NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

COMPUTING CONVEX HULLS IN HIGHER DIMENSIONS

by

Helena Klimo

B.A.(Pure Math), University of Calgary, 1971

Extended Studies Diploma, Simon Fraser University, 1983

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

December 1988

ISBN 0-315-48787-9

APPROVAL

Name: Helena Klimo

Degree: Master of Science

Title of thesis: COMPUTING CONVEX HULLS IN HIGHER DIMENSIONS

Examining Committee:

Chairman: Dr. J. Peters

Dr. Binay K. Bhattacharya
Senior Supervisor

Dr. Pavol Hell

Dr. Anthony H. Dixon
External Examiner

Date Approved: December 12, 1988

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Computing Convex Hulls in Higher Dimensions

Author: _____

(signature)

Helena Klimo

(name)

December 14, 1988

(date)

ABSTRACT

In this thesis we show how a point inclusion problem in a convex polyhedron, determined by a set of n points in R^d, can be solved in $\Theta(d \log F_{d,n})$ expected time, where $F_{d,n}$ is the number of facets of P. We also show how this result can be used to get fast on-line convex hull algorithm and how the "divide and conquer" technique, already used to compute convex hulls of two and three dimensional data sets, can be extended to higher dimensions.

The on-line algorithm uses the "beneath beyond" technique and can be applied to a general data set to solve either the Facet Problem (without maintaining some description of all the faces of a convex hull) or the Facial Lattice Problem. The "divide and conquer" technique solves the Facet Problem for simplicial data sets.

DEDICATION

To my husband, Paul

and our children Paul jr. and Monica

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my thesis supervisor Dr. Binay Bhattacharya for suggesting the topic and for his constant encouragement, unfailing interest and patience shown to me during my research.

Thank you, Binay.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The convex hull of a finite set S of n points, S being a subset of a d-dimensional Euclidean space R^d for $d \geq 2$, is one of the basic and important geometrical constructs. Stated in a very simple way, it can be defined as the smallest convex set containing a given set of points [Fig. 1.1].

The convex hull plays a central role in the field of computational geometry. A number of geometrical problems can be solved by transforming the original problem to the convex hull problem. There are many areas other than computational geometry where this geometric construct finds practical applications: image processing and pattern recognition, computer graphics, engineering, operations research, design automation, just to name few. In pattern recognition for example, questions such as separability or existence of linear decision rules can be easily answered through the computation of convex hulls [Rosenfeld(1969), Duda-Hart(1973), Toussaint(1978), Akl-Toussaint(1978-1)]. The following references discuss some interesting problems where the determination of a convex hull is needed: Freeman-Shapira(1975), Gilbert-Pollak(1986), Sklansky(1972).

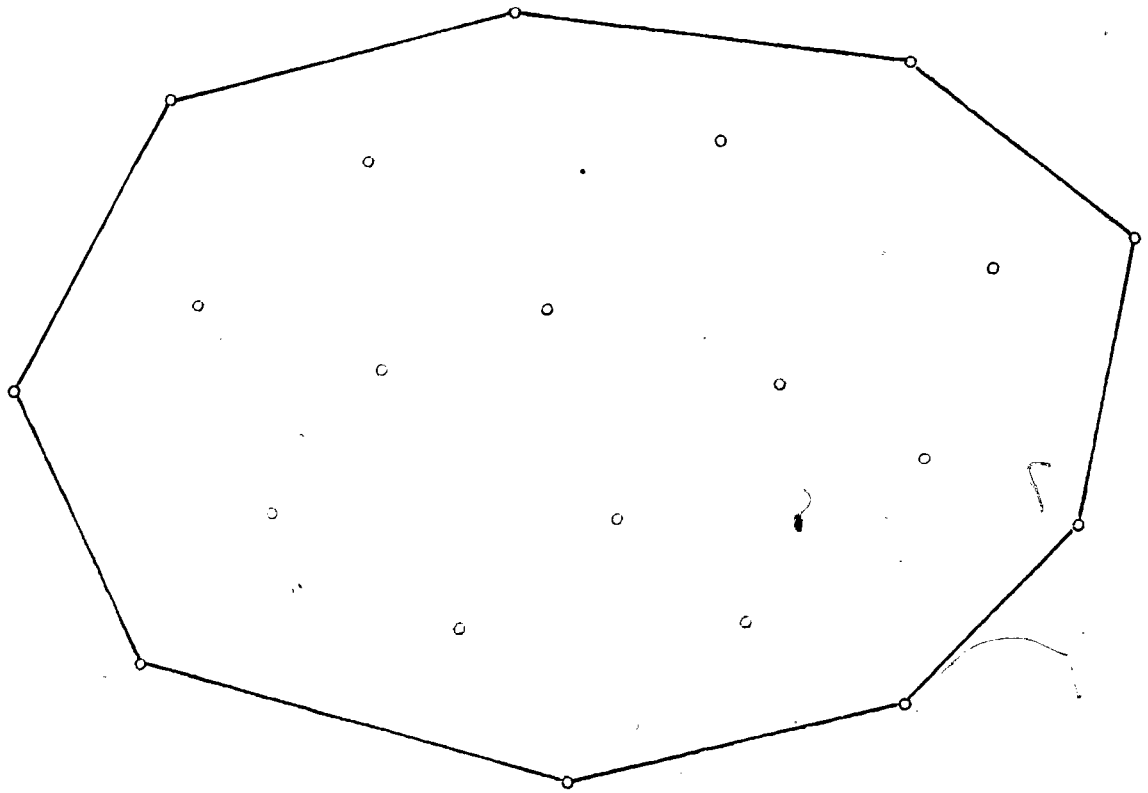


Fig. 1.1: Convex hull of a set of points in R^2.

1.1 The Convex Hull Problem

Various terms and definitions are going to be presented which will be used in this and subsequent sections. The reference to them can be found in Grünbaum(1967).

A set $S \subseteq R^d$, is called **affine** if for each pair of points x, y

from S, the line through x and y is contained in S. The **affine hull** of S, denoted as $aff(S)$, is the intersection of all affine sets containing S. A set S is **affinely independent** if no point of S is contained in the affine hull of the remaining points of S. Otherwise the set is called **affinely dependent.**

A set S is of **dimension** k, $k \leq d$, if S contains $(k+1)$ affinely independent points and every subset of S with $(k+2)$ points is affinely dependent.

A **hyperplane** in R^d, denoted as H^{d-1}, is an affine hull of d affinely independent points. In a mathematical way a hyperplane can be defined as follows:

Definition 1.1

A **hyperplane** H^{d-1} is the set of points $x = (x_1, \ldots, x_d)$ from R^d which satisfy the equation of the form

$$(x, \overline{n}) = x_1 n_1 + \ldots + x_d n_d - c = 0,$$

where c is a real number and $\overline{n} = (n_1, \ldots, n_d) \neq (0, \ldots, 0)$.

In this definition a $(d+1)$-dimensional vector $\overline{h} = (n_1, \ldots, n_d, -c)$ defines a unique hyperplane and is called a **determining vector** of that hyperplane. A **normal** to H^{d-1} is a vector parallel to $\overline{n}$. Every hyperplane defines two closed **half-spaces** whose intersection is the hyperplane itself and whose union is the space R^d. A **supporting hyperplane** of a set S is a hyperplane that intersects S and S is contained in one of the two closed

half-spaces defined by the hyperplane.

A set S is **convex** if for each pair of points x, y from S, all points in the straight line segment between x and y also belong to S. A **convex hull** of a set S, denoted as $CH(S)$, is the intersection of all convex sets containing S. A convex hull of a finite d-dimensional set of points is called a **d-polytope**. A finite set determines a unique bounded convex polytope.

The objective of a convex hull algorithm is the description of a convex hull polytope. A polytope can be described by means of its boundry which consists of faces. A **face** of a polytope P is the intersection of P with its supporting hyperplane. A **k-face** of P is a k-dimensional face of P. A $(d-1)$-dimensional face of a polytope is called a **facet**, a $(d-2)$-dimensional face is called an **edge**, a $(d-3)$-dimensional face is a **ridge** and a 0-dimensional face is a **vertex**. Polytope P itself is considered a d-face of P and the empty set is a (-1)-face of P. For every face f of P, f is the convex hull of all the vertices of P that are contained in f and f is also the intersection of all the facets of P that contain f.

Some types of polytopes deserve special attention. A **d-simplex** (or briefly, a **simplex**) is a convex hull of $(d+1)$ affinely independent points. It is the simplest type of a d-polytope. In two and three dimensions they are triangles and tetrahedrons respectively. A **simplicial polytope** is a polytope each of whose

facets are simplices[1] (i.e. $(d-1)$-dimensional faces containing exactly d vertices). A d-polytope whose every vertex is incident with exactly d edges is called a **simple polytope.**

Faces of a polytope can be graphically represented by a **facial graph.** It is an acyclic directed graph with one source and one sink. The nodes of this graph are in one-to-one correspondence with the faces of the polytope and there is an arc from k-dimensional face f to $(k-1)$-dimensional face g if g is contained in f. In this case g is called a **subface** of f which we denote as $(g\ sub\ f)$, and f is called a **superface** of g denoted as $(f\ super\ g)$. The Facial graph of a polytope P will be denoted as $fg(P)$. The size of the facial graph of P is the number of arcs plus the number of nodes in $fg(P)$.

There are three fundamental versions of the convex hull problem:

The Facet Problem

Given a set S of n points in R^d, enumerate all facets of $CH(S)$, where each facet is represented by the set of vertices contained in this facet.

The Vertex Problem

Given a set S of n points in R^d, identify those points of S that are vertices of $CH(S)$.

---

[1]Plural of simplex.

The Facial Lattice Problem

Given a set S of n points in R^d, produce the complete facial lattice of $CH(S)$, i.e. all the faces along with their inclusion relationship.

The Facet Problem and the Facial Lattice Problem are asymptotically at least as hard as the Vertex Problem since the output of the former becomes a valid solution to the Vertex Problem. For the same reason the Facial Lattice Problem is asymptotically at least as hard as the Facet Problem.

Because many practical applications of the convex hull construct require facet enumeration, we have focussed on solving the Facet Problem and the Facial Lattice Problem only.

1.2 Computational Complexity and Lower Bound

For an arbitrary finite set of n points in the plane, computing the convex hull is known to have $\Omega(n\log n)$ lower bound [Yao(1981)] which is restricted to the quadratic decision tree model. Several algorithms achieve this lower bound. Since any set of points in R^2 can be trivially embedded in R^d for $d > 2$, a lower bound result obtained for $d = 2$ remains a valid lower bound for $d > 2$ as well [Preparata-Shamos(1985)].

When establishing a bound to the running time of a convex hull algorithm, it is not only the input size that plays an important role, but also the size of the output produced. This is due to the fact that for a d-polytope with n vertices in R^d the number of facets can be as high as $O(\lfloor d/2 \rfloor ! n^{\lfloor d/2 \rfloor})$ [McMullen(1970)] and as low as $\Omega(n)$ [Barnette(1973)], which for $d > 3$ becomes a significant range. It is therefore desirable to include the size of the output as an additional measure to the time complexity function. The following section provides a limited overview of the convex hull algorithms known to us for data sets in dimensions higher than two.

1.3 Solving the Facet Problem and the Facial Lattice Problem in Higher Dimensions

In this section we will be talking about on-line and off-line convex hull algorithms. An algorithm which requires all of the data points to be present before any processing begins is called **off-line.** In many geometric applications, particularly those that run in real-time, this condition cannot be met. The computation must be done as the points are being received. In general, an algorithm that cannot look ahead at its input is referred to as an **on-line.**

Finding the convex hull of a finite set of points in R^d was one of the first problems explored in the field of computational geometry. A variety of algorithms have been proposed and analyzed for the planar convex hull problem [Graham(1972), Jarvis(1973), Preparata-Hong(1977), Bentley-Shamos(1978), Akl-Toussaint(1978-1), Akl-Toussaint(1978-2), Kirkpatrick-Seidel(1986)]. Graham's(1972) algorithm was historically the first publication to show that the planar convex hull can be computed in $O(n\log n)$ time in the worst case, which was later proved by Yao(1981) to be the optimal time.

For a set of points in R^3 Preparata and Hong(1977) presented an algorithm which is based on what is known as the "divide and

conquer" principle. The strategy employed is that the problem is first subdivided into subproblems of the same kind (divide), the subproblems are then recursively solved (conquer), and finally, the resulting convex hulls are combined to form a global solution (merge). The merge step is a crucial component of this method. Any algorithm that is based on the "divide and conquer" principle is efficient only if the solutions to the subproblems can be combined quickly. Let P_1 and P_2 be two non-intersecting convex hulls. To merge P_1 and P_2 means to determine the convex hull $CH(P_1,P_2)$ of P_1 and P_2. This is accomplished by constructing a "cylindrical wrap" which supports P_1 and P_2 and by removing from both P_1, and P_2 the respective portions which become internal to the resulting polytope. Preparata and Hong(1977) believed that the construction of this wrap is entirely guided by a circular sequence of vertices and edges which are successively acquired by the advancing steps of the wrapping process. As was later noticed by Edelsbruner(1987) this may not always be true - "a single vertex of a recursively constructed convex polytope can be encountered more than once when it is merged with another disjoint convex polytope". For a set of n points in R^3 the worst case computational complexity of this algorithm is $O(n\log n)$ which is optimal by Yao's(1981) results. This approach to solving the convex hull problem has not yet been extended to dimensions higher than three.

The first general algorithm offering a method for solving the Facet Problem in any dimension $d \geq 2$ was described by Chand and

Kapur(1970). Their idea is based on the observation that exactly two facets of a convex polytope intersect along one edge. The algorithm uses the so called "gift wrapping" principle where the polytope is generated by systematically computing the facets from the edges of the desired convex polytope. The computational complexity of this algorithm was analyzed by Bhattacharya(1982) who showed that in the worst case the time to compute the convex hull of n points in R^d is bounded above by $O(dnF_{d,n} + d^3F_{d,n}\log n)$, where $F_{d,n}$ is the number of facets of the computed polytope. A major drawback of this algorithm is that it computes correctly simplicial polytopes only. When more than d points lie in a convex hull facet, the determination of the edges associated with this facet is equivalent to determining the convex hull of the points contained in the facet. This may be considered as a convex hull problem in $(d-1)$-dimensional space and Chand and Kapur's algorithm can be applied again to solve it. However, due to the recursive nature of this approach the implementation is difficult. Swart(1985) later applied the "gift wrapping" principle to produce a structured representation of the convex hull, the facial lattice. When applied to this problem it has the worst case time complexity $O(dnL_{d,n} + d^5L_{d,n}\log n)$ for simplicial polytopes or $O(d^2nL_{d,n} + d^4L_{d,n}{}^2\log n)$ for non-simplicial ones, where $L_{d,n}$ is the size of the facial lattice produced.

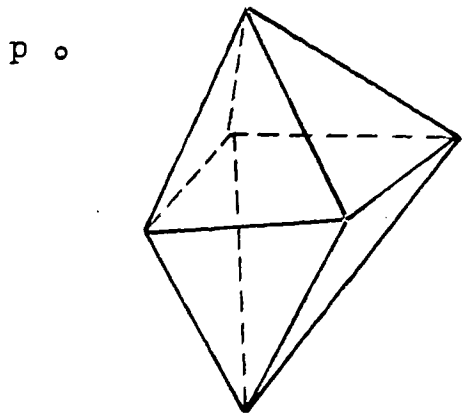
For a long time the "gift wrapping" method was the only known general technique to compute the convex hull of a finite point

set in d-dimensional space. A new technique, dubbed by Preparata and Shamos(1985) as the "beneath beyond" method was proposed independently by Kallay(1981), Seidel(1981) and Jozwik(1983) and was later also adopted by Rey and Ward(1985) in their convex hull algorithm. In a non-mathematical way it can be described as follows.

