COMPUTATIONAL GEOMETRY ON AN INTEGER GRID

by

J. MARK KEIL

B.Sc.(Hons) THE UNIVERSITY OF BRITISH COLUMBIA 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

April, 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 April 21, 1980

ABSTRACT

In this thesis we study a number of geometric problems in an integer grid domain. The worst case time complexity of many of the algorithms that solve geometric problems in the real plane is O(nlogn). This lower time bound is often proved by comparing the problems to Ω (nlogn) time comparison sorting. In the grid domain it is possible to sort coordinates, distances and angles in linear time. By taking advantage of linear integer grid sorting capabilities we are able to present linear time algorithms for the following geometric problems which have O(nlogn) time algorithms when set in the real plane: finding the convex hull of n points, finding a simple closed polygonal path through n points, finding the diameter of a set of n points, deciding the separability question for two point sets, finding the smallest enclosing circle for a set of points, finding a triangulation of a set of n points and finding the Voronoi polygon of one of a set of n points. We extend van Emde Boas' O(nloglogn) integer set manipulation tree structure so it will work on the $O(n^k)$ size integer grid. Using this extended structure we are able to present O(loglogn) search time algorithms for the problems of searching for a test point in a set of rectangles, in a rectilinear planar subdivision and in a restricted angle subdivision. We are also able to use the extended van Emde Boas tree to present O(nloglogn) time algorithms for the following intersection problems on the grid: detecting whether any two of n rectangles intersect, detecting whether any two of n rectilinear polygons intersect and detecting whether any two of n restricted angle polygons intersect.

Table of Contents

List of Figures

ACKNOWLEDGEMENT

I sincerely thank my supervisor David Kirkpatrick for working with me at every step, providing many ideas and much encouragement.

Computational Geometry on an Integer Grid

Computational Geometry

Computational geometry is the study of geometric problems in a computational setting. This involves analyzing the computational complexity of geometric problems and determining efficient algorithms for solving them. In many cases the algorithms becomes efficient because the geometric properties of the problem are exploited. Shamos was the first to explore a number of problems in computational geometry [Shamos 75a] [Shamos 75b] [Shamos 76]. There have been a number of areas of particular interest in computational geometry.

One area of particular interest has been the study of the problem of finding the convex hull of a set of plane points and the study of related problems. The convex hull of a set of n points on the plane is the smallest convex polygon whose corners lie at points of the set that encloses the set. Many algorithms for finding convex hulls and solving related problems have been proposed [Graham 72] [Anderson 78]. Shamos [Shamos 75a] considers such related problems as determining the separability of two plane point sets , determining the diameter of a set of n points in the plane and finding the intersection of two n-gons.

There has been interest in studying nearest neighbour problems such as finding the closest pair of a set of plane points. The Voronoi diagram is a powerful structure often used to solve nearest neighbour problems. The Voronoi polygon associated with each point p of a set of planar points is a region of the plane consisting of all points at least as near to p as to any

other point in the set. The Voronoi diagram of the set consists of the points in the plane which lie in two or more Voronoi polygons [Shamos 75a] [Shamos 75b] [Shamos 78]. Shamos and Hoey [Shamos 75b] study a number of problems involving the proximity of n points in the plane such as finding a Euclidean minimum spanning tree, the smallest circle enclosing the set, k nearest and farthest neighbours, the Voronoi diagram of the set, the two closest points and a proper straight-line triangulation.

Many intersection problems, such as detecting intersections among planar polygonal figures, have been studied [Bentley 79a] [Bentley 79b] [Brown 78] [vaishnavi 79a] [Vaishnavi 79b]. Shamos and Hoey [Shamos 76] study intersection problems such as detecting intersection among line segments, determining whether two simple plane n-gons intersect and determining whether any two of n circles in the plane intersect.

There has been interest in problems involving searching in the plane. A planar subdivision is a partition of the plane into polygonal regions. The problem of locating a point in a planar subdivision has been studied under various titles by Dobkin and Lipton [Dobkin 76], Lee and Preparata [Lee 77], Lipton and Tarjan [Lipton 77], Shamos [Shamos 78], Shamos and Bentley [Shamos 77] and Kirkpatrick [Kirkpatrick 79]. There are also many other geometric problems being studied under a computational light.

