C.1

DYNAMIC PROBLEMS IN COMPUTATIONAL GEOMETRY

by

IHOR GEORGE GOWDA

B.Sc.(Hons.), The University of Alberta, 1978

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

December 1980

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Nov. 27, 1980

ABSTRACT

Computational geometry is the study of algorithms for manipulating sets of points, lines, polygons, planes and other geometric objects. For many problems in this realm, the sets considered are static and the data structures representing them do not permit efficient insertion and deletion of objects (e.g. points). Dynamic problems, in which the set and the geometric data structure change over time, are also of interest. The topic of this thesis is the presentation of fast algorithms for fully dynamic maintenance of some common geometric structures. The following configurations are examined: planar nearest-point and farthest-point Voronoi diagrams, convex hulls (in two and three dimensions), common intersection of halfspaces (2-D and 3-D), and contour of maximal vectors (2-D and 3-D). The principal techniques exploited are fast merging of substructures, and the use of extra storage. Dynamic geometric search structures based upon the configurations are also presented.

Table of Contents

List of Figures

Acknowledgement

I would like to express my deepest thanks to my supervisor David Kirkpatrick, for his guidance,inspiration and faith in me. Also, I am grateful to Uri Ascher for being my second reader. Kris Barge (world's greatest librarian) aided me by obtaining a copy of Michael Shamos' thesis when it seemed unobtainable. Randy Goebel, Jim Little, Carl Pottle and Len Stoch kept me laughing in the midst of it all. My gratitude goes to Theresa Fong and Lindsey Wey for assistance with the typing. I also offer thanks to the Natural Sciences and Engineering Research Council of Canada for their financial support.

1. Introduction

1.1. Computational Geometry

Computational geometry is concerned with the design and analysis of algorithms for manipulating sets of points, lines, polygons, planes and other geometric objects in dimensions two and higher. Many computational problems are most naturally cast in a geometric setting, where fast algorithms can often be developed which exploit the structure of the problem provided by the geometry.

Several problem areas within computational geometry have had considerable research devoted to them. Construction of convex hulls is one such area (see [Be80]). The convex hull of a set of points **S** , mathematically defined to be the smallest convex set containing all the points of **S** , is an important basic geometrical structure that arises in many applications and as a component in the solution of many more involved problems.

Intersection problems, which occur in applications such as computer graphics, linear programming, and computer-aided design (especially integrated circuit design), have also received much attention recently ([ShHo76], [ChDo80]). Algorithms have been devised to determine intersections among lines, polygons, half-spaces and other objects.

Closest point problems (eg. element-uniqueness, finding

the two closest of n points, nearest-neighbor) arise in cluster analysis, pattern recognition, and construction of minimum spanning trees (see [Sh78]). The Voronoi diagram of n planar points is a structure containing proximity information which allows the efficient solution of many closest-point problems. There are geometric searching problems associated with Voronoi diagrams, maximal vectors, rectangular ranges, straight-line planar graphs and other geometric data structures.

Much of the pioneering work in computational geometry was done by Shamos, and [Sh78] is an indispensable account of early work in the discipline. Good surveys of portions of the field are provided in [MaOt79a] and [Br79]. An extensive bibliography of papers related to computational geometry is supplied in [EdvL80].

1.2. Maintenance of Dynamic Structures

For many problems in computational geometry, the sets considered are usually static and the data structures representing them are not designed to allow for efficient insertion and deletion of objects (eg. points). The data structures can be searched and otherwise manipulated very efficiently, but for some problems a complete reconstruction of the static data structure seems to be necessary to support the

insertion or deletion of even a single object.

1.2.1. Decomposable Searching Problems

Bentley [Be79] recently observed that for some problems, a distributed approach could provide reasonably efficient dynamic solutions. Rather than maintain the entire set of objects in one "global" data structure, it can be effective to partition the set into smaller pieces ("blocks"), each statically organized, and separately maintained. This approach can fail completely. It could prove more difficult to query a partitioned set (if it is indeed possible) than to query a set represented by a single, global data structure; and the approach will only be worthwhile if its associated overhead costs can be kept very low. Bentley [Be79] identified a large class of common problems (called decomposable searching problems) for which this general technique of converting static data structures into dynamic ones ("dynamization") is applicable. Bentley and Kung [BeKu79] have described how decomposable searching problems can be efficiently solved on a parallel computer.

Definition: A searching problem (or query) Q is said to be decomposable if for any set of objects **S** to which it applies and any partition of **S** into an arbitrary number of disjoint subsets

("blocks") $S_1,\ldots,S_k$, the answer $Q(S)$ can be synthesized in $O(k)$ time from the answers $Q(S_1),\ldots,Q(S_k)$ to the query for each separate block.

A typical example of a decomposable searching problem is the nearest-neighbor searching problem:

Problem: Determine which point of a given set is closest to some arbitrary test point in the plane.

Bentley and Saxe ([Be79], [SaBe79]) have presented several general dynamization techniques which can be applied to any decomposable searching problem, resulting in reasonable update (insertion or deletion) times and without excessively increasing the query times. An objection to their methods is that they primarily support insertions, deletions being much harder to handle in their framework.

1.2.2. Dynamization Methods

Every dynamization technique tends to employ its own method of partitioning a set into blocks and its own procedures for maintaining this decomposition. Bentley's primary method maintains a partition of a set of n points into distinct blocks of size 2^i for particular values of i. There is a block of the appropriate size for each "1" in the binary representation of n. There will exist some blocks of large size, but insertions will

most often require a repartition of points only at the "low end". This method usually adds a factor of lg n to the preprocessing time and the query time of the static data structure. It can be worthwhile to completely avoid large blocks, and maintain instead a partition of the set in which the sizes of the blocks are kept balanced in an optimal way. A dynamization of this type was exhibited by Maurer and Ottman [MaOt79b], although their method presupposed a limit on the largest set-size to occur over time and kept a fixed number of blocks. An improvement of this was shown by van Leeuwen and Wood [vLWo80], who present a fully dynamic scheme which adapts both the size limits on and the number of blocks dynamically at no extra cost. The worst-case bounds on update and query times are optimized, regardless of how the set-size varies. A major advantage of this method is that deletions can be processed quickly, even if the brute force approach of reconstructing an entire block as a new static structure afterwards is used. This is because the blocks which are broken up and rebuilt are much smaller than a single global block (representing the entire set). In [vLMa80], modifications to this "equal-blocks method" of dynamization are shown to improve the average-case bounds on insertion (and occasionally deletion) times.

As is noted in [vLWo80], it is important to recognize that general dynamization techniques shouldn't replace ingenuity in dynamizing a specific problem, but should be brought into action when individually-tailored approaches fail. Also, even though the general dynamization techniques for decomposable searching

problems are suitable for a wide variety of problems, there exist some common geometric configurations (eg. convex hulls [SaBe79]) for which the techniques are inapplicable. It is not possible to perform convex hull searching ("is point x inside the convex hull of point set F?") on a partitioned set. This is because F can be partitioned into two subsets such that a point x is not inside the hull of either subset, yet x is inside the convex hull of F. Some work has been done on dynamically maintaining non-decomposable geometric configurations including planar convex hulls. This will be discussed in more detail in section 4.2.2.

1.3. Topics of the Thesis: Problems and Results

In this thesis, we present several efficient, fully dynamic maintenance algorithms for a variety of geometric configurations. These enable the solution in a dynamic setting of problems which utilize these structures. Some of these problems are decomposable in Bentley's sense; others are not. In many cases, the bounds which we present are the best known to date, and for some problems full dynamization has not previously been discussed. The configurations which are studied are the following:

a) Voronoi diagrams (planar; nearest-point and

farthest-point).

b) Convex hulls (2-D and 3-D versions).

c) Common intersection of halfspaces (2-D and 3-D versions).

d) Contour of maximal vectors (2-D and 3-D versions).

Some important specific results presented in this thesis are:

- the Voronoi diagram of a planar point set can be dynamically maintained at a cost of O(n) time per insertion or deletion of a point, such that it can still be searched in O(lg n) time, where n is the current number of objects in the set (thus allowing the best known solution to the dynamic nearest-neighbor searching problem, and other closest-point problems). (section 2.1.1.2.)
- the convex hull of a planar point set can be maintained at a cost of $O(\lg^2 n)$ time per insertion or deletion (a consequence of an O(lg n) algorithm for computing the union of disjoint convex sets). (section 4.2.2.)
- the common intersection of a set of 2-D halfspaces can be maintained at a cost of $O(\lg^2 n)$ time per insertion or deletion of a halfspace (accomplished by transforming this problem to one dealing with convex hulls). (section 5.1.2.)
- the convex hull of a 3-D point set can be maintained at a cost of O(n) time per insertion or deletion (a similar result is also shown for 3-D halfspaces). (sections 3.2.

and 5.2.2.)

- the contour of maximal vectors of a 3-D point set can be maintained at a cost of O(n) per insertion or deletion. (section 6.2.2.)

Applications of these and other results are discussed throughout the thesis. One of the principal techniques which we employ is the exploitation of fast algorithms for "merging" of arbitrary or ordered substructures to create dynamic hierarchical data structures. Other issues considered in the thesis are the use of extra storage space to improve query and deletion times, and the investigation of tradeoffs among resources and solution characteristics. Specifically, a tradeoff between space and update time is explored, as well as a query vs. update time tradeoff.

1.4. Preliminary Definitions and Notation

We will characterize two distinct types of data structures for solving searching problems. A static structure is built once and can then be searched repeatedly; insertions and deletions of elements are not permitted. We define three functions of n (the number of elements currently in the set)

which portray the performance of a static structure **S** representing the set:

$P_S(n)$ = the preprocessing time required to build S,

$Q_S(n)$ = the query time required to perform a search in S,

$S_S(n)$ = the storage space required to represent S.

Unless otherwise stated, we consider worst-case cost functions throughout this thesis.

Another type of data structure is a dynamic structure, which represents a set whose size changes over time. The structure can be initially empty, and supports the operations of inserting a new element, deleting a current element, and performing a search to answer a query. The following functions of n (the number of elements currently in the set) are used to describe the performance of a dynamic structure **D** representing the set:

$I_D(n)$ = the time required to insert an element into D,

$D_D(n)$ = the time required to delete an element from D,

$Q_D(n)$ = the query time required to perform a search in D,

$S_D(n)$ = the storage space required to represent D,

$P_D(n)$ = the "preprocessing" time required to build D by performing n successive insertions into an initially empty structure,

$k(n)$ = the number of blocks into which D is partitioned.

2. Dynamic Voronoi Diagrams

2.1. Nearest-point Voronoi Diagrams

Voronoi diagrams have many applications in computational geometry as well as in other fields ([Sh78], [Br79]). The worst-case lower bound for their construction is $\Omega(n \lg n)$ time. This follows from Yao's $\Omega(n \lg n)$ lower bound for convex hulls [Ya79], and the fact that the convex hull can be derived from a Voronoi diagram in linear time [Sh78]. Shamos [Sh75] presents an $O(n \lg n)$ time divide-and-conquer algorithm for constructing the Euclidean Voronoi diagram of a set of n planar points. Kirkpatrick [Ki79a] has recently shown an $O(n \lg n)$ time algorithm for constructing the Voronoi diagram of n planar line segments.

Definition: Let **F** be a set of n points in the plane. The nearest-point planar Voronoi diagram of **F** is a network of n polygonal regions (Fig. 1). The region R_i associated with a point p_i in **F** is the set of all the points in the plane which are closer to p_i than to any other point in **F** . The vertices of the regions are called Voronoi points, and the polygonal boundaries of the regions form Voronoi polygons. Voronoi polygons can be bounded or unbounded.

Given an arbitrary point x in the plane, we can determine the point in **F** to which x is closest by finding which Voronoi

polygon contains point x. Thus, we see that searching for a point in a Voronoi diagram allows us to solve the nearest neighbor searching problem.

When we delete a point p_i from **F**, the Voronoi diagram of the revised set may become drastically different. Computing it can involve a complete reconstruction, costing O(n lg n) time. Without closer scrutiny, it might appear that inserting a point into **F** would also necessitate a total reconstruction.

However, Kirkpatrick [Ki79a] showed that two arbitrary Voronoi diagrams can be "merged" efficiently, linear separability of the two point sets involved being unnecessary. More specifically, he demonstrated the following:

LEMMA 2.1. [Ki79a] If **A** and **B** are arbitrary sets of planar points ($|\mathbf{A}|=n$, $|\mathbf{B}|=m$) , then their Voronoi diagrams can be merged to form the Voronoi diagram of (**A** union **B**) in O(n + m) time.

We will show that this result favors efficient dynamic nearest-neighbor searching and other dynamic applications of Voronoi diagrams.

2.1.1. Nearest-Neighbor Searching

The problem of static nearest-neighbor searching can be stated as follows:

Problem: Preprocess a set of n points in such a way that the nearest neighbor of a new test point can be found quickly.

As mentioned previously, finding the Voronoi polygon in which x is located immediately supplies the nearest neighbor of x. The Voronoi diagram of n points in the plane is a straight-line planar graph with O(n) vertices and O(n) edges [ShHo75]. It can be searched by any method for locating a point in a planar subdivision.

Shamos [ShHo75] provided an algorithm which performed planar subdivision search in O(lg n) time, using $O(n^2)$ storage and $O(n^2)$ preprocessing time. Lee and Preparata [LePr76] showed a method with $Q_S(n) = O(\lg^2 n)$, but with $S_S(n) = O(n)$ and $P_S(n) = O(n \lg n)$. Preparata [Pr80] later supplied an alternative algorithm with $Q_S(n) = O(\lg n)$, but with $S_S(n)$ and $P_S(n)$ both equal to O(n lg n). The most efficient approaches so far to static nearest-neighbor searching in the plane are represented by:

LEMMA 2.2. ([LiTa77], [Ki79b]) There exists a planar subdivision search structure with the following attributes:

$$Q_S(n) = O(\lg n)$$
$$S_S(n) = O(n)$$
$$P_S(n) = O(n \lg n) .$$

2.1.1.1. Dynamic Nearest-Neighbor Searching

Nearest-neighbor searching is a decomposable searching problem. The set **S** of n points can be partitioned into an arbitrary number of disjoint subsets $S_1,\ldots,S_k$, and the Voronoi diagrams of each subset maintained separately. The nearest-neighbor query Q can now be applied to each subset in turn, and the answer with minimum distance among all k answers provides the nearest neighbor to the test point.

The first treatment of dynamic nearest-neighbor searching was by Bentley [Be79]. His primary technique usually adds a factor of lg n to both the preprocessing and query times of the static data structure. Using an optimal underlying static structure (eg. lemma 2.2.) , Bentley obtains:

THEOREM 2.1. [Be79] There exists a dynamic nearest-neighbor search structure with the following characteristics:

$$Q_D(n) = O(\lg^2 n)$$
$$P_D(n) = O(n \lg^2 n)$$
$$I_D(n) = O(n \lg n)$$
$$S_D(n) = O(n) \quad .$$

A major drawback of Bentley's method is that it only supports insertion of points; deletions are prohibited. Overmars and van Leeuwen [OvvL79] presented a fairly intricate modification of Bentley's method (allowing slightly variable block-sizes) which allows deletions, at a cost of $D_D(n) = O(n \lg n)$.

These results can be improved by exploiting lemma 2.1. This permits not only linear insertion time but, as observed in

[SaBe79], efficient merging of substructures is superior to dismantling smaller structures followed by rebuilding larger structures from scratch. In fact, this merging permits us to shave a factor of lg n from the preprocessing time of the dynamic structure. Thus, employing this speedup on Bentley's (suitably modified) structure renders the following improved characteristics:

$$Q_D(n) = O(\lg^2 n)$$
$$P_D(n) = O(n \lg n)$$
$$I_D(n) = O(n)$$
$$D_D(n) = O(n \lg n)$$
$$S_D(n) = O(n) \quad .$$

The "equal blocks" dynamization method of van Leeuwen and Wood [vLWo80] offers possibilities for further improvement. They use an application-dependent sequence of "switchpoints" x_k, $k \geq 1$ with the following properties:

a) $x_k \in \mathbf{N}$

b) x_k is a multiple of k

c) $x_{k+1}/(k+1) \geq x_k/k$.