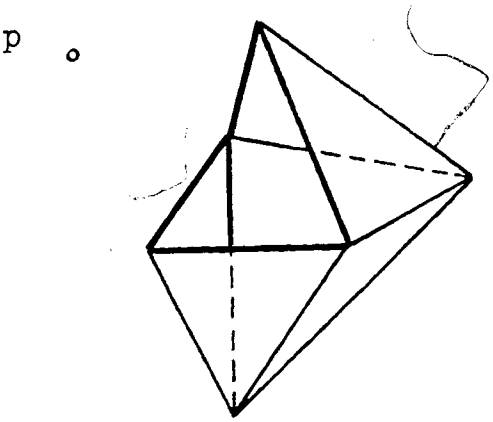
Given, a convex hull P of some point set and a new point p. Imagine point p sending out an intense light. If p is external to the current convex hull then all facets of P that receive light from this point are discarded, leaving the convex hull as an open shell with some "exposed edges" (i.e. the edges surrounding the opening of the shell). Every "exposed edge" together with the point p determines a new facet [Fig. 1.2]. If p is not external to the current convex hull, then p is already contained in P and need not be considered any further - p becomes a "throw away" point. The approach of this technique is incremental, meaning that the points are considered one at a time and the convex hull is updated every time a point, lying outside the convex hull computed so far, is encountered. This technique exhibits, in addition, the on-line property desirable by many applications.

Seidel's(1981) algorithm produces a facial lattice of the convex hull and the approach employed here is analogous to the "beneath beyond", only in the dual space. To achieve the best possible running time a point, when it is considered, has to lie outside of the current convex hull. To guarantee this condition, the

Current convex hull and a new point p.



Convex hull as an open shell with "exposed edges".



Updated convex hull to include p.

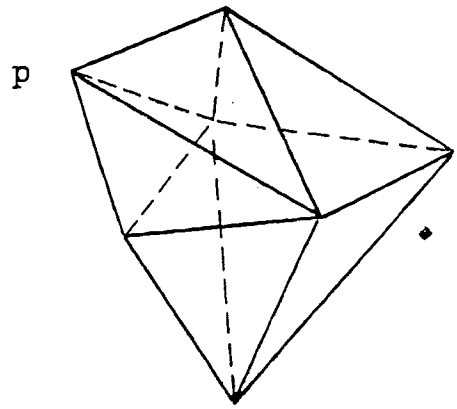


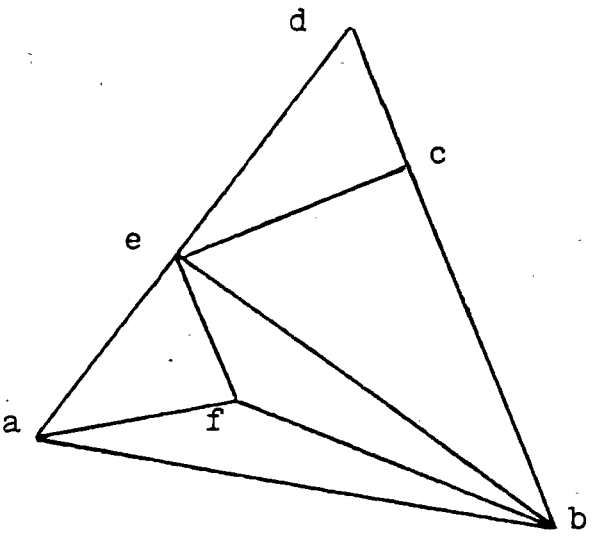
Fig. 1.2: Updating convex hull with a new point in R^3.

first step of Seidel's algorithm pre-sorts the initial point set into a lexicographical order. This means that the entire point

set has to be known in advance and the incremental approach used here for computing the convex hull loses its on-line property. Also, as a result of pre-sorting, every new point causes an update of the convex hull, which may be seen as a disadvantage of this approach (i.e. no "throw away" points). For a fixed dimension d, the worst case computational complexity of Seidel's algorithm is $O(n\log n + n^{\lfloor (d+1)/2 \rfloor})$ where, for even d, $O(n^{\lfloor (d+1)/2 \rfloor})$ is assymptotically equivalent to the largest possible output size. Seidel(1981) also argued that the algorithm is in the worst case optimal for even d. As was later shown by Swart(1985) this is true only if the complexity of the problem is measured in terms of the input size n alone. If the complexity is measured in terms of the actual output size, the algorithm can be far from optimal. Kallay(1984) later showed that the complexity of any incremental convex hull algorithm for n points in R^d is $\Omega(n^{\lfloor (d+1)/2 \rfloor})$ for a fixed d. This result makes Seidel's(1981) algorithm worst case optimal (in terms of the input size) for odd d as well.

Rey and Ward's(1985) algorithm solves the Facet Problem of a convex hull. This algorithm can be applied to compute non-simplicial facets but such facets are described as unions of several simplices [Fig. 1.3]. This solution to the problem of degeneracies is theoretically incorrect. In Rey and Ward's implementation, to establish that a new point is either interior to P_i or that this new point is external to P_i, the entire list of facets of P_i has to be searched. Keeping in mind that the

Facet (a,b,d) may be represented as a union of simplices (a,b,f), (b,c,e), (c,d,e), (a,f,e) and (b,e,f).



Non-simplicial facet (a,b,c,d,e) may be represented as a union of simplices (a,b,c), (a,c,e) and (c,d,e).

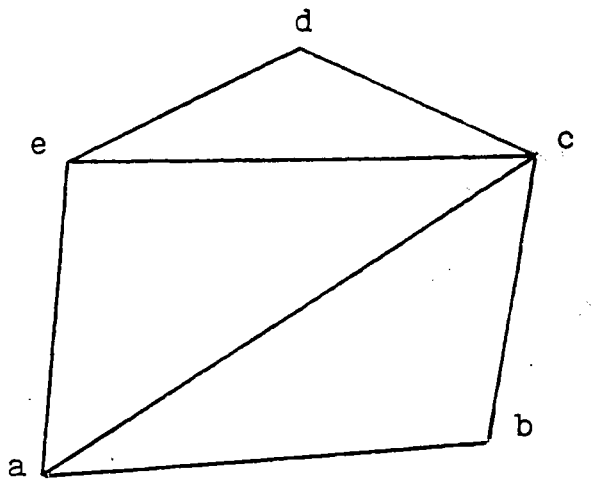


Fig. 1.3: Representation of a facet as a union of simplices.

number of facets, $F_{d,i}$, of a d-polytope with i vertices can be as high as $O(\lfloor d/2 \rfloor ! i^{\lfloor d/2 \rfloor})$ this "point inclusion" test in a convex polyhedron is of $\Theta(F_{d,i})$ complexity in the worst case and also on average, which is considered to be a major drawback of this implementation. For a fixed d, the worst case computational complexity of the algorithm is $O(n^{\lfloor d/2 \rfloor + 1})$. This worst case is the same as the worst case of Chand and Kapur's(1970) method.

Preparata and Shamos(1985) present an implementation of an on-line version of the "beneath beyond" technique for higher dimensions. Their implementation has the same drawback as Rey and Ward's(1985), namely the $\Theta(F_{d,i})$ "point inclusion" test. The algorithm solves the Facial Lattice Problem for a fixed dimension in the worst case time $O(n^{\lfloor d/2 \rfloor + 1})$. This time applies to degenerate cases as well.

Edelsbruner(1987) gives yet another description of the implementation of the "beneath beyond" technique for higher dimensions. This implementation is similar to Seidel's(1981) but no transformation to dual space is involved. The algorithm produces a facial lattice of a convex hull and can handle degeneracies. As in Seidel's implementation the entire point set is pre-sorted as a result of which every new point causes an update of the convex hull. The worst case computational complexity can be measured in terms of the input size only and it is the same as the worst case of Seidel's(1981) algorithm. According to Kallay's(1984) result, this implementation of an incremental technique is optimal.

The latest algorithm for computing the convex hull of a finite set of points in higher dimensions has again been proposed by Seidel(1986). It introduces a new technique, a straight line shelling of a polytope. For a set S of n points in R^d, the first step of the algorithm involves solving a linear program in $(d-1)$ variables and $(n+1)$ constraints for every point of S. Due to this step there may be objections to the practicality of this

approach which even Seidel considers to be well founded. The algorithm either enumerates all the facets of a convex hull, assuming that the convex hull is simplicial, or it constructs a facial lattice of a convex hull in the worst case time complexity of $O(n^2 + F_{d,n}\log n)$ or $O(n^2 + L_{d,n}\log n)$ respectively, for a fixed d. If the complexity of the problem is measured only in terms of the input size n, then this algorithm has the worst case time $O(\lfloor d/2 \rfloor ! n^{\lfloor d/2 \rfloor}\log n)$ which is the best worst case bound known for any technique for odd $d > 3$.

Although the objective may be, in many instances, the computation of just the facets of a convex hull, the problem of degenerate data sets has been solved by maintaining a facial graph of all the faces. One open problem still remains: the existence of an algorithm that would solve the Facet Problem for a general data set, without maintaining some description of all the faces, in time polynomial in n, d and $F_{d,n}$. Swart(1983) elaborates on this problem and conjectures that it may be possible to do so in time $O((F_{d,n})^2 d^{d+4}\log n)$, which is exponential in d.

One of the desirable properties of a convex hull algorithm is the on-line property. An on-line convex hull algorithm has to deal with the "point inclusion" problem. The existing on-line higher dimensional algorithms [Preparata-Shamos(1985), Rey-Ward(1985)] solve this problem in $\Theta(F_{d,i})$ time on average and in the worst case, where $F_{d,i}$ is the number of facets of a convex hull determined by a set of i points in R^d. Seidel(1981)

and Edelsbruner(1987) use an incremental on-line technique in their algorithm, but to avoid the time consuming "point inclusion" test, they pre-sort the entire point set and the resulting algorithms are off-line. In this thesis we show how the "point inclusion" problem can be solved in $\Theta(d\log F_{d,t})$ expected time. We also show how this result can be used to get a fast on-line convex hull algorithm and how the off-line "divide and conquer" technique, already used to compute convex hulls of two and three dimensional data sets, can be extended to higher dimensions.

The on-line algorithm we are proposing uses the "beneath beyond" technique and can be applied to a non-simplicial data set. It can be implemented to solve either the Facet Problem in the worst case time $O(d^4 nF_{d-1,n}F_{d-2,n})$, or the Facial Lattice Problem in $O(dnF_{d-1,n}\log F_{d,n})$ expected time, or $O(dnF_{d,n}\log F_{d,n})$ time in the worst case. This expected time is the best expected time of an on-line higher dimensional convex hull algorithm known to us. Our on-line algorithm is also the only one known to us that solves the Facet Problem for a non-simplicial data set without maintaining some description of all the faces of a convex hull. The off-line algorithm, based on the "divide and conquer" technique, solves the Facet Problem for a simplicial data set and its computational complexity is $O(dnF_{d,n})$ in the worst case.

In the following chapter we show how the "point inclusion" problem in a closed convex polyhedron P, determined by a set of

n points in R^d, can be solved in $\Theta(d\log F_{d,n})$ expected time, where $F_{d,n}$ is the number of facets of P. Chapter III describes and analyzes the computational complexity of the on-line convex hull algorithm for each one of the two problems (i.e. Facial Lattice Problem and Facet Problem). The convex hull algorithm based on the "divide and conquer" principle is presented and analyzed in Chapter IV. In Chapter V we discuss some experimental results and in the last chapter we summarize the work submitted in this thesis and point to some relevant questions open to further research.

CHAPTER II

THE POINT INCLUSION PROBLEM IN A CONVEX POLYHEDRON

The point inclusion problem in a convex polyhedron can be stated as follows:

> Given a closed convex polyhedron P in R^d, determine if an arbitrary point p lies inside of P.

The solution to this problem, which we propose in this chapter, is based on the properties of an important geometric construct, the Voronoi diagram [Voronoi(1908)]. We construct the Voronoi diagram as a first step of a transformation, where the point inclusion problem is transformed to a nearest neighbour problem, and we use a d-dimensional tree data structure [Bentley(1975), Friedman-Bentley-Finkel(1977)] to search for the nearest neighbour.

2.1 The Voronoi Diagram

One of the earliest definitions of what we now call the Voronoi diagram can be found in Dirichlet(1850). More than half a century later, mathematician G. Voronoi(1908) was the first to study this diagram in details. Some other names such as

Dirichlet(1850) tessellations, Thiessen(1911) polygons and Wigner-Seitz(1933) cells have been used in the literature, all referring to the same diagram. In the remainder of this section we derive the definition of this geometric construct.

Once more, let S be a set of n distinct points $p_1, \ldots, p_n$ in R^d. For any two points p_i, p_j with $i \neq j$, the locus of points equidistant to p_i and p_j is the perpendicular bisector $B(p_i, p_j)$ of the line segment joining p_i and p_j. $B(p_i, p_j)$ determines two closed half-spaces $H(p_i, p_j)$ and $H(p_j, p_i)$ the intersection of which is $B(p_i, p_j)$. The locus of points that lie as close or closer to p_i than to p_j is the closed half-space $H(p_i, p_j)$ that contains p_i[1]. If $p_i \neq p_j$, $H(p_i, p_j)$ and $H(p_j, p_i)$ are uniquely determined by $B(p_i, p_j)$. The locus of points at least as close to p_i as to any other point of S is then the intersection of the $(n-1)$ closed half-spaces $H(p_i, p_j)$ for $j = 1, \ldots, n$ and $j \neq i$. We denote this region by V_i and it is not hard to see that V_i is a convex region. In dimensions higher than two V_i is called the **Voronoi polyhedron** associated with p_i and p_i is the **generating point** of V_i. Each point of S is enclosed in a unique Voronoi polyhedron. For every p_i from S, let V_i be the Voronoi polyhedron associated with p_i. The n regions $V_1, \ldots, V_n$ partition R^d into a set of convex polyhedra and are referred to as the **Voronoi diagram** of S [Fig. 2.1].

---

[1]The word "close" in this context means the Euclidean distance between two points.

