, given in Corollaries 3.3.1 and 3.3.2 suggests that our bound on $C_d(v,n)$ might be improved by taking advantage of the more

The condition $n > v^2$ of part (ii) in turn implies $n_i \geq \frac{n}{v_i} \geq \frac{n}{v} > v$. By Corollary 3.3.3 it follows that

$$C_d(v,n) \leq \mathbf{O}(n\log v) + c\sum_{i=1}^{t}(d-1)n_i \log v_i\left(\max\left(1, \frac{d-4}{\log_{v_i} n_i - 1}\right)\right)^{d-4}$$

$$\leq \mathbf{O}(n\log v) + c\sum_{i=1}^{t}(d-1)n_i \log v_i\left(\max\left(1, \frac{d-4}{\log_{v} n - 2}\right)\right)^{d-4},$$

since $\log_{v_i} n_i \geq \log_v (n/v) = \log_v n - 1$,

$$\leq \mathbf{O}\left(d\, n \log v\left(\frac{d-4}{\log_v n - 2}\right)^{d-4}\right),$$

which proves part (ii).

Q.E.D.

As corollaries we obtain the following

Theorem 3.3.3:

For all $d \geq 4$ and $n \geq v^2$

$$C_d(v,n) \leq \mathbf{O}(n\,(\log v)^{d-3}).$$

Theorem 3.3.4:

For all $d \geq 5$ and $n = \Omega(v^{2+\alpha})$, for some $\alpha > 0$,

$$C_d(v,n) \leq \mathbf{O}(dn\log v).$$

3.4 Lower Bounds

Kung et al. [K-L-P] demonstrate the lower bound

$$C_d(n) \geq \Omega(n \log n), \quad \text{for } d \geq 2,$$

for all binary comparison tree algorithms. This bound is realized in situations where v, the number of maximal vectors, is $\Omega(n)$. If we compare this with the upper bounds of the preceding sections we are motivated to demonstrate a lower bound in terms of *both* n and v. We are able to show that

$$C_d(v,n) \geq \Omega(n \log v), \quad \text{for } d \geq 2,$$

for arbitrary fixed order algebraic decision tree algorithms. As a consequence, $C_d(v,n)$ is specified to within multiplicative constants, when $d = 2$ or 3, when $d = 4$ and $n \geq v^2$, and when $d \geq 5$ and $n \geq v^{2+\alpha}$, $\alpha \geq 0$.

The maximal vector problem, defined in section 1, might be more precisely called the *maximal vector set problem* (to distinguish it, for example, from variants which require the output to be presented in some specified order). For the purposes of establishing the most powerful lower bounds we will study what we call the *maximal vector cardinality verification problem*, namely, given a set $T \subset \mathbf{R}^d$ and an integer k, confirm whether or not the set V of maximal vectors in T satisfies $|V| = k$. It should be clear that a lower bound on this problem translates directly into a lower bound on more demanding variants.

Our lower bound result closely parallels the $\Omega(n \log H)$ lower bound for the two-dimensional convex hull problem (where n denotes the number of data points and H denotes the number of extreme points), presented in Chapter 2.

We will present only an outline of the proof of the maximal vector bound.

The *multiset size verification problem* takes a multiset $Z = \{z_1, z_2, \ldots, z_n\} \subset \mathbf{R}$ and an integer k, and asks whether Z has k distinct elements. In Corollary 2.5.1 we showed that the multiset size verification problem requires $\Omega(n \log k)$ steps, in the worst case, with any fixed order algebraic decision tree algorithm. The multiset size verification problem can be reduced to the maximal vector cardinality verification problem, yielding:

Theorem 3.4.1. :

The maximal vector cardinality verification problem requires $\Omega(n \log k)$ steps, in the worst case, using any fixed order algebraic decision tree algorithm.

Proof:

Let $Z = \{z_1, \ldots, z_n\}$ and k be an instance of the multiset size verification problem. Define $T = \{p_1, \ldots, p_n\} \subset \mathbf{R}^d$, by $p_i = (z_i, z_i, \ldots, z_i, -z_i)$, $1 \leq i \leq n$. It should be clear that T has exactly k distinct maximal vectors if and only if Z has exactly k distinct elements.

Q.E.D.

Observant readers may be disappointed to note that Theorem 3.4.1 takes full advantage of an assumption that sets of maximal vectors contain no duplicate points. Thus, Theorem 3.4.1 does not exclude the possibility of an algorithm for the maximal vector cardinality verification problem that, somehow,

exploits an assumption that all of its inputs are distinct, and runs in $\mathbf{o}(n \log k)$ steps. Fortunately, similarly to Theorem 2.5.2, also Theorem 3.4.1 can be strengthened to apply even in the presence of this input distinctness assumption. We state the stronger result without proof.

Theorem 3.4.2:

The maximal vector cardinality verification problem requires $\Omega(n \log k)$ steps, in the worst case, with any fixed order algebraic decision tree algorithm, even if the input vectors are assumed to be distinct.

Corollary 3.4.1:

$C_d(v,n) \geq \Omega(n \log v)$, for $d \geq 2$, using any fixed order algebraic decision tree algorithm.

4. Overlaying Planar Convex Subdivions

4.1 Preliminaries

A *convex subdivision* of the plane is a partition of the plane into a finite number of (relatively) open convex sets. It is not hard to see that these convex sets can be only of the following three types: a set can be a 2-dimensional *region*, i.e. it is an open convex polygon (possibly unbounded); a set can be a 1-dimensional *edge*, i.e. a (possibly unbounded) interval on some straight line; and a set can be a 0-dimensional *vertex*. Regions are separated by edges and vertices form endpoints of edges. Examples of convex subdivisions are Voronoi diagrams and straight line arrangements.

From any two planar convex subdivisions **B** and **G** one can form a new convex subdivision which we call the *overlay* of **B** and **G**. It consists of all nonempty sets of the form $B \cap G$ with $B \in \mathbf{B}$ and $G \in \mathbf{G}$. We are interested in computing the overlay of two convex subdivisions efficiently.

First we have to settle the question of representation. We assume that convex subdivisions are represented by some suitable planar graph data structure (for instance the quad edge structure [G-St]), appropriately augmented to be able to represent unbounded edges and regions. We also have to specify what we mean by "efficient computation." Note that the overlay of two convex subdivision **B** and **G** of size m and n respectively can consist of as many as $\mathbf{O}(mn)$ sets, as alluded to in Figure 4.1.1. On the other hand, it is not hard to exhibit examples where the overlay consists of only $\mathbf{O}(m+n)$ sets. It is thus

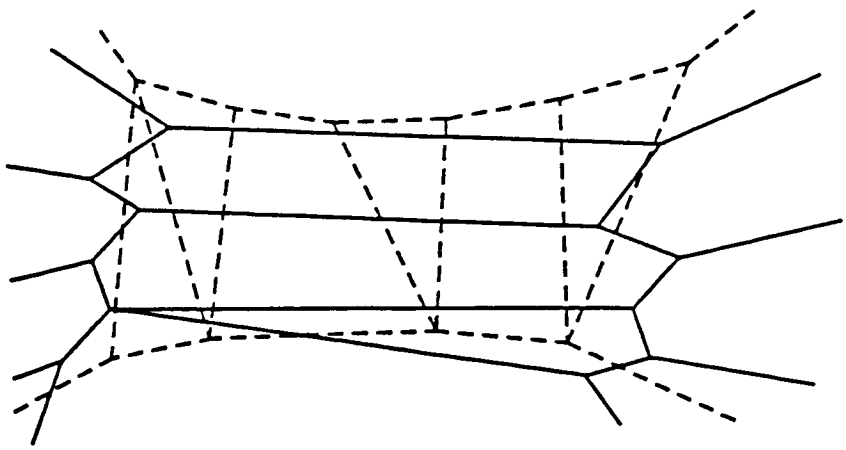


Figure 4.1.1: Overlay of two subdivisions

desirable that the running time of an overlay construction algorithm also depend on the size of the convex subdivision produced. In the following we let K denote the size of the overlay subdivision produced.

From now on we will designate the subdivision **B** and its sets as *blue* and the subdivision **G** and its parts as *green*. For ease of presentation we will initially assume that **B** and **G** are in "non-degenerate" position, i.e. they both have no vertical edges, no blue and green vertices coincide, and no blue (green) vertex lies on a green (blue) edge. In section 4.7 we shows how to deal with such degeneracies.

The obvious approach for solving the overlay problem is the plane sweep paradigm: sweep a vertical line from left to right over the two subdivisions and maintain the invariant that the part of the overlay that is totally to the left of the sweeping line has been correctly constructed. In its most natural form such

a sweeping algorithm takes time $\mathbf{O}((m+n+K)\log(m+n))$ (see e.g. [B-O]). By exploiting convexity and changing the sweeping invariant a bit (in the sense that parts of the overlay to the right of the sweeping line must also be computed) one can design a more sophisticated algorithm with running time $\mathbf{O}((m+n)\log(m+n)+K)$ [N-P]. We will use the idea of a "topological line sweep" as proposed by Edelsbrunner and Guibas [E-G] to construct the overlay in optimal $\mathbf{O}(m+n+K)$ time.

4.2 The Topological Line Sweep

The topological line sweep is a generalization of a straight line sweep. A line is swept over the plane from left to right and at all times the invariant is maintained that everything to the left of the line has been correctly computed already. However, the sweeping line is not a straight vertical line. It is allowed to bulge to a certain extent. In order to quantify the legal amount of bulging, and also for the purpose of dealing with such a line algorithmically, we must give a combinatorial characterization of what constitutes a correct sweeping line. We do this by identifying the line by the sequence of subdivision edges that intersect the line. We say an edge a of a subdivision lies *immediately above* an edge b iff there is a region R in the subdivision such that a is a top edge of R and b is a bottom edge of R. A sequence of edges $e_1, e_2, \cdots, e_s$ constitutes a *cut* of a subdivision $\mathbf{X}$ iff e_1 is a bottom edge of the top region of $\mathbf{X}$, e_s is a top edge of the bottom region of $\mathbf{X}$, and e_i lies immediately above e_{i+1} for $1 \le i < s$. (Note that because of convexity and the assumption of no vertical

edges all the notions involving "top" and "bottom" are well defined.) A line L is said to be a *valid sweep line* for a subdivision **X** iff the sequence of edges of **X** that intersect L forms a cut of **X** (see Figure 4.2.1). Note that for every cut C of **X** there is a valid sweep line L_C. One can define a partial order on the cuts of a subdivision **X** by saying that a cut C is to the left of cut $\bar{C}$ iff the set of vertices of **X** to the left of L_C is a subset of the set of vertices to the left of $L_{\bar{C}}$. There is a unique leftmost cut, the one with no vertices to the left of L_C, and a unique rightmost cut, the one with all vertices to the left of L_C. Sweeping a subdivision **X** entails starting with the leftmost cut C_0 and then going through a sequence of cuts C_i until the rightmost cut C_t is reached. Moreover, for all i, $0 \leq i < t$, C_i must be left of C_{i+1}, and the difference between C_i and C_{i+1} should be "small", i.e. one more vertex is to the left of C_{i+1} than to the left of C_i.

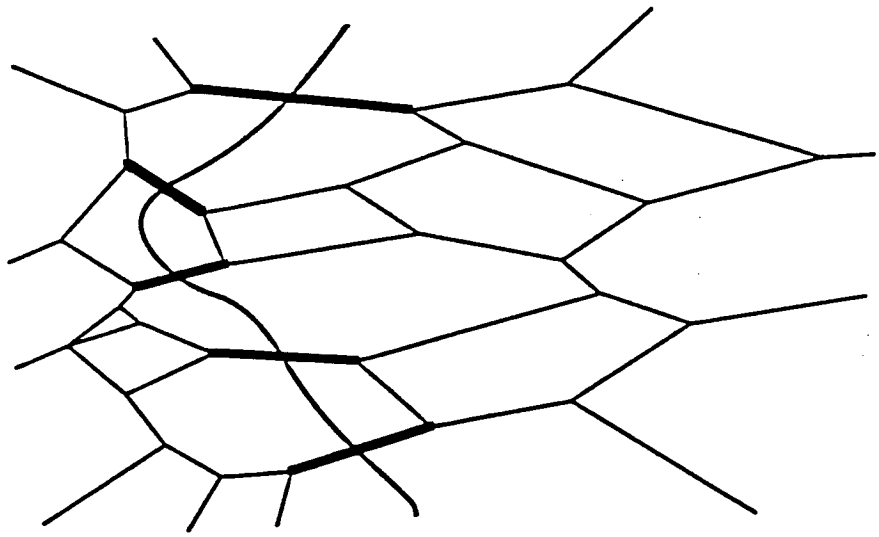


Figure 4.2.1: A cut and a sweep line

For a cut C_i in a sweep what can be the next cut C_{i+1}? Which vertex of the subdivision can be swept over next? It is easy to verify that any vertex v for which all left-edges (i.e. edges incident to and to the left of v) lie in the current cut C_i can be chosen for this purpose. The new cut C_{i+1} is then the same as C_i except that the left-edges of v are replaced by the right-edges of v in their natural vertical order (see Figure 4.2.2). We now know how a sweep can proceed. Is it possible that a sweep can get stuck? Is it possible that there is no vertex all of whose left-edges are in the current cut (provided, of course, we are not dealing with the rightmost cut)? The answer to this question is negative. Consider the leftmost of the vertices to the right of the current cut. Obviously all its left-edges must be in the cut. It is worth pointing out that the straight line sweep relies on this fact and always chooses as the next vertex to sweep over the one that is leftmost among the vertices to the right of the

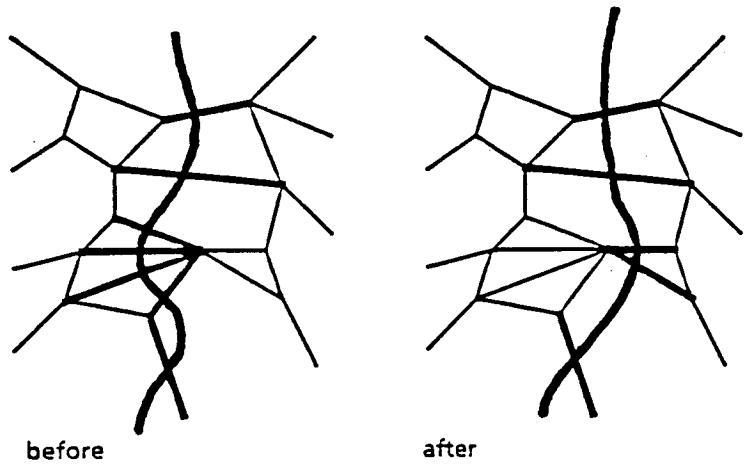


Figure 4.2.2: Sweeping over a vertex

sweeping line. Note that this is a global condition. The main advantage of the topological line sweep lies in the fact that it uses instead the purely local condition of "all left-edges in the current cut" for advancing the sweep, thus saving a logarithmic factor in the running time.

For the purpose of constructing the overlay **X** of two subdivisions **B** and **G** the sweeping approach will be applied as follows. Conceptually we will sweep over the result subdivision **X**. Of course we don't know **X** in advance. However, we know that it is made up of **B** and **G** and we will exploit the fact that every cut of **X** is a cut of **B** merged with a cut of **G**. Moreover, we will maintain the invariant that all parts of **X** that are completely to the left of the current sweep line have been correctly identified and constructed.