The set being maintained is partitioned into $k = k(n)$ blocks of approximately n/k elements each and a "dump" (also structured and varying in size from 0 to about n/k points), whenever n currently satisfies $x_k \leq n \leq x_{k+1}$. If the value of n falls below x_k or grows to x_{k+1}, then the number of blocks and the limits on their size are adapted correspondingly at negligible cost.

For dynamic nearest neighbor searching, they define their switchpoints by

x_k = the first multiple of k that is $\geq 2^k$.

This makes k = k(n) about lg n. They dynamize an optimal static solution w.r.t. to the stated switchpoint sequence to yield:

THEOREM 2.2. [vLWo80] There exists a dynamic nearest-neighbor search structure with the following characteristics:

$$Q_D(n) = O(\lg n \circledast Q_S(n/\lg n)) = O(\lg^2 n)$$
$$I_D(n) = O(P_S(n/\lg n)) = O(n)$$
$$D_D(n) = O(P_S(n/\lg n)) = O(n)$$
$$S_D(n) = O(\ S_S(n)\ ) = \ O(n)\ .$$

We can exploit lemma 2.1. to improve upon theorem 2.2., achieving $I_D(n) = O(n/\lg n)$.

The general question of optimal tradeoff between query and update times was thoroughly studied by Edelsbrunner [Ed79]. Using his analytic methods, we arrive at $k(n) = n^{1/2}$. The characteristics of the solution become:

$$Q_D(n) = O(n^{1/2} \lg n)$$
$$I_D(n) = O(\ n^{1/2}\ )$$
$$D_D(n) = O(n^{1/2} \lg n)$$
$$S_D(n) = O(\ n\ )\ .$$

These last two solutions in some sense represent the best known results for the dynamic nearest neighbor searching problem. These solutions have required only linear amounts of storage. We will demonstrate that it is possible, at the

expense of a limited amount of extra storage but without sacrificing optimal query time, to improve handling of deletion so that all updates are equally expensive. As above, the global static structure can be maintained as a substructure of the dynamic structure, ensuring that applications aside from queries can be processed just as efficiently as the static case. Because the nearest neighbor searching problem is decomposable, it will also be possible to simultaneously trade query time for update time. These tradeoffs provide a full spectrum of solutions to the dynamic nearest neighbor searching problem.

2.1.1.2. Using Extra Storage

First we will investigate a dynamic structure which uses some extra storage but does not exploit the decomposability of the problem. The base level of our structure is comprised of a system of van Leeuwen-Wood blocks with $k(n) = \lg n$, each block of size approximately $n/\lg n$. However, in addition to this we merge these $k(n)$ structures "upward" to obtain a single top-level structure. This results in a balanced binary tree of structures, with $\lg n$ structures at the leaf level. The height of this tree is $\lg \lg n$, and the entire set of points is represented once at each level of the tree. Thus, $O(n)$ storage is required at each level and the entire dynamic structure needs

O(n lg lg n) space. Since the Voronoi diagram of the complete set is maintained as the root of the structure, the query time remains the same as in the static case, namely O(lg n).

Both insertions and deletions in this structure occur at the leaf level, in the manner specified by van Leeuwen and Wood. After the changes at the leaf level have occurred, the necessary upward re-merging is done to reflect the appropriate changes higher up in the tree, right up to the root. Insertion into a leaf-level structure is performed at a cost proportional to the O(n/lg n) size of the structure, and the upward remerging phase takes O(n) time. For deletion, the cost of reconstructing a leaf-level structure is $O((n/\lg n) \circledast \lg(n/\lg n)) = O(n)$, and the upward remerging also takes O(n) time. Thus, we have

LEMMA 2.3. There exists a dynamic nearest-neighbor search structure with the following characteristics:

$$Q_D(n) = O(\lg n)$$
$$I_D(n) = O(n)$$
$$D_D(n) = O(n)$$
$$S_D(n) = O(n \lg \lg n) \quad .$$

This solution clearly exhibits the benefits of exploiting lemma 2.1. (the linear time merging of two arbitrary Voronoi diagrams).

If we are only willing to use O(n F(n)) storage, where $F(n) < \lg \lg n$, we can demonstrate a structure which uses less storage at the expense of greater deletion cost. Our structure becomes "shorter" than before; it is of height F(n). The number

of leaf-level structures, $k_L(n)$, is $O(2^{F(n)})$. Each one of these structures will be of size approximately $n/2^{F(n)}$. The cost of performing an insertion remains linear. The dominant cost in deletion from this dynamic structure is the actual cost of deletion from a leaf-level structure, not the time required for upward remerging. The deletion time is equal to $D_S(n/2^{F(n)}) = O((n/2^{F(n)}) \circledast \lg(n/2^{F(n)})) = O((n \lg n)/2^{F(n)})$. We thus have the following:

LEMMA 2.4. In the case that we have a unique highest-level structure (the global root structure) and if F(n) is O(lg lg n), then there exists a dynamic nearest neighbor search structure with the following characteristics:

$$Q_D(n) = O(\lg n)$$
$$I_D(n) = O(n)$$
$$D_D(n) = O((n \lg n)/2^{F(n)})$$
$$S_D(n) = O(n F(n)) .$$

2.1.1.3. Exploiting Decomposability

Now we will consider a slightly modified hierarchical dynamic structure, one which exploits the decomposability of the nearest neighbor searching problem. At the base (leaf) level, there are $k_L(n)$ structures. These are merged upwards as before,

but only until a height of lg lg n is achieved. Note that this will not necessarily produce a single root structure. In general, we will have a certain number of "highest-level" structures (denoted by H(n)), each of size about n/H(n). The entire overall structure is a forest of H(n) binary trees. The total number of leaf structures is $k_L(n) = H(n) \circledast \lg n$, each of size $O(n/(H(n) \circledast \lg n))$. This type of structure can be interpreted as a "band" of thickness lg lg n in a binary tree of structures with a unique global structure at the root and the individual points stored as leaves. H(n) can range betwen 1 and n/lg n (i.e. the band can be raised or lowered) to trade off query time against update time (see Fig. 2). Querying is always performed on the highest level structures, at a cost of $H(n) \circledast Q_S(n/H(n))$. Because the height of our structure is lg lg n, the dominant cost in both insertion and deletion is the cost of upward remerging. For deletions, the "break even point" (where the cost of reconstructing a leaf structure equals the upward remerging cost) occurs when H(n) = 1. At that point, both costs are O(n). The query-update tradeoff is formulated as follows:

LEMMA 2.5. In the case where F(n) = lg lg n (i.e. the maximum useful amount of storage is used) and H(n) highest-level structures are maintained ($1 \le H(n) \le n/\lg n$), we have a dynamic nearest neighbor search structure with attributes:

$$Q_D(n) = O(H(n) \circledast \lg(n/H(n)))$$

$$I_D(n) = O(\ n/H(n)\ )$$

$$D_D(n) = O(\ n/H(n)\ )$$

$$S_D(n) = O(n \lg \lg n) \quad .$$

We note that a query time-update time product of $O(n \circledast \lg(n/H(n)))$ is maintained. If we further restrict ourselves to the use of only $O(n\ F(n))$ storage, where $F(n) \leq \lg \lg n$, the result is a dynamic structure with $H(n)$ highest-level structures and $O(H(n) \circledast 2^{F(n)})$ leaf-level structures. We see that the storage-deletion tradeoff can be incorporated simultaneously with the query-update tradeoff.

Lemma 2.4. and lemma 2.5. can be combined to obtain the following result:

THEOREM 2.3. If $F(n)$ is $O(\lg \lg n)$ and $H(n)$ satisfies $1 \leq H(n) \leq n/\lg n$, then there exists a dynamic nearest neighbor search structure with the following attributes:

$$Q_D(n) = O(H(n) \circledast \lg(n/H(n)))$$
$$S_D(n) = O(n\ F(n))$$
$$I_D(n) = O(n/H(n))$$
$$D_D(n) = O((n \lg n)/(H(n) \circledast 2^{F(n)})).$$

We note that this implies a wide spectrum of efficient solutions to the dynamic nearest neighbor searching problem, and unifies earlier results.

2.1.2. Other Dynamic Applications

Recall that the very first hierarchical dynamic Voronoi diagram structure we discussed (lemma 2.3.) could process both insertions and deletions in O(n) time. This can be seen from theorem 2.3. by letting F(n) = lg lg n and H(n) = 1. Since this dynamic structure maintains the entire static Voronoi diagram as a substructure, other applications of Voronoi diagrams can be processed with no penalty over the static case.

The All Nearest Neighbors problem can be stated as follows:
Problem: Given a set of n points in the plane, find a nearest neighbor point (in the set) of each point.

Applications of this problem in mathematical ecology, geography and molecular physics are cited in [Sh78]. A solution to this problem is a set of n ordered pairs (a,b), where b is a nearest neighbor of a. Shamos [Sh78] has shown that, given the Voronoi diagram of the point set, all nearest neighbors can be found in linear time. The argument hinges on the fact that every nearest neighbor of a point p_i defines an edge of the Voronoi polygon V(i). To find a nearest neighbor of p_i, it is only necessary to examine each edge of V(i). Because every edge of the Voronoi diagram belongs to two Voronoi polygons, no edge will be scanned more than twice. Combining Shamos' algorithm with our dynamic Voronoi diagram structure (lemma 2.3.) produces the following result:

THEOREM 2.4. The set of All Nearest Neighbors pairs of a set of n points in the plane can be maintained dynamically at a cost of O(n) steps per insertion or deletion of a point.

Since one of these n ordered pairs is a pair of points that are closest together among all the points in the set, we see that the Closest Pair can also be dynamically maintained at O(n) per update.

Shamos [Sh78] defines a triangulation on n points in the plane to be a planar graph whose edges are straight line segments which join the points so that every region interior to the convex hull is a triangle. The problem of triangulation arises in the finite element method and in numerical interpolation (see [Sh78]). Shamos showed that the straight-line dual of the Voronoi diagram of a point set is a Delaunay triangulation.

Definition: A Delaunay triangulation of a point set is a triangulation with the property that the circumcircle of every triangle contains no points of the set.

Given the Voronoi diagram, the straight-line dual can be constructed in O(n) time by simply joining the pair of points that define each Voronoi edge.

THEOREM 2.5. The Delaunay triangulation of a set of n points in the plane can be maintained dynamically at a cost of O(n) steps per insertion or deletion of a point.

Definition: The Euclidean minimum spanning tree of n points in the plane is a tree of minimum total length whose vertices are the given points.

In the general case, construction of a minimum spanning tree is a graph problem. For arbitrary graphs of e edges and n

vertices, an O(e lg lg n) algorithm is presented in [ChTa76]. The Euclidean minimum spanning is described in [Sh78] as a common component in applications involving communications networks, clustering, pattern recognition and circuit design, and in obtaining approximate solutions to the Travelling Salesman Problem. Shamos [Sh78] shows that the Euclidean minimum spanning tree is a subgraph of the straight-line dual of the Voronoi diagram. It is therefore also a minimum spanning tree of the dual. The dual is a planar graph, for which a minimum spanning tree can be computed in O(n) time [ChTa76], so using our dynamic Voronoi diagram structure we have

THEOREM 2.6. A Euclidean minimum spanning tree of a set of n points in the plane can be maintained dynamically at a cost of O(n) steps per insertion or deletion of a point.

A problem posed in [Sh78] is the Largest Empty Circle.

Problem: Given a set of n points in the plane, find a largest circle that contains no points of the set yet whose center is interior to the convex hull.

This is a facility location problem, in which we would like to situate a new facility within a restricted region so that it is as far as possible from any of n existing ones.

Shamos showed that the convex hull of the set can be found in O(n) time, given the Voronoi diagram. His algorithm for solving the Largest Empty Circle problem requires O(n lg n) time even after the Voronoi diagram of the point set has been computed. He demonstrates that the center of the largest empty

circle must lie either at a Voronoi point or at an intersection of a Voronoi edge and a convex hull edge. The hull intersections he finds in O(n) time, but spends O(n lg n) time checking all of the Voronoi points for hull inclusion. This entire checking process can be accomplished in a slightly different manner, in O(n) time.

THEOREM 2.7. The largest empty circle defined by n points in the plane can be maintained dynamically at a cost of O(n) per insertion or deletion of a point.

Proof: First, we examine the semi-infinite rays of the Voronoi diagram. Each such ray r either intersects its corresponding convex hull edge E, or the circumcenter associated with r lies outside its Delaunay triangle and hence outside the hull. In the latter case we examine the Voronoi edges adjacent to r; either they or some edges "descendant" from them will intersect E. The intersection points are easily computed and then inspected. After this process has been carried out for all rays of the Voronoi diagram, the remaining Voronoi points are guaranteed to be interior to the convex hull, and need merely be inspected, not checked for hull inclusion. Since the number of Voronoi edges and Voronoi points are both O(n), this search for the center of the largest empty circle can be accomplished in O(n) time, given the Voronoi diagram. □

2.2. Farthest-point Voronoi Diagrams

Another type of Voronoi diagram, investigated in [ShHo75], is the planar farthest-point Voronoi diagram (hereafter denoted by FPVD). Shamos describes an O(n lg n) time algorithm for constructing the FPVD of a set of n planar points.

Definition: An FPVD of a set F of n points in the plane is a network of polygonal regions. Region R_i is the set of all points in the plane that are farther from point p_i than from any other point of F (see Fig. 3).

It is significant to note that only points that are vertices of the convex hull of F have associated farthest point regions. Furthermore, every farthest point region is unbounded. The vertices of the regions are called Voronoi points. Each Voronoi point V of the FPVD is equidistant from the three points of F that are farthest from V. A Voronoi point V is the center of a circle that passes through the three farthest points of F and contains all of the other n-3 points. Given an arbitrary point x in the plane, we can determine the point in F which is farthest from x by finding which farthest-point Voronoi region contains point x. Thus, we see that searching for the location of a point in a FPVD allows us to solve the farthest neighbor searching problem.

When a point is deleted from F, the FPVD may change drastically and require complete reconstruction at a cost of O(n lg n) time. This is primarily because deletion of a point on the convex hull of F may cause many points which were previously interior to become vertices of the hull. Insertion is somewhat easier to handle. Shamos [ShHo75] outlines a method

for "merging" the FPVDs of two linearly separated point sets in O(n) time. We will show that two arbitrary FPVDs can be merged efficiently, linear separability of the two point sets involved being unnecessary.

More specifically, we demonstrate the following:

LEMMA 2.6. If P and Q are arbitrary sets of points ($|P|=n$, $|Q|=m$), then their FPVDs can be merged to form the FPVD of (P union Q) in $O(n + m)$ time.

It follows trivially that maintenance of a FPVD upon insertion of a point can be accomplished in only O(n) time. This will aid efficient dynamic farthest-neighbor searching and other dynamic applications of FPVDs.

Our algorithm for merging FPVDs is similar in spirit to the algorithm presented in [Ki79a] for merging closest-point Voronoi diagrams. Suppose we are given two arbitrary disjoint point sets P and Q, as well as FPVD(P) and FPVD(Q). The point sets P and Q together impose a framework on the plane quite independent of their individual FPVDs. Defining distance to the set P (respectively Q) to be the maximum distance to an element of P (respectively Q), we can partition the plane into points farther from P, the P-region, points farther from Q, the Q-region, and points equidistant from P and Q, called the contour induced by P and Q. The contour is composed of straight line segments. It is formed from the edges of FPVD(P union Q) that separate regions associated with points in P from regions associated with points in Q. It is also the case that FPVD(P union Q) and FPVD(P) are identical in the P-region, as are FPVD(P union Q)

and FPVD(Q) in the Q-region. Thus, FPVD merging can be interpreted as the process of tracing out the components of the contour, along with appropriate piecing together of what remains. In general, the contour may have many connected components, which we will now describe how to initially locate and trace to termination.