Computational geometry has been studied primarily in the real plane or in higher diminsional real spaces. In these domains lower bounds on the worst case performance of many algorithms are proved by comparing the algorithm with the Ω (nlogn) worst case time for comparison based sorting. For example, Shamos [Shamos 75a] proves that Ω (nlogn) worst case time is a lower bound on the worst case time for finding the convex hull of a set of n points in the real plane by showing that any convex hull algorithm can be

used to sort. To prove the Ω (nlogn) lower bound on the worst case time for finding the convex hull of a set of n points in the real plane one can also reduce the problem of finding the maxima of a set of vectors to the problem of finding the convex hull. The Ω (nlogn) lower bound applies not only to convex hull algorithms that provide an ordered convex hull but also to algorithms which simply identify the hull points. Let A be a set of n vectors with real components. If the vectors are ordered so that u > v if $x_i(u) >= x_i(v)$ for i = 1,2 then a partial ordering has been applied to A. The maximal elements of this partially ordered set are called the maxima of A.

Lemma 1.0: There is an Ω (nlogn) lower bound on the time to find the maxima of a set of vectors.

Proof: [Kung 75].

Lemma 1.1: Ω (nlogn) is a lower bound on the time required to find the convex hull of n real plane points.

Proof: Let A be a set of real plane points such that the convex hull of A is all of A. There are four points in A a_t, a_b, a_r and a_l such that $x_2(a_t) = \max_i x_2(a_i)$, $x_2(a_b)$ l $\min_i x_2(a_i)$, $x_1(a_r) = \max_i x_1(a_i)$, $x_1(a_l) = \min_i x_1(a_i)$. These four points divide the hull points into four sets one of which will contain O(n) points. All of these are maxima vectors therefore by Lemma 1.0 the convex hull requires Ω (nlogn) time [Preparata 77].

The best worst case algorithms for many problems such as finding the Voronoi diagram of a set of n real plane points, finding a triangulation of a set of n real plane points and determining whether any two of n line segments in the real plane intersect have an O(nlogn) time bound.

Restricted Domains

Many problems naturally lie in restricted domains where comparison sorting is not necessary. To steer clear of the lower bounds imposed by comparison based sorting we look to these restricted domains. An example of such a domain would be an integer grid. On an integer grid all points have cartesian coordinates (x,y) where x and y are restricted to be integer. Another example would be a hexagonal or triangular grid where each point has six neighbours . Later we shall see that restricting lines on the grid to orientate at a fixed number of slopes can be exploited. For example, we may allow only horizontal lines, vertical lines and lines that lie at 45^{o} to the axes.

Applications

Many geometric problems are of practical as well as theoretical interest. Geographic data processing is one area where grid based computational geometry can be applied [Nagy 79a][Nagy 79b]. In many cases the points on maps are grid based. It is often necessary to solve geometric problems on maps. For example to locate a point on a map treat the map as a planar subdivision.

Printed circuit design is another application area for grid based computational geometry. The problem of detecting whether any two of n rectangles intersect has an important application in the area of very large scale integrated circuit artwork analysis [Baird 78] [Lauther 78]. Solving the rectangle intersection problem can identify locations where

circuit function is threatened by loss of edge acuity.

There are also applications of grid based computational geometry in computer graphics, pattern recognition, operations research, numerical analysis, linear programming and data base design [Preparata 77] [Freeman 79] [Sibson 78] [Vaishnavi 79a] [Shamos 75a] [Shamos 75b].

Shamos [Shamos 75b] mentions applications involving wire layout, facilities location and cutting stock. The construction of interpolating functions in two dimensions involves triangulating a set of points.

Models of Computation

The model of computation we shall use is a random-access machine (RAM) with the capability of performing rational number arithmetic. An equivalent RAM has the capability of performing integer arithmetic with multiplication. In the algorithms in Chapter 2 we also require that the machine will have the ability to calculate the floor function $\lfloor \ \rfloor$. Fortune and Hopcroft [Fortune 78] point out that the floor function can add substantially to the power of a machine. All the analyses which follow will use the uniform cost criterion with the RAM. [Aho 74] The uniform cost criteria with the RAM is the model of computation used when proving Ω (nlogn) worst case time is a lower bound for comparison based sorting. Each comparison takes one unit of time regardless of the size of the numbers.