4.3 Starting the Sweep

To start the sweep we have to construct the leftmost cut of **X**. Note that the leftmost cut of any convex subdivision consists of the edges which are unbounded to the left sorted according to their slope. Thus the leftmost cut of **X** can be formed from the leftmost cuts of **B** and **G** by a simple merge operation in linear time. Throughout the sweep we will augment the sequence of edges in the current cut with a sentinel edge on each end. These sentinel edges are taken to be both green and blue and they are assumed to be unbounded to the right.

4.4 Advancing the Sweep

In the resulting subdivision **X** there are really three types of vertices: (i) vertices that are formed by the intersection of blue and green edges, (ii) vertices of the blue subdivision, and (iii) vertices of the green subdivion. Note that the type (i) vertices are not known in advance but are discovered on the fly. We have three rules for advancing the sweeping line, according to the type of the vertex we are currently sweeping over.

Rule 1: If in the current cut there are two consecutive edges e_i and e_{i+1} of different colour which intersect so that their vertical order to the right of their intersection point is not reflected by their order in the current cut, then

interchange the two edges in the cut,

make their intersection point p, and the segments of the edges to the left of p part of **X**, etc.

Rule 2: If v is a *safe* blue vertex whose left-edges form a contiguous subsequence of the current cut, then

delete these edges from the cut, and

insert instead in their natural vertical order the right-edges of v.

Rule 3: Same as Rule 2 but interchange blue and green.

We owe a definition of what it means for a blue vertex v to be *safe*. Let e and e' be the green edges immediately above and below v, respectively. Vertex v is *safe* iff neither e nor e' is to the right of the current cut. Safety of green vertices is defined symmetrically. Note that once a vertex is safe during a sweep it stays safe.

We want to argue the correctness of Rules 1 to 3. Correctness in this context means that the sweeping invariant "all parts of **X** to the left of the sweeping line have been identified correctly" is maintained. Consider first Rule 1: Let e_i and e_{i+1} be edges satisfying the conditions of Rule 1. Without loss of generality let e_i be blue and e_{i+1} be green. Let L be some sweep line corresponding to the current cut and let q_i and q_{i+1} be the intersection points of L with e_i and L with e_{i+1}, respectively. Finally, let p be the intersection of e_i and e_{i+1} (see Figure 4.4.1). We need to show that no green edge can intersect e_i between q_i and p. By the conditions of Rule 1 no green edge intersects L between q_i and q_{i+1}. Similarly no green edge intersects e_{i+1}. Let p' be a point on e_i to the left of, but arbitrarily close to p. The points p' and q_i must lie in the same region of the green subdivision; they can be joined by a path running arbitrarily close along L and e_{i+1} that does not cross any green edges. By convexity the straight line segment joining p' and q_i (which is a subset of e_i) must

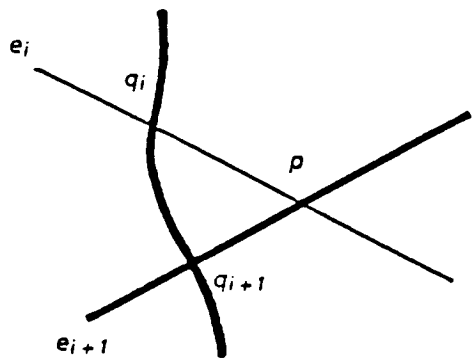


Figure 4.4.1: Correctness of Rule 1

be contained entirely in one green region. Thus no green edge can intersect this segment and hence no green edge can intersect e_i between q_i and p. By the same reasoning no blue edge can intersect e_{i+1} between q_{i+1} and p.

Next consider the correctness of Rule 2. Let v be a blue vertex satisfying the conditions of Rule 2. Let G be the region of the the green subdivision that contains v and let e and e' be the green edges bounding G that lie directly above and below v, respectively. By the safety of v the edges e and e' are either on or totally to the left of the current cut. Thus all bounding edges of G to the left of e and e' are completely to the left of the current cut and therefore, by the sweeping invariant, all intersections of these edges with blue edges have been correctly discovered already. For any left-edge of v let $\bar{e}$ be the last (right-most) intersecting green edge. Clearly $\bar{e}$ must be a bounding edge of G that is not to the right of e or e', which means $\bar{e}$ is either on or to the left of the current cut. If $\bar{e}$ is to the left of the current cut then all intersections with blue edges must have been discovered already because of the arguments above. If $\bar{e}$ is part of the current cut then all intersections with left-edges of v must have been discovered already because of the contiguity assumption for the left-edges of v in the current cut. Thus it is safe to sweep over vertex v and Rule 2 (and hence also the symmetrical Rule 3) is indeed correct.

4.5 Making Vertices Safe

In order to be able to apply Rules 2 and 3 we need to know which vertices are safe. The condition for safety appears to be easy to ascertain. However, it

turns out to be hard to maintain the set of all safe vertices at all times at reasonable cost. We circumvent this problem by maintaining just a subset of the safe vertices. For this purpose we have the following two rules.

Rule 4: In the current cut let e_i and e_k be two blue edges with e_j green for all j, $i<j<k$, and let B be the blue region between e_i and e_k. If the right endpoints of both e_i and e_k lie to the right of the right endpoint r of the edge e_l, for $l=i+1$ or $l=k-1$, then

declare r to be safe;

declare safe all so far unsafe, green vertices to the left of r that can be reached by an x-monotone green edge path from r that stays entirely within the blue region B;

Rule 5: Same as Rule 4, but interchange blue and green.

It should be clear that Rules 4 and 5 are indeed correct in the sense that they do not declare vertices safe which in reality are not. Also note that all the vertices declared safe by (say) Rule 4 can easily be identified by starting a depth-first search through the left edges of r that backtracks whenever e_i or e_k would be crossed, a cut edge e_j, $i<j<k$, is reached, or an already safe vertex is encountered. It remains to be shown that these rules are sufficiently encompassing, that it is impossible that Rules 1 to 3 cannot be applied because no appropriate safe vertex has been discovered via Rules 4 and 5.

So let us assume that Rule 1 cannot be applied (and we are not dealing with the rightmost cut). Call a (non-sentinel) edge e_i in the current cut *essential* iff e_{i-1} or e_{i+1} has a different colour than e_i. Let r be the leftmost right endpoint of the essential edges in the current cut. Without loss of generality

assume r is green. Rule 4 is obviously applicable to r. Let v be the leftmost green vertex that can be reached by an x-monotone edge path as specified in Rule 4. We claim that the left-edges of v form a contiguous subsequence of the current cut and thus Rule 2 is applicable. Since v is leftmost the only way that left edges of v might not lie in the current cut is for them to intersect e_i or e_k (as in Rule 4). But this would contradict the leftmost condition for r (see Figure 4.5.1). Thus all left edges of v lie in the current cut. They have to be contiguous since otherwise Rule 1 would still be applicable, contrary to assumption.

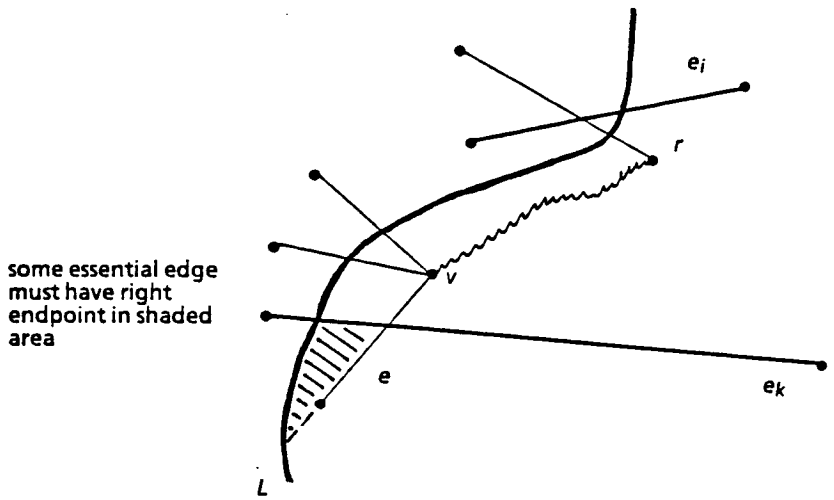


Figure 4.5.1: Applicability of Rule 4

4.6 Putting it together

In order to use Rules 1 to 5 efficiently in an algorithm one needs to maintain the following information: a doubly linked list of the edges in the current cut; for each of these edges a pointer to the next and to the previous edge of the same colour in the cut; a set of pairs of adjacent edges in the current cut for which Rule 1 is applicable; for each blue and each green vertex v a counter for how many of its left-edges are not in the current cut, a second counter tallying in the current cut all occurrences of adjacent edge pairs in which exactly one edge is a left-edge of v, and a flag telling whether v has been declared safe; finally, a set of vertices for which Rules 2/3 are applicable; this set contains all vertices for which the flag is on (i.e. the vertex is safe), and the two counters show 0 (all left-edges are in the current cut) and 2 (they are contiguous), respectively.

With this information the following type of algorithm can be performed efficiently: Apply Rule 1 as long as possible, otherwise Rule 2 or 3. After applying one of these rules update the information outlined above appropriately and check whether Rule 4 or 5 is applicable. If so, apply it immediately. As pointed out already, this can be done by a simple depth-first search.

We leave open further details of such an implementation. For the time analysis of this algorithm note first that with the information outlined above respective applicability of Rules 1–5 can be checked in constant time. An application of Rule 1 takes constant time. An application of Rule 2 or 3 takes

time proportional to the degree of the vertex being swept over. Since an application of Rule 1 yields a vertex in the resulting subdivision **X** of constant degree (i.e. degree 4) we can say that any application of Rule 1 through 3 takes time proportional to the degree of the produced vertex of **X**. The sum of the degrees of the vertices in **X** is $\mathbf{O}(K)$ and thus all applications of Rules 1 to 3 together take time $\mathbf{O}(K)$.

An application of Rule 4 takes time proportional to the number of blue edges traversed by the depth-first search. It is impossible to a give a good bound for this number for an individual application of Rule 4. However, every blue edge can be traversed by such a depth-first search at most once. Thus all applications of Rule 4 together require $\mathbf{O}(m)$ time. By the same argument all applications of Rule 5 together take $\mathbf{O}(n)$ time. Thus we can summarize:

Theorem 4.6.1:

For two convex subdivisions **B** and **G** of size m and n, respectively, the convex subdivision **X** formed by the overlay of **B** and **G** can be constructed via a topological line sweep in optimal time $\mathbf{O}(m+n+K)$, where K is the size of **X**.

It is worth pointing out that if we are only interested in enumerating the elements of the overlay subdivision **X**, then our algorithm can do this using only $\mathbf{O}(m+n)$ space.

4.7 Dealing with Degeneracies

Theorem 4.6.1 might be considered a bit sweeping in that the non-degeneracy assumptions made at the beginning of this chapter are somewhat surreptitiously omitted. In this section we want to show that these assumptions are sufficiently innocuous so that their omission in Theorem 4.6.1 is warranted.

Let us first state the possible degeneracies:

(i) vertical edges

(ii) a blue and a green vertex coincide

(iii) a vertex lies on an edge of opposite colour

(iv) two differently coloured edges overlap in a line segment

One of the standard techniques of dealing with degeneracies is "pertubation:" Conceptually perturb the blue and the green subdivision such that no degeneracy occurs. Compute the overlay of the perturbed subdivisions, and finally remove from the resulting subdivision all degenerate parts that do not exist in the overlay of the unperturbed subdivisions, i.e. 0-length edges, 0-area regions, identical vertices. Although this approach is correct and conceptually simple, we cannot afford it if we want to attain an $\mathbf{O}(m+n+K)$ time algorithm, where K is the true size of the unperturbed overlay. Perturbation can introduce too many extraneous elements in the overlay. Consider for instance a blue vertex coinciding with a green vertex. None of their incident edges intersect in the unperturbed case. However, in the perturbed case suddenly $\mathbf{O}(bg)$ blue-

green edge intersections can occur amongst these edges, where b and g are the degrees of the blue and green vertex, respectively. Moreover, if the incident edges are situated regularly about these two vertices, such an abundance of extraneous intersections arises for *every* possible perturbation.

Since the general perturbation appoach that would collectively deal with all degeneracies proves to be too expensive we have to resort to addressing each type of degeneracy individually.

(i) vertical edges:

Vertical edges do not really constitute a degeneracy but rather an inconvenience for the formulation of the algorithm, which relies on such notions as left endpoint or an edge lying above a region. For every vertical edge e, regard its lower endpoint as its left endpoint and its upper endpoint as its right endpoint. If e bounds a regions R from the left, regard e as on top of R; if e bounds R from the right, take e to be on the bottom of R. These conventions can be viewed as arising from a sufficiently small clockwise rotation of the entire subdivision.

(ii) a blue vertex b and a green vertex g coincide:

The combined vertex z in the resulting subdivision $\mathbf{X}$ can be swept over as soon as the union of the left-edges of b and g form a contiguous sequence of segments in the current cut. There is no need to check for safety. This observation suggests the following strategy: on discovering that b and g coincide (this

occurs during checking for applicability of Rule 1), pair them up and create two counters for the pair analogous to the counters associated with plain vertices: one for the combined number of left-edges of b and g that are not in the current cut, the other one for the number of adjacent edge pairs in the current cut for which exactly one edge is not a left-edge of b or g. When these counters show 0 and 2, respectively, delete duplicate left-edges of the joint vertex z (see (iv)), and sweep over z, i.e. replace the sequence of left-edges of b and g in the current cut by the appropriately merged sequence of right-edges.

(iii) a vertex v lies on an edge e of opposite colour:

On discovering this type of coincidence while checking for applicability of Rule 1, split edge e into two edges by introducing a new vertex w that coincides with v. Thus this type of degeneracy is readily reduced to one of type (ii).

(iv) two differently coloured edges overlap in a line segment s:

By virtue of the resolution of degeneracies of type (iii), the left and right endpoints of s, provided they exist, must be vertices arising from degeneracies of type (ii). Thus it only remains to ensure that s is not recorded twice in the result subdivision. If s has a right endpoint z, this is facilitated by the sweep step over z as specified in the treatment of degeneracies of type (ii). If s does not have a right endpoint, it must occur in the rightmost cut and this duplicate recording of s can be readily removed from that cut and from the result subdivision after termination of the entire sweep.

The reader might ask whether Rules 4/5 need any amendments in order to deal with degeneracies. Strictly speaking this is not necessary. However, for implementation purposes the following should be noted. Using the terminology of Rule 4, the interesting case happens when r is also the right endpoint of e_i or e_k (or both). In this case one has to make sure that the type of depth-first search described earlier for green vertices to be declared safe is started only through green left-edges of r that lie between e_i and e_k. Moreover, this has to be done irrespective of whether r had been declared safe earlier. This concludes our discussion of the handling of degenerate cases.

4.8 Applications

Why would one want to determine the overlay of two planar convex subdivisions? Computing the algebraic sum, also known as Minkowski sum, of two 3-dimensional polytopes was my original motivation; constructing arrangements of lines is another possible application. Since optimal specialized algorithms for the latter problem are known already ([E-O-S],[C-G-L],[E-G]), this section will concentrate exclusively on the former and show how an optimal solution to the planar convex overlay problem naturally yields an optimal solution to the Minkowski sum problem.