2.2.1. Tracing the FPVD Contour

Consider, for example, the situation shown in Fig. 4, where **P** = {A,B,C} and **Q** = {D,E,F}. From FPVD(P) and FPVD(Q), it is trivial to form the convex hulls of both P and Q. Next we intersect the two hulls; [Sh78] describes how to find the intersection of a convex m-gon and a convex n-gon in O(m + n) time. This allows us to form the convex hull of (P union Q). A list is maintained of the new edges introduced between a point from P and a point from Q. It is these pairs of points which determine the initiation of contour components. In the example, the vertices of the convex hull of (P union Q) are A, B, E and F. The "new" edges introduced are (A,F) and (B,E).

Tracing a contour component involves following the perpendicular bisector of the current farthest point in each FPVD until either (i) a V-edge is crossed, i.e. a current farthest point must be updated or (ii) the contour enters a

region where the two farthest points coincide with another pair on the "new edge" list, in which case that contour component is traced out to infinity and terminated. In the example, we initiate a contour component at infinity, along the perpendicular bisector of A and F. We trace along this bisector in a region where A and F are the current farthest points until we cross the Voronoi edge which is the bisector of A and B (to determine which V-edge intersects the contour first, scan the edges of one FPVD in a clockwise direction, those of the other counterclockwise [Le76]). The current farthest point in P is updated to become B, and we follow the perpendicular bisector of B and F until we cross the Voronoi edge which is the bisector of E and F. The new farthest point in Q becomes E, and we follow the perpendicular bisector of B and E. This bisector is traced out to infinity because (B,E) exists on the new hull edge list. In this example, the contour component which was just traced to completion was the only component since it accounted for both of the new hull edges (A,F) and (B,E). In general, there will be one half as many contour components as there are edges on the new edge list. Notice that the area above the contour component in Fig. 4 is the Q-region. Therefore, we append the portion of FPVD(Q) which lies in this region (the upper piece of the perpendicular bisector of E and F) to FPVD(P union Q). The P-region lies below the contour, and a corresponding part of FPVD(P) is also included in FPVD(P union Q).

We see from the construction that FPVD(P union Q) consists solely of some pieces of FPVD(P), some of FPVD(Q), and the

contour. It is easy to show that every V-edge (in both FPVD(P) and FPVD(Q)) is crossed at most twice by the contour. It follows that the total cost of tracing the entire contour (i.e. all components) is proportional to the total number of V-edges.

Our algorithm for computing FPVD(P union Q), given FPVD(P) and FPVD(Q), runs in $O(|P| + |Q|)$ time and makes no assumptions about the separability of the point sets P and Q. This result of course implies that maintenance of an FPVD on n points upon insertion of a new point can be done in O(n) time. It also leads in a recursive divide-and-conquer manner to an O(n lg n) algorithm for computing the FPVD of a set of n points.

2.2.2. Farthest-Neighbor Searching

Farthest-neighbor searching is closely related to nearest-neighbor searching, which was discussed in section 2.1.1.

Problem: Preprocess a set of n points in such a way that the farthest neighbor of a new test point can be found quickly.

Determining which farthest-point Voronoi region contains point x provides the farthest neighbor of x. The FPVD of n points is a straight-line planar graph with O(n) vertices and O(n) edges [ShHo75]. In exact analogy to the nearest-neighbor case, the FPVD can be searched by any of the methods for

locating a point in a planar subdivision. We can use (lemma 2.2.) the point-location algorithms of [LiTa77] or [Ki79b] to achieve the following bounds for static farthest-neighbor searching in the plane:

$$Q_S(n) = O(\lg n)$$
$$P_S(n) = O(n \lg n)$$
$$S_S(n) = O(n) \quad .$$

Farthest-neighbor searching is also a decomposable searching problem. A set S of n points can be partitioned into an arbitrary number of disjoint subsets $S_1,\ldots,S_k$, and the FPVDs of each subset maintained separately. The farthest neighbor query Q can now be applied to each subset in turn, and the answer with maximum distance among all k answers provides the farthest neighbor to the test point.

Since the insertion time for a static FPVD is O(n) and the deletion time is O(n lg n) and we can exploit linear-time merging, the parameters of the farthest-neighbor searching problem are identical to those of nearest-neighbor searching. Therefore, all the previously stated results for dynamic nearest-neighbor searching apply directly and remain unaltered for the problem of dynamic farthest neighbor searching. In particular, we can supply a dynamic solution (using only a linear amount of storage) with the following attributes:

$$Q_D(n) = O(\lg^2 n)$$
$$I_D(n) = O(n/\lg n)$$
$$D_D(n) = O(n)$$

$$S_D(n) = O(\ n\ ) \quad .$$

The storage-deletion and query-update tradeoffs discussed in sections 2.1.1.2. and 2.1.1.3. also apply in this new setting, and can be used to supply a similar wide spectrum of efficient solutions to the dynamic farthest-neighbor searching problem (see theorem 2.3.).

THEOREM 2.8. If $F(n)$ is $O(\lg \lg n)$ and $H(n)$ satisfies $1 \leq H(n) \leq n/\lg n$, then there exists a dynamic farthest neighbor search structure with the following attributes:

$$Q_D(n) = O(H(n) \circledast \lg(n/H(n)))$$
$$S_D(n) = O(n\ F(n))$$
$$I_D(n) = O(n/H(n))$$
$$D_D(n) = O(\ (n \lg n)/(H(n) \circledast 2^{F(n)})\ ) \quad .$$

2.2.3. Other Dynamic Applications

Letting $F(n) = \lg \lg n$ and $H(n) = 1$ in theorem 2.8. results in a hierarchical dynamic FPVD structure which can process both insertions and deletions in $O(n)$ time. Recall that this dynamic structure maintains the entire static FPVD as a

substructure. This allows other applications of farthest point Voronoi diagrams to be processed with no penalty over the static case.

Definition: Given a set S of n points in the plane, the diameter of the set is the maximum distance between any two of its points.

In [ShHo75] it is stated that the two farthest points of S can be found in O(n) time, given FPVD(S). This is done by examining each edge of the diagram and computing the distance between the points determining it. The greatest of these distances is the diameter of S. Combining this algorithm with our dynamic FPVD structure allows us to maintain a set of points at O(n) time per update, such that the diameter of the set can be computed in O(n) time when desired. Some applications of set diameter in clustering are mentioned in [ShHo75]. We will return to the problem of maintaining set diameter in chapter four.

The Smallest Enclosing Circle problem can be stated as follows:

Problem: Given n points in the plane, find the smallest circle enclosing them.

This problem is one of minimax facilities location, in which a point x (the center of the circle) is sought, whose greatest distance to any point of the set is a minimum. It is discussed in [ShHo75], which mentions applications in siting emergency services, where worst-case response time can be an important consideration, and in locating radio transmitters.

An algorithm is presented in [ShHo75] which is based on the

FPVD of the set. The smallest enclosing circle is determined either by three points of the set or two points which define a diameter. If the circle determined by the diameter encloses the set, the problem is solved. Otherwise, it is shown that the center of the smallest enclosing circle is a vertex of the farthest point Voronoi diagram. The FPVD contains only O(n) vertices, so the circumradii can be found in O(n) time, and the vertex with maximum circumradius is the center of the smallest enclosing circle. Therefore, the smallest enclosing circle can be computed in O(n) time, given the FPVD of the point set. Combining this algorithm from [ShHo75] with our dynamic FPVD structure results in the following:

THEOREM 2.9. The smallest enclosing circle of n points in the plane can be maintained dynamically at a cost of O(n) steps per insertion or deletion of a point.

3. Restricted Linear Merging and Dynamic 3-D Convex Hulls

3.1. Dynamization using Restricted Linear Merging

In chapter two, we efficiently maintained dynamic Voronoi diagrams by taking advantage of linear algorithms for the merging of arbitrary structures (lemma 2.1. and lemma 2.6.) and the van Leeuwen-Wood [vLWo80] technique for dynamically maintaining a set. For some other geometric data structures, there exist linear algorithms to merge "separable" structures, but we know of no such algorithm for the merging of arbitrary structures. The convex hull of a set of points in 3-space [PrHo77] is an example of such a geometric data structure. It turns out that this restricted linear merging speeds up the dynamic maintenance of the associated data structures by the same amount as the unrestricted linear merging discussed in chapter two.

The technique described in [vLWo80] basically allows us to dynamically maintain approximately equal-sized subsets of a set of n points in logarithmic update time. As presented, the technique does not directly lend itself to maintaining linearly (or planarly in 3-d) separable subsets of points. However, we demonstrate that it is possible to preserve separability with no (asymptotic) increase in the amount of work.

LEMMA 3.1. If k(n) is monotonic increasing, it is possible to dynamically maintain a set of n points in O(k(n)) linearly (or planarly) separated subsets each of size O(n/k(n)) at a cost of O(lg n) steps per update.

Proof: Our technique is a variation on the van Leeuwen-Wood scheme. The set of n points is maintained in such a manner that each subset has a size in the range $[\ \lfloor n/k(n) \rfloor, 2\lfloor n/k(n) \rfloor + 2\ ]$. Note that this will automatically also keep the total number of subsets within bounds. If the size of a particular subset lies at either extreme of this range, we will call it critical. The following strategy has the property that it maintains the invariant condition: all subset sizes lie inside the specified range and at most $k(n) \circledast (\lfloor n/k(n) \rfloor + 1) - n$ subsets are critical.

All subsets are maintained in a priority queue, ordered by subset size. Upon insertion (deletion) a dictionary is checked to determine membership and to identify the subset which should obtain (contains) the specified element. This appropriate subset is then updated. Next, assuming at least one subset is critical, some global maintenance is performed to preserve the invariant condition. A subset of size $\lfloor n/k(n) \rfloor$ (or smaller, if one exists) is merged with one of its "neighboring" separated subsets, and the resulting subset is inserted back into the priority queue. Also, a set of size $2\lfloor n/k(n) \rfloor + 2$ (or larger, if one exists) is split into two (nearly) equal-sized separated subsets, both of which are inserted back into the priority

queue.

Since each update involves at most one dictionary reference and a constant number of priority queue operations both on sets of size O(n), (which are all O(lg n) operations) the result follows . □

We can now proceed much as we did in the previous chapter, maintaining a balanced binary tree of geometric data structures, with the global structure at the root, and k(n) separated subset structures at the leaves. Insertions and deletions are first processed at the leaf level (possibly involving complete reconstruction of a leaf structure), then the remainder of the tree is updated by re-merging along the entire path to the root, as well as where rebalancing has affected the tree. It is important to preserve the separability condition when rebalancing takes place.

This technique of using extra storage to improve deletion cost applies to any data structures that can be merged (exploiting "separability" if necessary) faster than they can be totally reconstructed. This efficacy of "separable merging" permits some new applications of the basic dynamic tree technique. In the next section , we discuss the dynamization of the convex hull of points in 3-space. In later chapters we will see that the technique also applies to the common intersection of halfspaces in 3-space, and to maximal vectors of a set of points in 3-space.

3.2. Dynamic 3-d Convex Hulls

The convex hull of a set of points in 3-space is a geometric data structure which has applications in computer graphics, pattern classification, and in other problems of computational geometry ([PrHo77], [Sh78], [Br79]). Preparata and Hong [PrHo77] supply an algorithm which computes the 3-d convex hull of n points in O(n lg n) time. From a dynamic viewpoint, the deletion of a hull point can necessitate a complete O(n lg n) reconstruction, while a restricted linear merge permits an O(n) insertion cost. Preparata and Hong [PrHo77] show that convex hulls of planarly separable point sets in 3-space can be merged (to form the convex hull of their union) in linear time.

Since we have a linear merge algorithm for separable subsets, we can employ the tree structure of the previous section for efficient dynamization of 3-d convex hulls. If we maintain a tree structure with O(lg n) leaf sets each of size O(n/lg n), we obtain,

THEOREM 3.1. The convex hull of a set of n points in 3-space can be maintained dynamically at a cost of O(n) steps per insertion or deletion.

Note that the tree has height O(lg lg n), and thus uses O(n lg lg n) storage. The cost of reconstruction at the leaf level is O((n/lg n) ⊛ lg(n/lg n)), that is O(n), which equals

the cost of updating the remainder of the tree.

The same idea can be used with a tree of fewer levels, consequently using less storage. This will have the effect of saving some storage, at the expense of increased deletion cost. Let $F(n)$, $1 \leq F(n) \leq \lg \lg n$, be the height (number of levels) to which we maintain the 3-d convex hull tree structure. Then the same algorithm leads to,

Corollary 3.1. The 3-d convex hull of a set of n points can be dynamically maintained in $O(n)$ time per insertion and $O((n \lg n)/2^{F(n)})$ time per deletion, using $O(n F(n))$ storage.

The problem of determining whether a given test-point is inside a 3-d convex hull is not a decomposable searching problem; the query cannot be answered by inspecting subsets, but must refer to the global structure. Since our dynamization method maintains the entire convex hull, it makes possible fast dynamic hull inclusion queries. The problem of determining whether a point in three-space lies within a convex polyhedron can be transformed to locating a point within two planar straight-line graphs [Br79], as follows. Break the polyhedron into UPPER and LOWER parts, and project both parts orthographically to planar graphs in the xy plane. If a 3-d point t projects to a 2-d point in region A of the UPPER graph and region B of the LOWER graph, then t lies within the convex hull iff t lies below face A and above face B of the polyhedron. The point location can be solved in $O(\lg h)$ time, where h is the number of hull points (lemma 2.2.).

If the inclusion query were decomposable, then subset search structures (analogous to nearest neighbour search structures of chapter two) could have been utilized, since they are compatible with the general dynamization technique. This would provide the additional tradeoff, between query and update times. Examples of this are presented in chapters five and six for decomposable searching problems concerning the common intersection of halfspaces and the contour of maximal vectors in three dimensions.

4. Dynamic Planar Convex Hulls

4.1. Planar Convex Hulls

The convex hull of a set of n points in the plane is defined to be the smallest convex set that contains all of the points (Fig. 5). Aside from being a fundamental tool in computational geometry, convex hulls are important in practical applications such as cluster analysis and statistics. Shamos [Sh 78] discusses many different algorithms for determining the convex hull of a set of n points in the plane. The algorithms usually run in O(n lg n) time, which is optimal [Ya79], and operate on a static set of points. Shamos was the first to suggest and present an on-line convex hull algorithm, which received one point at a time and updated the convex hull accordingly. Preparata [Pr79] recently described an algorithm which constructed the convex hull by successive insertions, with an O(lg n) time bound per insertion.

All of the previous algorithms support only insertions at best, none being fully dynamic. Restoring the convex hull upon deletion of points from the set can be a much more difficult task than insertion. Any fully dynamic convex hull algorithm cannot discard points in the interior of the current convex hull. This can be demonstrated as follows. Intuitively, a stretched rubber band surrounding the set, when released, will

assume the shape of the convex hull. If we allow a deletion of a current hull point to occur, the hull can "snap inward", causing old formerly interior points to become part of the new updated convex hull. The first fully dynamic convex hull algorithm was recently presented by Overmars and van Leeuwen [OvvL80]. They showed that the convex hull of a set of n points can be dynamically maintained at a cost of $O(\lg^3 n)$ per insertion or deletion.

In section 4.2, we will present an $O(\lg(n+m))$ algorithm for computing the convex hull of the union of two disjoint convex polygons. Several applications of the algorithm are presented and discussed. In particular, it leads to improvements on the time bounds claimed by Overmars and van Leeuwen for dynamic maintenance of planar convex hulls.

4.2. Union of Disjoint Convex Polygons

Consider two linearly separable convex polygons A and B in the plane, having n and m vertices respectively. By linearly separable , we mean that there exists a line l of some orientation which isolates the two polygons in separate half-planes. To find the convex hull of the union of A and B, it suffices to find the two non-intersecting segments whose defining lines are mutually tangent to A and B, and hence the

vertices of A and B which are the endpoints of those segments (see Fig. 6). The two such segments (mutual supporting tangents, or "bridges") found become edges of the union hull, and the edges of A and B which are inside the resulting polygon are discarded. This process of "merging" convex hulls is a generalization of Preparata's [Pr79] approach to updating planar convex hulls in real-time (O(lg n) bound on update time per inserted point; see Fig. 7).