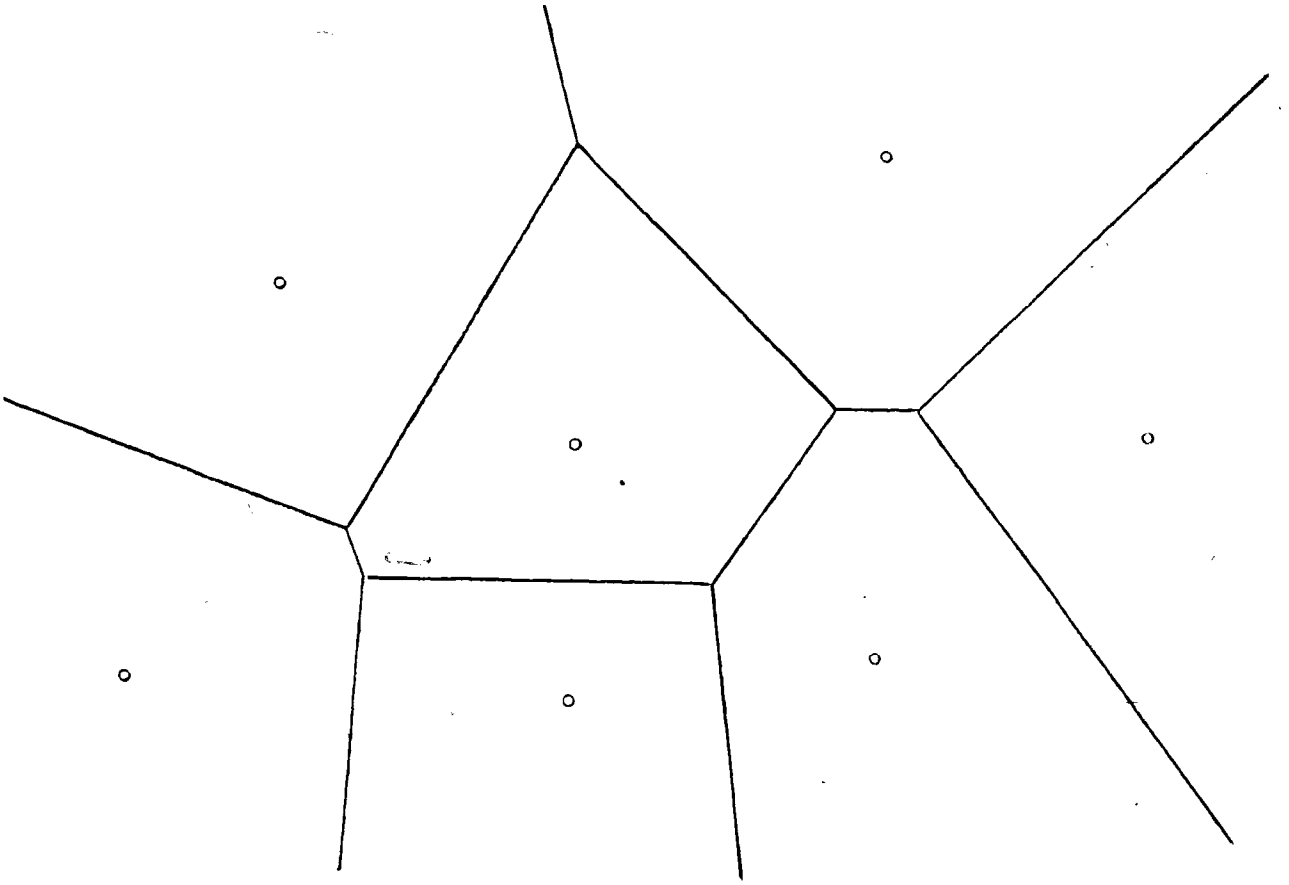


Fig. 2.1: Voronoi diagram of a set of points in R^2.

2.2 The *d-D* Tree Data Structure

The *d-D* tree is a generalization of a simple binary tree [Friedman-Bentley-Finkel(1977)]. In our implementation the *d-D* tree is built over a set of points. Every node of the tree represents a subset of the points in the set and a partitioning of that subset. The root of the tree represents the entire set. Each non-terminal node has two son nodes. These son nodes

represent the two subsets defined by the partitioning in the parent node. The terminal node, often called a **bucket**, represents either a single point from the set or a small subset of points.

In d-dimensions, a point is represented by its d real valued coordinates. Any one of the d-dimensions can be used as a discriminator for partitioning the subset represented by a particular non-terminal node. In Friedman-Bentley-Finkel(1977) both the discriminator and partition value for each non-terminal node, as well as the bucket size for terminal nodes, are chosen to obtain the best expected cost of searching for nearest neighbour(s). This yields what is called the optimized d-D tree. The prescription for optimizing is to choose at every non-terminal node the dimension with the largest spread in coordinate values as the discriminator and to choose the median of the coordinate values of the discriminator as the partition. At the level of terminal nodes, the bucket size should be made as small as possible. The effect of the optimized d-D tree partitioning is a division of the coordinate space into approximately hypercubical subregions, each containing very nearly the same number of points. To minimize the upper bound on the number of points examined by each search, the buckets should each contain one point only [Friedman-Bentley-Finkel(1977)].

The geometric boundaries of the subregion of every non-terminal node are determined by the partitions defined at the nodes above it in the tree. The volume of these subregions is smaller for

subsets defined by nodes deeper in the tree. The geometric boundary of the root node is defined as plus and minus infinity on every dimension. When search is performed, if the node under investigation is terminal, all the points in the bucket are searched for the nearest neighbour, and the point found to be the closest is maintained. If the node under investigation is not terminal, the recursive procedure is called for the node representing the subset on the same side of the partition as the query point. When control returns, a test is made to determine if it is necessary to consider the points on the side of the partition opposite to the query point. It is necessary to consider that subset only if the geometric boundaries delimiting this subregion intersect the ball centered at the query point with the radius equal to the distance to the closest point encountered so far. This is referred to as the "bounds overlap ball" test. If this test fails, none of the points on the opposite side of the partition can be the closest neighbour to the query point. If the bounds do overlap the ball then the points of that subtree must be considered and the procedure is called recursively for the node representing that subset. A "ball within bounds" test is made before returning to determine if it is necessary to continue the search. This test determines whether the ball is entirely within the geometric boundaries of the subregion represented by the node. If so, the current nearest neighbour is correct for the entire set and the search can be terminated. This recursive search procedure is described in detail in Friedman-Bentley-Finkel(1977).

The computation time required to organize the *d-D* tree data structure over a set of *n* points in R^d is proportional to *dn*log*n* and the expected computation time to perform each search is proportional to log*n*. This expected search time is independent of the probability distribution of the points in R^d and for large data sets, the expected number of point examinations required by the search (i.e. the number of terminal nodes searched) is shown to be independent of the value of *n* [Friedman-Bentley-Finkel(1977)].

In our implementation of the *d-D* tree, a bucket contains one point only and every non-terminal node represents not only a subregion of the coordinate space but also a cell within this subregion. A cell is determined by a set of points contained in the subregion and its geometric boundaries tightly enclose this set of points [Fig. 2.2]. When "bounds overlap ball" test is performed, it is the geometric boundaries delimiting the cell that are considered rather than the boundaries of the subregion that contain the cell. During the search, if a node under investigation is not terminal, the recursive procedure is called for the node representing the subset on the same side of the partition as the query point only if the "bounds overlap ball" test is true. In many instances, the boundaries of a subregion may overlap the ball, but the boundaries of a cell contained in it may not. As we show in Chapter V this change greatly improves the average number of nodes considered when search is performed. The expected computation time to perform each search is, in our

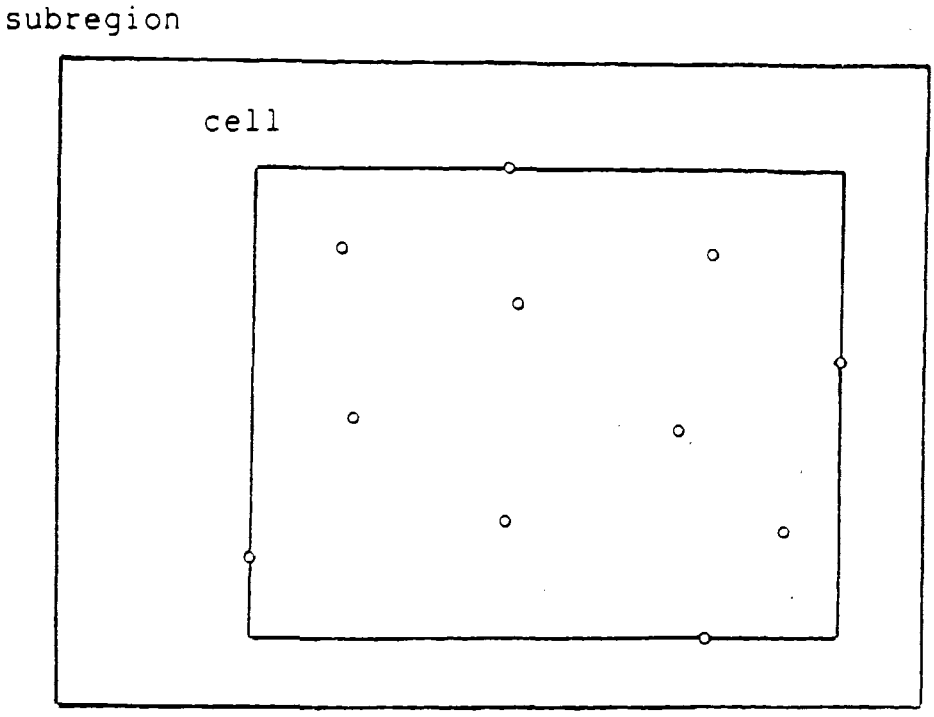


Fig. 2.2: Subregion and a cell in R^2 as represented by a non-terminal node of a *d-D* tree.

implementation of the *d-D* tree, proportional to $d \log n$.

2.2.1 Dynamic *d-D* Tree

To implement our on-line convex hull algorithm, we must be able to maintain the *d-D* tree data structure dynamically. To our best knowledge this has not been done yet and we are proposing the

following way of updating the tree.

To insert a new point into the *d-D* tree structure, the tree is traversed from the root node down to the terminal node, along the path determined by the nodes representing the subset on the same side of the partition as the new point. When a terminal node is reached, this node is changed to non-terminal and the point which the terminal node represented plus the new point become the sons of the newly created non-terminal node. In this non-terminal node the dimension with the largest spread in coordinate values is chosen to be the discriminator and the lesser value of the (two) coordinate values of the discriminator determines the partition.

To delete a point from the *d-D* tree, let *Node* be the terminal node representing this point. The following procedure details the deletion of *Node*.

Delete *Node*

Let *NodeParent* be the parent of *Node* and let *OtherSon* be the son of *NodeParent* such that *OtherSon* ≠ *Node*.

NodeParent = *OtherSon*

Delete *Node*

Delete *OtherSon*

This guarantees that every non-terminal node has two sons and therefore there is no unnecessary partitioning (i.e. one side of a partition being empty).

When a *d-D* tree is updated, the geometric boundaries of the cells associated with the non-terminal nodes along the path from the terminal node back to the root node may have to be adjusted to reflect this update. The volume of a cell can therefore "grow" after an insertion or it may "shrink" as a result of a deletion, but the region containing this cell remains unchanged.

The amount of time it takes to update a *d-D* tree is proportional to dH, where H is the height of the tree. For a binary search tree, when points are inserted and deleted at random, the height of the tree is found to be proportional to $\log n$, where n is the number of terminal nodes in the tree [Wirth(1976)]. As our test results (presented in Chapter V) show, this is observed to be true for a *d-D* tree data structure as well. Therefore, we assume that an update of a *d-D* tree with n terminal nodes is expected to take $\Theta(d\log n)$ operations.

2.3 Transformation of a Closed Convex Polyhedron into a Voronoi Polyhedron

Let P be a closed convex polyhedron determined by a set of n points in R^d and let x be an arbitrary point interior to P. We assume that P is represented by the set of its facets. For every facet f_i of P we generate point q_i such that $aff(f_i)$ becomes a

perpendicular bisector of a line segment joining x and q_i. Thus $aff(f_i)$ partitions R^d into two closed half-spaces: $H(x,q_i)$ containing x and $H(q_i,x)$ containing q_i. P can then be expressed as the intersection of all $H(x,q_i)$ for $i = 1, \ldots, F_{d,n}$, where $F_{d,n}$ is the number of facets of P. We have thus transformed P into a Voronoi polyhedron enclosing its generating point x [Fig. 2.3]. For a given point p we can now use the nearest neighbour information using the set $\{q_1, \ldots, q_{F_{d,n}}, x\}$ to answer the point inclusion problem in a closed convex polyhedron P. To determine if a new point p lies inside of P we store the generated set of q-points $Q = \{q_1, \ldots, q_{F_{d,n}}\}$ in a d-D tree data structure and we search the tree to determine if a point from Q is closer to p than x is. In other words we are determining if there exists a point in Q that lies inside the hypersphere centered at p with the radius equal to the euclidean distance between the points p and x. As soon as we find such a point we know that p is outside of P and the search can be terminated. If no such point exists, it means that x is the point closest to p and p is therefore contained in P.

The expected time to solve the point inclusion problem in a convex polyhedron P, determined by a set of n points in R^d, is proportional to $d\log F_{d,n}$, where $F_{d,n}$ is the number of facets of P. Since $F_{d,n}$ is no higher than $O(\lfloor d/2 \rfloor ! n^{\lfloor d/2 \rfloor})$, the point inclusion problem can be solved in the expected $O(d^2 \log \lfloor d/2 \rfloor ! n)$ time.

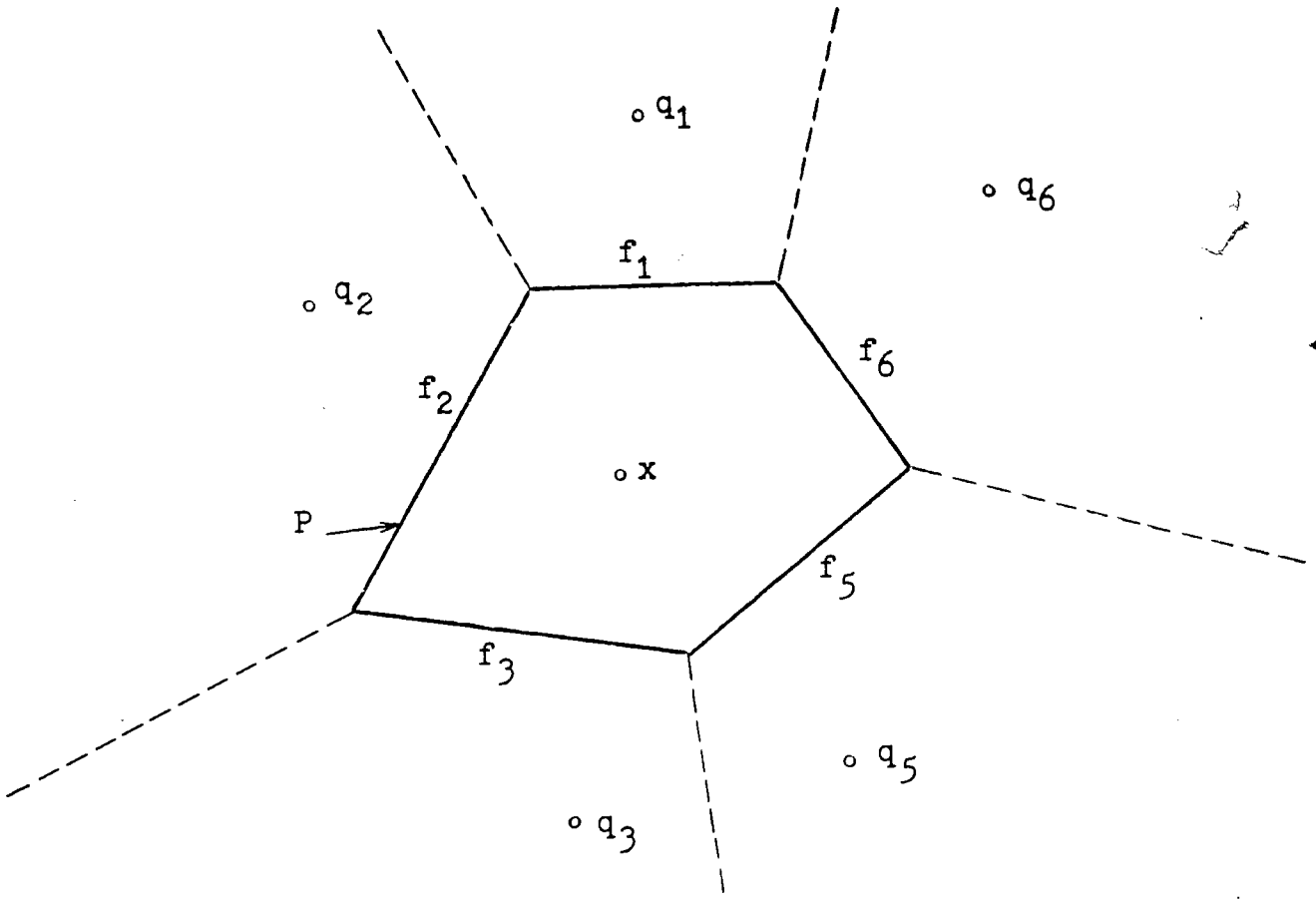


Fig. 2.3: Polygon *P* transformed into a Voronoi Polygon.