For two sets $S,T \subset \mathbf{R}^d$ define their *set sum* $S+T$ to be the set $\{s+t \mid s \in S, t \in T\}$. For two polytopes P and Q their *Minkowski sum* $P+Q$ is simply their set sum, i.e. $P+Q = \{p+q \mid p \in P \text{ and } q \in Q\}$. The Minkowski sum

establish that the width of P with respect to a is equal to the distance between the origin and either of the two tangent planes of the polytope formed by $P-P$ that are orthogonal to a. It follows that the globally largest and smallest widths of P are determined by the vertex of $P-P$ furthest from the origin and the facet of $P-P$ closest to the origin, respectively. Thus these extremal widths can be computed easily as soon as $P-P$ is available.

Let us now attack the problem of computing the Minkowski sum of two 3-polytopes P and Q. Let S denote the set of m vertices of P and T the set of n vertices of Q. We already mentioned that $P+Q$ is nothing but the convex hull of $S+T$. This characterization already yields an algorithm: From S and T form $S+T$ and then construct the convex hull of this set. Using Preparata and Hong's convex hull algorithm [P-H],[E] this method would require $\mathbf{O}(mn\log mn)$ time since $S+T$ can contain up to mn elements. Note that this rather straightforward approach discards almost all information about the facial structure of P and Q. Accordingly, in order to improve upon this algorithm, we shall try to express the facial structure of $P+Q$ in terms of the structure of P and Q. For this purpose let us define the *support set* of face F of a polytope R to be the set of all directions[1] a for which the supporting hyperplane of R with outward normal a intersects R precisely in F. We denote this support set by $A(F,R)$. Equivalently, in more algebraic terms, a is in $A(F,R)$ iff F comprises exactly all $x \in R$ that maximize $<x,a>$.

---

[1]We call a vector a a direction if $a \neq 0$.

Let a be some direction and let F, G, and H be faces of P, Q, and $P+Q$, respectively, such that $a \in A(F,P)$, $a \in A(G,Q)$, and $a \in A(H,P+Q)$. Thus F comprises all $x \in P$ that maximize $<x,a>$, G consists all $y \in Q$ that maximize $<y,a>$, and H is consists of all $z \in P+Q$ that maximize $<z,a>$. Recall that $z \in P+Q$ iff $z=x+y$ with $x \in P$ and $y \in Q$. From the maximality conditions and the additivity of the inner product it therefore follows that $z \in H$ if and only if $z=x+y$ with $x \in F$ and $y \in G$; in other words, $H = F+G$. This reasoning gives rise to the following characterization:

Lemma 4.8.1:

Let F and G be faces of polytopes P and Q, respectively.

(i) If $A(F,P) \cap A(G,Q) \neq \emptyset$, then $F+G$ is a face of $P+Q$ and

$$A(F+G,P+Q) = A(F,P) \cap A(G,Q).$$

(ii) Every face of $P+Q$ arises in such a fashion.

This characterization is based on support sets. In order to exploit it algorithmically we need to know more about the structure of support sets. First note that every direction a is in the support set of exactly one face of a polytope R. (That it cannot be in more than one support set is clear. That a has to be in the support set of some face of R can be rigorously proven using the fact that R must be bounded and closed and hence compact, which implies that $<x,a>$ realizes a maximum for some $x \in R$.) Thus the support sets of the faces of R form a partition of the set of all directions. We call this partition the *support map* of R.

Next note that the support set of a face F of R must be closed under positive combination, i.e. $a,b \in A(F,R)$ imply $\lambda a+\mu b \in A(F,R)$ for any $\lambda,\mu \geq 0$, $\lambda\mu \neq 0$. For if $x \in F$ maximizes $<x,a>$ and $<x,b>$, then x also maximizes $<x,\lambda a+\mu b>$ for positive λ and μ. Therefore $A(F,R)$ forms a convex cone and the support map of R is a partition of space into a finite set of convex cones, one for each face of R.

Since a support set $A(F,R)$ is always a convex cone we can restrict it to directions of unit length. Denote such a restricted support set by $\bar{A}(F,R) = \{a \in A(F,R) \mid |a|=1\}$. The support map of a polytope R can thus be viewed as a "convex" partition $\Sigma(R)$ of the unit sphere. In other words, $\Sigma(R)$ forms a "convex" subdivision of the unit sphere.

What does $\Sigma(R)$ look like when R is a 3-polytope in $\mathbf{R}^3$? If F is a facet, then $\bar{A}(F,R)$ is just a singleton point, the unit length outward normal of F. For an edge e incident to facets F and G the restricted support set $\bar{A}(e,R)$ is the open (shorter) great arc on the unit sphere connecting the points $\bar{A}(F,R)$ and $\bar{A}(G,R)$. If v is a vertex incident to edges $e_1,\cdots,e_k$ and facets $F_1,\cdots,F_k$, then $\bar{A}(v,R)$ is the open spherical "convex polygon" bounded by the arcs $\bar{A}(e_i,R)$ and the corners $\bar{A}(F_i,R)$, $1 \leq i \leq k$. It follows that $\Sigma(R)$ can be viewed as a planar graph; moreover this graph is the geometric dual of the 1-skeleton of the polytope R. Thus $\Sigma(R)$ taken as a planar graph provides a rather explicit encoding of the facial structure of R. Augmented with additional geometric information such as the location of vertex v for each region $\bar{A}(v,R)$

the support map $\Sigma(R)$ provides a complete description of the polytope R.

Our approach to constructing the Minkowski sum of two 3-polytopes should be clear now. First construct the augmented support maps $\Sigma(P)$ and $\Sigma(Q)$. This is straightforward since they are just the geometric duals of the 1-skeletons of P and Q. From these support maps construct the augmented map $\Sigma(P+Q)$ which, in turn, gives a complete description of $P+Q$,

How do we get $\Sigma(P+Q)$ from $\Sigma(P)$ and $\Sigma(Q)$? Lemma 4.8.1 implies that $\Sigma(P+Q)$ consists exactly of all non-empty intersections of some support set in $\Sigma(P)$ with some support set in $\Sigma(Q)$. In other words, the spherical "convex" subdivision $\Sigma(P+Q)$ is the overlay of the spherical "convex" subdivisions $\Sigma(P)$ and $\Sigma(Q)$. The problem of constructing the overlay of two *spherical* "convex" subdivision can of course be easily reduced to two *planar* convex subdivision overlay problems by cutting the unit sphere into two hemispheres and radially projecting each hemisphere onto the plane that is tangent to its pole. Finally note that augmenting $\Sigma(P+Q)$ with the required geometric information is easy. The support region $\bar{A}(v,P+Q)$ of a vertex v of $P+Q$ arises uniquely as $\bar{A}(p,P) \cap \bar{A}(q,Q)$ for a vertex p of P and a vertex q of Q. Moreover, by Lemma 4.8.1, the vertex v must then be $p+q$. Thus we can summarize:

Theorem 4.8.1:

The Minkowski sum of two 3-polytopes P and Q can be constructed in time $\mathbf{O}(m+n+K)$, where m, n, and K are the number of vertices of P, Q, and $P+Q$, respectively.

5. Constructing Higher-Dimensional Convex Hulls

5.1. Introduction

The convex hull problem, finding the smallest convex set containing a given set of m points in $\mathbf{R}^d$, has been a central problem in computational geometry. It is a basic and intuitive problem, it frequently appears as subproblem in the solution of other geometric problems, and a number of important geometric problems, such as intersecting halfspaces or constructing Voronoi diagrams, have been shown to be just convex hull problems in disguise.

The complexity of the problem is well understood for dimension $d \leq 3$. For $d=1$ the problem amounts to finding the maximum and minimum of a set of m reals and it is well known that $\lceil 3m/2 - 2 \rceil$ comparisons are sufficient and necessary for this task. The case $d=2$, finding the smallest convex polygon containing a planar point set, has been the topic of many a paper in computational geometry. The bottom line is that $\Theta(m \log m)$ time is necessary and sufficient to solve this problem. Graham [Gra] was the first to show sufficiency, followed by many others using different algorithms, and necessity was proven much later by Yao [Y] and Ben-Or [BO] for realistic models of computation. Furthermore, as shown in chapter 2 of this thesis, if the complexity of the problem is measured in terms of input size m and output size H (the number of vertices of the produced polygon), then $\Theta(m \log H)$ is necessary and sufficient.

For the case $d=3$, finding the smallest convex polyhedron containing a given point set, Ben-Or's $\Omega(m \log m)$ lower bound carries over. In contrast to the

2-dimensional case, only one algorithm, a clever divide-and-conquer technique by Preparata and Hong [P-H] (see also [E]), is known that shows that $\Theta(m \log m)$ time is also sufficient.

For $d \geq 4$ the whole situation changes considerably. The main reason is that the number faces (or facets) of a d-polytope with m vertices can range anywhere between $\Omega(m)$ and $\mathbf{O}(m^{\lfloor d/2 \rfloor})$. Starting with $d=4$ this gap is non-trivial and it becomes desirable to express the complexity of the convex hull problem in terms of the output size as well.

Essentially only two types of algorithms for constructing convex hulls in $\mathbf{R}^d$, $d \geq 4$ are known. One approach, dubbed the "beneath-beyond method" in Preparata's and Shamos's book [P-S], was proposed independently by Kalai [Kal], Seidel [S81], and Rey and Ward [R-W]. The running time of these algorithms can only be measured in terms of m, and for fixed dimension d the more refined algorithm in [S81] has worst case running time $\mathbf{O}(m^{\lfloor (d+1)/2 \rfloor})$. If the complexity of the convex hull problem is measured only in terms of m, then this algorithm is worst case optimal for even $d \geq 4$. However, if the problem complexity is also measured in terms of the actual output size, then the algorithm can be far from optimal, as shown by Swart [Sw]. The second approach is generally known as the "gift-wrapping method." It was first proposed in [C-K], and later refined and analyzed by Swart and Bhattacharya [Sw],[B]. Here two versions of the convex hull problem need to be distinguished. One just asks to enumerate all facets of the convex hull of m points, where it is assumed that no

degeneracies exist. The other version asks to produce the facial graph of the convex hull, i.e. all the faces along with a minimal description of their inclusion relationships. Swart [Sw] gave an implementation of the gift-wrapping approach that solved the facet enumeration problem in worst case time

$$\mathbf{O}(dFm + d^4F\log m)\ , \quad \text{where } F \text{ is the number of facets.}$$

For the facial graph problem he presented an algorithm with worst case time complexity

$$\mathbf{O}(\ d^2K_1m + d^3K_2\log K_0)\ .$$

Here K_i are parameters of the size of the produced facial graph: K_0 is the number of nodes in this graph, K_1 the number of arcs, and $2K_2$ the number of paths of length two.

In this chapter we outline a new approach for solving the convex hull problem. The new method is based on the notion of "shelling" a polytope. This approach yields an algorithm for the facet enumeration problem whose worst case time complexity is

$$\mathbf{O}(\ m\,\lambda(d-1,m-1) + d^4F\log m\ ).$$

For the facial lattice problem our approach yields an algorithm whose worst case running time is bounded by

$$\mathbf{O}(\ dm\,\lambda(d-1,m-1) + K_2(d^2+\log K_0)\ ).$$

In these bounds $\lambda(d,m)$ denotes the time necessary to solve a linear program with at most m constraints in at most d variables. For fixed dimension d the bound $\lambda(d,m)$ has been shown to be $\mathbf{O}(m)$. Thus for fixed d our new algorithms

yield better bounds than the gift-wrapping method as long as the number of faces of the constructed polytope is $\Omega(m)$. Moreover, if the complexity of the problem is only measured in terms of the input size m, our algorithms have worst case time $\mathbf{O}(m^{\lfloor d/2 \rfloor} \log m)$, which is the best bound known for odd $d > 3$.

It should also be noted here that the dual form of the facet enumeration problem, namely enumerating the vertices of a linearly constrained feasibility set, has been dealt with extensively in the Operations Research literature. Mattheiss and Rubin [M-R] and also Dyer [D83] survey the different solutions to this problem. Dyer also analyzes the complexity of the different approaches. He shows that of the currently known methods only the so-called "pivoting methods" yield algorithms whose time complexity can reasonably be related to the output size. It turns out that "pivoting" is exactly the dual notion to "gift-wrapping", and the time bound derived by Dyer for his implementation of a pivoting method agrees with the bound of Swart's implementation of the gift-wrapping method.

By duality our approach can be used to solve the vertex enumeration problem also. For fixed dimension d the worst case time will be $\mathbf{O}(v \log m)$ plus the time necessary to solve m linear programs, each with $m-1$ constraints in $d-1$ variables. where m denotes the number of constraints and v the number of enumerated vertices. This is, at least in the asymptotic sense, an improvement over the currently known vertex enumeration algorithms.

5.2. Polytopes

In this section we first give a short synopsis of relevant definitions and results in the theory of convex polytopes. For more detail the reader is referred to [M-S] or [G]. We conclude the section with precise statements of the problems this paper wants to solve.

The following discussion is about point sets in $\mathbf{R}^d$. We will denote the usual inner product between points $x,y \in \mathbf{R}^d$ by $<x,y>$. For non-zero $a \in \mathbf{R}^d$ and arbitrary $c \in \mathbf{R}$ the set $H = \{x \mid <x,a> = c\}$ is called a *hyperplane* and the set $H = \{x \mid <x,a> \leq c\}$ is called a *halfspace* (bounded by H). For a set S in $\mathbf{R}^d$ the *convex hull* of S, for short conv S, is defined to be the intersection of all halfspaces that contain S. Equivalently, conv S can be defined as the set of all convex combinations of members of S, i.e. the set of all points that are representable as $\sum_{p \in S} \lambda_p p$ with $\sum_{p \in S} \lambda_p = 1$, and $0 \leq \lambda_p \leq 1$ for all $p \in S$. The convex hull of a finite set is called a *polytope*. A polytope P is called a *k-polytope*, if k is the dimension of the smallest affine subspace of $\mathbf{R}^d$ that contains P. A hyperplane $H = \{x \mid <x,a> = c\}$ is said to *support* a polytope P iff $<x,a> \leq c$ for all $x \in P$ and $<x,a> = c$ for some $x \in P$. The intersection of a polytope P and a supporting hyperplane is called a *face* of P. Every face of P is a polytope. A face F of P is called a *k-face* if it is a k-polytope. For a k-polytope P the $(k-1)$-faces are called *facets*, the $(k-2)$-faces are called *ridges*, and the 0-faces are called *vertices*. The empty set is considered a (-1)-face of P and P itself is considered a k-face of P. The relation "being a face of" is

transitive. Every face F of P is the intersection of the facets of P that contain F. It is also the convex hull of the vertices of P contained in F. The faces of P under inclusion form a complete lattice which is called the *facial lattice*.

A *k-simplex* is a k-polytope formed by the convex hull of $k+1$ points in $\mathbf{R}^d$, $d \geq k$. A k-simplex has $k+1$ facets and $\begin{bmatrix} k+1 \\ 2 \end{bmatrix}$ ridges. A polytope P is called *simplicial* if every face of P (except for P) is a simplex. If P is the convex hull of a set S in non-degenerate position (i.e. no $d+1$ points in S lie on a common hyperplane), then P is simplicial.

The *facial graph* of a polytope is an acyclic directed graph with one source and one sink. The nodes in this graph are the faces of P. An arc connects faces F and G iff G is a facet of F. The facial graph is the Hasse diagram of the facial lattice of P.