This same problem of finding "bridges" was solved by Shamos [Sh78] as a step in computing the Voronoi diagram of a planar point set; he presented an O(n+m) algorithm. Our faster solution does not, however, result in an asymptotic improvement in the run-time of Shamos' O(n lg n) Voronoi diagram algorithm. Preparata and Hong [PrHo77] also find bridges between hulls, that being the merge step of their divide-and-conquer algorithm for computing convex hulls of planar point sets. Their merge is performed in O(n+m) steps. When their merge step is replaced by one of sublinear complexity (eg. our O(lg(n+m)) algorithm), the result is an algorithm which finds the convex hull of n points in only O(n) time, after all the points have been sorted. This was noted by Overmars and van Leeuwen [OvvL80], who find a bridge between two separated hulls in $O(\lg^2(n+m))$ time as part of a dynamic convex hull algorithm.

4.2.1. The Algorithm

When finding bridges, we observe that the upper and lower portions of the polygons can be treated quite separately. To find the "tops" and "bottoms" of both convex polygons, we do the following: consider a line l which separates polygon A from polygon B and project all the vertices of both polygons onto the perpendicular bisector of line l (see Fig. 8). For each polygon we then find the vertices with minimum and maximum coordinates on the axis of projection (those vertices marked with X in Fig. 8). These will be the vertices which split the polygons into "top" and "bottom" chains. One bridge will connect a vertex from the top of A to a vertex from the top of B, while the other bridge connects two vertices from the bottom chains. Mutual separating tangents, which are discussed in section 3.1, connect a vertex from the top of A to one from the bottom of B, and a vertex from the bottom of A to one from the top of B.

For ease of illustration and without loss of generality, consider the following situation: we have two convex polygons A and B separated by a vertical line, and we want to find the "upper" bridge between them (finding the "lower" bridge is analogous). Observe that this "upper" bridge will connect two points from the "upper" chains of edges of A and B (defined with respect to the leftmost and rightmost points of both A and B). The edges of the polygons are in an ordered representation. We locate the "middle" edges m_A and m_B of A and B, extend them in both directions, and examine the three possible cases (see Fig. 9):

i.(Fig. 9a) The extensions of m_A and m_B "overshoot" (are above) one another. In this case, we find the intersection point of the extensions of m_A and m_B, and determine to which side of the dividing line it lies. In the example, the intersection point lies on the same side of the dividing line as A. By convexity, segments to the left of m_A have steeper slopes than m_A, so considering their extensions instead would result in intersection points even further to the left. Since polygon B is constrained to lie to the right of the separating line and beneath the extension of m_B, the edges of A which are to the left of m_A could not supply possible tangent vertices and therefore can be eliminated from further consideration.

ii.(Fig. 9b) The extension of m_A overshoots m_B; extension of m_B falls below m_A (the opposite situation is analogous). In this case, observe that the edges of B to the left of m_B have even steeper slopes than m_B, hence their extensions would fall even lower below m_A. Therefore, their vertices are impossible candidates for tangent points and are discarded.

iii.(Fig. 9c) The extensions of m_A and m_B "undershoot" (are below) one another. In this case, we see that the edges of A to the right of m_A and those of B to the left of m_B can both be eliminated from further consideration due to where the extensions of these steeper-sloped edges would lie.

In each case, after some edges have been discarded, we find the "middle" edge of the remaining edges in that polygonal chain, extend it to a line and determine which of the previous situations occurs. This process is iterated until one of the polygonal chains is reduced to a unique point (which will be an endpoint of the upper bridge). At this stage, we find the corresponding endpoint on the other polygonal chain by performing a one-point convex hull update [Pr79] at logarithmic cost.

Observe that at each stage of the algorithm, we discarded at least half of the edges of at least one of the polygonal chains. The "lower" bridge is found analogously. Hence we have

LEMMA 4.1. Let A and B be two disjoint convex polygons in the plane, with n and m vertices respectively. The two mutual supporting tangents of A and B can be computed in O(lg (n+m)) time.

We note that in the general case, a separating line between A and B will not be vertical, but can be found in O(lg (n+m)) time by a method due to Chazelle and Dobkin [ChDo80] .

4.2.1.1. An Application: Mutual Separating Tangents

A very similar method can be employed to find the two mutual separating tangents of two non-intersecting convex

polygons (see Fig. 10). Without loss of generality, assume we have two convex polygons A and B separated by a vertical line, and we want to find the mutual separating tangent between them which connects a vertex from the top of A to a vertex from the bottom of B (the case for the mutual separating tangent from bottom of A to top of B is analogous).We locate the "middle" edges m_A and m_B of A and B, extend them in both directions, and examine the three possible distinct cases (Fig. 11):

i.(Fig. 11a) The extension of m_A undershoots m_B; extension of m_B overshoots m_A. In this case, we find the intersection point of the extensions of m_A and m_B and determine whether it lies to the right of m_B, or to the left of m_A. In the example, it lies to the right of m_B. We note that the edges of B which are to the left of m_B have even steeper slopes than m_B and that by convexity A is constrained to lie below the extension of m_A. The segments of B to the left of m_B would overshoot A even more drastically, and the furthest left that the intersection point could occur at would be beneath the leftmost point of B. Hence, the segments of B to the left of m_B can be eliminated from further consideration.

ii.(Fig. 11b) The extension of m_A lies above m_B; extension of m_B lies above m_A. In this case, we note that the segments of A to the left of m_A have extensions which would lie even higher above m_B than does the extension of m_A.

Hence, none of their vertices are possible tangent points so the edges can be discarded.

iii.(Fig. 11c) The extension of m_A lies above m_B; extension of m_B lies below m_A. The edges of A to the left of m_A can be discarded by the same reasoning employed in (ii); we also note that the edges of B to the right of m_B can be discarded because their slopes are greater and their extensions would fall even lower below m_A, so the points on those edges are ineligible.

In each case, after edges have been discarded, we find the "middle" edge of the remaining edges, extend it to a line and determine which of the three previous cases occurs, until we perform a one-point update, as in section 4.2.1. At each stage, we discarded at least half of the edges of at least one of the polygonal chains. Hence we have

LEMMA 4.2. Let A and B be two disjoint convex polygons in the plane, with n and m vertices respectively. The two mutual separating tangents of A and B can be found in O(lg (n+m)) time.

An interesting observation is that, unlike the algorithm in section 4.2.1., here we only make use of the orientation of the separating line, not its actual position. We will see in chapter five the importance of mutual separating tangents in maintaining the common intersection of a set of halfspaces.

4.2.2. Dynamically Maintaining Planar Convex Hulls

Most existing convex hull algorithms find the convex hull of a static set of n points in the plane in O(n lg n) time, and are not suited for insertions or deletions to the point set. Preparata [Pr79] exhibits an algorithm for inserting a point into a set and updating the convex hull accordingly in O(lg n) time, but his algorithm cannot handle deletions. As previously mentioned, the first fully dynamic algorithm for maintaining planar convex hulls is due to Overmars and van Leeuwen [OvvL80] . They represent separately the right and left faces of the hull (we will represent the upper and lower faces, for later convenience; see Fig. 12). They also maintain some information about the arrangement of the points currently in the interior of the hull, because interior points could become hull points upon a deletion. Points of convex hull fragments are stored in concatenable queues, which are associated with nodes of a balanced binary search tree. They employ efficient splitting and joining together of concatenable queues. The clever data-structuring techniques are necessary to fully exploit a fast "bridge-finding" hull merge; it is necessary to avoid swamping this sublinear time by the time spent copying portions of data structures.

More precisely, Overmars and van Leeuwen represent hull-faces of horizontally separated subsets of points, and "merge" separated hull-faces of n and m points (i.e. find a

bridge) in $O(\lg^2 (n+m))$ time. They are thus able to maintain the convex hull of n points at a cost of $O(\lg^2 n)$ per insertion and $O(\lg^3 n)$ per deletion ([OvvL 80], thm. 4.5). Our improved hull-merging technique (lemma 4.1.) results in a less expensive deletion cost, and hence the following:

THEOREM 4.1. : The convex hull of a set of n points in the plane can be dynamically maintained at a cost of $O(\lg^2 n)$ per insertion or deletion.

4.2.3. Applications of the Dynamic Convex Hull Algorithm

Theorem 4.1. leads to subsequent improvements in the applications of the dynamic convex hull algorithm investigated by Overmars and van Leeuwen, for example:

- a set of n points in the plane can be "peeled" in $O(n \lg^2 n)$ steps (important in statistics; previous bound was $O(n \lg^3 n)$).
- the joint convex layers of a set of n points in the plane can be computed in $O(n \lg^2 n)$ steps (previous bound $O(n \lg^3 n)$).
- a "spiral" connecting all n points in the plane can be found in $O(n \lg^2 n)$ steps (previously $O(n \lg^3 n)$).

Saxe and Bentley [SaBe79] posed a problem regarding dynamic convex hull searching ("is test-point t within the convex hull of a set of points **F** ?"), to which their methods were inapplicable. Our result improves upon that of Overmars and van Leeuwen.

THEOREM 4.2. A set **F** of n points in the plane can be maintained dynamically at a cost of $O(\lg^2 n)$ per insertion or deletion, such that convex hull searching queries can be answered in only $O(\lg h)$ time (h = number of points on the hull).

Yet another application involves separability of two sets in the plane. Two sets are linearly separable iff their convex hulls are disjoint ([Sh78]). Employing the static separability algorithm of Chazelle and Dobkin[ChDo80], we have the following:

THEOREM 4.3. Two sets **A** and **B** in the plane can be maintained dynamically at a cost of $O(\lg^2 n)$ per insertion or deletion (n = current size of set), such that separability, whenever desired, can be decided in $O(\lg n)$ time.

Shamos [Sh78] showed that the diameter of a planar point set (the maximum distance between any two points) occurs between two points on the convex hull of the set, and can be found in time linear in the number of hull points. It follows that a planar point set can be maintained dynamically at $O(\lg^2 n)$ per insertion or deletion such that the diameter of the set can be found in $O(h)$ time.

5. Dynamic Intersection of Halfspaces

5.1. Common Intersection of Halfplanes

The common intersection of a set of n halfspaces is a convex polygonal region (possibly open or empty) bounded by at most n edges. Shamos [Sh78] showed that the intersection of two convex planar polygonal regions with n and m edges can be found in O(n + m) time, and used this to find the common intersection of n halfplanes in O(n lg n) time. Brown [Br79] uses a geometric transform to map the problem of intersecting halfplanes to two problems of constructing the convex hull of a planar point set, and a simple intersection problem, also resulting in an O(n lg n) algorithm. His algorithm works as follows:

The halfplanes are partitioned into two sets, UPPER and LOWER, where a halfplane is in UPPER if the line at its boundary is above the rest of the halfplane (analogously for LOWER). Then,

a) Construct U, the intersection of the UPPER halfplanes.

b) Construct L, the intersection of the LOWER halfplanes.

c) Find the intersection of regions U and L (involves finding points of intersection P and Q; see Fig. 13).

To construct U, Brown uses a transform which maps the UPPER halfplanes $y \leq a_i x + b_i$ to the points (a_i,b_i). He then proves that the non-redundant halfplanes correspond to those points which are on the lower face of the convex hull of the points (a_i,b_i), thus reducing the problem of intersecting n UPPER halfplanes to the problem of constructing the lower face of the convex hull of n points.

5.1.1. Merging Chains of Halfplanes

Note that non-redundant halfplanes contribute to the common intersection in the order of their slopes. Using Brown's transform, we observe that finding the common intersection of two ordered chains of halfspaces can be accomplished using our algorithm for union of two convex hulls. For example, finding the point where two chains (ordered by slope) of UPPER halfplanes intersect transforms to finding the "lower" bridge of two "lower" convex hulls (see Fig. 14). The two points which form the endpoints of the bridge map to the two halfplanes which create the intersection point, and the points which are not on the newly-updated lower convex hull correspond to the redundant UPPER halfplanes.

Let $\{h_1, \ \ldots \ , h_n\}$ be a set of n arbitrary halfplanes, sorted by angle (or slope). We observe that for any $1 \leq i \leq n$,

the two "slope-separated" sets of halfspaces $\{h_1 \ldots h_i\}$ and $\{h_{i+1} \ldots h_n\}$ are transformed into two separated (by a vertical line) point sets. Hence, the convex hulls of the point sets generated by Brown's transform are linearly separable and thus amenable to the use of our convex hull merging algorithm.

Brown performs the merge step of his algorithm (finding the common intersection of U and L, i.e. finding P and Q in Fig. 13) in O(n) time, working directly with the chains of halfspaces. This can be improved to O(lg n) by using a simple technique which works with the transformed representation (i.e. convex hulls). Recall that U, the intersection of the UPPER halfplanes, is represented by a lower convex hull and L by an upper convex hull. Brown's transform [Br79], which maps the line $y = ax + b$ to the point (a,b) and the point (a,b) to the line $y = -ax + b$, has several interesting features. For instance, distances in the y-coordinate between points and lines are preserved by the transform. Thus, the incidence of point on line is preserved, as well as above-belowness between points and lines. From the properties of the transform, the two hulls will be linearly separable (assuming non-empty intersection of U and L), and hence we can find their two mutual separating tangents ($line_P$ and $line_Q$ in Fig. 15). These can be found in O(lg n) time as described earlier (lemma 4.2.). Note that the case in which the two hulls are not separable (which implies that the common intersection of U and L is empty) can be detected in O(lg n) time by the method of Chazelle and Dobkin [ChDo80].

The points in Fig. 15 labelled U_P and L_P (defining $line_P$) correspond to precisely those two halfplanes, one UPPER and one LOWER, which intersect at point P (see Fig. 13). A similar situation holds for U_Q and L_Q. This can be seen from the following considerations. Points which are inside (U intersect L) get mapped by the transform to lines which separate the upper hull from the lower, because all such points are above all the LOWER halfplanes and below all the UPPER halfplanes. The point P, which has minimum x-coordinate among all these points, gets mapped by the transform (recall, (a,b) maps to y = -ax + b) to the separating line which has maximum slope; and Q (maximum x-coordinate) gets mapped to $line_Q$, which is the separating line with the smallest slope. The separating lines $line_P$ and $line_Q$ are mutual tangent lines, incident on one point each from the upper and lower hulls because the transform preserves incidence, and points P and Q are each incident on one LOWER and one UPPER halfplane. The points on the (lower) hull <u>between</u> U_P and U_Q correspond to those UPPER halfplanes which actually participate in forming the common intersection of U and L, and similarly for the points on the (upper) hull between L_P and L_Q. The encircled points on the hulls in Fig. 15 correspond to redundant halfspaces.

5.1.2. <u>Dynamic Halfspace Intersection</u>

Brown [Br79] mentions dynamic maintenance of the common intersection of halfspaces as an open problem. Overmars and van Leeuwen [OvvL80] solve this by representing separately the two "directions" of halfspaces (left and right, or UPPER and LOWER), using dynamic data structures similar to the ones used for planar convex hulls in chapter four. They utilized an $O(\lg^2 n)$ procedure to find the common intersection of two ordered chains of n halfspaces. Our O(lg n) algorithm for finding the common intersection leads to improvements on the results presented in [OvvL80].

THEOREM 5.1. The common intersection of a set of n halfspaces in the plane can be dynamically maintained at a cost of $O(\lg^2 n)$ steps per insertion or deletion.

As mentioned in [OvvL80], a byproduct of such a theorem is an algorithm to construct the common intersection of a set of halfspaces which takes only O(n) time after the halfspaces have been sorted by direction. One application of theorem 5.1. is a halfspace searching problem which is decomposable ([Be79],[SaBe79]);it is amenable to general, but in this case less efficient, dynamization methods.
Problem: Does test-point t belong to the common intersection of a set **F** of n halfspaces?".

THEOREM 5.2. The common intersection of a set of n halfspaces in the plane can be dynamically maintained at a cost of $O(\lg^2 n)$ per insertion or deletion, such that a halfspace

searching query can be answered in O(lg h) time, where h is the number of non-redundant halfspaces.

Improvements can also be made to two other applications in [OvvL80] :

- the kernel of a (simple) n-gon can be maintained at a cost of $O(\lg^2 n)$ per insertion or deletion of an edge.
- the feasible region of a linear programming problem in two variables can be dynamically maintained at a cost of $O(\lg^2 n)$ per insertion or deletion of an inequality.