CHAPTER III

THE ON-LINE CONVEX HULL ALGORITHM

In this chapter we present an on-line convex hull algorithm that can solve both the Facet Problem and the Facial Lattice Problem for a data set of any dimension $d \geq 2$. The algorithm is based on the "beneath beyond" technique, the global strategy of which has been described in Chapter I.

3.1 Implementation Details and Computational Complexity

Given set S of n points, $S \subseteq R^d$, let S' be a subset of S of $(d+1)$ affinely independent points $p_i = (p_{i1}, \ldots, p_{id})$ for $i = 1, \ldots, d+1$. The first step of the algorithm determines S' and computes the initial convex hull, the d-simplex $P_{d+1} = CH(S')$.

Since every facet is contained in a unique supporting hyperplane, we are using the determining vector of that hyperplane to represent a facet. Every set of d distinct points of S' determines a facet of P_{d+1} for which the determining vector of its supporting hyperplane needs to be computed. The following section describes how this can be accomplished.

Let (w.l.o.g.) f_{d+1} be the facet of P_{d+1} determined by vertices $p_1, \ldots, p_d$ and $\bar{n} = (n_1, \ldots, n_d, -c)$ be the determining vector of the supporting hyperplane H^{d-1} of f_{d+1}. Then by Definition 1.1, for every point p_i, $i = 1, \ldots, d$

$$n_1 p_{i1} + \ldots + n_d p_{id} - c = 0. \qquad [3.1]$$

When $c = 0$, hyperplane H^{d-1} passes through the origin of a coordinate system. By transforming the origin to an arbitrary point x interior to P_{d+1} we eliminate this possibility[1]. This transformation also increases the numerical stability of the algorithm. According to Solomon(1985) "it is best to carry out all geometric computations as near to the origin as possible".

We can then divide [3.1] by c to obtain

$$h_1 p_{i1} + \ldots + h_d p_{id} - 1 = 0$$

where $h_i = n_i / c$. The determining vector $\bar{h} = (h_1, \ldots, h_d, -1)$ can now be uniquely computed by solving the linear system of equations

$$h_1 p_{i1} + \ldots + h_d p_{id} - 1 = 0$$

for $i = 1, \ldots, d$, where $(p_{i1}, \ldots, p_{id})$ are the transformed coordinates of a point p_i.

Let P_{i-1} be the convex hull polytope computed so far. When a new

---

[1]To determine x one can, for example, compute the centroid of S'.

point p_i is considered the algorithm performs the "beneath beyond" test to determine the position of p_i with respect to the facets of the current convex polytope P_{i-1}. To be able to perform this test the determining vector $\overline{h}$ of the supporting hyperplane of every facet f of P_{i-1} is computed in such a way, that for every point p from P_{i-1}, $(p,\overline{h}) \leq 0$. Geometrically this means that the normals to the supporting hyperplanes of all facets of P_{i-1} point outside of P_{i-1}. This is an important requirement for the following definition of the "beneath beyond" test.

Definition 3.1

A point p lies **beneath** (**beyond**) a facet f of a polytope P if and only if $(p,\overline{h}) < 0$ ($(p,\overline{h} > 0)$, where $\overline{h}$ is the determining vector of a supporting hyperplane of a facet f.

Polytope P_{i-1} needs to be updated to include point p_i only if p_i lies beyond some facet(s) of P_{i-1}. To determine if an update is necessary, we solve the point inclusion problem in a convex polyhedron P_{i-1} for point p_i and we apply the solution described in the previous chapter to solve this problem.

Let x be a point interior to P_{i-1}, $F_{d,i-1}$ the number of facets of P_{i-1} and let $Q = \{q_1, \ldots, q_{F_{d,i-1}}\}$ be the set of generated q-points such that for every facet f_j of P_{i-1}, $1 \leq j \leq F_{d,i-1}$, $aff(f_j)$ is the perpendicular bisector of the line segment joining x and q_j. If x is at least as close to p_i as any point

from Q, then no update of P_{i-1} is necessary. However, if for some j, $1 \leq j \leq F_{d,i-1}$, q_j is closer to p_i than x is, then facet f_j is such that p_i lies beyond it.

3.1.1 The Algorithm

Input: Set S of n points, $S \subseteq R^d$, and dimension d.

Output: Description of $P_n = CH(S)$.

Step 1: {Compute d-simplex}.

1. Determine $S' \subseteq S$ of any $(d+1)$ affinely independent points and set $S'' = S - S'$.
   Let (w.l.o.g.) $S' = \{p_1, \ldots, p_{d+1}\}$ and $S'' = \{p_{d+2}, \ldots, p_n\}$.
2. Determine point x that is interior to the d-simplex defined by S'.
3. Translate the points of S' and S'' such that the origin is at x.
4. *Compute* P_{d+1} where $P_{d+1} = CH(S')$.
   Let $\{f_1, \ldots, f_{d+1}\}$ be the set of facets of P_{d+1}.
5. For $i = 1$ to $(d+1)$ do
   a. Compute the determining vector of $aff(f_i)$.
   b. Compute point q_i such that $aff(f_i)$ becomes a perpendicular bisector of the line segment joining q_i and x.

6. Construct d-D tree over the set $\{q_1, \ldots, q_{d+1}\}$.

Step 2: {Process points of S"}.

Let $F_{d,i-1}$ be the number of facets of P_{i-1}.

1. For $i = (d+2)$ to n do

   a. Search d-D tree for a point that is closer to p_i than x is.

      If x is the closest point to p_i then

         $P_i = P_{i-1}$.

      Else

         Let q_j, $1 \leq j \leq F_{d,i-1}$, be the point that is closer to p_i than x and let f_j be the facet corresponding to q_j
         (i.e. $aff(f_j)$ is the perpendicular bisector of the line segment joining x and q_j).

         Update P_{i-1} *to include* p_i.

Step 3: {Clean Up}.

   For every vertex p of P_n do

      Undo the translation performed in Step 1.

The details of procedures *Compute* P_{d+1} in Step 1 and *Update* P_{i-1} *to include* p_i in Step 2 depend on the particular version of a convex hull problem the algorithm is solving and we will therefore describe them separately.

3.1.2 The Facial Lattice Problem

We are using facial graph to represent faces of a convex hull polytope. The initial facial graph contains one node only which corresponds to the empty set. The facial graph of a d-simplex consists of $2^{(d+1)}$ nodes, and it can be constructed by introducing one point of S' at a time. It takes exactly $(d+1)$ iterations before the polytope reaches full dimension d. We first obtain 0-face, then 1-face, and so on. The construction mechanism of the facial graph of a d-simplex is based on the following theorem and lemma:

Theorem 3.1 [Grünbam(1967)]

Let P be a convex polytope in R^d of dimension $k < d$, and let p be a point that is not contained in $aff(P)$. Define $P' = CH(P \cup \{p\})$. Then each face of P' is one of the following types:

1. a face f of P is also a face of P' or
2. if f is a face of P then $f' = CH(f \cup \{p\})$ is a face of P'.

Polytope P', as defined in Theorem 3.1, is usually called a **pyramid** with **base** P and **apex** p.

Lemma 3.1 [Grünbaum(1967)]

Let P and P' be defined as in Theorem 3.1. For faces f and g of P, define $f' = CH(f \cup \{p\})$ and $g' = CH(g \cup \{p\})$.

Then

1. (f *sub* g) in P' if and only if (f *sub* g) in P,
2. (f *sub* g') if and only if f=g, and
3. (f' *sub* g') if and only if (f *sub* g).

The following is an outline of the procedure *Compute* P_{d+1} in Step 1.

Compute P_{d+1}

Let $P_0 = \varnothing$.

For $i = 1$ to $(d+1)$ do

Construct $fg(P_i)$ where $P_i = CH(P_{i-1} \cup \{p_i\})$

[see "Pyramidal Update", Edelsbruner(1987)].

To construct $fg(CH(P_{i-1} \cup \{p_i\}))$ from $fg(P_{i-1})$ after P_{i-1} reaches dimension d, Edelsbruner(1987) introduces a coloring scheme which he uses to classify facets according to their relative position to point p_i. For f, a facet of P_{i-1}, this classification is defined as follows:

f is *red* if p_i lies beyond f,

f is *blue* if p_i lies beneath f, and

f is *yellow* if p_i belongs to $aff(f)$ [Fig. 3.1].

No color is ever assigned to the one only d-face of P_{i-1} which is polytope P_{i-1} itself. If f is a k-face of P_{i-1}, with $k < (d-1)$, then f is assigned a color which is determined by the mixture of colors assigned to all facets which contain f in

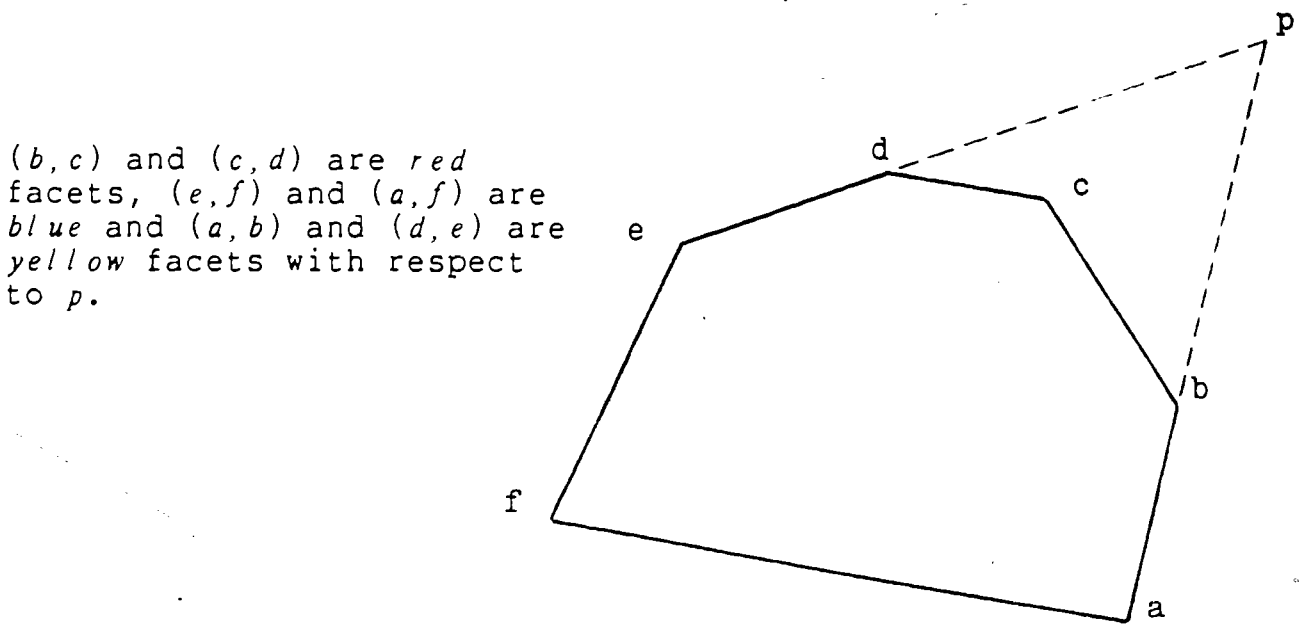


Fig. 3.1: Three types of facets with respect to point *p*.

their boundaries:

f is *orange* if it belongs to the boundary of *red* and *yellow* facets,

f is *green* if it belongs to the boundary of *yellow* and *blue* facets,

f is *purple* if it belongs to the boundary of *red* and *blue* facets, and

f is *brown* if it belongs to the boundary of *red*, *yellow* and *blue* facets.

Additional color groups are defined as follows:

f has a *blue component* if it is *blue*, *green*, *purple* or *brown*,

f has a *red component* if it is *red*, *orange*, *purple* or *brown*, and

f has a *yellow component* if it is *yellow*, *orange*, *green* or *brown*.

With this color scheme we can define the update mechanism.

Theorem 3.2

If P is a d-polytope and p a point in R^d, let $P' = CH(P \cup \{p\})$. Then each face of P' is one of the following types:

1. a face f of P is also a face of P' if and only if f has a *blue component*
2. if f is a face of P then $f' = CH(f \cup \{p\})$ is a face of P' if and only if either
   a. f has a *blue* and a *red component*, or
   b. f is *yellow*.

Proof: [Edelsbruner(1987)]

Theorem 3.2 defines the faces of P' in terms of the faces of P, that is, the nodes of $fg(P')$ in terms of the nodes of $fg(P)$. To completely define $fg(P')$ we also need to define its arcs.

Lemma 3.2

Let P and P' be defined as in Theorem 3.2 and let f and g be two faces of P. Define $f' = CH(f \cup \{p\})$ and $g' = CH(g \cup \{p\})$ if f and g have a *red* and a *blue*

component and $f^* = CH(f \cup \{p\})$ and $g^* = CH(g \cup \{p\})$ if f and g are *yellow*. Then

1. $(f\ sub\ g)$ in P' if and only if $(f\ sub\ g)$ in P and f and g have a *blue component*
2. $(f\ sub\ g')$ if and only if $f = g$
3. $(f\ sub\ g^*)$ if and only if $(f\ sub\ g)$ and f has a *blue component*
4. $(f'$ sub $g')$ if and only if $(f\ sub\ g)$
5. $(f'\ sub\ g^*)$ if and only if $(f\ sub\ l)$ and $(l\ sub\ g)$ for some subface l of g, and
6. $(f^*\ sub\ g^*)$ if and only if $(f\ sub\ g)$.

Moreover, the only d-face of P' (i.e. polytope P' itself) is a superface of all facets of P'.

Proof: [Edelsbruner(1987)]

Lemma 3.3

Let $(f\ sub\ l)$ and $(l\ sub\ g)$ be three faces of a polytope P with f *brown* and g *yellow*. Face l is uniquely determined by faces f and g and the requirement that l has to be *orange*.

Proof: [Edelsbruner(1987)]

From the above information we can see that only *red* and *yellow* facets are important for an update, and we are now going to describe how to identify them. One *red* facet is determined by searching the *d-D* tree. Having discovered one *red* facet, we can

identify all the other *red* and *yellow* facets (if any) by performing depth first search of a facial graph. The search examines nodes that correspond to the facets of a polytope and advances through the arcs corresponding to its edges backtracking every time a *blue* facet is reached. Due to the following theorem, we know that this graph is well defined.

Theorem 3.3

Every edge of a convex polytope lies in two and exactly two facets of this polytope.

Proof: [Chand-Kapur(1970)]

We are now ready to specify the update procedure in Step 2 of the algorithm.

Update P_{i-1} *to include* p_i

1. Determine all *red* and *yellow* facets with respect to p_i using facet f_j as the first *red* facet.
2. For every *red* facet do

   Delete its corresponding q-point from d-D tree.
3. *Construct* $fg(P_i)$ where $P_i = CH(P_{i-1} \cup \{p_i\})$ [see "Non-Pyramidal Update", Edelsbruner(1987)].

   Let $F_{d-1,i-1}$ be the set of all facets that contain p_i in their boundary.
4. For every facet f from $F_{d-1,i-1}$ do
   a. Compute the determining vector of $aff(f)$.
   b. Compute point q such that $aff(f)$ becomes a perpendicular bisector of the line segment joining x

and q.

c. Update d-D tree to contain point q.

Data Structures

A node in the facial graph stores the following information:

SuperFace - points to a list of super faces,

SubFace - points to a list of subfaces,

Color - stores color of a face represented by this node,

NodeCopy - points to a copy of this node
(also used to mark and unmark facets).