For a polytope P let $\varphi(P)$ denote the number its facets and $\rho(P)$ the number of its ridges. Define the three quantites

$$K_0(P) = \sum_{F \text{ a face of } P} 1$$

$$K_1(P) = \sum_{F \text{ a face of } P} \varphi(F)$$

$$K_2(P) = \sum_{F \text{ a face of } P} \rho(F) \ .$$

K_0 denotes the number of faces of P, or equivalently, the number of nodes in the facial graph of P. K_1 is equal to the number of arcs in the facial graph, and, since every ridge is in exactly two facets (see Fact 5.2.1), $2K_2$ is the

number of directed paths in the facial graph with length 2. It is known that for a d-polytope P with m vertices, $K_i(P)$ is in $\mathbf{O}(m^{\lfloor d/2 \rfloor})$ and in $\Omega(m)$ for $i=0,1,2$. This follows immediately from the upper bound theorem [M-S] for $i=0,1$,

Two versions of the convex hull problem are considered in this chapter:

The facial graph problem:

Given a set S of m points in $\mathbf{R}^d$, construct the facial graph of the polytope formed by conv S.

The facet enumeration problem:

Given a set S of $m>d$ points in $\mathbf{R}^d$ in non-degenerate position, enumerate all the facets of conv S.

In the technical discussion of our algorithms we will need the following facts about polytopes. For proofs see [M-S].

Fact 5.2.1: Every ridge of a polytope is contained in exactly two facets.

Fact 5.2.2: Let P be a d-polytope with $b \in \operatorname{int} P$. For every facet F of P let a_F be the vector such that $<a_F, b-x> = 1$ for all $x \in F$ and $<a_F, b-x> \leq 1$ for all $x \in P$. Let G be some face of P and let Φ be the facets of P that contain G. $H = \{x \mid <a, b-x> = 1\}$ is a supporting hyperplane of P that contains G iff $a = \sum_{F \in \Phi} \lambda_F a_F$ with $0 \leq \lambda_F \leq 1$ for all F and $\sum_{F \in \Phi} \lambda_F = 1$.

We say a point p is *beyond* a facet F of a d-polytope P iff the hyperplane that contains F separates p and P. Point p is said to be *beneath* F otherwise.

Fact 5.2.3: Let P be a d-polytope in $\mathbf{R}^d$ and let p be a point not in P that is not contained in the affine hull of any facet of P. Let $P' = \text{conv}(P \cup \{p\})$.
G' is a face of P' that contains p if and only if G' is of the form $\text{conv}(G \cup \{p\})$, where G is a face of P that is contained in some facet of P for which p is beneath and in some facet of P for which p is beyond.

Fact 5.2.4: Let v be a vertex of a d-polytope P, let H be a hyperplane that separates v from the remaining vertices of P, and let $P' = P \cap H$. The lattice of the faces of P that contain v is isomorphic to the facial lattice of P'.

5.3. Shelling a Polytope

A *shelling* of a d-polytope P is an ordering of the facets of P, say $F_1, \ldots, F_n$ ($n = f_{d-1}(P)$) so that for each i, $1 < i < n$,

$$F_i \cap \left[\bigcup_{j<i} F_j \right]$$

is the union of the first $k < f_{d-2}(F_i)$ facets of F_i in a shelling of F_i.
Any ordering of the two facets of a 1-polytope is a shelling.

Thus a shelling is a very controlled enumeration of the facets of P and through the recursiveness of its definition it induces a controlled ordering on all the faces of P. If the reader feels uncomfortable with this recursive definition,

the following property of a shelling might be more appealing to intuition: for each i, $1<i<n$,

$$\bigcup_{j<i} F_i$$

is a topological $(d-1)$-ball.

The reader may convince himself that an ordering $e_1, \ldots, e_n$ of the facets (i.e. edges) of a convex $2-$polytope (i.e. polygon) is a shelling iff for $1<i<n$ the union $\bigcup_{j\leq i} e_j$ forms one contiguous chain of edges. For $3-$polytopes an ordering of facets is a shelling if for $1<i<n$ the intersection of F_i and $\bigcup_{j<i} F_j$ is a contiguous chain of edges.

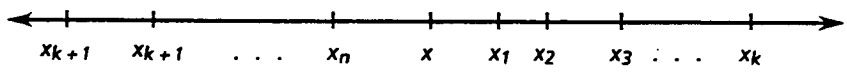
It is not hard to show that for every $d>0$ any ordering of the facets of a $d-$simplex is a shelling.

It is not at all clear, though, that every $d-$polytope must have a shelling. A construction due to Bruggesser and Mani [B-M] proves the shellability of all polytopes. The intuition behind that construction is as follows.

Imagine an observer travelling along a directed straight line L that lies in "suitable general position" and intersects the interior of a $d-$polytope P. He starts his journey at some point of L in the interior of P and moves in the direction of L. At some point the observer passes through some facet F and thus leaves the interior of P. This facet F is the first in our shelling order. Just beyond F, all the observer can see of P when looking back is F. But as he con-

tinues moving away from P along L more and more facets will become visible. The order in which they appear constitutes the first part of our shelling order. After some point no more facets will become visible as the observer moves along L toward infinity. The observer then jumps back (or "moves through infinity", if you will) to the "beginning" of L. He now sees exactly all the facets of P he could not see before. Still moving along L, but now towards P, facets will become invisible. The order in which they disappear constitutes the second part of our shelling order.

The mathematical formulation of this construction is as follows: Let L_d be any directed straight line intersecting P's interior that is *admissible* for P. This means that L_d intersects the affine hull of each facet of P and that all these intersection points are distinct. Number these intersection points x_i along L_d as shown below, where x is some point on L_d in the interior of P.



Then this ordering of the corresponding facets of P is a shelling, which we call a *straight line shelling* generated by the line L_d.

The proof for this fact is constructive. We will give the idea of the construction because we exploit it in our algorithm.

The proof is by induction on d. For every i, $1<i<n$, one needs to exhibit a shelling of the $(d-1)-$polytope F_i such that $F_i \cap \bigcup_{j<i} F_j$ is the union of the first

k_i $(d-2)$-faces in this shelling of F_i. Of course we want this shelling of F_i to be a straight line shelling. Thus we have to exhibit an apporpriate shelling line L_{d-1} for F_i. It turns out that we can choose *any* line that contains x_i (the intersection point of L_d and the affine hull of F_i), intersects the relative interior of F_i, is admissible for F_i, and is directed such that the relative position of x_i and F_i along L_{d-1} agrees with the relative position of x_i and P along L_d. The reader is referred to [B-M] for details.

5.4. The Facet Enumeration Problem

In this section we present an efficient algorithm for the following facet enumeration problem: We are given a set S of $m > d$ points in $\mathbf{R}^d$ in non-degenerate position, i.e. no $d+1$ points of S lie in a common hyperplane. We want to enumerate all facets of $P = \operatorname{conv} S$.

Because of the non-degeneracy assumption P must be a simplicial d-polytope. Thus all facets of P must be $(d-1)$-simplices. Enumerating all facets then means reporting all d-tuples of points in S that span a facet of P.

We want to enumerate the facets in a straight line shelling order. Let b be some point in the interior of P and let a be some non-zero vector in $\mathbf{R}^d$. We assume without loss of generality that the line L through b that is spanned by a is an admissible shelling line for P. This means that for no two facets of P do the two hyperplanes containing them intersect L in the same point. Moreover, no facet is parallel to L. This assumption is justified since admissability of the

shelling line can be simulated using standard perturbation techniques (see section 5.6).

Mimicking the journey of the "travelling observer" of the previous section and for reasons that will soon become clear, we parameterize the shelling line L by $x(t) = b - \frac{1}{t}a$, with $-\infty < t < \infty$. We will ignore the singularity $t = 0$.

Let $F_1, \ldots, F_n$ be the facets of P in shelling order according to L. For $1 \le i \le n = f_{d-1}(P)$ let $x(t_i)$ be the intersection point of L and the hyperplane containing F_i. By virtue of our parameterization we have $t_i < t_j$ for $i < j$. Of course this ordering is not known in advance. After all, just given the point set S we do not even know the facets of $P = \operatorname{conv} S$. But it is possible to discover all the facets in this order in an efficient manner. For this purpose the notion of horizon is important.

For $t \in R$ call a face G of P a *horizon face* at (time) t iff G is contained in two facets F_i and F_j of P with $t_i < t$ and $t_j \ge t$. *Horizon ridges* and *horizon peaks* are horizon $(d-2)$- and $(d-3)$-faces, respectively. We call the set of all horizon faces at t simply the *horizon* at t. The horizon faces satisfy a number of simple but important properties.

Lemma 5.4.1:

Let G be a ridge of P, and let F_i and F_j be the two facets containing G, with $i<j$.

(i) G is a horizon ridge for t exactly when t is in the interval $(t_i,t_j]$.

(ii) If G is a horizon ridge for t then the hyperplane spanned by G and $x(t)$ supports P.

Proof:

(i) is trivial. For (ii) let n_i be the normal vector for the hyperplane containing F_i such that $<n_i, b-x> = 1$ for all $x \in F_i$ and $<n_i, b-x> < 1$ for all $x \in P$. Let n_j be the analogous normal vector for F_j. It is easy to check that now $t_i = <n_i, a>$ and $t_j = <n_j, a>$. Let t be between t_i and t_j. Then t can be represented as $t_s = st_i + (1-s)t_j$, with $0 \le s < 1$. It is easy to check that the unique hyperplane H_s spanned by $x(t_s)$ and G can be represented as

$$H_s = \{x \mid <sn_i + (1-s)n_j, b-x> = 1\}.$$

Fact 5.2.2 now implies our claim.

Q.E.D.

Lemma 5.4.2:

For $t_1 < t < t_n$ the horizon for t is isomorphic to a $(d-1)$-polytope.

Proof: (outline)

Since the horizon remains invariant over intervals $t_i < t < t_{i+1}$ we assume w.l.o.g. that $t \neq t_i$ for $1 < i < n$ and also $t \neq 0$.

Consider the d-polytope $P' = \text{conv}(P \cup \{x(t)\})$. As a consequence of Fact 5.2.3 there is a 1-1 correspondence between the horizon faces of P at t and the faces of P' that contain the new vertex $x(t)$. By Fact 5.2.4 in turn, the lattice of faces of P' that contain $x(t)$ is isomorphic to the face lattice of the $(d-1)$-polytope $H \cap P'$, where H is a hyperplane that separates $x(t)$ from P.

Q.E.D.

The last lemma has some important consequences. By Fact 5.2.1, if g is a horizon peak then it is contained in (and the intersection of) exactly two horizon ridges, say G_1 and G_2. Together these three faces span a unique hyperplane H_g. This can easily be seen from the dimension formula

$$\dim(G_1 \bigcup G_2) = \dim G_1 + \dim G_2 - \dim(G_1 \bigcap G_2)$$

when taking into account that $G_1 \bigcap G_2 = g$. Let t_g be the real such that H_g intersects the shelling line L in $x(t_g)$.

Let us assume inductively that we have found the first $i-1$ facets in the shelling order and that we know the horizon for t_{i-1}. We want to find F_i. We do this by maintaining a set $\mathbf{H}_i$ of hyperplanes one of which must contain F_i. These hyperplanes are naturally ordered by their intersection points with L. The one with the "least" intersection point will be the hyperplane that contains F_i.

By the properties of a shelling the intersection of F_i and $\bigcup_{j<i} F_j$ must be the union of one or more horizon ridges. We distinguish two cases:

1. This intersection consists of the union of more than one horizon ridge.

   By the properties of a shelling two of these horizon ridges must intersect in a horizon peak g. It follows immediately that F_i must be contained in H_g. Thus for every horizon peak g we include H_g in $\mathbf{H}_i$.

2. The intersection consists of exactly one horizon ridge G.

   In this case we cannot expect to be able to deduce F_i from the structure of the horizon. However, F_i must contain a vertex $p \in S$ that is not contained in any horizon ridge. Moreover, p is not contained in any F_j with $j<i$. In other words, F_i is the first facet in the shelling that contains p. Let us assume that somehow we can determine for every $p \in S$ whether p is a vertex of P and, if it is, we can determine the first facet in the shelling that contains p. Let H_p be the hyperplane that contains that facet and let $x(t_p)$ be the intersection of H_p and L. We include all such H_p's in $\mathbf{H}_i$.

Lemma 5.4.3:

Let $\mathbf{H}_i$ be as defined above but remove all hyperplanes H_ξ with $t_\xi \leq t_{i-1}$ (where ξ stands for a horizon ridge g or a vertex p).

The hyperplane H_ξ in $\mathbf{H}_i$ with the smallest t_ξ contains the next facet F_i.

Proof: By construction the hyperplane H that contains F_i is contained in $\mathbf{H}_i$. Every H_ξ in $\mathbf{H}_i$ that supports P contains some facet F_j with $j \geq i$. For such supporting hyperplanes we therefore have $t_\xi = t_j \geq t_i$. We are done if we show that for every hyperplane H_ξ in $\mathbf{H}_i$ that does not support P we have $t_\xi > t_j$ for some $j \geq i$.

Let G be a horizon ridge that is contained in H_ξ and let F_k and F_j be the two facets containing G such that $k<j$. Because G is a horizon ridge we have $k<i$ and $j \geq i$. Lemma 5.4.1 now implies that $t_\xi < t_k$ or $t_\xi > t_j$. The case $t_\xi < t_k$ cannot occur since $t_k \leq t_{i-1}$ and all H_ξ with $t_\xi \leq t_{i-1}$ were removed from $\mathbf{H}_i$. In the other case we have $t_\xi > t_j \geq t_i$ as desired.

Q.E.D.

It is now not too difficult to specify an incremental algorithm that enumerates the facets of P in shelling order. In essence it only needs to keep track of the horizon and the set $\mathbf{H}_i$. There is only one major problem left. We assumed we can determine for every $p \in S$ whether is a vertex of P and, if it is, we can determine the first facet in the shelling that contains p. We now show that this can be indeed be done, without knowing P, via linear programming.

If p is a vertex of P then there must be a supporting hyperplane H of P that contains p. We want to find the supporting hyperplane H_p that intersects the shelling line L in point $x(t_p)$ such that t_p is minimal. The hyperplane H_p certainly does not contain b, which lies in the interior of P. Thus H_p consists of points $z \in \mathbf{R}^d$ that satisfy $< n , b-z > = 1$, where n is some non-zero vector that is to be determined. Since p is to lie in H_p the equation $< n , b-p > = 1$ must hold. Now H supports P iff the closed halfspace bounded by H that contains b also contains all points of S. Thus the inequality $< n , b-q > \leq 1$ must hold for each $q \in S$. The intersection point $x(t_p) = b - \frac{1}{t}a$ of H_p and the shelling line L must satisfy $< n , b-x(t_p) > = 1$. This implies $t_p = < n , a >$. Our

goal is to choose the vector n such that t_p is minimized. Thus we obtain the following linear program:

$$\begin{aligned} &\text{minimize } t_p = \langle n, a \rangle \text{ such that} \\ &\qquad \langle n, b-p \rangle = 1 \\ &\qquad \langle n, b-q \rangle \leq 1 \qquad \text{for all } q \in S,\ q \neq p \end{aligned}$$

This linear program can easily be transformed into an equivalent one with $m-1$ constraints and $d-1$ variables.

Let $H_p = \{z \mid \langle n, b-z \rangle = 1\}$ be the hyperplane resulting from the solution of the above linear program. It is clear that H_p must be a supporting hyperplane of P containing p. If the optimum solution of the linear program is chosen as a basic feasible solution, then $d-1$ of the constraints must be tight, i.e. $d-1$ points of S must lie on H_p besides p. From the non-degeneracy assumptions made at the beginning of this section it follows that this basic feasible solution is unique. Moreover, these d points on H_p must span a facet F of P. Clearly F must be the first facet in the shelling order that contains p, which is what we wanted to show.