5.2. Common Intersection of Halfspaces in Three Dimensions

The common intersection of a set of n halfspaces in three dimensions is a convex polyhedral region bounded by at most n faces, and having O(n) edges and vertices. Both Zolnowsky [Zo78] and Preparata and Muller [PrMu79] have presented O(n lg n) algorithms for solving the general problem of intersecting three-dimensional halfspaces. Brown [Br79] applies a (3-d) point/plane transform to construct the intersection of n UPPER halfspaces in O(n lg n) time.

The transform used by Brown in three dimensions is a straightforward extension of the two-dimensional transform; planes transform to points, and points transform to planes. The

formulae of the transform are:

$z = ax + by + c \quad \rightarrow \quad (a,b,c)$, and

$(x,y,z) \quad \rightarrow \quad c = -xa + -yb + z.$

The transform preserves the distance between a point and a plane in the z coordinate, as well as the sense of above/belowness (in the z coordinate) between points and planes.

Brown's algorithm for constructing U (the intersection of the UPPER halfspaces) in three dimensions is analogous to the algorithm for the two-dimensional case. He first transforms the n UPPER halfspaces to n points in abc space and constructs the bottom part of the convex hull of the points in O(n lg n) time. For the hull construction, he utilizes the 3-d convex hull algorithm of Preparata and Hong [PrHo77]. Brown proves that the points above the bottom part of the convex hull correspond to "redundant" halfspaces and can be discarded. To form U, he then simply applies an inverse transform to the bottom part of the convex hull.

5.2.1. Merging Chains of Halfspaces

Let $\{h_1,\ldots,h_n\}$ be a set of n arbitrary UPPER halfspaces, sorted by x-coefficient (i.e. ordered on a's in $z \geq a_i x + b_i y + c_i$). We observe that for any $1 \leq i \leq n$, the two "x slope-separated" sets of halfspaces $\{h_1 \ldots h_i\}$ and $\{h_{i+1} \ldots h_n\}$

get mapped by Brown's transform into two planarly-separated point sets. The plane with equation $x = a_i$ will serve to separate the two point sets. Hence, the convex hulls of the point sets generated by Brown's transform are planarly separable and thus amenable to the use of the linear time convex hull merging algorithm of Preparata and Hong [PrHo77]. We use this algorithm to find the (bottom part of the) convex hull of the union of the two existing planarly separated hulls. The points which are not on the newly-updated lower convex hull correspond to the redundant halfspaces. We observe then, that finding the common intersection of two ordered sets of halfspaces can be accomplished using an algorithm for merging convex hulls (which works in linear time).

The common intersection of a set of n LOWER halfspaces, denoted by L, can be found in a very similar fashion. The major difference is that one constructs the top part of the convex hull of the points, rather than the bottom. The points below the top part of the convex hull now correspond to redundant halfspaces, and the convex hull merging algorithm remains essentially unchanged because hulls to be merged remain planarly separated as before.

Separately representing U and L is sufficient for most applications, but a complete representation would also require finding the common intersection of U and L. It turns out that the transformed representation (i.e. convex hulls) is also suitable for this. U is represented by a lower convex hull, and L by an upper convex hull. From the properties of Brown's

transform, the two hulls will be planarly separable (assuming non-empty intersection of U and L). Analogously to the two-dimensional case of section 5.1., we find all the mutual separating tangent planes of the two hulls. The points on the hulls incident to these tangent planes correspond to the planes of U and L which intersect with one another.

The mutual separating tangent planes can be found using the linear time convex hull merging algorithm of Preparata and Hong [PrHo77], only slightly modified. Their algorithm essentially finds all mutual supporting tangent planes between two planarly separated hulls, starting off with a mutual supporting tangent line of a projection of the hulls. Instead, we supply a mutual separating tangent line as a start (see section 4.2.1.1.), and then continue the algorithm. Thus, all the mutual separating tangent planes and hence the common intersection of U and L, can be found in linear time.

5.2.2. Dynamic 3-D Halfspace Intersection

In the previous section, we demonstrated how to use 3-d convex hulls in the representation of the common intersection of 3-d halfspaces. By exploiting some properties of Brown's transform [Br79] and a modification of the 3-d convex hull algorithm of [PrHo77], we can apply the dynamization results of

chapter three to arrive at the following:

THEOREM 5.3. The common intersection of a set of n three-dimensional halfspaces can be dynamically maintained at a cost of O(n) steps per insertion or deletion (of a halfspace).

An immediate application of theorem 5.3. results in the following:

Corollary 5.3. The feasible region of a linear programming problem in three variables can be dynamically maintained at a cost of O(n) per insertion or deletion of an inequality.

Another application of theorem 5.3. is a halfspace searching problem, which is decomposable. The query involved is "does test-point t belong to the common intersection of a set of n three-dimensional halfspaces?". Since the dynamic techniques of chapters two and three, as previously mentioned, are compatible with the nature of decomposable searching problems, we can claim the following result (compare theorem 2.3.):

THEOREM 5.4. If $1 \leq F(n) \leq \lg \lg n$ and $1 \leq H(n) \leq n$, then there exists a 3-d halfspace searching structure with attributes:

$$Q_D(n) = O(H(n) \lg(n/H(n)))$$

$$S_D(n) = O(n F(n))$$

$$I_D(n) = O(n/H(n) + \lg n)$$

$$D_D(n) = O((n \lg n)/(H(n)2^{F(n)}) + \lg n) .$$

6. Dynamic Maximal Vectors of a Set

6.1. Maximal Vectors in Two Dimensions

Definition: A vector (point) v in the plane is maximal in a set **S** when $v \in \mathbf{S}$ and none of the other vectors in **S** are greater in both x and y coordinates.

The maximal vectors of a planar set can be considered to form a one-sided contour, evoking an image of a side of a "rectilinear convex hull" (see Fig. 16). Maximal vectors have applications in pattern classification, operations research and statistics ([Ku75], [OvvL80]).

Kung, Luccio and Preparata [Ku75] have presented an algorithm for determining the maximal vectors of a static set of n planar points in O(n lg n) time. Recently, Overmars and van Leeuwen [OvvL80] have investigated dynamic maintenance of maximal vectors in two dimensions. They keep the points sorted by x-coordinate and store the contour of current maximal elements in a concatenable queue. This allows them, by a simple binary search on one of the contours to construct the contour of the union of the contours of two linearly separated subsets in O(lg n) time.

Their resulting "composite" contour consists of regular pieces from the contours of both (separated) halves, and they use a fully dynamic structure resembling their planar convex

hull structure (see chapter four) to claim the following result:

THEOREM 6.1. [OvvL80] One can dynamically maintain the maximal elements of a set of n points in the plane, at a cost of only $O(\lg^2 n)$ steps per insertion or deletion.

As they point out, this also permits a new algorithm for maximal vectors of a static set which, after sorting the points by x-coordinate, takes only O(n) time. A final application mentioned in [OvvL80] is that the (decomposable) searching problem "is x dominated by an element of the set" can be dynamically solved in O(lg n) time at a cost of $O(\lg^2 n)$ per update.

6.2. Maximal Vectors in Three Dimensions

Given a set of n vectors in three dimensions, a vector of the set is maximal if none of the other n-1 vectors are greater in all three coordinates. Kung, Luccio and Preparata [Ku75] have shown an (n lg n) lower bound for the problem of finding the maximal vectors of a static set of n vectors in three-space. They have also presented an algorithm that finds the maxima in $O(n (\lg n)^{k-2})$ time in $k \geq 3$ dimensions. Their algorithm thus runs in O(n lg n) time in three dimensions, but it does not fit the classic divide-and-conquer mold.

In order to employ the dynamization techniques presented in

chapter three, we must have a linear-time algorithm to merge two planarly separable subsets of maximal vectors. We present such an algorithm in the next section.

6.2.1. Description of the Merging Algorithm

Consider a set **V** of n three-dimensional vectors, where $v_i = (x_i, y_i, z_i)$. **V** has been sorted by x-coordinate so that $x(v_1) > x(v_2) > \ldots > x(v_n)$. The objective is to find V_M, the set of maximal vectors of **V**, given the maximal vectors of two planarly separated subsets of **V**. We use a separating plane with equation $x = x(v_{n/2})$. We have a description of R_M, the set of maximal vectors of the subset $\{ v_1, \ldots v_{n/2} \}$, and of S_M, which is the set of maximal vectors of the subset $\{ v_{n/2+1}, \ldots v_n \}$.

We observe that, because of the x-coordinate ordering, the elements of R_M are also members of V_M. The elements of S_M, however, are not necessarily maximal elements of V. In fact, an element of S_M is a member of V_M iff it is not dominated by any element in R_M (one vector is said to dominate another if it is greater in all coordinates). Therefore, we want to determine T_M, the set of elements in S_M which are not dominated by any element in R_M. V_M will then become (R_M union T_M).

The projection onto a plane (e.g. x = 0) of a 3-d contour

of maximal vectors is a planar subdivision composed of rectilinear regions. Associated with each region is its maximal element, found at the top right corner of the region.

Illustrated in Fig. 17 is an example of two subsets of contours of maximal vectors projected onto the yz plane. Note that R_M (shown in thick lines) and S_M (shown in thinner lines) constitute two rectilinear planar subdivisions. Each subdivision is kept triangulated, with triangulation links from a maximal point to each nonadjacent point of its region (only S_M is shown triangulated in Fig. 17). The triangulation serves as a "navigation aid", as we will later explain.

R_M is the "superior" subset with respect to the x coordinate, and all of R_M is retained in V_M (see Fig. 18). The principle of the merge algorithm is to trace along the outermost contour ("profile") of R_M, from v_{init} to v_{final}, recording along the way which points of S_M are retained and added to the R_M structure, and discarding the rest of S_M.

As we follow R_M's outer contour, starting from v_{init} (where $z = 0$, y = maximum y of R_M), we must keep track of which region of S_M we are located in (i.e. which S_M point, if any, is currently dominating). Whenever we intersect a triangulation link of S_M, we eliminate it. Upon intersecting an edge of S_M's contour, we record the intersection point, and add a triangulation link from the point to the dominating S_M point of the current region. We also "snip off" the part of S_M contour edge which is interior to the profile of R_M. Whenever we turn a corner (i.e. change direction) while following the R_M profile,

we set up a triangulation link between the corner point and the current dominating point of S_M (if one exists). We keep tracing upward and along the outer contour of R_M, following these rules, until we have completed it (i.e. reached v_{final}). The complete contour of V_M, the result of the merging algorithm, is depicted in Fig. 18.

Since the R_M profile is monotone nondecreasing and rectilinear (going from lower right to upper left), it crosses any contour edge or triangulation link at most once. When the traced search path enters a particular triangle, it can leave by only one of two other possible edges. Hence if the two triangulated contours are appropriately represented (each triangle associated with its bounding edges and vice versa), the total time for the merge process is proportional to the total number of contour edges and triangulation links. Thus it requires time which is linear in the total number of vectors.

THEOREM 6.2. Let **V** be a set of 3-d vectors. Given the maximal vectors of two planarly-separated subsets of **V**, their contours can be merged to form the contour of maximal vectors of **V** in O(n) time.

6.2.2. Dynamic 3-D Maximal Vectors

Recall that in chapter three we presented techniques for

dynamically maintaining data structures on separable subsets of points. The efficient use of these techniques relied upon the availability of linear algorithms for merging structures of separated subsets. Given the merge algorithm of section 6.2.1., we can apply the dynamization method of chapter three to obtain the following result:

THEOREM 6.3. The maximal vectors of a set of n vectors in 3-space can be maintained dynamically at a cost of O(n) steps per insertion or deletion.

It should be noted that a complete description of the contour of maximal vectors is maintained by our scheme, enabling fast dynamic domination queries.

Definition: A domination query asks whether a given test vector is dominated by any in the set.

The query can be answered in logarithmic time by performing a planar subdivision search (see lemma 2.2.) to find which region of the projection of the contour of maximal vectors (onto the yz plane) the test vector lies in. Once the region has been located, we simply compare the "heights" (x-coordinates) of the test vector and the region in whose projection it is located. The "height" of the region is the x-coordinate of its dominating point. If the test vector is above the region's surface, then it is not dominated by any vector in the set. Just as in two dimensions, 3-d domination searching is a decomposable searching problem. We can thus apply the results of chapter three again

to get the following:

THEOREM 6.4. If $1 \leq F(n) \leq \lg \lg n$ and $1 \leq H(n) \leq n$, then there exists a dynamic 3-d vector domination search structure with attributes:

$$Q_D(n) = O(H(n) \circledast \lg(n/H(n)))$$

$$S_D(n) = O(n\ F(n))$$

$$I_D(n) = O(n/H(n) + \lg n) \text{ and}$$

$$D_D(n) = O(((n \lg n)/(H(n) \circledast 2^{F(n)})) + \lg n) \quad .$$

7. Conclusions

7.1. New Results and Techniques

In this thesis, we have presented several efficient algorithms for fully dynamic maintenance of a wide variety of geometric structures. We have shown how to reduce the cost of performing deletions at the expense of using extra storage, a general technique applicable to any structures which can be merged asymptotically faster than they can be completely reconstructed. Linear algorithms have been discussed for "merging" of separable or (in some cases) arbitrary structures. Our algorithms also enable the efficient solution of some searching problems which query dynamic search structures based upon geometric configurations. The techniques presented are compatible with but do not depend on the notion of decomposability. Tradeoffs among resources and solution characteristics for dynamic geometric search structures have been explored. A tradeoff between space and update time was investigated, as well as a query vs. update time tradeoff. We have also exhibited a logarithmic merging algorithm for some common planar configurations.

Some of the major new results are:

- The Voronoi diagram of a 2-d point set can be

dynamically maintained in O(n) time per insertion or deletion of a point. (section 2.1.1.2.)

- The convex hull of a 2-d point set can be maintained in $O(\lg^2 n)$ time per insertion or deletion. (section 4.2.2.)
- The common intersection of a set of 2-d halfspaces can be maintained in $O(\lg^2 n)$ time per insertion or deletion of a halfspace. (section 5.1.2.)
- The convex hull of a 3-d point set can be maintained in O(n) time per insertion or deletion (a similar result is also shown for 3-d halfspaces). (sections 3.2. and 5.2.2.)
- The contour of maximal vectors of a 3-d point set can be maintained in O(n) time per insertion or deletion. (section 6.2.2.)

Applications of these and other results have been discussed throughout the thesis.

7.2. Directions for Further Research

The list below describes several open problems regarding the work which has been presented in this thesis.

- Prove optimality of all the worst-case bounds presented. Otherwise, improve the stated upper bounds.
- Find linear algorithms for merging arbitrary

substructures of 3-d convex hulls and 3-d contour of maximal vectors.

- Present other geometric data structures which can be efficiently dynamized using the methods we have discussed.
- All of our techniques have been aimed at improving the worst case cost of dynamic maintenance. Devise algorithms which provide good average case performance. A step in this direction is provided by van Leeuwen and Maurer [vLMa80].

References

[Ah74] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithms , Addison-Wesley (1974).

[Be79] Bentley, J.L., "Decomposable Searching Problems", Info. Proc. Lett. 8,5 (1979), 244-251.

[Be80] Bentley, J.L., "Notes on a Taxonomy of Planar Convex Hull Algorithms", preprint, Dept. of Computer Science, CMU, Pittsburgh, Pennsylvania (1980).

[BeKu79] Bentley, J.L. and Kung, H.T., "Two Papers on a Tree-Structured Parallel Computer", Tech. Rep. CMU-CS-79-142, Carnegie- Mellon University (1979).

[Br79] Brown, K.Q., "Geometric Transforms for Fast Geometric Algorithms", Tech. Rep. CMU-CS-80-101, Carnegie-Mellon University (1979).

[ChDo80] Chazelle, B. and Dobkin, D.P., "Detection is Easier than Computation", Proc. 12th Annual ACM Symposium on Theory of Computing , (1980), 146-153.

[ChTa76] Cheriton, D. and Tarjan, R.E., "Finding Minimum Spanning Trees", SIAM Journal of Computing 5,4 (1976), 724-742.