We have already mentioned some possible grid definitions. For the purposes of this thesis a grid is defined to be a regular bounded rectangular integer grid of size m. A point on the grid has the cartesian coordinates (x,y) where x and y are integers in the range 0 to m. "m" can

be a polynomial function of n. n is the size of the problem. For example n could be the number of points in a set or the number of edges in a set of polygons. Although most of the results apply for arbitrary m we will restrict our attention to the case where m is $O(n^k)$ for some fixed k.

Integer Sorting and Initialization Method

We are able to avoid using comparison based sorting because the domain is of a fixed range integer type. When dealing with sufficiently restricted sets of integers sorting can be performed in linear time. By using an extended radix sort we will be able to sort in the grid domain in linear time.

On several occasions we will use the technique of using a data structure without initializing more of it than is strictly necessary.

Lemma 1.2: It is possible to perform inserts, deletes and membership tests in a data structure that can contain subsets of a universe set consisting of a bounded subset of the natural numbers in time proportional to the number of operations being performed. This allows us to build data structures whose space requirements exceed their preprocessing requirements.

Proof: [Aho 74] p. 71.

Chapter Summarys

In chapter 2 we show how to sort coordinates, distances and angles in the grid domain in linear time. O(n) time algorithms are given for the following problems in the grid case: Finding the convex hull of n points, finding a simple closed polygonal path through n points, finding the

diameter of a set of n points, deciding the separability question for two point sets, finding the smallest enclosing circle for a set of points, finding a triangulation of a set of n points and finding the Voronoi polygon of one of a set of n points.

In chapter 3 we review an O(nloglogn) initialization time, O(loglogn) search time set manipulation structure due to van Emde Boas. [van Emde Boas 77] The structure is extended so that it will work on an $O(n^k)$ grid set without having an increased asymptotic time bound. We also extend the structure to handle integers ranging from 1 to n^k selected from a set of size O(n) in only $O(n^2)$ space.

In chapter 4 we exploit the dense extended van Emde Boas structure to give O(loglogn) time algorithms for searching in a set of rectangles, in a rectilinear subdivision and in a restricted angle subdivision.

In chapter 5 we exploit the dense extended van Emde Boas structure to give O(nloglogn) time algorithms for detecting intersections among rectangles, among rectilinear polygons and among restricted angle polygons. Chapter 6 presents our conclusions.

Chapter 2 Grids and Sorting

Sorting Methods

The grid domain we have defined has points (x,y) where x and y are integers which fall into the range 0 to m. A comparison based sort is not needed because the points are integers. The radix sort is a fast integer sort that works on a finite range [Aho 74].

Lemma 2.1: A sequence of n grid points can be sorted by x or y coordinates or lexicographically in O(n + m) time using the radix bucket sort.

Proof: [Aho 74].

Lemma 2.2: A sequence of n grid points can be sorted by x or y coordinates or lexicographically in $O(n\log_n m)$ time which is O(kn) time if m is $O(n^k)$.

Proof: by expressing the grid points in n-ary notation, a multipass n bucket sort could be used on a sequence of n grid points. Each grid point in n-ary notation would have at most k digits when $m = n^k$ [Aho 74]. On the first pass the least significant digit would be sorted using an n bucket sort in O(n) time. On the next pass the next most significant digit would be sorted. There will be at most k of these passes. The multipass sort will require $O(n\log_n m)$ which is O(kn) time if $m = n^k$. If k is a fixed constant the k pass sort requires only O(n) time.

If n is a power of two division is not needed to do the sort. Shifting will do to express m in n-ary notation.

A set of intergridpoint distances may have to be sorted quickly.

Lemma 2.3: We can sort a set of n distances on the grid in $O(n\log_n m)$

time.

Proof: The distance d between two grid points in the Euclidean metric is $(x_2 - x_1)^2 + (y_2 - y_1)^2$. $d^2 = (x_2 - x_1)^2 - (y_2 - y_1)^2$ is an integer. A sequence of points sorted by d^2 from a given point will be in the same order as if they were sorted by d from the given point. Fast integer sorting techniques can be used on d^2 because d^2 is an integer quantity. On the grid d will range from 0 to m and therefore d^2 will range from 0 to m^2. If d^2 is expressed in n-ary notation there will be at most $2\log_n m$ digits. A sequence of n grid points can be sorted with respect to d^2 and thus also to d from a given point in $O(n\log_n m)$ time using a $2\log_n m$ pass bucket sort.

Corollary to 2.3: We can sort a set of n distances on a $O(n^k)$ size grid in O(n) time.