We now have everything ready to outline an algorithm for the facet enumeration problem. We have a set S of $m > d$ points in $\mathbf{R}^d$ in general position. Let b be a point in the interior of $\operatorname{conv} S$ (e.q. $b = \frac{1}{m} \sum_{p \in S} p$). Choose some non-zero vector $a \in \mathbf{R}^d$ and assume that the line through b spanned by a is an admissable shelling line for $\operatorname{conv} S$.

Because of the non-degeneracy assumption all the faces of the convex hull are simplices. We therefore identify each k-face X with the set of its $k+1$ vertices. In our algorithm we need to keep track of the horizon ridges and the horizon peaks. They are most naturally stored as a $(d-1)$-regular graph with the ridges as nodes and the peaks as edges. An edge is incident to a node iff the corresponding peak is contained in the corresponding ridge (recall Lemma 5.4.2 and Fact 5.2.1). We call this graph the *horizon graph*.

Besides the horizon graph we also need to maintain a priority queue containing the t_p and t_g values explained in the beginning of this section. Finally, we need a dictionary of $(d-1)$-sets in S. Its use is explained below.

Algorithm 5.4.1:

1. For each point $p \in S$ solve a linear program as outlined above. If the program is infeasible then eliminate this point from S. It cannot be a vertex of $\operatorname{conv} S$. If it is feasible let t_p be the optimal solution (it is impossible that this LP is unbounded), and let $Q_p \subset S$ be the set of $d-1$ points that provide the tight constraints of the optimal basic feasible solution. The convex hull of Q_p will be the horizon ridge that is contained in the first facet of the shelling that contains p.

   Enter all these t_p-values in a priority queue and enter all the Q_p's in the dictionary. This dictionary will be used to locate the ridge formed by a Q_p in the horizon graph.

2. Let t be the minimum value in the priority queue. Because of our non-degeneracy assumption there will be a set T of exactly d points $p \in S$ for which $t_p = t$.

   Report T as the first facet.

   Build the horizon graph. It is a complete graph on d nodes having a node for every $(d-1)$-subset $T_p = T - \{p\}$ of T and an edge for every $(d-2)$-subset of T. Note that $T_p = Q_p$. Create links from the dictionary entries Q_p to the corresponding nodes in the horizon graph.

   Delete the t_p values from the priority queue for all $p \in T$.

3. Repeat the following until the priority queue is empty:

   Let t be the minimum value in the priority queue.

   Case 1: t is realized by a t_p value for some $p \in S$:

      Report $Q_p \cup \{p\}$ as the next facet in the shelling.

      Change the horizon graph so as to reflect the new horizon after the addition of the new facet in the shelling. This, in essence, involves finding the node in the horizon graph that corresponds to Q_p using the dictionary, and then replacing this node by a complete graph on $d-1$ nodes, one for each $(d-1)$-set in $Q_p \cup \{p\}$ with the exception of Q_p itself.

      Delete t_p from the priority queue.

Case 2: t is realized by t_g for a set Γ of horizon peaks g.

For some $g \in \Gamma$ let G_1 and G_2 be the two horizon ridges that contain g.

Report the set of vertices of $G_1 \cup G_2$ as the next facet in the shelling.

The peaks $g \in \Gamma$ are the edges of a k-clique K in the horizon graph, $1<k<d$. Replace this clique by a $(d-k)$-clique, whose nodes are all $(d-1)$-sets in F that do not appear as nodes in K.

During the updates of the horizon graph outlined above remove from the priority queue all t_g values of deleted horizon peaks g. Similarly, update or insert the t_g values of newly created or changed horizon peaks g, provided $t_g>t$. Finally, for every new horizon ridge created, check whether it occurs in the dictionary. If it does, create a link from the dictionary to the corresponding node in the horizon graph.

4. When the priority queue is empty, the horizon graph will be a d-clique K. Report $\bigcup\{G|G \text{ a node in } K\}$ as the last facet in the shelling.

Every linear program in step 1 can be transformed into one with $m-1$ constraints in $d-1$ variables. Thus step 1 takes time $\mathbf{O}(m\lambda(m-1,d-1))$. Whenever a facet is reported in steps 2 to 4 at most $\mathbf{O}(d^2)$ elements of the horizon graph are changed and as many lookups and updates are performed in the priority queue and the dictionary. For each horizon peak that is updated in the priority queue, the hyperplane that contains it along with its two incident horizon ridges has to be computed. This can be done in $\mathbf{O}(d)$ time. See the

detailed general algorithm in the next section. The size of the dictionary is $\mathbf{O}(m)$, and the size of the priority queue cannot exceed $\mathbf{O}(m^{\lfloor d/2 \rfloor})$ (since conv S can have as most this many ridges). Therefore steps 2 to 4 together take time $\mathbf{O}(d^4 F \log m)$, where F is the number of facets reported. Thus we have the following theorem:

Theorem 5.4.1:

For a set S of $m > d$ points in $\mathbf{R}^d$ in non-degenerate position Algorithm 5.4.1 enumerates the F facets of conv S in time

$\mathbf{O}(m\lambda(m-1,d-1) + d^4 F \log m)$.

The linear-time linear programming algorithms for fixed dimension d [M84],[D86],[C] imply the following corollary:

Corollary 5.4.2:

For fixed d the F facets of the convex hull of a set of m points in non-degenerate position in $\mathbf{R}^d$ can be enumerated in time $\mathbf{O}(m^2 + F \log m)$.

Finally, by the upper bound theorem $F = \mathbf{O}(m^{\lfloor d/2 \rfloor})$, which implies the following result:

Corollary 5.4.3:

For fixed $d > 3$ the facets of the convex hull of a set of m points in non-degenerate position in $\mathbf{R}^d$ can be enumerated in time $\mathbf{O}(m^{\lfloor d/2 \rfloor} \log m)$.

5.5. Constructing the Facial Graph

In this section we show how the shelling approach can be used to construct the facial graph of the convex hull of a set S of m points in $\mathbf{R}^d$. The points in S need not be in non-degenerate position. Thus the proper faces of conv$S$ need not be simplices, but they can be arbitrary polytopes. This generality caused problems for previous algorithms that constructed the facial graph facet by facet. Typically such an algorithm would proceed roughly as follows (see [Sw]): somehow determine which points of S lie in a common facet; construct the facet by applying the convex hull algorithm recursively (one dimension down) on these points; combine the facial graph of the new facet with the facial graph of the previously obtained facets; somehow find a not yet computed facet and repeat.

Obvious problems arise in such an algorithm. As lower dimensional faces are shared by many faces, they will be discovered and recomputed many times redundantly unless a dynamic programming approach is taken and computed faces are stored and can be looked up in a special data structure. It is not clear how the next facet to be computed is determined; in particular, if there are several choices, it is not clear which facet would be best to choose next. Computing the facets in a straight line shelling order solves all these problems at once.

In the previous section we showed how one can predict the next facet in a straight line shelling by keeping track of the horizon and doing a certain amount of preprocessing in the form of linear programming. Thus choice and determination of the next facet to be computed are not a problem any more. The other problem, which faces of the new facet F have already been computed, is rendered vacuous by the very properties of a shelling: these previously computed faces are exactly the horizon faces that are contained in F. Moreover, these faces form the initial segment of a straight line shelling of F. Thus, applying our convex hull algorithm recursively to F, we do not need to recompute this initial segment of the shelling of F, but we can pick up the construction at the appropriate point in the middle of the shelling.

Before describing such a recursive procedure in any more detail we need to discuss some representational issues and data structure considerations. We are to construct the facial graph of the polytope P that constitutes the convex hull of a set S of m points in $\mathbf{R}^d$. Recall that the nodes of this directed graph are the faces of P and that it has an arc from node F to node G iff F is a facet of G. This graph is acyclic and it has one source, the empty face, and one sink, the polytope P itself. We shall augment this graph with arc labels: an arc from F to G will be labeled by some vertex of G that is not a vertex of F. In the data structure that encodes this augmented facial graph we need space at each node F to store the following information:

n_F $\cdots$ a vector in $\mathbf{R}^d$;

super(F) $\cdots$ a pointer to a superface of F (i.e. a face that F is a facet of);

neighbors(F) $\cdots$ a set of up to two pointers to superfaces of F;

a pointer into a priority queue (to be described below);

various mark bits.

We will not spell out the exact implementation of such a data structure in any more detail. However, we will assume that, besides accessing the listed information at a node in constant time, the following operations can be performed within in the given time bounds:

- creating a new node in constant time
- introducing an arc between two given nodes in constant time
- enumerating the set of facets of a given face F at constant cost per facet
- enumerating the set of superfaces of F at constant cost per superface

As a second data structure our algorithm requires a priority queue (PQ) for mimicking the journey of the travelling observer along the shelling line. As in the algorithm of the previous section there are two types of entries into PQ. The first is of the form (t,n,p,T), where the key t is a real number, n is a vector in $\mathbf{R}^d$, p is a point of S, and T is a subset of S. The intuitive meaning of such an entry can be given roughly as follows: At time t the first facet becomes "visible" that contains the point p; this facet will be the convex hull of T and has n as normal vector.

The second type of entry is of the form (t,n,G), where t and n are as above, and G is a face. The meaning of this entry is approximately: at time t a facet becomes "visible" that contains the horizon peak G, is normal to n and is also contains the two horizon ridges $neighbors(G)$. We assume that this type of priority queue is implemented such that insertions and deletions can be performed at logarithmic cost. Performing a DELETEMIN operation in PQ should return a quintuple (t_0,n,HG,U,T), where t_0 is the smallest key in PQ, HG is the union of all G with (t_0,n_G,G) in PQ, and U is the union of all p with (t_0,n_p,p,T) in PQ. T will be the same for all such p. Moreover, if U is empty, then T is given as empty as well. Finally, all vectors in $\{n_\xi \mid \xi \in U \cup HG\}$ will be multiples of each other. Any one of them can be used for n. The cost of such a DELETEMIN operation is assumed to be $\mathbf{O}((|HG| + |U|)\log M)$, where M is the size of PQ.

We will now describe our recursive shelling procedure SHELL for constructing or completing the facial graph of Q, the polytope formed by the convex hull of a set T of points in $\mathbf{R}^d$. Formally, SHELL is a function that returns as its value the node in the facial graph that represents Q. SHELL takes the following input parameters:

(a,b) : a pair of vectors describing the shelling line L: $x(t) = b - \frac{1}{t}a$, where $b \in \text{relint}\, Q$ and $a \neq 0$;

t_0 : a real number describing how far the shelling has progressed along L;

FF : the set of facets of Q that are known already and form the initial segment of facets in the shelling of Q along L, up to and including time t_0;

HR : the set of horizon ridges of Q at time t_0;

U : the set of points in T that do not lie in any facet in **FF**;

T : the set of points whose convex hull is Q; this parameter is only needed when $U \neq \emptyset$; in case $U = \emptyset$, T will be given as empty as well;

N : a maximal set of independent vectors that span the orthogonal complement of the affine hull of Q;

Before the procedure SHELL can be invoked the first time to construct the convex hull of a set S of m points in $\mathbf{R}^d$ the following computations need to be performed: a maximal set A of independent vectors spanning the orthogonal complement of the affine hull of S has to be found. Using standard linear algebra procedures this can be done in time $\mathbf{O}(d^2m)$. A point x in the relative interior of conv$S$ needs to be computed, e.g. $x = \frac{1}{m} \sum_{p \in S} p$, and some (if at all possible) non-zero vector y that is normal to all $n \in N$ has to be found. After this preprocessing SHELL is called with parameters

$(a,b) = (y,x); \quad t_0 = -\infty;$

FF = **HR** = $\emptyset$;

$U = T = S;$

$N = A.$

First we give a short outline of the procedure SHELL:

1. Create a new node for Q and create arcs between all facets in **FF** and Q;
2. Bottom out the recursion in case you are dealing with a 0-dimensional polytope;
3. Set up the horizon, i.e. find the horizon peaks;
4. set up PQ, the priority queue, which tells in what order to generate the new facets of Q;
5. Generate the new facets of Q; i.e.

   while PQ $\neq \varnothing$ do

   5.0. DELETEMIN

   5.1. prepare to shell next facet F of Q;

   5.2. call SHELL recursively to generate F;

   5.3. update PQ;

6. Label the arcs between Q and its facets.

Before expanding on each of these parts of the function SHELL we will state some preconditions that are expected to be satisfied when SHELL is invoked, and some postconditions that are guaranteed to be satisfied after execution of SHELL. These conditions involve the additional information that is stored along with the nodes of the facial graph.

Preconditions:

Pre 0: The facial graph of all faces of Q that are contained in facets $F \in \mathbf{FF}$ has been correctly built already and all its arcs are properly labelled.

Pre 1: For every facet $F \in \mathbf{FF}$ the vector n_F is a normal vector to F that is parallel to the affine hull of Q, i.e. $< n_F, n > = 0$ for all $n \in N$.

Pre 2: For every horizon ridge $R \in \mathbf{HR}$, the pointer $super(R)$ is set to the unique facet in **FF** that contains R.

Pre 3: Let R be a horizon ridge in **HR** and let $F = super(R)$;

n_R is set to a normal vector of R that is orthogonal to n_F and to all $n \in N$. Therefore n_R and n_F span all vectors that are orthogonal to R and parallel to Q.

Postconditions:

Post 0: The facial graph of Q has been built correctly.

Post 1: The arcs in this facial graph have been properly labelled.

Post 2: For every facet F of Q, the vector n_F has been set such than n_F is orthogonal to F and to all vectors in N.

Post 3: For every facet F of Q, the set $neighbors(F)$ contains Q.

Post 4: For every "new" facet F of Q, i.e. $F \notin \mathbf{FF}$, the pointer $super(F)$ has been set to Q.

Post 5: For every ridge R of Q that is incident to a "new" facet F of Q the facet F is contained in $neighbors(R)$.

Note that the preconditions are satisfied vacuously the first time SHELL is invoked, since in that case the sets **FF** and **HR** are both empty.

We will now describe the different steps of the function SHELL in more detail. **Step 1** does not need much explanation. Create a node in the facial graph representing Q, introduce arcs between each facet $F \in \mathbf{FF}$ and Q, and finally, in order to satisfy postcondition 3, insert Q into $neighbors(F)$.

Next **step 2:** The function SHELL "bottoms out" when it is called to construct the facial graph of a 0-dimensional polytope. This case is easily recognizable by the condition $|T| = 1$. Let p be the only point in T. The only facet of the 0-dimensional polytope $Q = \{p\}$ is the empty face. Thus if **FF** contains an element E, it must be the empty face and all that needs to be done is to label the only arc into Q with p. If, on the other hand, **FF** is empty, then a node E for the empty face has to be created along with an arc from it to Q labelled with p. To satisfy postconditions 2 and 4 the vector n_E must be set to 0, and $super(E)$ must be set to Q. Finally, return, without executing any further steps of SHELL.

Step 3 is fairly straightforward as well. One wants to find the set **HP** of all horizon peaks, and for each $G \in \mathbf{HP}$ one wants to know the two horizon ridges that contain it. The following code segment achieves this:

```
HP := ∅
for all F ∈ HR do
    for all facets G of F do
        insert G into HP
        insert F into neighbors(G)
```

Step 4 consists of two parts: 4.1, "scheduling" the points of U in PQ, and 4.2, "scheduling" each horizon peak $G \in \mathbf{HP}$. Let us consider step 4.1 first.