[Ed79] Edelsbrunner, H., "Optimizing the Dynamization of Decomposable Searching Problems", Report 35, Inst. fur Informationsverarbeitung, TU Graz, Austria (1979).

[EdvL80] Edelsbrunner, H. and van Leeuwen, J., "Multidimensional Algorithms and Data Structures (Bibliography)", EATCS Bulletin (1980), 46-68.

[GoKi80a] Gowda, I.G. and Kirkpatrick, D.G., "Exploiting Linear Merging and Extra Storage in the Maintenance of Fully Dynamic Geometric Data Structures", Proc. 18th Annual Allerton Conference on Communication, Control and Computing (1980).

[GoKi80b] Gowda, I.G. and Kirkpatrick, D.G., "A Fast Algorithm for Union of Convex Sets and its Applications", Dept. of Computer Science, UBC, Vancouver, Canada (in preparation).

[Gr72] Graham, R.L., "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set", Info. Proc. Lett. 1,4 (1972), 132-133.

[Ki79a] Kirkpatrick, D.G., "Efficient Computation of Continuous Skeletons", Proc. 20th Annual IEEE Symp. on Foundations

of Computer Science , (1979), 18-27.

[Ki79b] Kirkpatrick, D.G., "Optimal Search in Planar Subdivisions", preprint, Dept. of Computer Science, UBC, Vancouver, Canada (1979).

[Ku75] Kung, H.T., Luccio, F. and Preparata, F.P., "On Finding the Maxima of a Set of Vectors", J.ACM 22,4 (1975), 469-476.

[Le76] Lee, D.-T., "On finding k-nearest neighbors in the plane", Coordinated Science Lab. Report R-728, University of Illinois, Urbana, Illinois (1976).

[LePr76] Lee, D.-T. and Preparata, F.P., "Location of a Point in a Planar Subdivision and its Applications", SIAM Journal of Computing 6,3 (1977), 594-606.

[LiTa77] Lipton, R.J. and Tarjan, R.E., "Applications of a Planar Separator Theorem", Proc. 18th Annual IEEE Symp. on Foundations of Computer Science (1977), 162-170.

[MaOt79a] Maurer, H.A. and Ottman, Th., "Manipulating Sets of Points - A Survey" in: Applied Computer Science 13 ; Carl Hanser, (1979), 9-29.

[MaOt79b] Maurer, H.A. and Ottman, Th., "Dynamic Solutions of Decomposable Searching Problems, Report 33, Institut fur Informationsverarbeitung, TU Graz, Austria (1979).

[OvvL79] Overmars, M.H. and van Leeuwen, J., "Two General Methods for Dynamizing Decomposable Searching Problems, Tech. Rep. RUU- CS-79-10, Dept. of Computer Science, University of Utrecht (1979).

[OvvL80] Overmars, M.H. and van Leeuwen, J., "Dynamically Maintaining Configurations in the Plane", Proc. 12th Annual ACM Symposium on Theory of Computing (1980), 135-145.

[Pr79] Preparata, F.P., "An Optimal Real-Time Algorithm for Planar Convex Hulls", Comm. ACM 22,7 (1979), 402-405.

[Pr80] Preparata, F.P., "A New Approach to Planar Point Location", preliminary draft (1980).

[PrHo77] Preparata, F.P. and Hong, S.J., "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm. ACM 20,2 (1977), 87-93.

[PrMu79] Preparata, F.P. and Muller, D.E., "Finding the Intersection of a Set of n Halfspaces in Time O(n log n), Theoretical Computer Science 8,1 (1979), 45-55.

[SaBe79] Saxe, J.B. and Bentley, J.L., "Transforming Static Data Structures to Dynamic Structures", Proc. 20th Annual IEEE Symp. on Foundations of Computer Science (1979), 148-168.

[Sh75] Shamos, M.I., "Geometric Complexity", Proc. 7th Annual ACM Symposium on Theory of Computing (1975), 224-233.

[Sh78] Shamos, M.I., "Computational Geometry", Doctoral dissertation, Yale University (1978).

[ShHo75] Shamos, M.I. and Hoey, D.,"Closest-Point Problems", Proc. 16th Annual IEEE Symp. on Foundations of Computer Science (1975), 151-162.

[ShHo76] Shamos, M.I. and Hoey, D., "Geometric Intersection Problems", Proc. 17th Annual IEEE Symp. on Foundations of Computer Science (1976), 208-215.

[vLMa80] van Leeuwen, J. and Maurer, H.A., "Dynamic Systems of Static Data-Structures", Report 42, Institut fur Informations- verarbeitung, TU Graz, Austria (1980).

[vLWo80] van Leeuwen, J. and Wood, D., "Dynamization of Decomposable Searching Problems", Info. Proc. Lett. 10,2 (1980), 51-56.

[Ya79] Yao, A.C., "A Lower Bound to Finding Convex Hulls", Rep. STAN-CS-79-733, Stanford University (1979).

[Zo78] Zolnowsky, J., "Topics in Computational Geometry", Ph.D. Thesis, Stanford University (1978).

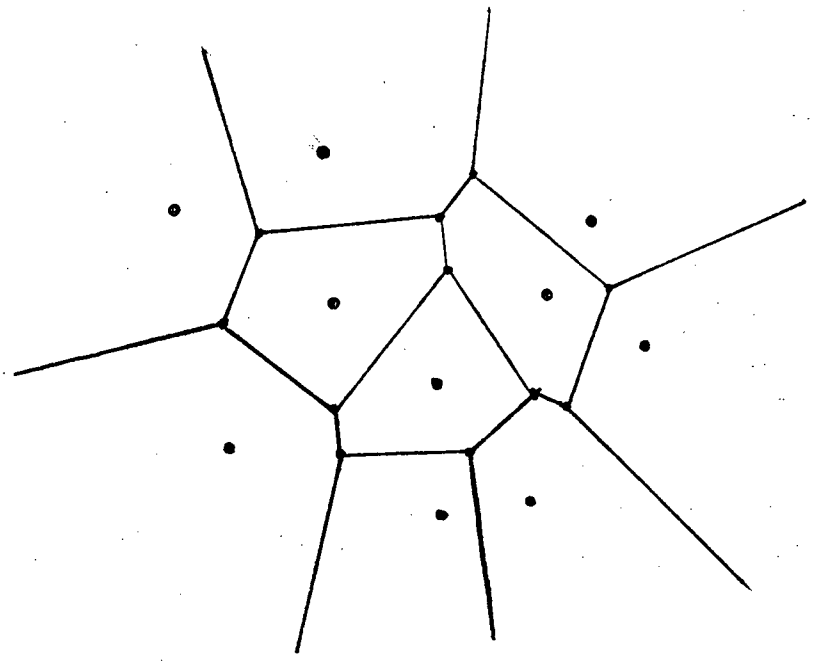


Figure 1. Planar nearest-point Voronoi diagram

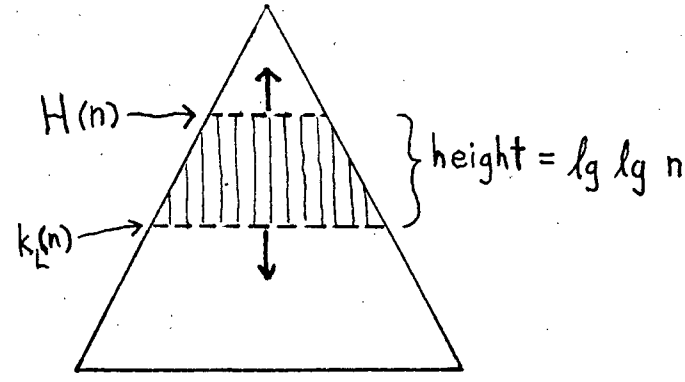


Figure 2. A band in a tree of structures

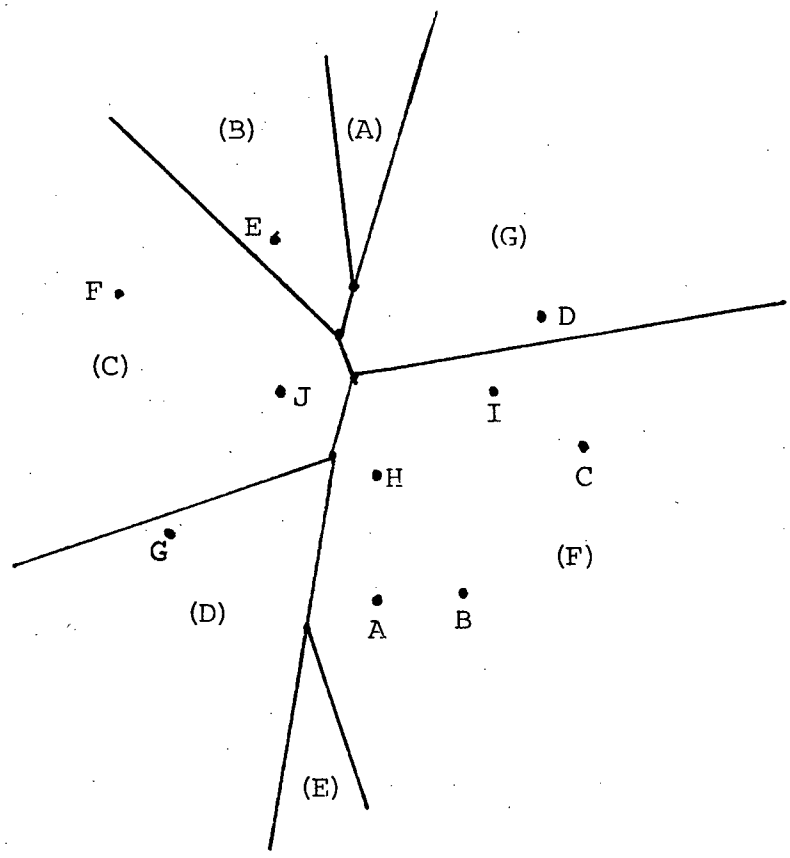


Figure 3. Planar farthest-point Voronoi diagram

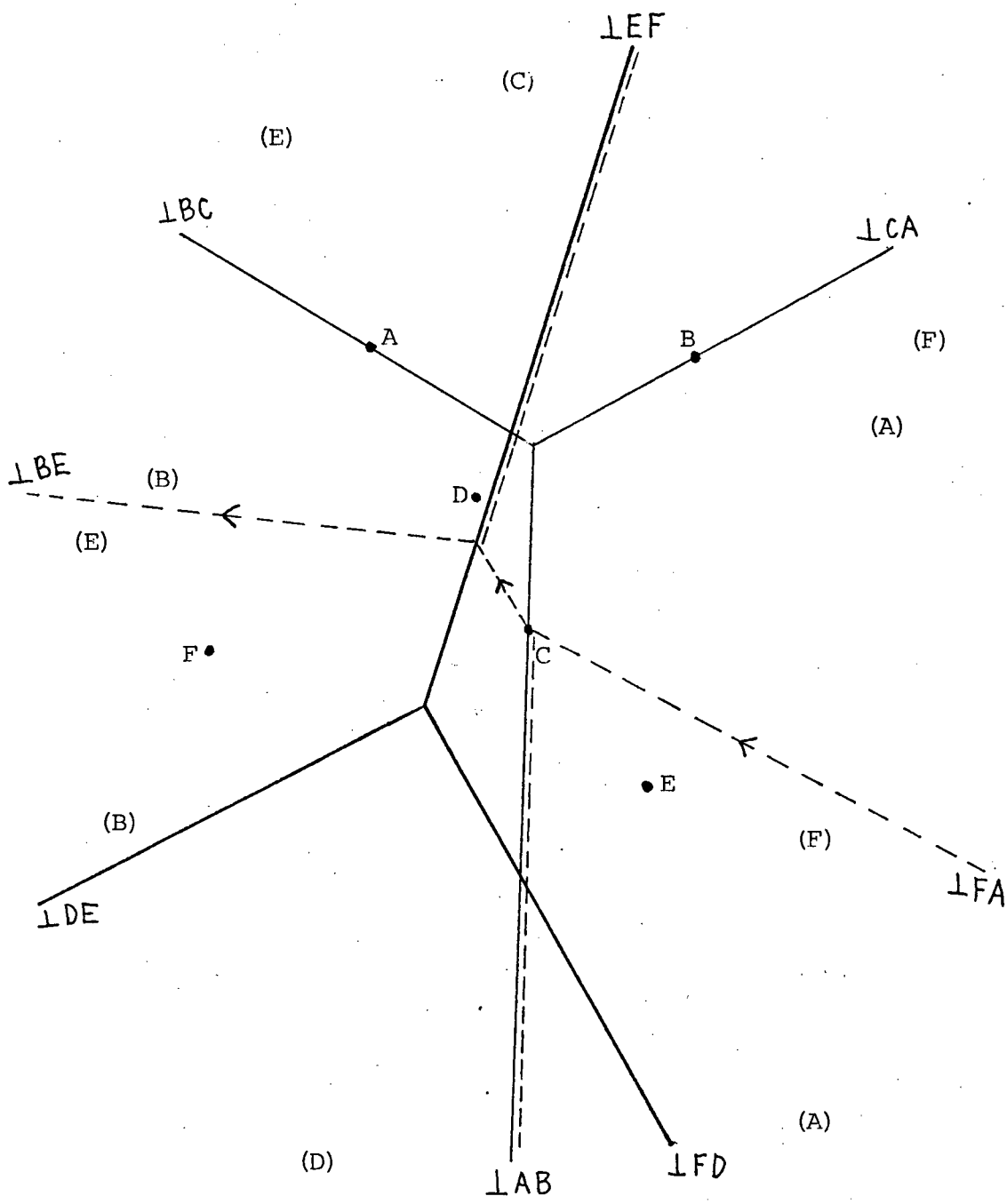


Figure 4. Merging two arbitrary FPVDs.