In addition, every node representing a facet stores the determining vector of the supporting hyperplane of the facet and a pointer to the terminal node in the d-D tree which corresponds to point q associated with this facet. Every node representing a vertex stores the transformed coordinates of the vertex. In the initial one node of a facial graph (representing the empty set) all pointers are initialized to nil and color to *unspecified*.

A node in the d-D tree stores the following:

Terminal Node

qPoint - an array, coordinates of point q,

Parent - points to the parent node,

Facet - points to the node in the facial graph which corresponds to the facet associated with point q.

Non-Terminal Node

Parent - points to the parent node,

Discr - coordinate chosen as the discriminator,

DiscrValue - value of the discriminator,

LowCellBound - an array, for every coordinate the low bound of a cell,

HighCellBound - an array, for every coordinate the high bound of a cell,

LeftSon - points to the left son node,

RightSon - points to the right son node.

Computational Complexity

Before we analyze the computational complexity of the algorithm, we enlist the following results which will be used in the analysis.

Lemma 3.4

Let P be a polytope with n vertices in R^d, and let p be a vertex of P. The number of faces of P which contain p in their boundary plus the number of incidences among them is $O(F_{d-1,n})$.

Proof: [Edelsbruner(1987)]

Lemma 3.5

Every k-face of a d-polytope P, $k \leq d$, is the intersection of at least $(d-k)$ facets of P.

Proof: [Grünbaum(1967)]

We are now going to analyze the computational complexity of the algorithm for its expected time and its worst case performance in terms of length of the input.

Step 1: Set S' can be determined in $O(nd^3)$ time in the worst case.

Computation of point x requires $O(d)$ operations and the required translation can be applied to all points of S' and S'' in $O(dn)$ time.

Facial graph of P_{d+1} is obtained by induction and it can be constructed in $O(dF_{d,d+1})$ time.

Polytope P_{d+1} has exactly $(d+1)$ facets. Given a facet of P_{d+1}, the determining vector of its supporting hyperplane can be computed in $O(d^3)$ operations and point q corresponding to this facet can be computed in $O(d)$ time.

Finally, the construcion of a d-D tree over a set of $(d+1)$ points can be done in $O(d\log d)$ time.

Running Time: $O(nd^3)$ in the worst case.

Step 2: We will analyze this step for one update only. That is we will determine the time needed to update the existing convex hull P_{i-1} with point p_i.

As we have already mentioned, one *red* facet of P_{i-1} is obtained by searching the *d-D* tree. The number of terminal nodes in the tree is equal to the number of facets of P_{i-1} which is bounded above by $F_{d,i-1}$. The expected time to search the tree is therefore $\Theta(d\log F_{d,i-1})$.

The determination of all the other *red* and *yellow* facets is proportional to the number of *red* and *yellow* facets. The worst case occurs when no *yellow* facets exist [Seidel(1981)], since it is always possible to perturb p_i slightly such that all *yellow* facets become *red* and colors of other facets remain unchanged. As shown by Seidel(1981) this perturbation does not decrease the number of faces or face-subface relations in P_i. As a consequence, we can analyze our algorithm for simplicial polytopes only and the time bound we develop will apply to general cases as well.

The computation of $fg(P_i)$ is proportional to $D(p_i)$ and $N(p_i)$ where $D(p_i)$ is the number of faces and incidences that are deleted from $fg(P_{i-1})$ when $fg(P_i) = fg(P_{i-1} \cup \{p_i\})$ is computed and $N(p_i)$ is the number of new faces and incidences that appear in $fg(P_i)$

after the update is completed. It is clear that each face and each incidence between two faces can be deleted at most once, and it is also clear that every face that is going to be deleted had to be created first. Therefore $D(p_{d+2})+ \ldots + D(p_n) \leq (d+1) + N(p_{d+2})+ \ldots + N(p_n)$ and it is sufficient to establish upper bound on $N(p_i)$ only. From Lemma 3.4 the upper bound on $N(p_i)$ is $F_{d-1,i-1}$.

To compute the point q for every new facet takes $\Theta(dF_{d-1,i-1})$ operations and to store the new q-points in the d-D tree structure (containing q-points of all the remaining facets of P_{i-1}) takes $\Theta(dF_{d-1,i-1}\log F_{d,i-1})$ expected time. Also the determining vector has to be computed for each new facet which requires $\Theta(d^3F_{d-1,i-1})$ operations.

Running Time:

$\Theta(d(F_{d-1,d+1}\log F_{d,d+1}+ \ldots + F_{d-1,n-1}\log F_{d,n-1})) =$

$O(dnF_{d-1,n-1}\log F_{d,n-1}) = O(dnF_{d-1,n}\log F_{d,n})$ expected time.

Step 3: To reverse the translation performed in Step 1 takes at most $O(nd)$ computations.

Running Time: $O(nd)$ in the worst case.

From this analysis we can see that Step 2 is the dominating step

of our algorithm. The algorithm computes the convex hull of a set of n points in R^d in $O(dnF_{d-1,n}\log F_{d,n})$ expected time. This time is better than the $O(nF_{d,n})$ expected time of the other on-line higher dimensional convex hull algorithms known to us [Preparata-Shamos(1985), Rey-Ward(1985)].

The validity of the expected time of our algorithm depends on the height of the d-D tree throughout the computation. It is hard to know, in general, how the tree will "grow". We can only predict, that if the tree is updated randomly, the average path length grows logarithmically with the number of nodes in the tree, even so, in the worst case, the path length grows linearly. To maintain the tree balanced, we can reconstruct the tree every time a new polytope is computed. If $F_{d,i}$ is the number of facets of P_i for $(d+2) \leq i \leq n$, then $F_{d,i}$ is also the number of points that have to be stored in the d-D tree after P_i is computed. To reconstuct the tree therefore takes $\Theta(dF_{d,i}\log F_{d,i})$ operations. Under these circumstances $\Theta(dF_{d,i}\log F_{d,i})$ dominates the computational complexity of our algorithm.

The worst case performance can be determined by establishing an upper bound on the running time of the search and the update of the d-D tree, since these are the only two procedures in the analysis of the algorithm where we have given expected times. In the worst case the search can examine every node in the tree for which $\Theta(dF_{d,i-1})$ is the upper bound on the number of operations. As for the update, the worst case occurs when the tree is

reconstructed which, as we have already mentioned, takes $\Theta(dF_{d,i}\log F_{d,i})$ operations. To conclude, the algorithm computes the facial lattice of the convex hull of a set of n points in R^d in $\Theta(d(F_{d,d+1}\log F_{d,d+1} + \ldots + F_{d,n-1}\log F_{d,n-1})) = O(dnF_{d,n}\log F_{d,n})$ time in the worst case.

3.1.3 The Facet Problem

To solve the Facet Problem for a non-simplicial data set without maintaining some information about all the faces of a convex hull polytope, we propose the following data structures:

A *FacetList* which contains information about individual facets and a *VertexList* containing vertices of a convex hull polytope.

For every facet f in the *FacetList* there is a *FacetNeighbors*(f) list - a list of facets that share a common edge with f; and a *FacetVertices*(f) list - a list of vertices that are contained in f. Similarly, for every vertex v in the *VertexList* we maintain a *VertexFacets*(v) list - a list of facets containing v in their boundary.

As in the previous implementation, a facet is represented by the determining vector of its supporting hyperplane and a vertex by its d real valued coordinates.

The computation of P_{d+1} is quite simple. Every point in S' is a vertex of the initial d-simplex. It takes exactly $(d+1)$ iterations to compute all the facets of P_{d+1}, each iteration computing one of the facets. In a d-simplex, a facet shares an edge with the remaining d facets, and a vertex is contained in a boundary of exactly d facets.

Compute P_{d+1}

1. Set $FacetList = \emptyset$, $VertexList = \emptyset$.
2. For $i = 1$ to $(d+1)$ do
   Add p_i to $VertexList$.
   $VertexFacets(p_i) = \emptyset$.
3. For $i = 1$ to $(d+1)$ do
   a. Compute facet f_i determined by the set of points $\{\{p_1, \ldots, p_{d+1}\} - \{p_i\}\}$.
   b. Add f_i to $FacetList$.
   $FacetVertices(f_i) = \emptyset$.
   $FacetNeighbors(f_i) = \emptyset$.
4. For $i = 1$ to $(d+1)$ do
   For $j = 1$ to $(d+1)$ do
   If $i \neq j$ then do
   Add f_i to $FacetNeighbors(f_j)$.
   Add f_i to $VertexFacets(p_j)$.
   Add p_j to $FacetVertices(f_i)$.

Next comes the update mechanism of a d-polytope P_{i-1}. Again here, as in the previous implementation, we have to determine

all *red* and *yellow* facets with respect to some new point p_i. Having discovered one *red* facet by searching the *d-D* tree, we use this facet to determine all the other *red* and *yellow* facets by a process called *Peeling*. During this process we look at every facet that shares an edge with a *red* facet and perform the "beneath beyond" test to determine its color. When a *red* facet is discovered, we perform the "beneath beyond" test on its neighbouring facets and so on, marking every facet that has been tested. As one can see, the *Peeling* process is proportional to the number of *red* and *yellow* facets obtained.

Knowing all *red* facets, it is not difficult to determine the "exposed edges" and to compute new facets. Let f' and f'' be two neighbouring facets such that f' is *blue* and f'' is *red*, and let E be the set of vertices common to f' and f''. Then E determines what we call an "exposed edge", and every vertex contained in E is an "exposed vertex". Let f be the new facet defined by $\{E \cup \{p_i\}\}$. Then $FacetVertices(f) = \{E \cup \{p_i\}\}$ and for every vertex v from E we can add f to $VertexFacets(v)$. We also know that f' and f share an edge and we can therefore add f to $FacetNeighbors(f')$ and set $FacetNeighbors(f) = \{f'\}$. One crucial step of the algorithm still remains to be resolved. It is the determination of the neighbouring facets to f (other than f') and the new neighbouring facets to all *yellow* facets.

Let f be a facet, either new or *yellow*, for which its neighbouring facets are to be determined, and let F_{yellow} be the set of *yellow* facets with respect to p_i and F_{new} the set of new

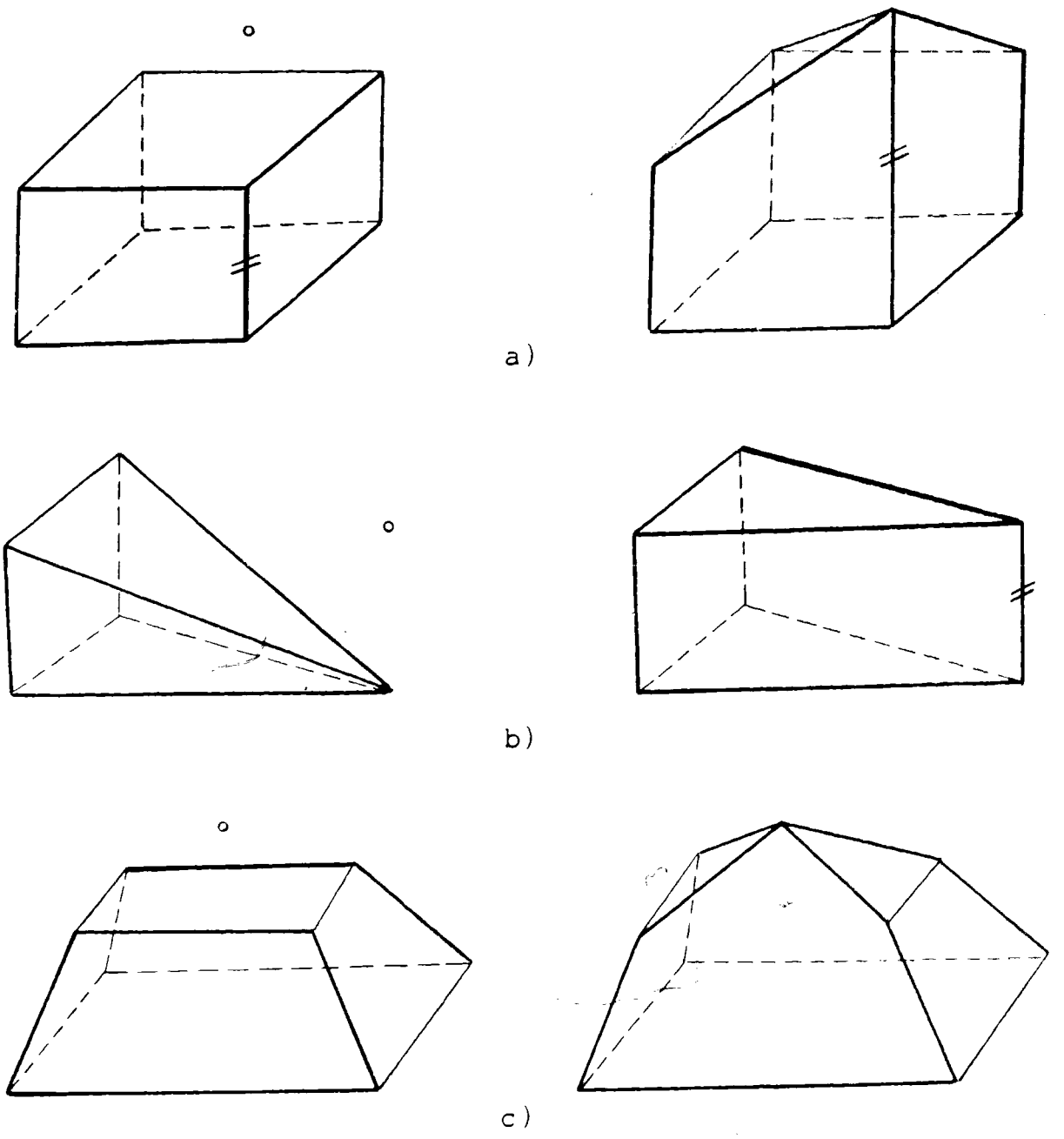
facets of P_i computed so far. The brute force method would be to look at every facet in F_{new} and F_{yellow} and determine the number of affinely independent vertices, *VCount*, shared by this facet with f. If for any of the facets checked the *VCount* is $(d-1)$ then f shares an edge with this facet. However, we can do better than this by considering only those facets from F_{new} and F_{yellow} that contain a vertex from *FacetVertices*(f) in their boundary. This is done by moving all *yellow* facets to the end of the *VertexFacets* lists and by maintaining a count, *FCount*, of all *yellow*/new facets in each list (i.e. new facets are added to the end of the list as well).

Special care has to be taken when *VCount* is computed between two *yellow* facets. This is because two *yellow* facets may: a) already be neighbours in P_{i-1}, b) become neighbours only in P_i, or c) be neighbours in neither P_{i-1} nor P_i[2] [Fig. 3.2]. Only in case b) the neighbourhood information of the *yellow* facets needs to be updated. If we determine the *VCount* in P_i then the numbers obtained in a) and b) would be the same (i.e. $(d-1)$). In order to clearly recognize cases a) and b) we determine the number of affinely independent vertices between *yellow* facets in P_{i-1}, obtaining $(d-1)$ or $(d-2)$ in a) or b), respectively.

In the last step of this implementation of the *Update* P_{i-1} *to include* p_i procedure we use Lemma 3.5 to remove vertices of P_{i-1} which are no longer vertices in P_i.

---

[2]Note that for two new facets or a new and a *yellow* facet only cases a) or c) can occur.



Yellow facets (heavy lined) sharing an edge a) before and after update, b) after update only, and c) neither before nor after update.

Fig. 3.2: Two *yellow* facets of a convex hull polytope before and after update.

Update P_{i-1} *to include* p_i

Let F_{red} and F_{yellow} be the sets of *red* and *yellow* facets of P_{i-1} with respect to p_i obtained by the *Peeling* process (as already described).