As in the algorithm of the previous section, a linear program has to be solved for each $p \in U$. This linear program yields t_p, the first time point p becomes visible from the shelling line, and n_p a normal vector to the first facet of Q that contains p. This LP has a slightly different form from the one given in the previous section because the shelling line now does not necessarily contain the origin and because n_p should lie in aff$Q$ in order to fulfill postcondition 2. Thus **step 4.1** looks as follows:

For each $p \in U$ do

Solve the linear program

$$\text{minimize } t_p = \langle n_p, a \rangle \text{ such that}$$

$$\langle n_p, b - q \rangle \leq 1 \qquad \text{for all } q \in T - \{p\}$$

$$\langle n_p, b - p \rangle = 1$$

$$\langle n_p, n \rangle = 0 \qquad \text{for all } n \in N$$

If it was feasible and bounded and if $t_p > t_0$ then

Let $T' = \{q \in T \mid \langle n_p, b - q \rangle = 1\}$.

Insert (t_p, n_p, p, T') into the priority queue PQ.

In **step 4.2** we have to "schedule" each horizon peak $G \in \mathbf{HP}$. This means we have to calculate (a normal vector n_G of) the unique hyperplane of aff$Q$ that is spanned by G and the two horizon ridges of Q that contain G. The intersection

point $x(t_G)$ of the shelling line L and this hyperplane has to be found, and the appropriate information has to be stored in PQ under key t_G. This can be done as follows:

For each $G \in$ **HP** do

Let R and R' be the horizon ridges that contain G, i.e. $neighbors(G) = \{R,R'\}$.

Let p and p' be the labels of the arcs between G and R, and G and R', respectively.

Let $F = super(R)$ be the unique facet of Q that contains R.

The normal vector n_G of the sought hyperplane must be spanned by n_F and n_R (this follows from precondition 3). To be precise,

$$n_G = <n_R, p-p'>n_F - <n_F, p-p'>n_R .$$

The intersection point $x(t_G)$ of L and the sought hyperplane is then given by

$$t_G = \frac{<n_G, a>}{<n_G, b-p>} .$$

If the denominator of the fraction defining t_G is zero, then the hyperplane intersects the shelling line L in point b, which lies in the relative interior of Q. Thus the hyperplane cannot possibly contain a facet of Q, and G need not be scheduled.

If t_G well defined and $t_G > t_0$ then

Insert (t_G,n_G,G) into PQ

Set the pointer from node G to this priority queue entry.

Note that n_G would satisfy postcondition 2. By its construction it is orthogonal to all $n \in N$.

Next we explain **step 5,** the most complicated part of the procedure SHELL. It is a loop that generates the remaining facets of Q in shelling order. Before giving a detailed description of this loop we state a few loop invariants, conditions that are to hold whenever the body of the loop is to be executed. For this purpose we have to define several values which will only be used to state the invariants. They will not appear in the algorithm per se.

Let **F** be the set of facets of Q that have been discovered already and let $x(t_{current})$ be the intersection point of the shelling line L and the last of the facets in **F** (in shelling order). Let **CHR** be the set of current horizon ridges of Q, i.e. the ridges of Q that lie in exactly one facet $F \in \mathbf{F}$, and let **CHP** be the set of current horizon peaks, i.e. the peaks of Q that lie in a ridge $R \in \mathbf{CHR}$.

Loop Invariants:

Inv 0: The facial graph for all faces in the facets $F \in \mathbf{F}$ of Q has been built correctly, including a node for Q and arcs between $F \in \mathbf{F}$ and Q. All the arcs, except for the ones into Q, are properly labelled.

Inv 1: For every facet $F \in \mathbf{F}$ a normal vector n_F is known with $< n_F, n > = 0$ for all $n \in N$, the pointer $super(F)$ is set to Q, and Q is also in $neighbors(F)$.

Inv 2: For every ridge $R \in \mathbf{CHR}$ the pointer $super(R)$ is set to the unique $F \in \mathbf{F}$ that contains R.

Inv 3: For every ridge $R \in \mathbf{CHR}$ a normal vector n_R is known with $< n_R , n > = 0$ for all $n \in N$, and $< n_R , n_{super(R)} > = 0$.

Inv 4: For every known ridge R of Q, every "new" facet F of Q (i.e. $F \notin \mathbf{FF}$) that contains R is in $neighbors(R)$.

Inv 5: For every horizon peak $G \in \mathbf{CHP}$ the set $neighbors(G)$ consists of the two horizon ridges in **CHR** that contain G.

Inv 6: For every horizon peak $G \in \mathbf{CHP}$ the value t_G (in the sense of step 4.2) has been computed, and G is stored in the priority queue PQ, provided $t_G > t_{current}$.

Note that initially all theses invariants are satisfied. Invariant 0 holds because of precondition 0 and step 1. Invariants 1 to 3 are guaranteed to hold by the preconditions 1 to 3 of the procedure SHELL. Invariant 4 holds vacuously, and invariants 5 and 6 must hold because of steps 3 and 4.2.

Now we describe the individual parts of the loop that forms step 5.

Step 5.0 is straightforward: DELETEMIN returns a quintuple $(t_F, n_F, \mathbf{G}, U', T')$. Its interpretation is as follows: The next facet F in the shelling of Q lies in a hyperplane normal to n_F that intersects L in $x(t_F)$. The set **G** contains all the horizon peaks G of Q for which both horizon ridges of Q that contain G are facets of F. The set U' contains points of T that lie in F but in no other

currently known facet $F' \in \mathbf{FF}$ of Q. The set T' consists of all the points in T that lie in aff$F$. It is only given when U' is non-empty.

Step 5.1 is much more involved. In it all parameters necessary to apply SHELL recursively to construct F have to be set up. In addition it has to ensure that all the preconditions of SHELL are met.

Step 5.1.1: What shelling line L': $x'(t') = b' - \frac{1}{t'}a'$ is to be used for the shelling of F', and at what point $x'(t'_0)$ should the shelling "pick up?"

By the properties of straight line shellings we can pick for L' any line that contains $x(t_F)$, the intersection point of L and the affine hull of F, and any point b' in the relative interior of F.

How do we find a point in relint$F$? This is easy when $T' \neq \emptyset$. In this case we can simply use

$$b' := \frac{1}{|T'|} \sum_{p \in T'} p.$$

If on the other hand $T' = \emptyset$ we have to find a maximal set V of affinely independent points of F and use

$$b' := \frac{1}{|V|} \sum_{p \in V} p.$$

Such a set V can be formed as follows: Let G be any face in $\mathbf{G}$ (which is guaranteed to be non-empty in this case). Find a maximal set W of affinely independent vertices of G. This can be done by collecting the arc labels along any directed path between G and the empty face. Add to W the labels of the

arcs between G and the two faces in $neighbors(G)$ to yield V.

When setting up a', the directional vector of L', and t'_0, the pickup point for the shelling, one has to keep in mind that L' has to be oriented in such a way that the relative order of b' and $x(t_F)$ along L' agrees with the relative order of b and $x(t_F)$ along L. Finally, $x(t_F)$ is a point at infinity when $t_F = 0$. In this case F is parallel to L, which means that L' should be chosen parallel to L.

case $t_F < 0$: $a' := b - \frac{1}{t_F} a - b'$; $t_0' := -1$;

case $t_F = 0$: $a' := a$; $t_0' := 0$;

case $t_F > 0$: $a' := b' - b + \frac{1}{t_F} a$; $t_0' := 1$;

Since the length of a' is unimportant, all three cases can be subsumed into one:

$a' = t_F(b' - b) + a$ and $t_0' := t_F$.

Step 5.1.2: What is the set **FF'** of known facets of F? The known facets of F must be horizon ridges of Q. In order to discover them we have to distinguish three cases:

case 1: $\mathbf{G} \neq \emptyset$

This means more than one facet of F is known already and **FF'** comprises all horizon ridges of Q that contain horizon peaks of Q that lie in **G**.

```
FF' := ∅
For all G ∈ G do
    For both F' ∈ neighbors(G) do
        insert F' into FF';
```

case 2: $\mathbf{G}=\emptyset$ and $U' \neq T'$

This condition implies that exactly one facet F' of F is known already. Which horizon ridge constitutes F'? Note that the set of vertices of this face F' must be a subset of the vertices of F which in turn is a subset of T'. This observation justifies the following method to discover F'. Discover a maximal directed path in the currently know partial facial graph of Q that starts at the empty face and all of whose arc labels are in T'. The endpoint of any such path must be F'. Such a path can be discovered greedily in $\mathbf{O}(dm)$ time since its length is at most $d+1$ and since the outdegree of any node in the facial graph is at most m.

case 3: $\mathbf{G}=\emptyset$ and $U'=T'$

In this case F is the first facet of Q and therefore no facets of F are known yet. Thus $\mathbf{FF'} = \emptyset$.

We also have to ensure that precondition 1 will be satisfied when SHELL is invoked for F. This means for every $F' \in \mathbf{FF'}$ we want the vector $n_{F'}$ to be normal to F' and also parallel to F, i.e. normal to the orthogonal complement of F. This orthogonal complement is spanned by N, the orthogonal complement of Q, together with n_F, the normal vector of F in aff$Q$. As F' is a current horizon ridge of Q we know from invariant 3 that the vectors $n_{F'}$ and $n_{super(F')}$ span the set of vectors that are normal to F' and that lie in the orthogonal complement of F. So all that needs to be done is to create a linear combination of these two vectors that is orthogonal to n_F. This can be done easily:

For all $F' \in \mathbf{FF}'$ do

$$n_{F'} := < n_{F'}, n_F > n_{super(F')} - < n_{super(F')}, n_F > n_{F'}.$$

Step 5.1.3: What is the set $\mathbf{HR}'$ of current horizon ridges of F?

Precisely those ridges of F that lie in exactly one known facet $F' \in \mathbf{FF}'$ of F.

```
HR' := ∅;
For all F' ∈ FF' do
    For all facets R' of FF' do
        delete F' from neighbors(R')
        if R' ∈ HR' then delete R' from HR'
                  else insert R' into HR'
                       set super(R') to F'
```

The last line in this code fragment ensures that precondition 2 will be satisfied when SHELL is invoked for F in step 5.2. What about precondition 3? Let $R' \in \mathbf{HR}'$ be a horizon ridge of F and let $F' = super(R')$ be the unique facet of F that contains R' (which is known because of invariant 2). We need to calculate a normal vector $n_{R'}$ of R' that is orthogonal to $n_{F'}$ and to the orthogonal complement of F, i.e. to all $n \in N \cup n_F$.

R' is a horizon peak of Q and thus contained in two horizon ridges of Q, one of which is F'. Let G' be the other horizon ridge of Q that contains R' and let $G = super(G')$ be the facet of Q that contains G'. Note that G' can be easily identified from R'. It is the only element left in $neighbors(R')$. By invariant 3 the vectors $n_{G'}$ and n_G are independent and are both normal to G' as well as the orthogonal complement of Q. Since G' does not lie in F, it follows that $n_{G'}$, n_G, and n_F are linearly independent. Thus the projections

$$n_1 = <n_F, n_F> n_{G'} - <n_{G'}, n_F> n_F \quad \text{and}$$

$$n_2 = <n_F, n_F> n_G - <n_G, n_F> n_F$$

are linearly independent. Moreover, they are normal to R', and orthogonal to n_F and all $n \in N$. It just remains to calculate a linear combination of them that is also normal to $n_{F'}$. Thus we get the desired

$$n_{R'} = <n_1, n_{F'}> n_2 - <n_2, n_{F'}> n_1.$$

This straightforward calculation has to be performed for every $R' \in \mathbf{HR}'$.

Step 5.1.4: What is to be used as the remaining parameters for SHELL?

U' and T' are correctly given in the tuple returned by the DELETEMIN operation of step 5.0. For N', a set of vectors spanning the orthogonal complement of F, one can simply use $N \cup \{n_F\}$.

Step 5.2 is easy after everything has been laboriously prepared:

$F :=$ SHELL($(a',b'),t'_0, \mathbf{FF}', \mathbf{HR}', U', T', N'$)
create an arc from F to Q;
set *super*(F) to Q;
set n_F;
set *neighbors*(F) to $\{Q\}$;

The body of the loop that makes up step 5 is almost complete. Let us check whether all of the seven stated loop invariants are satisfied now:

Invariant 0 remains true because of step 5.2 and the postconditions 0 and 1, since the only new face in $\mathbf{F}$ is F

Invariant 1 certainly continues to hold also, since the only additional facet is F and the required information was supplied in step 5.2.

Invariant 2: The horizon ridges that are now in **CHR** and were not there before are exactly the "new" facets of F. Thus postcondition 4 of SHELL guarantees that this invariant still holds.

Invariant 3 is implied by postcondition 2 of SHELL.

Invariant 4: F is the only additional "new" facet of Q. Thus by postcondition 3 of SHELL invariant 4 remains true.

Invariant 5: The "new" horizon peaks of Q, i.e. the ones that either were not in **CHP** before or are now contained in a different horizon ridge of Q than before, are exactly the ridges of F that are contained in a "new" facet of F. Thus postcondition 5 of SHELL implies invariant 5.

Invariant 6, however, is not satisfied yet. The "new" or changed horizon peaks of Q are exactly the ridges of F that lie in a "new" facet of F. If for such a ridge G both containing facets of F are "new," then these two facets form *neighbors*(G). It follows that in this case, the two horizon ridges of Q that are in *neighbors*(G) of the horizon peak G of Q span aff$F$, which intersects the shelling line L in $x(t_F)$. As t_F is the new $t_{current}$, this means such a G need not be stored in the priority queue PQ.

If, on the other hand, a ridge G of F is contained in only one "new" facet of F, it is necessary to try to schedule such a horizon peak of Q in PQ. It is straightforward to determine these ridges of F. They are exactly **HR**′, the initial horizon ridges of F.

Step 5.3 schedules these horizon peaks of Q in PQ. The computations involved are analogous to the ones in step 4.2.

For all $G \in \mathbf{HR}'$ do
Delete the entry corresponding to G from PQ (using the pointer stored with G).
Let $\{R,R'\} = neighbors(G)$;
Let p and p' be the labels of the arcs between G and R, and G and R', respectively.
Let $n_G := <n_R, p-p'>n_{super(R)} - <n_{super(R)}, p-p'>n_R$;
Let $t_G = \dfrac{<n_G, a>}{<n_G, b-p>}$;
If t_G well defined and $t_G > t_0$ then
Insert (t_G,n_G,G) into PQ;
Set the pointer from node G to this priority queue entry.

This completes the loop that constitutes step 5. The reader may check that upon termination of this loop its invariants imply most of the postulated postconditions of the procedure SHELL. In particular invariant 0 implies postcondition 0, invariant 1 implies postconditions 2 to 4, and invariant 4 implies postcondition 5. The only postcondition that is not satisfied is the one numbered 1 that concerns arc labels. By invariant 0 all arcs except for the ones into Q have been labelled. **Step 6** takes care of the remaining ones.