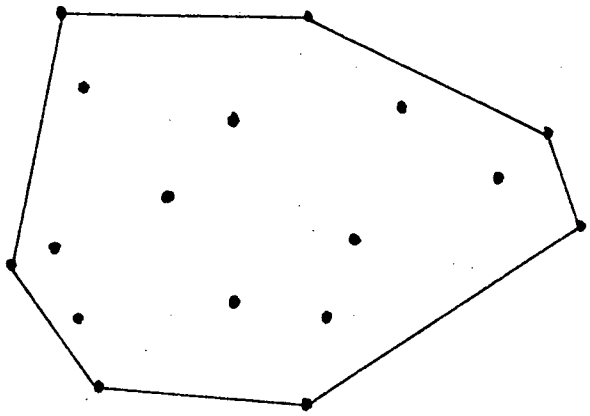


Figure 5. Convex hull of a planar point set

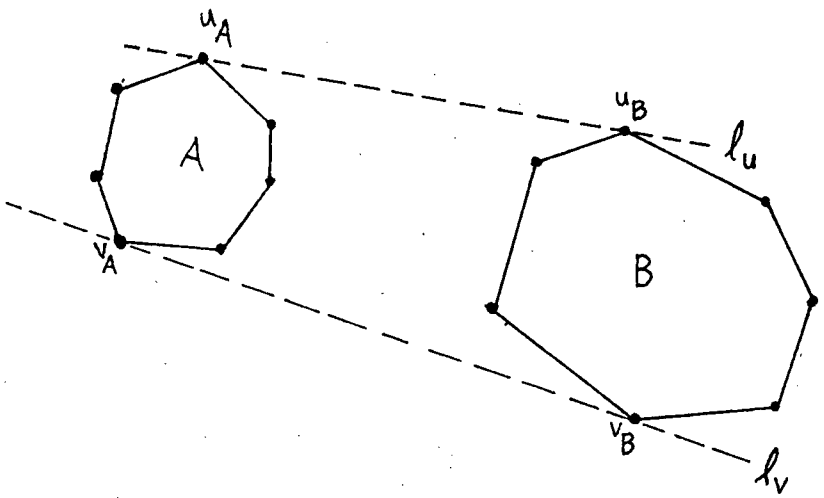


Figure 6. Merging convex hulls

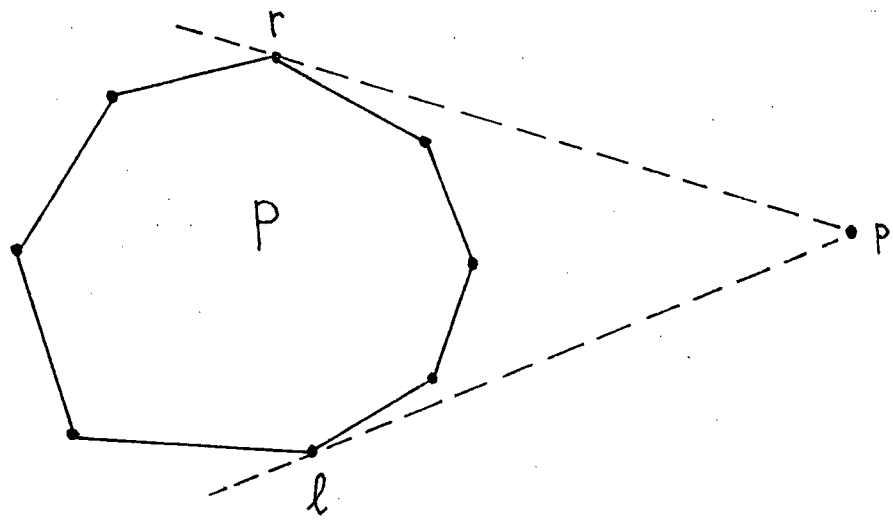


Figure 7. Single point update of convex hull

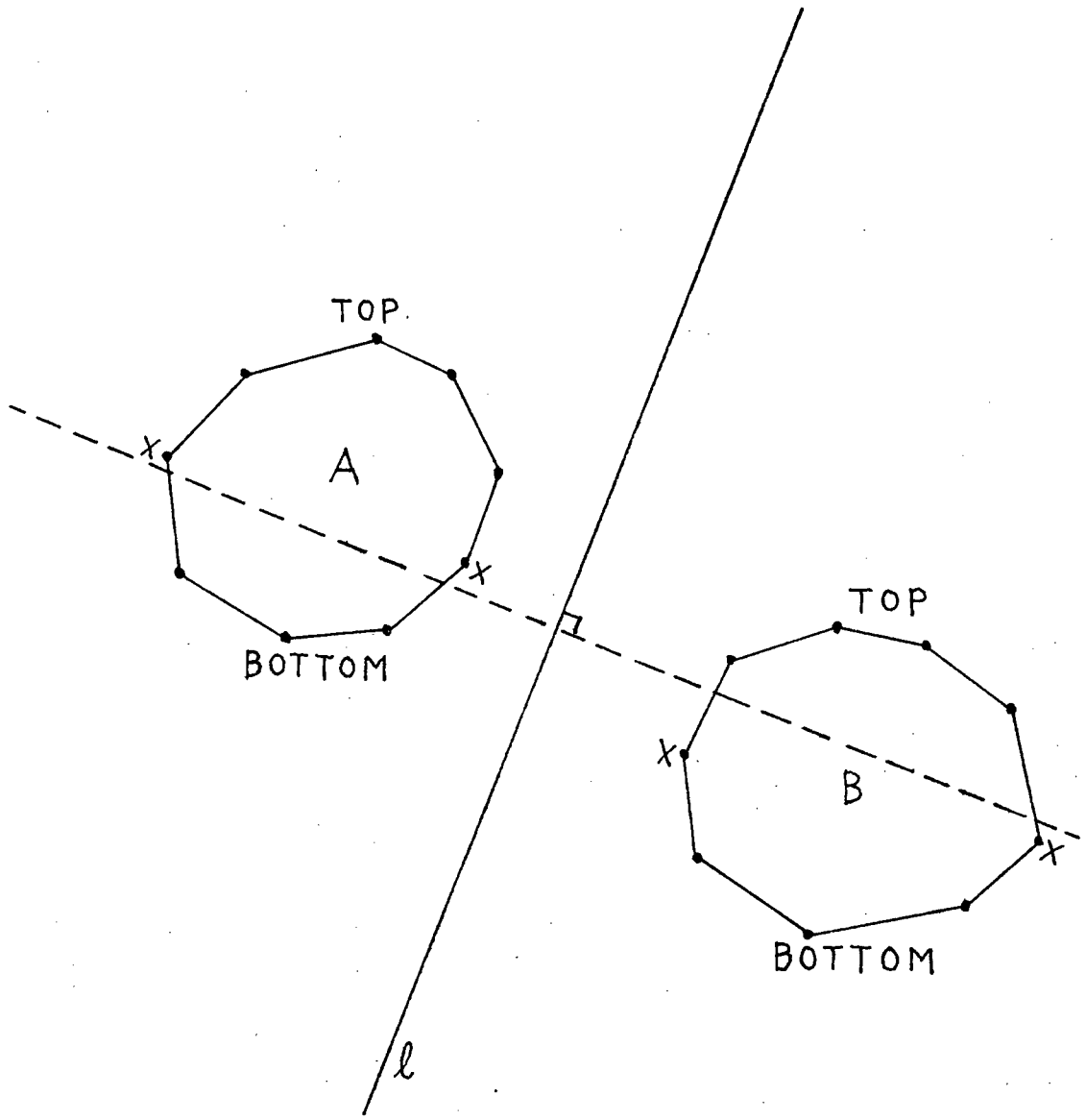


Figure 8. Tops and bottoms of convex polygons

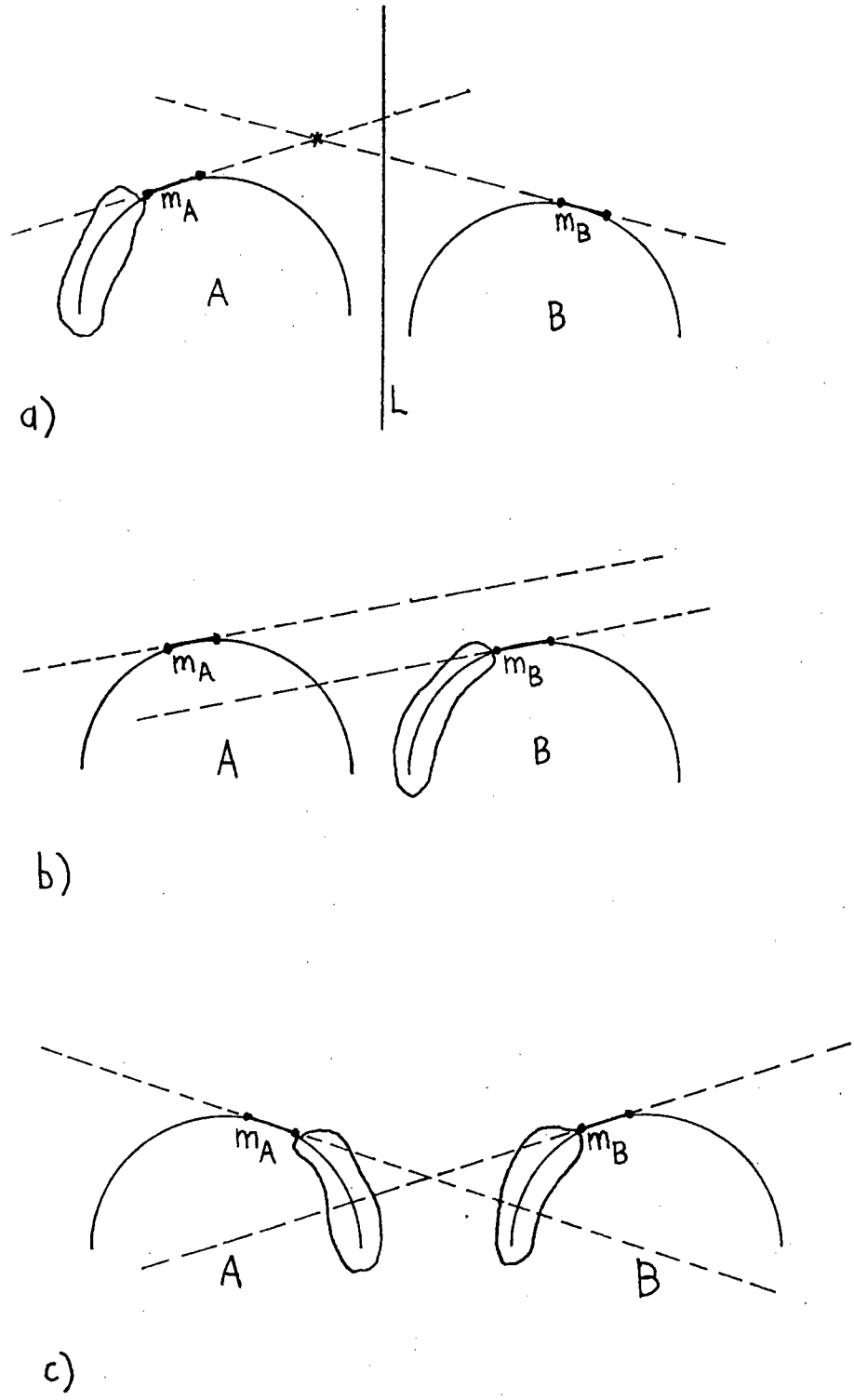


Figure 9. Algorithm for mutual supporting tangents

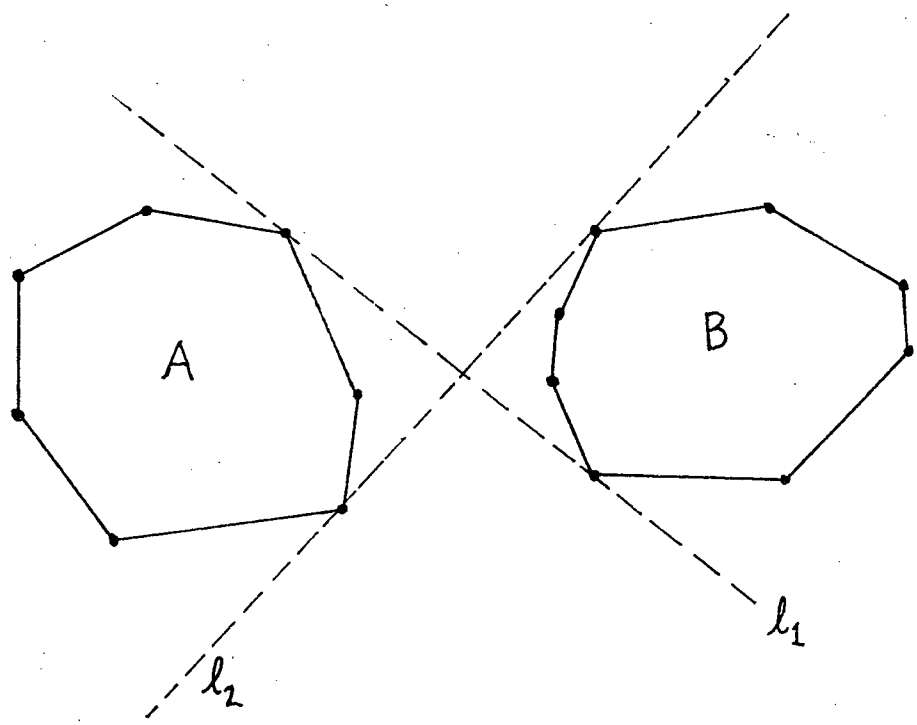


Figure 10. Mutual separating tangents

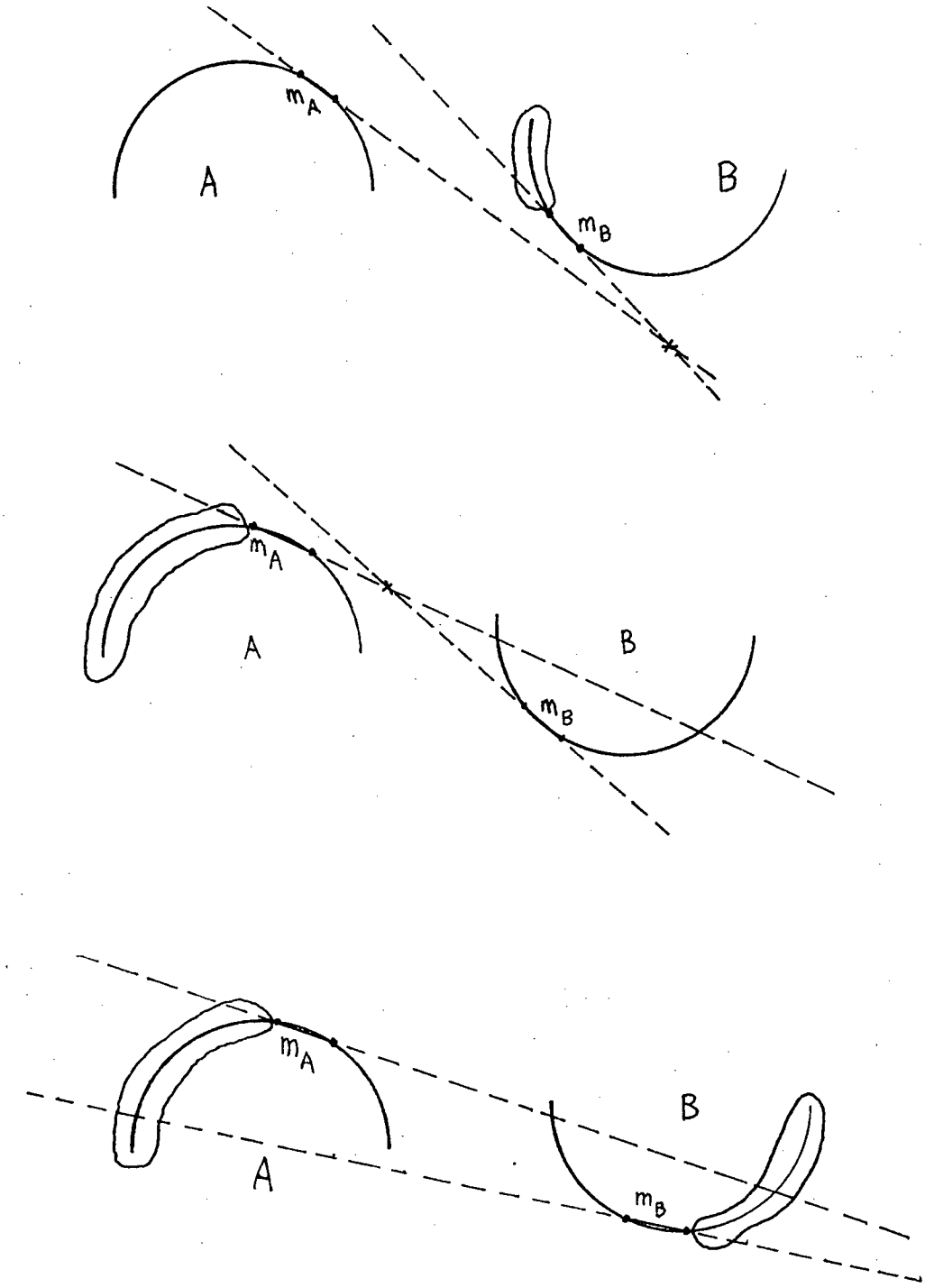


Figure 11. Algorithm for mutual separating tangents

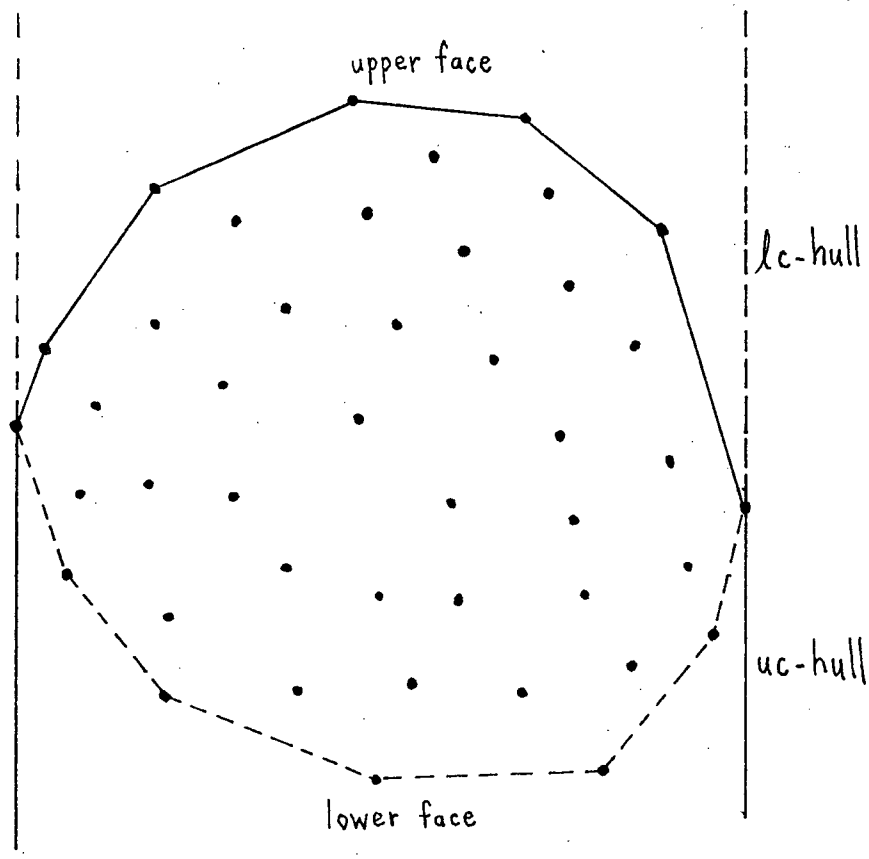