1. {Determine neighbouring facets among *yellow* facets}

   For every f from F_{yellow} do

      Let f_{yellow} be the set of *yellow* facets (other than f) such that each one contains at least one vertex of f in its boundary.

      For every f' from f_{yellow} do

         If the number of affinely independent vertices common to f' and f is $(d-2)$ then do

            Add f to $FacetNeighbors(f')$.

            Add f' to $FacetNeighbors(f)$.

2. Add p_i to $VertexList$.

   $VertexFacets(p_i) = \varnothing$.

3. {Compute new facets}

   For every f' from F_{red} do

      For every f'' from $FacetNeighbors(f')$ do

         If f'' is *blue* then do

            Compute new facet f determined by $\{E \cup \{p_i\}\}$, where E is the set of vertices common to f' and f''.

            Add f to $FacetList$.

            $FacetNeighbors(f) = \{f''\}$.

            Add f to $FacetNeighbors(f'')$.

            For every v from E do

Add f to $VertexFacets(v)$.

Add v to $FacetVertices(f)$.

Determine neighbouring facets to f.

4. {Add p_i to the boundary of every new facet}

   Let F_{new} be the set of new facets computed in the previous step.

   For every f from F_{new} do

      Add f to $VertexFacets(p_i)$.

      Add p_i to $FacetVertices(f)$.

5. {Add p_i to the boundary of every *yellow* facet}

   For every f from F_{yellow} do

      Add f to $VertexFacets(p_i)$.

      Add p_i to $FacetVertices(f)$.

6. *Remove red facets from P_i.*

Determine neighbouring facets to f

Let $f_{new/yellow}$ be the set of *yellow* and already computed new facets such that each one contains at least one vertex of E in its boundary.

For every f' from $f_{new/yellow}$ do

   If the number of affinely independent vertices common to f and f' is $(d-2)$ then do

      Add f to $FacetNeighbors(f')$.

      Add f' to $FacetNeighbors(f)$.

Remove red facets from P_i

For every f' from F_{red} do

For every f'' from $FacetNeighbors(f')$ do

Remove f' from $FacetNeighbors(f'')$.

Remove f'' from $FacetNeighbors(f')$.

For every v from $FacetVertices(f')$ do

Remove f' from $VertexFacets(v)$.

Let *FCount* be the number of facets in $VertexFacets(v)$.

If $FCount = 0$ then do

Remove v from $VertexList$.

Else

If $FCount < d$ then do

For every f in $VertexFacets(v)$ do

Remove v from $FacetVertices(f)$.

Remove f from $VertexFacets(v)$.

Remove v from $VertexList$.

Data Structures

Vertex List

Vertex - an array, transformed coordinates of a vertex,

FCount - counter, used to count the number of new or *yellow* facets containing this vertex in their boundary,

Flag - used to mark and unmark this record,

FirstVF, *LastVF* - pointers to the first and the last record, respectively, of *VertexFacet* list associated with this vertex,

PrevVertex, *NextVertex* - pointers to the previous and the next record, respectively, in *VertexList*.

FacetList

Facet - an array, facet normal representing a facet,

Node - points to a terminal node in the *d-D* tree that stores point *q* associated with this facet,

VCount - counter, used to count the number of vertices common to this facet and a facet for which the *FacetNeighbors* information is being established,

Flag - used to mark and unmark this record,

FirstV, *LastV* - pointers to the first and the last record, respectively, in *Vertices* list,

FirstFV, *LastFV* - pointers to the first and the last record, respectively, in *FacetVertices* list,

FirstFN, *LastFN* - pointers to the first and the last record, respectively, in *FacetNeighbors* list,

FirstN, *LastN* - pointers to the first and the last record, respectively, in *Neighbors* list,

PrevFacet, *NextFacet* - pointers to the previous and the next record, respectively, in *FacetList*.

VertexFacets

Vertex - points to the vertex record in *VertexList* associated with this *VertexFacets* list,

Facet - points to the facet record in *FacetList* that contains vertex associated with this *VertexFacets* list in its boundary,

PrevFV, *NextFV* - pointers to the previous and the next record, respectively, in *FacetVertices* list,

PrevVF, *NextVF* - pointers to the previous and the next record, respectively, in *VertexFacets* list.

FacetNeighbors

Facet - points to the facet record in *FacetList* associated with this *FacetNeighbors* list,

Neighbor - points to the facet record in *FacetList* that is a neighbour to the facet associated with this *FacetNeighbors* list,

PrevN, *NextN* - pointers to the previous and the next record, respectively, in *Neighbors* list,

PrevFN, *NextFN* - pointers to the previous and the next record, respectively, in *FacetNeighbors* list.

FacetVertices

the same as *VertexFacets*.

Neighbors

the same as *FacetNeighbors*.

(This list is for the purpose of efficiency of the algorithm only).

Vertices

Vertex - points to a vertex record in *VertexList*,

NextV - points to the next record in *Vertices*.

(This list is used to store pointers to the vertices that are contained in a facet associated with this list and another facet for which the *FacetNeighbors* information is being established).

A node in a *d-D* tree stores the same information as in the previous implementation except for the *Facet* field in the terminal node which points to a facet record in the *FacetList* associated with point *q* represented by this node.

Computational Complexity

The performance of the algorithm will be analyzed for its worst case in terms of length of the input. For the same reason as given in the analysis of the algorithm for the previous implementation, the worst case occurs when no *yellow* facets exist and we will therefore analyze the algorithm for simplicial polytopes only.

Step 1: It is easy to see that the computational complexity of *Compute* P_{d+1} is $O(d^4)$ and the overall complexity of this step remains unchanged in comparison to the previous implementation.

Running Time: $O(nd^3)$ in the worst case.

Step 2: Only procedure *Update* P_{i-1} *to include* p_i needs to be analyzed here, since no change has been made to the other procedures within this step. We will establish an upper bound on the number of operations needed to update P_{i-1}.

For every "exposed edge" in P_{i-1} there is a new facet in P_i. From Lemma 3.4 we know that the number of new facets is bounded above by $F_{d-1,i-1}$. The q-points and the determining vectors of all new facets can be computed in $\Theta(dF_{d-1,i-1})$ and $\Theta(d^3F_{d-1,i-1})$ operations respectively. The d-D tree can be constructed in $\Theta(dF_{d,i}\log F_{d,i})$ time, where $F_{d,i}$ is the number of facets of P_i. For all new facets computed, the *VertexFacets* and the *FacetVertices* lists can be updated in $\Theta(dF_{d-1,i-1})$ time.

To update the *FacetNeighbors* information we do the following operations: for every new facet we 1. look at its "exposed vertices" and 2. for every "exposed vertex" we in turn look at every new facet that contains this vertex in its boundary, and 3. we test the vertices that

are common to these two facets to determine if they are affinely independent. It can be shown that the set of "exposed edges" and "exposed vertices" is isomorphic to a $(d-1)$ polytope [Siedel(1986)], and this establishes $F_{d-2,i}$ as the upper bound on the number of new facets (or "exposed edges") intersecting at a given "exposed vertex". In the worst case then the number of operations performed to update the *FacetNeighbors* information is $\Theta(d^4 F_{d-1,i-1} F_{d-2,i-1})$.

Running Time:

$\Theta(d^4(F_{d-1,d+1}F_{d-2,d+1} + \cdots + F_{d-1,n-1}F_{d-2,n-1})) =$

$O(d^4 n F_{d-1,n-1}F_{d-2,n-1}) = O(d^4 n F_{d-1,n}F_{d-2,n})$ in the worst case.

Step 3: Running Time: $O(dn)$ in the worst case.

The analysis shows that our on-line algorithm computes the facets of the convex hull of a set of n points in R^d in $O(d^4 n F_{d-1,n}F_{d-2,n})$ time in the worst case. This algorithm is the only one known to us that solves the Facet Problem for a non-simplicial data set without maintaining some representation of all the faces of a convex hull. Our worst case time is better than the worst case time conjectured for this problem by Swart(1983) (i.e. $O((F_{d,n})^2 d^{d+4} \log n)$).

CHAPTER IV

THE DIVIDE AND CONQUER ALGORITHM

In this chapter we present an off-line convex hull algorithm solving the Facet Problem for higher dimensional, simplicial data sets. The algorithm is off-line because it requires that all the data points be present before any processing begins. The technique is based on the well known "divide and conquer" principle. In the following section we describe the details of the algorithm and establish its computational complexity.

4.1 Implementation Details and Computational Complexity

When describing the algorithm we will refer to the Voronoi diagram and a *d-D* tree data structure which we have introduced in Chapter II. We will therefore assume reader's familiarity with the meaning of the terms.

The basic data structures used in the implementation of this algorithm are the same as specified for the implementation of our on-line solution to the Facet Problem described in Chapter III (i.e. *FacetList*, *VertexList* and the lists of pointers that relate the two).

The "divide and conquer" principle, in general, involves partitioning the original problem into several subproblems (divide), recursively solving each subproblem (conquer), and combining the solutions to the subproblems to obtain the solution of the original problem (merge). In our particular case this means that we partition the given set S of n points in R^d into two subsets S_1 and S_2 of approximately the same size, separately and recursively compute the convex hulls of both subsets, and then "merge" the convex hulls into a single polytope $P = CH(P_1 \cup P_2)$ where $P_1 = CH(S_1)$ and $P_2 = CH(S_2)$.

For efficiency reasons, we pre-sort the points of S with respect to the value of the first coordinate and partition S into S_1 and S_2 such that for any p (q) of S_1 (S_2), $x_1(p) < c$ $(x_1(q) > c)$ for some c, where $x_1(p)$ is the first coordinate of p. We call S_1 and S_2 the left and the right subsets of S. Thus, S_1 and S_2 are separable by the hyperplane $x_1 = c$.

The merge step is the most important step of the algorithm. It is this step that actually computes the facets of a convex hull. Every invocation of the *Merge* procedure is going to be represented by a node in a binary tree. The inputs to the *Merge* procedure are two non-intersecting polytopes and the output is the convex hull of their union. Let S_i, S_j be two subsets (separable by a vertical hyperplane) of some set S_{ij} such that S_i and S_j are left and right subsets of S_{ij} respectively, and let $P_i = CH(S_i)$ and $P_j = CH(S_j)$. If the *Merge* procedure is invoked with P_i and P_j as its input, then we create a node in a

binary tree representing $P_{ij} = CH(P_i \cup P_j)$ with its *LeftSon* pointer pointing to a node associated with P_i and its *RightSon* pointer pointing to a node associated with P_j. To compute polytope P_{ij} we first delete those facets of P_i and P_j that are not going to be facets of P_{ij}, thus leaving the two polytopes as "open shells", and then we compute the "wrap around" portion W_{ij} that connects the "open shells" [Fig. 4.1]. P_{ij} is therefore composed of W_{ij} and the remaining portions of P_i and P_j.

Let F_W be the set of facets of W_{ij} and let x be some point that is interior to P_{ij}. Every f from F_W determines a hyperplane $aff(f)$ that partitions R^d into two closed half-spaces intersection of which is $aff(f)$. Let $H(f,x)$ be the closed half-space that contains x. Then the intersection of all $H(f_i,x,)$ for f_i from F_W, is a convex polyhedron which we denote P_W. If for every f_i we compute point q_i such that $aff(f_i)$ becomes a perpendicular bisector of the line segment joining x and q_i, we transform P_W into a Voronoi polyhedron with point x being its generating point [Fig. 4.2]. Let Q_W be the set of q-points computed for the facets of F_W. For reasons that will become clear later, we will construct a d-D tree storing the points of Q_W and associate this tree with the node in a binary tree representing P_{ij}.

Because of the non-degenerate assumption on S, a vertex from P_i can lie either beneath or beyond a supporting hyperplane of a facet of P_j and the same is true for a vertex from P_j in relation to a supporting hyperplane of a facet of P_i. Therefore,

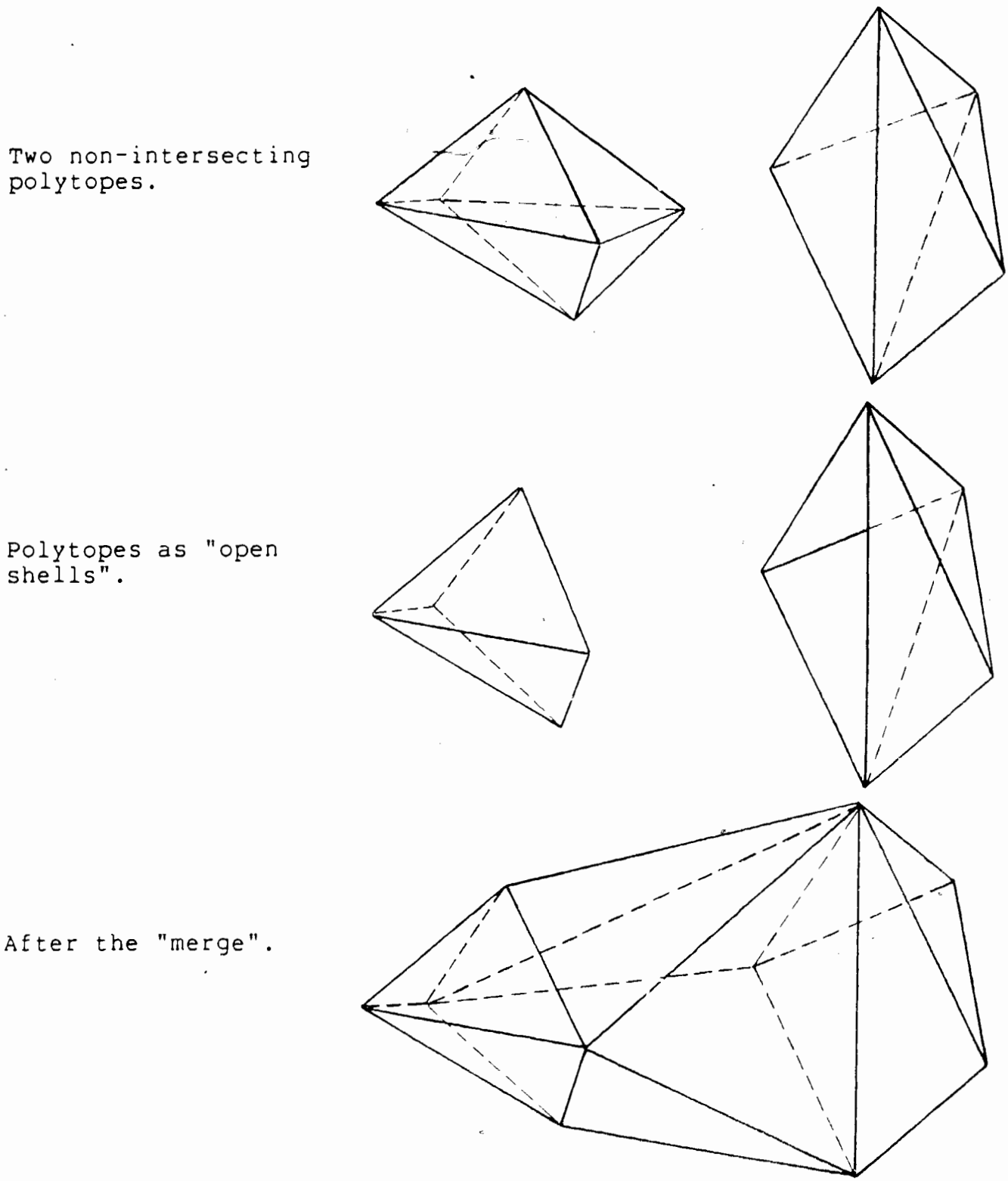


Fig. 4.1: "Merging" of two polytopes in R^3.