In order to label an arc between a facet F of Q and Q itself one has to find a vertex of Q that is not in F. Here is a simple method: Let R be a ridge of Q that is contained in F and let H be the other facet of Q that contains R. The label p of the arc between R and H is a vertex of $H \subset Q$ that is not in $R = F \cap H$. Thus p is a vertex of Q that is not contained in F, as desired. This approach poses only one slight problem in that it is not straightforward to dis-

cover "the other" facet of Q that contains a given ridge R. During the shelling algorithm Q can be just a face of a larger polytope. Thus R can have many superfaces. The following algorithm solves this problem. In order to avoid alloting yet more space to each node in the facial graph this algorithm saves *neighbors*(R) in a FIFO queue and reuses the freed space. It is assumed that the enumeration order of the facets of a face is fixed.

```
For all facets F of Q do
    For all facets R of F do
        if R unmarked then mark R
                           insert neighbors(R) into FIFO queue
                           set neighbors(R) := {F}
                      else unmark R
                           insert F into neighbors(R)

For all facets F of Q do
    Let R be some facet of F
    Let H be the member of neighbors(R) that is not F
    Let p be the label of the arc between R and H
    Label the arc from F to Q with p

For all facets F of Q do
    For all facets R of F do
        if R unmarked then mark R
                           set neighbors(R) to first element in FIFO
                                                              queue
                      else unmark R
```

Of course it is possible to include this arc labelling into the loop of step 5. For the sake of presentation, however, we found it clearer to separate out this aspect into a step of its own. This ends the presentation of the procedure SHELL. Hopefully, the given preconditions, postconditions, and loop invariants have aided the reader to convince himself of the correctness of this procedure.

5.6. Making the Shelling Line Admissible

Unfortunately, the procedure SHELL still has one flaw. The shelling lines it uses cannot in general be guaranteed to be admissible. Recall that a line L is admissible for a polytope P iff no facet of P is parallel to L and the affine hulls of no two facets intersect L in the same point. Intuitively, no two facets must "become visible at the same time." If a non-admissible shelling line happens to be used in our algorithm, horizon peaks that belong to different facets will be stored in the priority queue under the same key and thus become indistinguishable. The observant reader might object that it is still possible to differentiate between these entries in the priority queue by comparing the normal vectors n_G that are stored with them. This is correct. However, the question which facet to shell next would still remain unresolved. (There are examples that show that such ties must not be broken arbitrarily.)

Of course, when the algorithm begins, the facets of the polytope to be constructed are not known already. Thus there is no reasonable way of telling in advance whether a proposed shelling line will be admissible. Fortunately, this seemingly insurmountable problem can be solved using perturbation techniques.

This is the idea: Use as shelling line an arbitrary line L that intersects the interior of P. When a tie occurs – more than one facet becomes "visible" at once – resolve it by determining what would happen if a line slightly different from L had been used.

Formally this can be achieved as follows. Let $L: x(t) = b - \frac{1}{t}a$ be the proposed shelling line, let H be a hyperplane with normal vector n that contains some point p but does not contain b. Then H intersects L at

$$t = \frac{<n, a>}{<n, b-p>},$$

where $t=0$ means H and L intersect at infinity, i.e. they are parallel.

Now let $\vec{a}(\varepsilon)$ be a d-vector whose components are algebraically independent polynomials $a_i(\varepsilon)$ of degree $d-1$, such that $\vec{a}(0) = a$. Algebraically independent here means that the $d \times d$ matrix A formed by the coefficients of these polynomials is non-singular. For every ε let L_ε be the line defined by $x(t) = b - \frac{1}{t}\vec{a}(\varepsilon)$. Note that for $\varepsilon=0$ the line L_ε is nothing but L. For small positive ε the line L_ε is just a small perturbation of L with $L_\varepsilon \cap L = \{b\}$. For every ε the hyperplane H intersects L_ε at

$$t = t(\varepsilon) = \frac{<n, \vec{a}(\varepsilon)>}{<n, b-p>}.$$

Note that formally $t(\varepsilon)$ is a degree $d-1$ polynomial in ε, since the numerator of its defining expression is a sum of d degree $d-1$ polynomials and the denominator is a non-zero constant (as $b \notin H$ and $p \in H$). Also note that $t(\varepsilon)$ cannot be the zero polynomial since the component polynomials of $\vec{a}(\varepsilon)$ were assumed to be algebraically independent.

Now let $\bar{H}$ be a hyperplane with normal vector $\bar{n}$ that contains a point $\bar{p}$ but does not contain b. The intersection of L_ε and $\bar{H}$ is determined by

$$\bar{t} = \bar{t}(\varepsilon) = \frac{<\bar{n}, \vec{a}(\varepsilon)>}{<\bar{n}, b-\bar{p}>} .$$

Clearly H and $\bar{H}$ intersect L in the same point iff $t(0)=\bar{t}(0)$. However, for arbitrarily small positive ε the hyperplanes H and $\bar{H}$ cannot intersect L_ε in the same point unless they are identical. To prove this we just have to show that the polynomials $t(\varepsilon)$ and $\bar{t}(\varepsilon)$ are the same iff $H=\bar{H}$. But this follows from the following observations: Scaling n leaves $t(\varepsilon)$ unchanged. The inner product $<n, b-p>$ is constant for all $p \in H$. The polynomials defining $\vec{a}(\varepsilon)$ are algebraically independent. Analogous statements hold for $\bar{H}$ as well.

If for sufficiently small $\varepsilon>0$ the hyperplanes H and $\bar{H}$ do not intersect L_ε in the same point, which one intersects "first?" Which of the polynomials $t(\varepsilon)$ and $\bar{t}(\varepsilon)$ is smaller for arbitrarily small $\varepsilon>0$? This can clearly be deduced from the coefficients defining the polynomials: The relative order between the corresponding lowest degree non-agreeing coefficients of $t(\varepsilon)$ and $\bar{t}(\varepsilon)$ determines the relative order between these two polynomials for all sufficiently small $\varepsilon>0$. In other words, which of H and $\bar{H}$ intersects the shelling line L_ε "first" for arbitrarily small $\varepsilon>0$ can be decided by a lexicographic comparison between the coefficient vectors of the polynomials $t(\varepsilon)$ and $\bar{t}(\varepsilon)$.

Everything seems nice and dandy now. Represent $\vec{a}(\varepsilon)$ by its $d \times d$ coefficient matrix A, and whenever something is to be stored in the priority queue with single value key

$$t = \frac{<n, a>}{<n, b-p>} ,$$

use the vector key

$$\vec{t} = \frac{1}{< n , b - p >} n^T A$$

instead, employing lexicographic comparisons in place of the single value comparisons.

There is a lacuna, however, in this wholesale reasoning. In general, we will shell some k-dimensional polytope Q in $\mathbf{R}^d$, with $k \leq d$ but not necessarily $k = d$. We cannot allow the shelling line to be perturbed arbitrarily, but it has to stay in the affine hull of Q. Call this affine hull X and let N be the orthogonal complement of X. If the shelling line is to stay in X, then the directional vector $\vec{a}(\varepsilon)$ has to be of the form $\sum_{0 \leq i < d} a_i \varepsilon^i$, where each a_i is a d-vector orthogonal to N. Of course, if the dimension of X is k, then only k independent vectors of this kind can be found. Thus if $k < d$, the matrix $A = (a_0, \ldots, a_{d-1})$ is singular, which means that the d component polynomials that constitute $\vec{a}(\varepsilon)$ are not algebraically independent, which in turn invalidates the claim that the polynomials $t(\varepsilon)$ and $\bar{t}(\varepsilon)$ are different for distinct hyperplanes H and $\bar{H}$.

Fortunately, we can ignore this problem. In our algorithm only hyperplanes occur whose normal vectors are orthogonal to N. As it turns out, for these hyperplanes the correspondence with the polynomial $t(\varepsilon)$ is 1-1. Why is that so? Assume there are two such hyperplanes H and $\bar{H}$ with equal corresponding polynomials $t(\varepsilon)$ and $\bar{t}(\varepsilon)$. Then $\frac{1}{< n , b - p >} n^T A$ must be equal to $\frac{1}{< \bar{n} , b - \bar{p} >} \bar{n}^T A$, which means

$$\left(\frac{1}{<n, b-p>}\, n - \frac{1}{<\bar{n}, b-\bar{p}>}\, \bar{n}\right)^T A = 0 ,$$

i.e. a linear combination of n and $\bar{n}$ is in N, which is clearly impossible if both these vectors are orthogonal to N.

Having filled (hopefully) all gaps and dismissed all objections, we can now describe the exact modifications to the procedure SHELL that will make every shelling line appear admissible.

We introduce an additional parameter k, which will denote the dimension of the polytope that is being shelled. The pair of d-vectors (a,b) in the parameter list representing the shelling line L we replace by a pair (A,b) describing the perturbing shelling line L_ε, where $A = (a_0, \ldots, a_{d-1})$ is a $d \times k$ matrix representing the directional vector $\vec{a}(\varepsilon) = \sum_{0 \le i < d} a_i \varepsilon^i$ of L_ε, and b is a point in the relative interior of the polytope, as before.

The parameter t_0, the real number describing how far a shelling has progressed along L, we replace by a k-tuple $\vec{t_0}$, which represents the polynomial $t(\varepsilon)$ that describes for each ε how far a shelling has progressed along L_ε. Similarly, the priority queue will be changed, so that the keys are now k-tuples $\vec{t}$ of reals instead of simple real numbers t, and lexicographic comparisons will be used between these keys. Note that by the arguments in the preceding paragraphs it suffices to consider only degree $k-1$ polynomials when perturbing the shelling line for a k-polytope. This is the reason why we can use k-tuples instead of d-tuples.

Before SHELL is invoked the first time for a point set S in $\mathbf{R}^d$, the dimension $k(S)$ of the affine hull of S has to be determined and a set B of $d-k(S)$ linearly independent vectors has to be found that span the orthogonal complement of affS. In addition $k(S)$ linearly independent vectors $a_0, \ldots, a_{k(S)-1}$ have to be determined that are all orthogonal to B. SHELL is then called with parameters $k = k(S)$, $A = (a_0, \ldots, a_{k(S)-1})$, and $\vec{t_0} = (-\infty, \ldots, -\infty)$. The other parameters are set as before.

In step 4.1 of SHELL we would now like to solve for every $p \in T$ the following "lexicographic linear program:"

$$\text{lexicographically minimize } \vec{t_p} = (<n_p, a_1>, <n_p, a_2>, \ldots, <n_p, a_k>)$$

$$\text{such that } <n_p, b-q> \leq 1 \qquad \text{for all } q \in T-\{p\}$$

$$<n_p, b-p> = 1$$

$$<n_p, n> \leq 0 \qquad \text{for all } n \in N$$

Such a lexicographic minimization can be achieved by solving a sequence of k ordinary linear programs:

For $j=1$ to k do

minimize $t_{p,j} = <n_p, a_j>$ such that

$$<n_p, b-q> \leq 1 \qquad \text{for all } q \in T-\{p\}$$

$$<n_p, b-p> = 1$$

$$<n_p, n> \leq 0 \qquad \text{for all } n \in N$$

$$<n_p, a_i> = t_{p,i} \qquad \text{for } 1 \leq i < j$$

However, most linear programming algorithms (in particular pivoting methods and Megiddo's elimination algorithm) can be modified so that at little additional cost lexicographic linear programs can be solved directly.

After solving this lexicographic linear program and determining the the set T', the tuple (n_p,p,T') has to be inserted into the priority queue with key $\vec{t_p} = (t_{p,1}, \ldots, t_{p,k})$.

In steps 4.2 and 5.3 the tuple (n_G,G) has to be inserted into the priority queue with key $\vec{t_G} = \frac{1}{<n_G, b-p>} n_G^T A$.

Finally, there is step 5.1.1, where a representation for the perturbing shelling line L_ε' is to be computed. The computation of b' remains the same. The required $d \times (k-1)$ matrix A' is found by computing

$$A'' = (b'-b)\vec{t_F}^T + A$$

and deleting the last column from this matrix.[1] The $(k-1)$-vector $\vec{t_0}$ must be set to $\vec{t_F}$, omitting the last component. For the sake of completeness: The recursive call in step 5.2 should include the parameter $k' = k-1$.

---

[1]That the $d \times k$ matrix A'' has deficient rank is clear from the geometric context. That it indeed has rank $k-1$ can be proven algebraically by showing that, considering how $\vec{t_F}$ was defined, A'' must be representable as a product $B \cdot A$, where B is a $d \times d$ matrix of rank $d-1$ whose null space is spanned by the vector $b'-b$, which in turn is in the column space of A. The details are left to the reader.

5.7. Time Analysis and Conclusions

We want to analyze the running time of the procedure SHELL when applied to a point set S of m points in $\mathbf{R}^d$ whose convex hull is the polytope P. We will measure the size of the facial graph of P in three parameters $K_i(P)$, $0 \le i \le 2$. Recall that $\varphi(Q)$ and $\rho(Q)$ denote the number of facets and ridges of polytope Q. These quantities define the values K_i as follows:

$$K_0(P) = \sum_{F \text{ a face of } P} 1$$

$$K_1(P) = \sum_{F \text{ a face of } P} \varphi(F)$$

$$K_2(P) = \sum_{F \text{ a face of } P} \rho(F)$$

Of course $K_0(P)$ is the number of nodes in the facial graph of P, $K_1(P)$ is the number of arcs, and the reader may convince himself that the quantity $K_2(P)$ is half the number of directed paths in the facial graph of P with length 2. With these quantities the analysis of the running time of the presented procedure becomes straightforward.

For every face of P (except for the empty face E) SHELL is invoked exactly once, and for each face of P, except for P, E, and the vertices of P, the body of the loop in step 5 is executed exactly once. We will thus first analyze the cumulative cost of all invocations of SHELL without step 5. Second we will analyze the cumulative cost of all executions of the body of the loop in step 5 in all invocations of SHELL.

Let Q be some face of P for whose construction SHELL has been invoked. Step 1 of SHELL certainly does not take more than $\mathbf{O}(\varphi(Q))$ time. Step 2 takes constant time.

Step 3 takes constant time per discovered horizon peak since every such peak is in two ridges from **HR**. This cost can be considered subsumed by the cost of step 4.2, which takes time $\mathbf{O}(d^2+\log M)$ per horizon peak, where M is some upper bound to the size of the priority queue PQ. (One would expect a $d\log M$ term in this bound for the modified version of SHELL that implements the perturbation technique described in the previous section. However, this extra factor of d can be avoided if one uses a special data structure for lexicographic search that has access and update time of $\mathbf{O}(d+\log M)$; see [Meh, p.285].) We claim that we can ignore this cost and charge it to step 5. For this purpose consider a more expensive version of SHELL that inserts every horizon peak in **HP** into PQ irrespective of whether $t_G > t_0$ (with undefined t_G set to $-\infty$). Of course, the DELETEMIN operation of step 5 would be modified as well so that it would return the least elements in PQ with key greater than $t_{current}$. Note that this more expensive version would also terminate with an empty priority queue. The deletions in step 5.3 would eliminate the extraneous entries. Thus we assume that every horizon peak discovered in step 3 is inserted into PQ and we charge the $\mathbf{O}(d^2+\log M)$ cost that it incurred to its eventual deletion from PQ in step 5.

Step 6 clearly takes time $\mathbf{O}(\rho(Q))$, and thus the overall charge for SHELL

invoked for Q, without step 5 and step 4.1 is $\mathbf{O}(\varphi(Q)+\rho(Q))$. This means taken over all invocations of SHELL the cumulative cost is $\mathbf{O}(K_1(P)+K_2(P))$.