Figure 12. Upper and lower hull faces

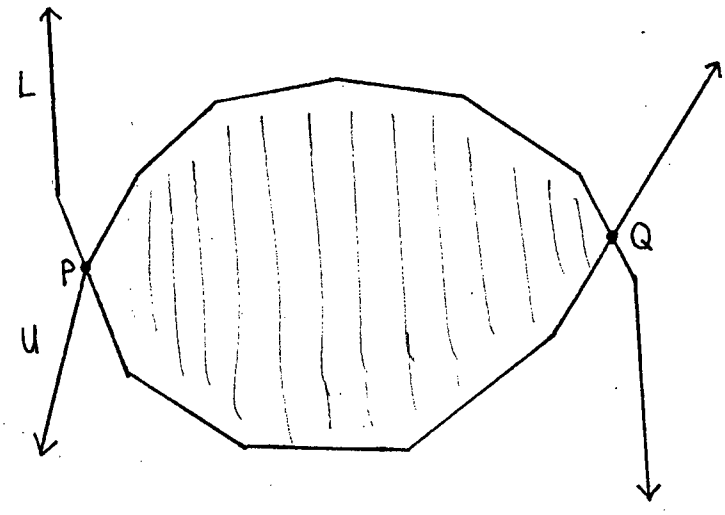
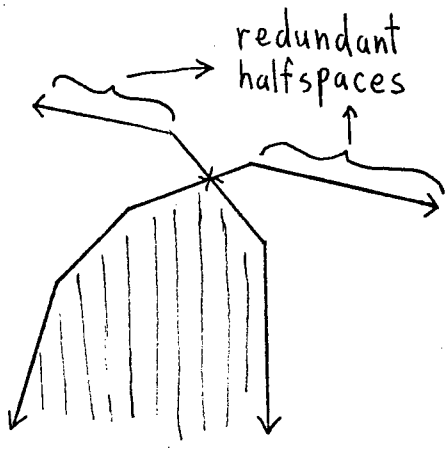
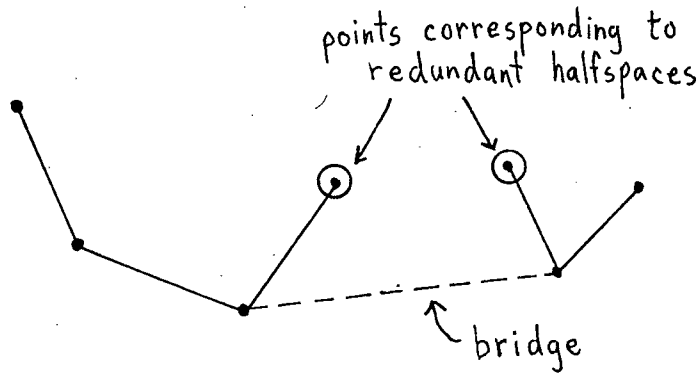


Figure 13. Common intersection of halfplanes



a)



b)

Figure 14. Brown's transform for intersecting halfplanes

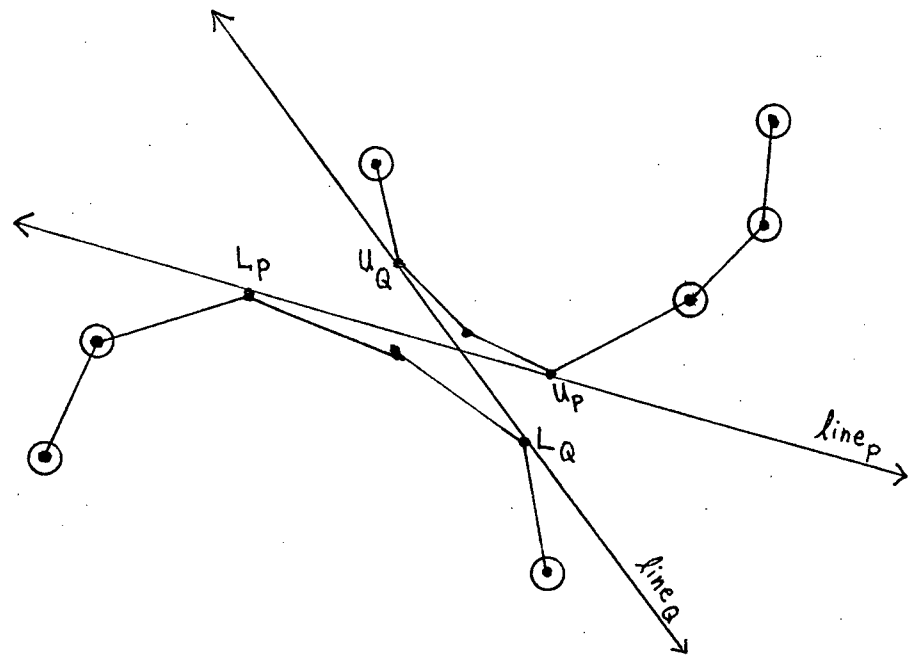


Figure 15. Brown's transform and mutual separating tangents

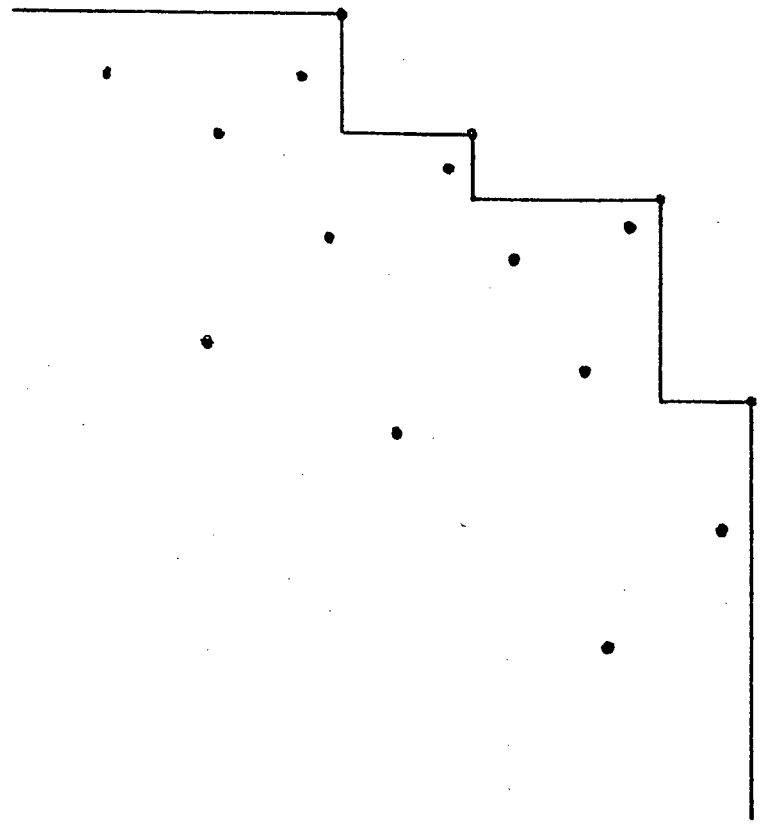


Figure 16. Maximal vectors of a planar set

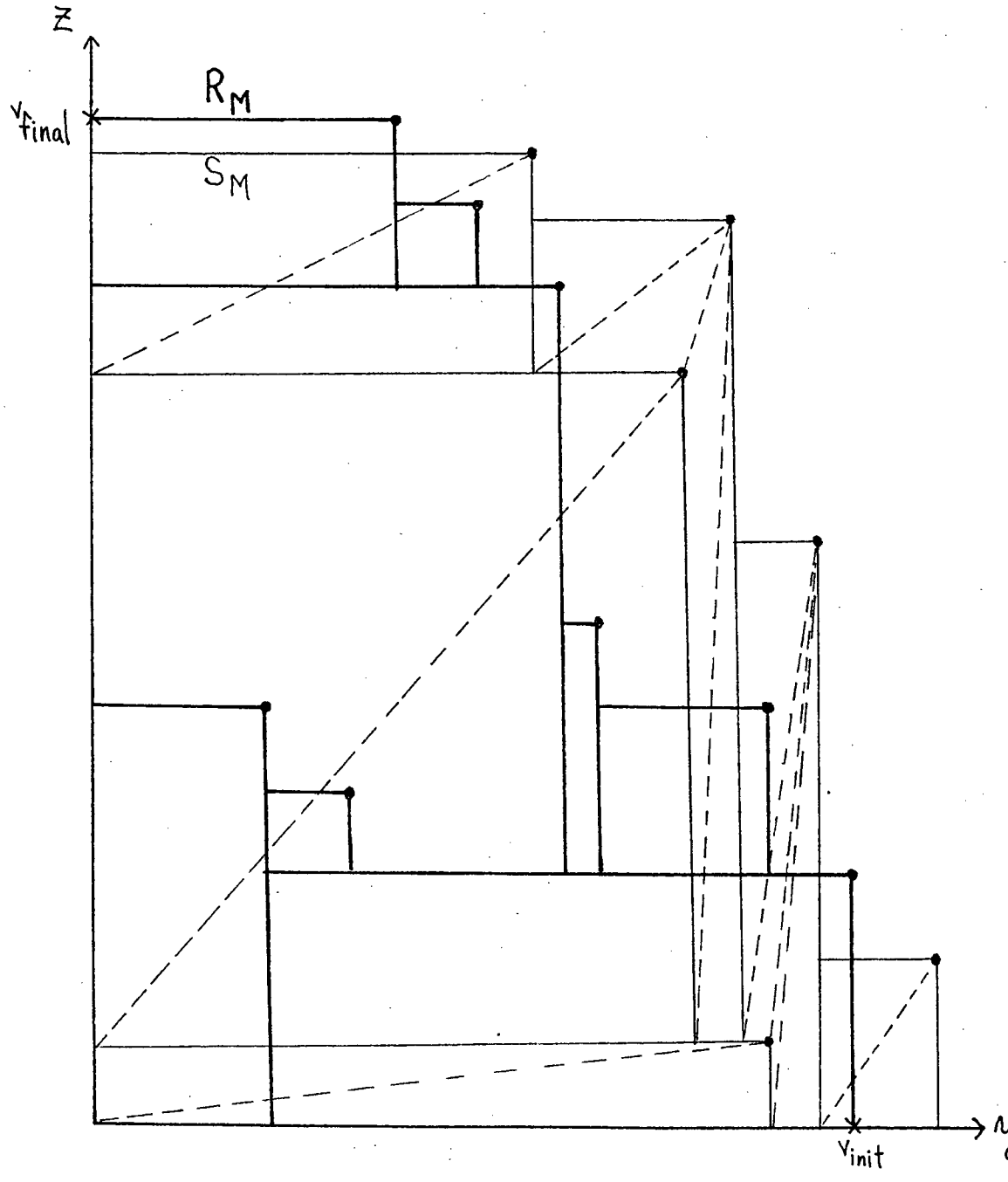


Figure 17. Initial state of merge algorithm

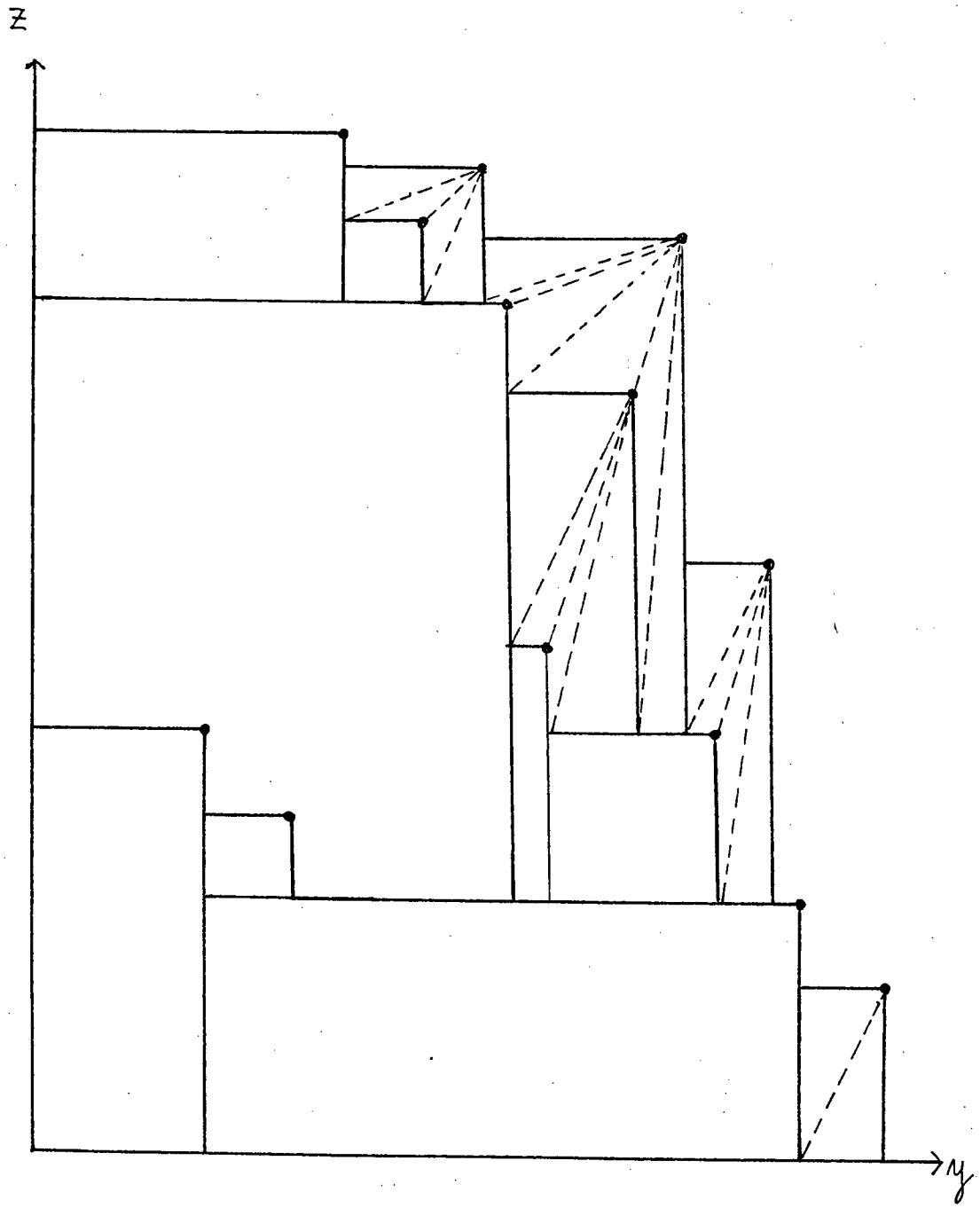


Figure 18. Final state of merge algorithm