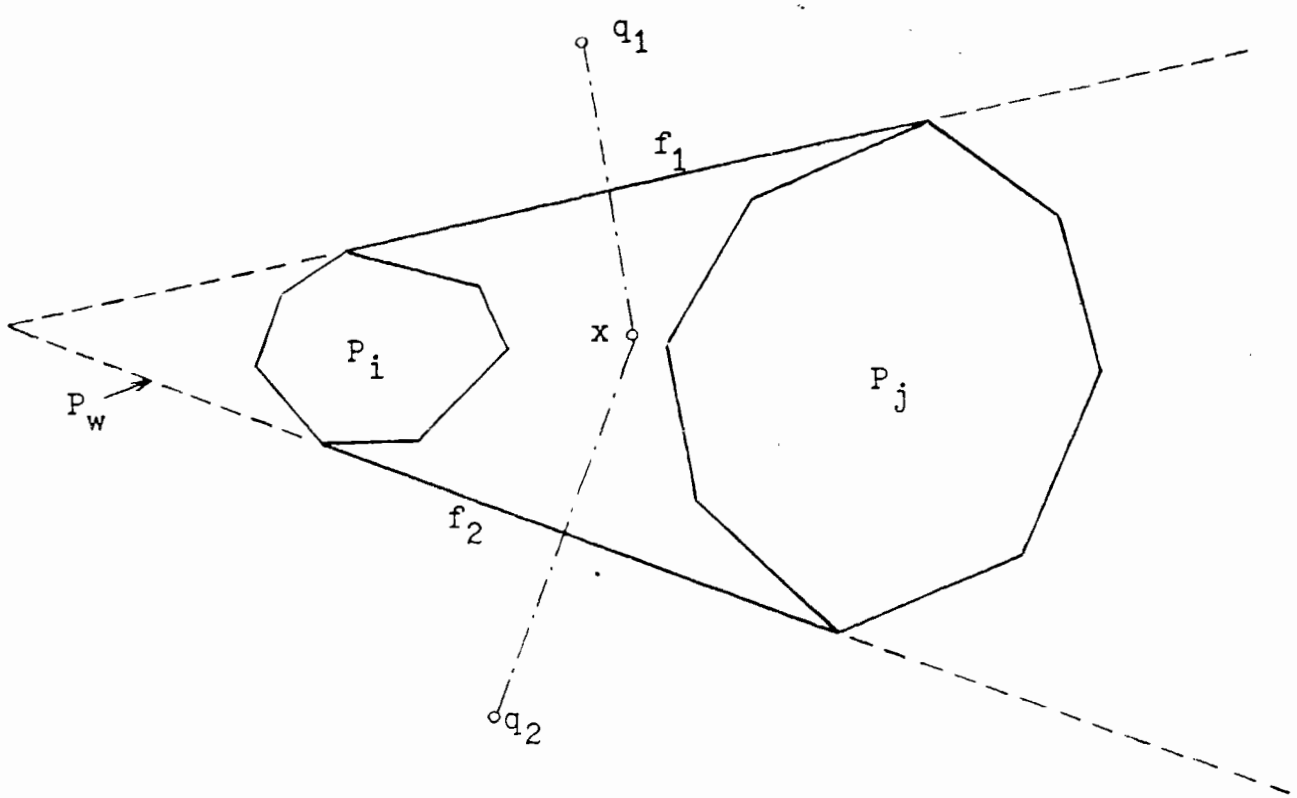


Fig. 4.2: "Wrap around" portion connecting polytopes P_i and P_j extended to form polygon P_w and its transformation into a Voronoi polygon.

according to Definition 3.1 we can only have *red* or *blue* facets (i.e. no *yellow*). The process of determining *red* facets is similar to the one we have described in Chapter II for the on-line algorithm. However, here we are determining *red* facets

of one polytope with respect to all the vertices of some other polytope. We accomplish this by considering one vertex at a time, discovering one *red* facet with respect to this vertex first, and then applying the *Peeling* process (as described in Chapter III) to determine all the other *red* facets (if any). In the following section we will show how to discover the first *red* facet.

Assume $p = (p_1, \ldots, p_d)$ is a vertex of P_j, and P_i is the polytope whose first *red* facet with respect to p is to be determined. Further assume that $x_1 = c$ is the hyperplane that separates S_i and S_j. We start by searching the d-D tree associated with the node in the binary tree representing P_i. If the search is successful, we have determined our first *red* facet (i.e. it is the facet corresponding to the point q discovered by the search). When the q-points stored in d-D tree were computed, the origin of the the coordinate system was transformed to some point x interior to the polyhedron $P_{ij} = CH(P_i \cup P_j)$. For every d-D tree, the coordinates of its corresponding point x are stored in the node of the binary tree that the d-D tree is associated with. It is necessary to transform the coordinates of p such that the origin is at x, before searching the d-D tree. If the search is not successful, we will search the d-D trees associated with the nodes of the binary tree along the path determined by the *RightSon* pointer of every node visited, terminating the search as soon as a *red* facet is discovered[1]. In

---

[1]In a reverse situation when p is a vertex of P_i and P_j is the polytope whose *red* facets with respect to p are to be

the following we will explain why it is possible to search along one path only.

Let P_{ij} and P_{kl} be two polytopes that are to be merged. Then they are separable by a hyperplane $x_l = c$ for some c. Let P_{ij} (P_{kl}) be the left (right) polytope with respect to the separating hyperplane. To determine one red facet of P_{ij} with respect to some vertex p of P_{kl} we traverse down the binary tree starting at the node representing P_{ij} and searching the d-D tree associated with every node visited. A d-D tree stores q-points of some "wrap around". Let W_{ij} be the "wrap around" connecting polytopes P_i and P_j. Then again there exists a separating hyperplane $x_l = c'$ that separates P_i and P_j. This hyperplane defines two closed half-spaces H_i and H_j containing P_i and P_j respectively. Let P_j be the right polytope with respect to the separating hyperplane. Some of the facets of P_i and P_j were deleted when W_{ij} was computed, leaving S_i and S_j as the sets of the facets of P_i and P_j, respectively, of their remaining "open shells". Define P_w to be the polytope obtained as the intersection of all $aff(f)$ for f, a facet of W_{ij}. Let p be inside P_w (i.e. if p is not inside P_w then at least one facet of P_w must be *red*). Since p is inside of P_w and it is also in H_j (i.e. $c' < c$), it must be in $(P_w \cap H_j)$. But for every point x from $(P_w \cap H_j)$ x is beneath all the facets of S_i and therefore p is beneath all the facets of S_i as well. The q-points for the

---

[1](cont'd) determined, we will search the d-D trees associated with the nodes of the binary tree along the path determined by the *LeftSon* pointers.

facets in S_i are stored in the d-D tree associated with the node pointed to by *LeftSon* pointer of its parent node (i.e. the node associated with P_{ij}) and therefore the path determined by the *LeftSon* pointer need not be traversed.

As facets are deleted, the corresponding q-points are deleted from the d-D trees as well. Therefore a d-D tree may not contain all the q-points it contained when it was originally created. Let W be the "wrap around" from which some facets have been deleted. The d-D tree corresponding to W is associated with a node in the binary tree representing some polytope P. The new facets that replaced the facets deleted from W must have been computed after the facets of W were, and therefore their corresponding q-points must be stored in the d-D tree associated with some nodes of the binary tree at a level higher than the node representing P. Since our search always starts at the highest level of the binary tree, we must have tested p against those relevant facets already and p must have been found beneath each one of them.

To compute new facets (i.e. facets of the "wrap around" connecting two "open shells") we can apply the "gift wrapping" technique [Chand-Kapur(1970)]. This technique is based on Theorem 3.3. The principle involved here is to compute the facets of a convex hull from the known edges. We start with an initial convex hull facet. Assuming that the facet is simplicial it determines d $(d-2)$-dimensional edges. The initial facet is then rotated about each edge to obtain new facets. This process

is repeated for every new facet and the edges determined by the facet. In the following paragraph we show how the initial facet can be obtained.

Each one of the two "open shells" determines a set of "exposed edges" and a set of "exposed vertices" (i.e. the vertices contained in the exposed edges). Let V_i and V_j be the sets of "exposed vertices" of P_i and P_j respectively. For every "exposed edge" one of the two facets that contain this edge is known and the other facet needs to be computed. Let e be an exposed edge of P_i and f the known facet that contains e. The initial facet can be computed by seeking among all the hyperplanes determined by the edge e and a point from V_j the one such that all other vertices of V_j are beneath this hyperplane.

To compute the facets of the "wrap around" efficiently, we introduce a new representation for vertices, facets and edges. For every vertex we generate a unique vertex number, a positive integer. A simplicial facet can then be represented by a vector of length d that contains the generated vertex numbers of the d vertices that define the facet, in ascending order. Similarly, to represent an edge we use a vertex of length $(d-1)$ containing the vertex numbers of the vertices that determine the edge, also in ascending order.

As the new facets are computed, they are stored in a queue which we call *FacetQueue*. In this queue a facet is represented by its vector. If (w.l.o.g.) $\{p_1, \ldots, p_d\}$ is the set of vertices that

determine a new facet and $(n_1, \ldots, n_d)$ its corresponding vector, then every n_i for $1 \leq i \leq d$ can be used as a label for an edge of this facet. An edge will have a label n_i if it is determined by the set of vertices $\{p_1, \ldots, p_d\} - \{p_i\}$. Therefore, a facet vector represents not only a facet but also the labels of its edges.

The new edges are stored in a height balanced AVL tree [Aho-Hopcroft-Ullman(1974)]. At the time of inserting an edge as a node into the tree, we also store a pointer pointing to the facet that contains this edge, stored in *FacetQueue*. Two edges e_i and e_j with vectors $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$, respectively, are compared lexicographically by the following rule:

Let k be the smallest subscript for which $i_k \neq j_k$.

If $i_k < j_k$ then e_i is "smaller" than e_j.

Else e_i is "larger" that e_j.

It is possible that an edge of a newly computed facet is already present in the tree. This means, that the edge is now associated with two facets and therefore need not be considered any further. This can be indicated by changing the edge label in the vectors of both facets in *FacetQueue* to its negative value. When a facet from the *FacetQueue* is considered, new facets are computed only for those edges for which the facet label is positive. For a given edge e we consider only the two sets of "exposed vertices" as possible candidates for the vertex that will determine the new facet containing e. These sets are only

subsets of the entire set of vertices of the resulting polytope, which contributes to the efficiency of the algorithm.

Every one of the two polytopes being merged has its own *FacetsList* and *VertexList*. We append one to the end of the other thus obtaining only one *FacetList* and only one *VertexList*. As new facets are being computed, they are added to the end of *FacetList*. The update of the *FacetVertices*, *VertexFacets* and *FacetNeighbors* lists is, as we have shown for the Facet Problem implementation of the on-line algorithm, proportional to the number of new facets computed.

Chand and Kapur presented a very clever way of computing the new facets from the known edges [Chand-Kapur(1970)]. We will not describe the details of their technique here, but refer the reader to their publication.

4.1.1 The Algorithm

Input: Set S of n points, $S \subseteq R^d$, and dimension d.
Output: Description of the facets of $P = CH(S)$.

Step 1: {Sort}

Sort point of S with respect to the value of the first coordinate.

Let $S = \{p_1, \ldots, p_n\}$ be the sorted set.

Step 2:

If $n \leq d$ then do

Construct the convex hull of S by using any trivial algorithm and stop.

Else

Do Step 3.

Step 3:

1. {Divide}

   Set $k = \lfloor n/2 \rfloor$ and divide S into

   $S_1 = \{p_1, \ldots, p_k\}$ and

   $S_2 = \{p_{k+1}, \ldots, p_n\}$.

2. {Conquer}

   Compute $P_1 = CH(S_1)$ and $P_2 = CH(S_2)$ recursively.

3. {Merge}

   Merge the two convex hulls to form $P = CH(P_1 \cup P_2)$.

The only non-trivial part of this algorithm is the *Merge* procedure in Step 3 which combines two convex hulls. Following is an outline of the steps of this procedure.

Merge

1. Determine facets that need to be deleted.
   a. Determine one facet that needs to be deleted from one polytope for every vertex of the other polytope and vice versa.

b. Use *Peeling* process to determine all the other facets that need to be deleted.

2. Compute new facets.

3. Store new facets in a *d-D* tree.

4.1.2 Data Structures

A node in a *d-D* tree stores the same information as specified in Chapter III for the Facet Problem solving implementation of the on-line algorithm.

VertexList

Vertex - an array, original coordinates of a vertex,

VertexNo - a positive integer generated to represent a vertex,

Flag - used to mark and unmark this record,

FirstVF, *LastVF* - pointers to the first and the last record, respectively, of *VertexFacet* list associated with this vertex,

PrevVertex, *NextVertex* - pointers to the previous and the next record, respectively, in *VertexList*.

FacetList

Facet - an array, facet normal representing a facet,

PointX - an array, coordinates of point x to which the origin of the coordinate system was transformed when a facet represented by this record was computed,

Node - points to a terminal node in the *d-D* tree which stores point q associated with this facet,

Flag - used to mark and unmark this record,

FirstFV, *LastFV* - pointers to the first and the last record, respectively, in *FacetNeighbors* list,

FirstN, *LastN* - pointers to the first and the last record, respectively, in *Neighbors* list,

PrevFacet, *NextFacet* - pointers to the previous and the next record, respectively, in *FacetList*.

VertexFacets, *FacetNeighbors*, *FacetVertices* and *Neighbors* lists store the same information as specified in Chapter III for the Facet Problem solving implementation of the on-line algorithm.

A node in the binary tree stores the following:

TreeNode - points to the root node of a *d-D* tree asssociated with this node,

PointX - an array, coordinates of point x to which the origin of the coordinate system was transformed when the facets of the "wrap around" represented by this node were computed,

LeftSon, *RightSon* - pointers to the left and the right son nodes, respectively, in this tree.

FacetQueue

FVertices - an array, contains d vertex numbers of the vertices that are contained in this facet, in ascending order,

Facet - points to the facet record in *FacetList* associated with this facet.

A node in the AVL tree stores the following:

EVertices - a search key, an array containing $(d-1)$ vertex numbers of the vertices that are contained in this edge, in ascending order,

FQueue - points to the facet in *FacetQueue* that contains this edge,

SmallSon - pointer, points to the son node with "smaller" value of search key than the value of search key in this node, *LargeSon* - pointer, points to the son node with "larger" value of search key than the value of search key in this node.

4.1.3 Computational Complexity

Step 1:

The amount of time needed to pre-sort the points of S is

$O(n\log n)$.

Step 2:

If $n \leq d$ the construction of $P = CH(S)$ can be done in $O(dF_{d,n})$ operations.

Step 3:

Let $T(n,d)$ denotes the time needed by the algorithm to compute $P = CH(S)$. Then assuming that n is a power of two we have the following recurrence relation:

$$T(1, d) = \text{constant}$$

$$T(n,d) = 2T(n/2,d) + M(n,d)$$

where $M(n,d)$ denotes the time it takes to compute the convex hull of the union of two polytopes with $n/2$ vertices each.

The solution to this recurrence relation is obtained by establishing an upper bound on $M(n,d)$. To merge two polytopes we do the following:

1. For every vertex of one polytope determine a facet (if it exists) of the other polytope such that the vertex lies beyond it, and vice versa.
   The height of the binary tree is $\lceil \log n \rceil$.
   Let $F_{*,n}$ be the number of facets of a "wrap around" connecting two d-polytopes P_1 and P_2 of $O(n)$ vertices each. Then the height of the d-D tree is $\lceil \log F_{*,n} \rceil$ and the tree can be searched in

$\Theta(d\log F_{*,n})$ expected time or $O(dF_{*,n})$ time in the worst case.