The cost of step 4.1 is clearly dominated by the time to solve the (lexicographic) linear program. This program has at most $m-1$ inequality constraints, and using the equality constraints it can easily be transformed into one with at most $d-1$ variables. Note that considering all invocations of SHELL, step 4.1 can be executed at most d times for every $p \in S$; once for each dimension up to d. This means overall this step is executed at most dm times. Thus the cumulative cost of this step is $\mathbf{O}(dm\lambda(m-1,d-1))$, where $\lambda(m,d)$ denotes the cost of solving a (lexicographic) linear program with m constraints in d variables.

At last we turn to the analysis of step 5. Let F be the face that is constructed during some execution of the body of the loop in step 5. The DELETE-MIN operations of step 5.0 involves $|\mathbf{G}|+|U'|$ deletions from PQ. We dispose of the $|U'|$ factor by noting that the cost of insertion of the points $p \in U'$ was accounted for in the analysis of step 4.1 and thus we can charge the cost of their deletion to their insertion. All the elements of $\mathbf{G}$ will be ridges of F. Noting that we also have to take care of the insertion charge for every $G \in \mathbf{G}$ the total cost incurred by step 5.0 is certainly not more than $\mathbf{O}(\rho(F)(d^2+\log M))$.

The time required by step 5.1.1 is $\mathbf{O}(d^2)$ in the "normal" case when $T'=\varnothing$. The other case, $T' \neq \varnothing$ requires time $\mathbf{O}(dm)$. But it can occur only at most md times over all executions of the body of the loop in all invocations of SHELL.

Thus cumulatively, the latter case requires time $\mathbf{O}(d^2m^2)$.

Step 5.1.2 presents a similar story: case 2, $\mathbf{G}=\varnothing$, can happen at most dm times and then requires time $\mathbf{O}(dm)$ incurring an accumlative $\mathbf{O}(d^2m^2)$ cost. In case 1, the time required in the first part of this step is proportional to $|\mathbf{G}|$ which is less than $\rho(F)$. The time necessary for the second part of step 5.1.2 is $\mathbf{O}(d)$ for every facet of F that is in $\mathbf{FF}'$ and thus bounded by $\mathbf{O}(d\varphi(F))$.

The time required for step 5.1.3 is clearly bounded by $\mathbf{O}(d\rho(F))$ and step 5.1.4 takes constant time. So does step 5.2, excluding the time taken by the recursive call.

In step 5.3 the cost for every $G \in \mathbf{HR}'$ is $\mathbf{O}(d^2+\log M)$. Since every such G is a ridge of F the overall cost of this step is bounded by $\mathbf{O}(\rho(F)(d^2+\log M))$.

The total cost of one execution of the body of the loop in step 5 (excluding the special cases) is thus $\mathbf{O}(\,\rho(F)(d^2+d\log M) + d^2 + d\varphi(F)\,)$. Taken over all executions of the loop over all invocations the time required for step 5 (including the special cases) is thus

$$\mathbf{O}(\,d^2m^2 + d^2K_0(P) + dK_1(P) + (d^2+\log M)K_2(P)\,).$$

The value M in this expression is any upper bound on the size of priority queue PQ. Clearly $K_0(P)$, the number of faces of P, can be substituted for M, since every face can be stored in PQ at most once.

Before combining all the individual time bounds let us note that it is realistic to assume that $\lambda(m,d) = \Omega(dm)$. Just writing down the m constraints

would take this much time. Thus any $\mathbf{O}(d^2m^2)$ term is subsumed by the $\mathbf{O}(dm\lambda(m-1,d-1))$ cost of step 4.1. For similar reasons the cost of preprocessing before SHELL is called the first time can be thought to be subsumed by the cost of step 4.1. Finally note that $K_0(P) = \mathbf{O}(K_1(P))$ and $K_1(P) = \mathbf{O}(K_2(P))$. We summarize:

Theorem 5.7.1:

When applied to a set S of m points in $\mathbf{R}^d$ the procedure SHELL constructs the augmented facial graph of the convex hull P of S in time $\mathbf{O}\left(dm\lambda(m-1,d-1) + K_2(P)(d^2+\log K_0(P)) \right)$.

Analogous to Theorem 5.4.1 we have the following corollaries.

Corollary 5.7.2:

For fixed d the facial graph of the polytope P formed by the convex hull of a given set S of m points in $\mathbf{R}^d$ can be constructed in time $\mathbf{O}(m^2 + K_2(P)\log m)$.

Corollary 5.7.3:

For fixed $d>3$ the facial graph of the polytope P formed by the convex hull of a given set S of m points in $\mathbf{R}^d$ can be constructed in worst case time $\mathbf{O}(m^{\lfloor d/2 \rfloor}\log m)$.

Some comments are in order. First of all, my original claim in [S86] that the

running time of SHELL depended on $K_1(P)$ instead of $K_2(P)$ turned out to be wrong. This dependence on $K_2(P)$ raises some interesting questions. Are there polytopes P for which the three quantities $K_0(P)$, $K_1(P)$, and $K_2(P)$ differ significantly? It is not hard to see that $K_1(P) \leq mK_0(P)$ and $K_2(P) \leq mK_1(P)$ holds, where m denotes the number of vertices of P. Can m be replaced in these inequalities by quantities that depend on the dimension d only? As $K_2(P)$ also arises in the time bound for Swart's gift-wrapping algorithm, it is conceivable that dependence on this quantity is inherent to the complexity of the facial graph problem. As (admittedly inconclusive) support to this hypothesis consider the corresponding decision problem: Is a given directed arc-labelled graph the facial graph of some polytope P? So far I have been unable to find an algorithm that decides this question in time $\mathbf{o}(K_2(P))$.

Finally, some remarks about the use of linear programming in the new algorithms: I certainly do not advocate the actual use of Megiddo's $\mathbf{O}(m)$ linear programming algorithm [M84], since its running time depends doubly exponentially on d. The recent new $\mathbf{O}(m)$ algorithms of Dyer [D86] and Clarkson [C] whose dependence on the dimension is 3^{d^2} do not present a viable alternative either. It is very likely, however, that even using an ordinary simplex method, the running times of the proposed algorithms will be comparable to, if not better than, the running time of the corresponding gift-wrapping based algorithms. This claim is based on the observation that gift-wrapping and pivoting are dual notions, and that it is likely that the total number of pivot steps in all

the linear programs to be solved in the shelling algorithm is smaller that the number of gift-wrapping steps in the gift-wrapping algorithm. It remains to be tested whether this really is the case.

The proposed algorithms assume exact real arithmetic at unit cost per operation as the underlying model of computation. This is certainly somewhat unrealistic. It should be clear that rational arithmetic could also be used. In this case it would be interesting to analyze the running time of the algorithms in terms of the logarithmic cost model. Also, note that the polynomial-time linear programming algorithms [A-S],[L-Y],[Kar] could be used in this case.

References

[A-H-U] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, **The Design and Analysis of Computer Algorithms.** Addison-Wesley, Reading, MA (1974).

[A-S] B. Aspvall and R.E. Stone, *Khachiyan's Linear Programming Algorithm.* J. of Algorithms 1 (1980) 1-13.

[A] D. Avis: *On the Complexity of Finding the Convex Hull of a Set of Points.* McGill Univ., Montreal, School of Computer Science, Report SOCS 79.2 (1979).

[A-E-S] D. Avis, H. ElGindy, and R. Seidel, *Simple On-Line Algorithms for Convex Polygons,* in **Computational Geometry**, G. Toussaint (ed.), North Holland (1985) 23-42.

[B] B. Bhattacharya, *Application of Computational Geometry to Pattern Recognition Problems.* Simon Fraser Univ. CS Tech.Rep. 82-3 (1982).

[BO] M. Ben-Or, *Lower Bounds for Algebraic Computation Trees.* Proc. 15th ACM STOC, (1983) 80-86.

[BKST] J.L. Bentley, H.T. Kung, M. Schkolnick, and C.D. Thompson, *On the Average Number of Maxima in a Set of Vectors and Applications.* Journal of the ACM 25 (1978) 536-543.

[B-O] J.L. Bentley and T.A. Ottmann, *Algorithms for Reporting and Counting Geometric Intersections.* IEEE Trans. Computers, C-28, 9 (1979) 643-647.

[B-S] J.L. Bentley and M.I. Shamos, *Divide and Conquer for Linear Expected Time.* Information Processing Letters 7, (1978) 87-91.

[B-M] H. Bruggesser and P. Mani, *Shellable Decompositions of Cells and Spheres.* Math. Scand. 29 (1971) 197-205.

[C-K] D.R. Chand and S.S. Kapur, *An Algorithm for Convex Polytopes.* Journal of the ACM 17 (1970) 76-86.

[C-G-L] B. Chazelle, L.J. Guibas, D.T. Lee, *The Power of Geometric Duality.* Proc. 24th IEEE Symposium on FOCS (1983) 217-225.

[C] K.L. Clarkson, *Linear Programming in* $\mathbf{O}(n3^{d^2})$ *time.* Information Processing Letters 22 no.1, (1986) 21-24.

[D] M.E. Dyer, *Linear-Time Algorithms for Two- and Three-Variable Linear Programming.* SIAM J. on Computing (Feb. 1984) 31-45.

[D83] M.E. Dyer, *The Complexity of Vertex Enumeration Methods.* Math. Oper. Res. 8 (1983) 381-402.

[D86] M.E. Dyer, *A Multidimensional Search Technique and its Applications to the Euclidean One-Center Problem.* Manuscript 1986.

[Ed] U.F. Eddy, *A New Convex Hull Algorithm for Planar Sets.* ACM Transaction on Mathematical Software 3, (1977) 398-403 and 411-412.

[E82] H. Edelsbrunner, *Intersection Problems in Computational Geometry.* Tech. Univ. of Graz; Ph.D. Thesis (1982).

[E] H. Edelsbrunner, **Arrangements and Geometric Computation.** To be published.

[E-G] H. Edelsbrunner and L.J. Guibas, *Topologically Sweeping an Arrangement.* Proc. of the 18th ACM STOC (1985) 389-403.

[E-O-S] H. Edelsbrunner, J. O'Rourke, and R. Seidel, *Constructing Arrangements of Lines and Hyperplanes with Applications.* SIAM J. on Computing (May 1986) 342-363.

[G-B-T] H.N. Gabow, J.L. Bentley, and R.E. Tarjan, *Scaling and Related Techniques for Geometry Problems.* Proc. 16th ACM STOC (1984) 135-143.

[Gra] R.L. Graham, *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set.* Information Processing Letters 1, (1972) 132-133.

[G] B. Grünbaum, **Convex Polytopes.** Interscience (1967).

[G-Se] L.J. Guibas and R. Seidel *Computing Convolutions by Reciprocal Search.* Proc. 2nd ACM Symposium on Computational Geometry (1986) 90-99.

[G-St] L.J. Guibas and J. Stolfi, *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams.* ACM Trans. on Graphics 4 (1985) 74-123.

[J] R.A. Jarvis, *On the Identification of the Convex Hull of a Finite Set of Points in the Plane.* Information Processing Letters 2, (1973) 18-21.

[Kal] M. Kallay, *Convex Hull Algorithms for Higher Dimensions.* Manuscript (1981).

[Kar] N. Karmarkar, *A New Polynomial Time Algorithm for Linear Programming.* Combinatorica 4, (1984) 373-411.

[K-S85] D.G. Kirkpatrick and R. Seidel, *Output-Size Sensitive Algorithms for Finding Maximal Vectors.* Proc. 1st ACM Symposium on Computational Geometry (1985) 89-96.

[K-S86] D.G. Kirkpatrick and R. Seidel, *The Ultimate Planar Convex Hull Algorithm?* SIAM J. on Computing (Feb. 1986) 287-299.

[K] D.E. Knuth, **The Art of Computer Programming**, Volume 3. Addison-Wesley, Reading, MA (1973).

[K-L-P] H.T. Kung, F. Luccio, and F.P. Preparata, *On Finding the Maxima of a Set of Vectors.* Journal of the ACM 22 (1975) 469-476.

[L-Y] L.A. Levin and B. Yamnitsky, *An Old Linear Programming Algorithm runs in Polynomial Time.* Proc. of the 23rd IEEE Symposium on FOCS (1982) 327-328.

[M-S] P. McMullen and G.C. Shepard, **Convex Polytopes and the Upper Bound Conjecture.** London Math. Soc. Lecture Notes Series, vol 3, Cambridge Univ. Press (1971).

[M-R] T.H. Mattheiss and D.S. Rubin, *A Survey and Comparison of Methods for Finding all Vertices of Convex Polyhedral Sets.* Math. Oper. Res. 5 (1980) 167-185.

[Meh] K. Mehlhorn **Data Structures and Algorithms 1: Sorting and Searching.** EATCS Monographs on Theoretical Computer Science; Springer-Verlag (1984).

[M83] N. Megiddo, *Linear-Time Algorithms for Linear Programming in R^3 and Related Problems.* SIAM J. on Computing (Nov. 1983) 759-776.

[M84] N. Megiddo, *Linear Programming in Linear Time when the Dimension is Fixed.* Journal of the ACM 31 (1984) 114-127.

[N-P] J. Nievergelt and F.P. Preparata, *Plane-sweep algorithms for intersecting geometric figures.* Communications of the ACM 25 (1982) 739-747.

[O] M.H. Overmars *The Design of Dynamic Data Structures.* Univ. of Utrecht; Ph.D. Thesis (1983).

[Pre] F.P. Preparata, *An Optimal Real Time Algorithm for Planar Convex Hulls.* Communications of the ACM 22, (1979) 402-405.

[P-H] F.P. Preparata and S.J. Hong, *Convex Hulls of Finite Sets of Points in Two and Three Dimensions.* Communications of the ACM 20, (1977) 87-93.

[P-S] F.P. Preparata and M.I. Shamos, **Computational Geometry.** Springer Verlag (1985).

[R] P.V. Ramanan, *Topics in Combinatorial Algorithms*, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign; Ph.D. Thesis (1985).

[R-W] C. Rey and R. Ward, *An On-Line Algorithm for Determining Convex Polytopes.* Manuscript, Dept. of EE, Univ. of B.C. (1984).

[Sk] R. Sedgewick, *Quicksort.* Stanford University, Stanford, California; Ph.D. Thesis, (1975).

[S81] R. Seidel, *A Convex Hull Algorithm Oprimal for Point Sets in Even Dimensions.* Univ. of BC, CS Tech.Rep. 81-14 (1981).

[S86] R. Seidel. *Constructing Higher-Dimensional Convex Hulls at Logarithmic Cost per Face.* Proc. of the 18th STOC (1986) 404-413.

[Sh] M.I. Shamos, *Computational Geometry.* Yale University, New Haven, Connecticut; Ph.D. Thesis, (1978).

[S-Y] J.M. Steele and A.C. Yao, *Lower Bounds for Algebraic Decision Trees.* Journal of Algorithms 3, (1982) 1-8.

[Sw] G. Swart, *Finding the Convex Hull Facet by Facet.* J. of Algorithms 6 (1985) 17-48.

[T-H] G.T. Toussaint and M.E. Houle, *Computing the Width of a Set.* Proc. 1st ACM Symposium on Computational Geometry (1985) 1-7.

[vE] P. van Emde Boas, *On the* $\Omega(n\log n)$ *Lower-Bound for Convex Hull and Maximal Vector Determination.* Information Processing Letters 10, (1980) 132-136.

[Y] A.C. Yao, *A Lower Bound to Finding Convex Hulls.* Journal of the ACM 28, (1981) 780-789.