Running Time:

$O(dnF_{*,n})$ in the worst case or

$O(dn\log n\log F_{*,n})$ expected time.

2. Determine all the other facets that need to be deleted.

The determination of facets that need to be deleted is proportional to the number of of facets found which can be $O(F_{d,n})$ in the worst case. These facets have to be deleted from the *FacetList* which takes $O(dF_{d,n})$ operations, and also from the *d-D* tree, which can be done in $O(dF_{d,n}\log F_{*,n})$ operations.

Running Time: $O(dF_{d,n}\log F_{*,n})$ in the worst case.

3. Compute new facets.

To compute new facets using the "divide and conquer" technique takes $O(dnF_{*,n})$ operations in the worst case.

Running Time: $O(dnF_{*,n})$ in the worst case.

4. Store new facets.

New facets have to be stored in the *d-D* tree which takes $\Theta(dF_{*,n}\log F_{*,n})$ operations and also in *FacetList* which takes $O(dF_{*,n})$ time.

Running Time: $\Theta(dF_{*,n}\log F_{*,n})$ expected time and in the worst case.

The upper bound on $F_{*,n}$ is equal to the upper bound on the number of facets computed to obtain $P = CH(P_1 \cup P_2)$. Since P_1 and P_2 are separable by some hyperplane H, $F_{*,n}$ is also equal to the upper bound on the number of edges of a facet that is obtained when P is cut by H. Since in the worst case $O(F_{d,n})$ facets of P can be cut by H, $F_{*,n} = F_{d,n}$. Therefore, our off-line algorithm computes the facets of a convex hull of n points in R^d in $O(dnF_{d,n})$ time in the worst case.

CHAPTER V

EXPERIMENTAL RESULTS

The on-line Facet Problem solving convex hull algorithm, presented in Chapter III, has been implemented using the Pascal language. We have tested the performance of the algorithm on randomly generated data sets with normal and uniform distribution and on "real life" data set. The convex hulls were computed for two, three, four and five dimensional data sets. The results obtained are presented in tables 1, 2 and 3.

The "real life" data set consists of 2,998 records of cervical cell data, each cell represented by a six-dimensional feature vector. Because of a very large number of facets computed for a high dimensional data set of this size, we considered only up to five features. We have observed that a large number of points, when tested against the convex hull, are found to be in its interior. These are what we call "throw away" points because they require no further processing. The larger the number of the "throw away" points, the more efficient our algorithm is.

The number of facets of the resulting polytope in comparison to the total number of facets computed seems to be quite small. Even so, we have a very large number of points that do not cause computation of new facets (i.e. the "throw away" points). This would suggest that our algorithm is more efficient on average than Seidel's(1981) and Edelsbruner's(1987), where, because of

pre-sorting, new facets are computed for every point of the data set

From the test results, we can see, that the *d-D* tree remains reasonably balanced throughout the computation. The maximum height of the tree does not increase significantly, although there was a considerable amount of updating being done.

The test results show that as an incremental technique for computing convex hulls, our algorithm is efficient. It performed well on a "real life" data set as well as on a randomly generated one.

Table 1: Performance of the on-line convex hull algorithm on a cervical cell data set

Dimension	2	3	4	5
File size	2,998	2,998	2,998	2,998
No. of vertices	15	48	109	250
No. of facets	15	92	546	4,182
Total no. of facets computed	144	1,080	7,365	46,284
No. of "throw away" points	2,924	2,801	2,616	2,415
Average no. of terminal nodes searched	6	8	42	207
Average no. of terminal nodes in the tree	15	79	460	3,429
Maximum height of the tree	9	14	22	33

Table 2: Performance of the on-line convex hull algorithm on a uniformly distributed data set

Dimension	2	3	4	5
File size	3,000	3,000	2,000	750
No. of vertices	14	85	230	285
No. of facets	14	161	1,201	5,215
Total no. of facets computed	141	1,316	9,746	29,779
No. of "throw away" points	2,928	2,758	1,543	350
Average no. of terminal nodes searched	3	14	34	85
Average no. of terminal nodes in the tree	14	130	859	2,948
Maximum neight of the tree	8	14	22	26

Table 3: Performance of the on-line convex hull algorithm on a normally distributed data set

Dimension	2	3	4	5
File size	3,000	3,000	2,000	750
No. of vertices	11	40	105	157
No. of facets	11	76	540	2,442
Total no. of facets computed	107	720	4,872	19,366
No. of "throw away" points	2,945	2,863	1,730	459
Average no. of terminal nodes searched	2	5	12	22
Average no. of terminal nodes in the tree	12	69	428	1,550
Maximum height of the tree	7	13	20	25

CHAPTER VI

CONCLUSION

One of the problems an on-line convex hull algorithm has to solve is the point inclusion problem in a convex polyhedron. The existing on-line convex hull algorithms [Preparata-Shamos(1985), Rey-Ward(1985)] do not solve this problem efficiently. For a polyhedron P determined by a set of n points in R^d their solution takes $\Theta(F_{d,n})$ time on average and in the worst case, where $F_{d,n}$ is the number of facets of P. Seidel(1981) and Edelsbruner(1987) also use an on-line technique in their convex hull algorithms, but avoid the time consuming "point inclusion" test by pre-sorting the entire point set and the resulting algorithms are off-line. We have proposed a method which solves the point inclusion problem in $\Theta(d\log F_{d,n})$ expected time. In short, we transform the point inclusion problem to the nearest neighbour problem and we use a d-dimensional tree structure, called the d-D tree, to search for the nearest neighbour.

When searching for the nearest neighbour, the search space is partitioned into regions. We have shown how the number of nodes examined by the search can be reduced by creating "cells" within the regions. A cell of a particular region tightly encloses the set of domain points contained in the region. We have also proposed a way of dynamically updating the d-D tree in time proportional to dH where H is the height of the tree.

We have used the above results in the design of two convex hull algorithms for higher dimensional data sets. The first algorithm is an on-line algorithm which can be applied to a non-simplicial data set. It can be used to solve either the Facial Lattice Problem or the Facet Problem. The Facet Problem is solved without maintaining information about all the faces of a convex hull polytope and this is the only algorithm known to us to do so. For a set of n points in R^d the algorithm solves the Facet Problem in $O(d^4 nF_{d-1,n}F_{d-2,n})$ time in the worst case. The Facial Lattice Problem can be solved in $O(dnF_{d-1,n}\log F_{d,n})$ expected time or $O(dnF_{d,n}\log F_{d,n})$ time in the worst case. This expected time is the best expected time known to us for an on-line higher dimensional convex hull algorithm.

We have implemented the on-line algorithm for the Facet Problem in Pascal language. Our test results show, that when the convex hull is computed for either a randomly generated data set or a "real world" data set, there are a large number of points which, when tested against the current convex hull polytope, are found to be in its interior. This finding contributes to the efficiency of our algorithm since we are only spending expected $\Theta(d\log F_{d,i})$ time to process these points, where $F_{d,i}$ is the number of facets of the current d-polytope determined by the first i points.

The "divide and conquer" principle has been used by Preparata and Hong(1977) in the design of a convex hull algorithm for two and three dimensional data sets. The technique used by them to

determine facets that need to be deleted when two polytopes are merged cannot be extended to higher dimensions. We have solved this problem by transforming it to a nearest neighbour problem. To compute new facets we are using the "gift wrapping" principle. In our implementation the number of vertices that need to be considered when new facets are computed is only a subset of the number of vertices of the resulting polytope. The worst case performance of this algorithm is $O(dnF_{d,n})$.

6.1 Open Questions

The nearest neighbour problem for a domain of n points in R^d in a dynamic environment is solved in $O(d\log n)$ expected time. The q-points that we generate for the facets of a convex polyhedron (with respect to some point x interior to the polyhedron) form a special geometric structure. Whether this structure can be exploited successfully to develop an algorithm that would solve the nearest neighbour problem in $O(d\log n)$ time in the worst case is an open question.

Our on-line algorithm, presented in Chapter III, maintains the d-D tree data structure dynamically. Generally, it is hard to predict how the tree will "grow". We conjecture, that if the tree is updated randomly, the expected height of the tree grows

logarithmically with the number of terminal nodes in the tree. Whether it is possible to update the tree so that it will remain balanced, or to balance the tree without recreating it is an open question.

In order for the "divide and conquer" technique to be extended to higher dimensions, the following two problems have to be solved efficiently:

1. The determination of the facets that need to be deleted during the merge step.
2. The computation of the new facets during the merge step.

In this thesis we have solved the first problem efficiently. To compute new facets, we use the "gift wrapping" technique. This technique computes a facet in $O(n)$ time. Whether there is a technique that would compute a facet in $O(\log n)$ time, is an open question.

BIBLIOGRAPHY

Aho, A.V., Hopcroft, J.E. and Ullman, J.D. (1974)
"The Design and Analysis of Computer Algorithms"
Addison-Wesley Publishing Comp., 1974.

Akl, S.G. and Toussaint, G.T. (1978-1)
"Efficient Convex Hull Algorithms for Pattern Recognition Applications"
Proc. 4th Int'l Joint Conf. on Pattern Recognition, Kyoto, Japan, pp. 483-487.

Akl, S.G. and Toussaint, G.T. (1978-2)
"A Fast Convex Hull Algorithm"
Info. Proc. Letters, Vol. 7, No.5, 1978, pp.219-222.

Barnette, D.W. (1973)
"A Proof of the Lower Bound Conjecture for Convex Polytopes"
Pacific Journal of Mathematics 46, 1973, pp. 349-354.

Ben-Or, M. (1983)
"Lower Bounds for Algebraic Computation Trees"
Proc. 15th ACM STOC, 1983, pp. 80-86.

Bentley, J.L. (1975)
"Multidimensional Binary Search Trees Used for Associative Searching"
Comm. of the ACM, Vol. 18, No. 9, 1975, pp. 509-517.

Bentley, J.L. and Shamos, M.I. (1978)
"Divide and Conquer for Linear Expected Time"
Info. Proc. Letters, Vol. 7, No. 2, Fef. 1978, pp. 87-91.

Bhattacharya, B.K. (1982)
"Application of Computational Geometry to Pattern Recognition Problem"
Simon Fraser University, CS Tech. Rep. 82-3, 1982.

Chand, D.R. and Kapur, S.S. (1970)
"An Algorithm for Convex Polytopes"
Journal of the ACM, Vol. 17, No. 1, 1970, pp. 78-86.

Dirichlet, G.L. (1850)
"Uber die Reduction der Positiven Quadratischen Formen mit Drei Umberstimmten Ganzen Zahlen"
Journal fur die Reine Angew. Math., Vol. 40, 1850, pp. 209-227.

Duda, R.D. and Hart, P.E. (1973)
"Pattern Classification and Scene Analysis"
Wiley, New York, 1973, pp. 166-171.

Edelsbruner, H. (1987)
"Algorithms in Combinatorial Geometry"
Springer-Verlag, 1987.

Freeman, H. and Shapira, R. (1975)
"Determining the Minimum Area Incasing Rectangle for an Arbitrary Closed Curve"
Comm. ACM, Vol. 18, No. 7, July 1975, pp. 409-413.

Friedman, J.H., Bentley, J.L. and Finkel, R.A. (1977)
"An Algorithm for Finding Best Matches in Logarithmic Expected Time"
ACM Transactions on Mathematical Software, Vol. 3, No. 3, Sept. 1977, pp. 209-226.

Gilbert, E.N. and Pollak, H. (1986)
"Steiner Minimal Trees"
SIM J. Appl. Math., 16, 1986, pp. 1-29.

Graham, R.L. (1972)
"An Efficient Algorithm foe Determining the Convex Hull of a Finite Planar Set"
Info. Proc. Letters, Vol. 1, 1972, pp. 132-133.

Grünbaum, B. (1967)
"Pure and Applied Mathematics", Vol. XVI: Convex Polytopes
Wiley Interscience Publishers, New York, 1976.

Jarvis, R. A. (1973)
"On the Identification of the Convex Hull of a Finite Set of Points in the Plane"
Info. Proc. Letters, Vol. 2, 1973, pp. 18-21.

Jozwik, A. (1983)
"A Method for Solving the n-dimensional Convex Problem"
Pattern Recognition Letters, Vol. 2, 1983, pp. 23-25.

Kallay, M. (1981)
"Convex Hull Algorithms in Higher Dimensions"
Univ. of Oklahoma, Dept. of Mathematics, unpublished manuscript.

Kallay, M. (1984)
"The Complexity of Incremental Convex Hull Algorithms in R^d"
Inf. Proc. Letters, Vol. 19, 1984, p.197.

Kirkpatrick, D.G. and Seidel, R. (1986)
"The Ultimate Planar Convex Hull Algorithm ?"
SIAM Journal on Computing, Vol. 15, No. 1, Feb. 1986, pp.287-299.

Knuth, D.E. (1976)
"Big Omicron and Big Omega and Big Theta"
SIGACT News 8, No. 2, pp.18-24.

McMullen, P. (1970)
"The Maximum Number of Faces of a Convex Polytope"
Mathematica 17, 1970, pp. 179-184.

McMullen, P. and Shephard, G.C. (1971)
"Convex Polytopes and the Upper Bound Conjecture"
London Mathematical Society Lecture Notes Series, Vol. 3, Cambridge University Press, 1971.

Preparata, F.P. and Hong, S.J. (1977)
"Convex Hulls of Finite Sets of Points in Two and Three Dimensions"
Comm. of the ACM, Vol. 20, No. 7, 1977, pp. 87-93.

Preparata, F.P. and Shamos, M.I. (1985)
"Computational Geometry"
Springer Verlag (1985)

Rey, C. and Ward, R. (1985)
"An On-line Algorithm for Determining Convex Polytopes"
IEEE Transactions, 1985, pp. 87-91.

Rosenfeld, A. (1969)
"Picture Processing by Computers"
Academic Press, New York, 1969.

Seidel, R. (1981)
"A Convex Hull Algorithm Optimal for Point Sets in Even Dimensions"
Univ. of British Columbia, CS Tech. Rep. 81-14, 1981.

Seidel, R. (1986)
"Constructing Higher Dimensional Convex Hulls at Logarithmic Cost per Face"
Proceedings of 18th Annual ACM STOCK, New York, 1986.

Sklansky, J. (1972)
"Measuring Concavity on a Rectangular Mosaic"
IEEE Trans. Comptrs. C-21, Dec. 1972, pp. 1355-1364.

Solomon, B.J. (1985)
"Surface Intersection for Solid Modelling"
University of Cambridge, Ph.D. Thesis, 1985.

Swart, G. (1985)
"Finding the Convex Hull Facet by Facet"
Journal of Algorithms 6, 1985, pp. 17-48.

Thiessen, A.H. (1911)
"Precipitation Averages for Large Areas"
Monthly Weather Review, Vol. 39, 1911, pp. 1082-1084.

Toussaint, G.T. (1978)
"The Convex Hull as a Tool in Pattern Recognition"
Proc. AFOSR Workshop in Communication Theory and Applications,
Cape Cod, Mass., Sept. 1978.

Voronoi, G. (1908)
"Nouvelles Applications des Parameters Continues a la Theorie des Formes Quadratiques"
Denxieme Memoire, Recherches sur les Paralleloedres Primitifs,
Journal Reine Angew. Math., Vol. 134, 1908, pp. 198-287.

Wigner, E. and Seitz, F. (1933)
"On the Constitution of Metallic Sodium"
Physical Review, Vol. 43, 1933, pp. 804-810.

Wirth, N. (1976)
"Algorithms + Data Structures = Programs"
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1976.

Yao, A.C. (1981)
"A Lower Bound to Finding Convex Hulls"
Journal of the ACM, Vol. 28, 1981, pp. 780-789.

INDEX