Proof: The $O(n\log_n m)$ time sort takes O(kn) time on this grid. Since k is a constant the distances can be sorted in O(n) time.

In some algorithms it is necessary to sort points by the angle they form with respect to a given origin and axis.

Lemma 2.4: A set of angles determined by n sets of three grid points can be sorted in $O(n\log_n m)$ time.

Proof: One grid point is selected to be the origin and the points are to be sorted with respect to the angle they form with the x-axis. This angle can be determined from the quadrant and the slope within the quadrant. The quadrant can easily be determined by the signs on the coordinates. Given the points in one quadrant the slope is sufficient to determine the angle. The slope is of the form y/x where x and y are integers in the range 0 to m. These slopes are not integer. If the linear integer sorting techniques are to be used on the slopes they have to be transformed to

integers in such a way as to preserve the order between the slopes.

The difference between two slopes is $y_1/x_1 - y_2/x_2 = (y_1x_2 - y_2x_1)/x_1x_2$. The smallest difference between two slopes is $1/x_1x_2$. Any transform used must separate slopes that are separated by $1/x_1x_2$. Slopes that are separated by $1/x_1x_2$ must be mapped to different integers . Since $1 \leq x_i \leq m$ then $1 \leq m^2/x_1x_2$. The expression $\lfloor m^2 \cdot \text{Slope} \rfloor$ is the required transform. By multiplying the slopes by m^2 and rounding down slopes are mapped to integers with order preserved. These integers will range from 0 to m^3 and can be expressed in $3\log_n m$ diget n-ary notation. Using a multiple pass bucket sorting technique these integers which correspond to the slopes can be sorted in $O(n\log_n m)$ time. Since angles with respect the origin are determined by the slope in each quadrant, these angles can be sorted in $O(n\log_n m)$ time.

Corollary to 2.4: A set of angles determined by n sets of three grid points on an $O(n^k)$ size grid can be sorted in linear time.

Proof: From Lemma 2.2 the angles can be sorted in $O(n\log_n m)$ time which on this grid is $O(kn)$ time. Angles can be sorted in linear time on an $O(n^k)$ size grid.

The $O(n)$ time algorithms found in the rest of the chapter which depend on integer grid sorting capabilities all assume an $O(n^k)$ size grid. These algorithms will work on a general size m grid in $O(n\log_n m)$ time.

Convex Hull and Related Algorithms

The convex hull of a set of n real plane points can be found in $O(n\log n)$ time using Graham's algorithm [Graham 72]. In Graham's algorithm the first step is to find a point interior to the convex hull. Then all

points are sorted by angle around this point. Initially all points p_i are considered to be part of the convex hull. A series of three point tests are made beginning with the point with the smallest angle which test whether to remove the middle point from the convex hull. If the angle $p_1p_2p_3$ is greater than 180 degrees than p_2 is on the convex hull so continue by testing $p_2p_3p_4$. If angle $p_1p_2p_3$ is less than 180 degrees than eliminate p_2 from the convex hull and continue by testing $p_1p_2p_4$. These tests continue until all the points have been examined. None of the points need to be examined more than a constant number of times. Graham showed that the complexity of the algorithm can be decomposed into two parts: one for sorting the points by angle which takes O(nlogn) time and the other for testing which points belong on the hull which takes O(n) time.

Theorem 2.5: By taking advantage of grid integer sorting capabilities the convex hull of n grid points can be found in O(n) time.

Proof: Anderson [Anderson 78] noticed that it is not necessary to start the convex hull algorithm with a point interior to the hull. Instead select the leftmost bottom grid point x_0. This point can be found in linear time. Next order the points x_i by the angle x_0 -- x_i forms with the horizontal line passing through x_0. We perform the angle sort for grid points in linear time. If more than one point lies at the same angle eliminate the one closer to x_0. Finish the algorithm by performing the three point tests as before. The entire algorithm is of time complexity O(n) in the grid domain. Shamos suggested a different technique for a linear convex hull algorithm on a size n lattice in an early draft of his thesis. The coordinate sorting based algorithm of Andrew [Andrew 79] could also be adapted to an O(n) algorithm on the grid.

A number of results follow from the linear convex hull algorithm.

Theorem 2.6: (a) The diameter of n grid points can be found in O(n) time.

(b) the separability question for two grid point sets can be decided in O(n) time.

(c) the smallest circle enclosing a set of n grid points can be found in O(n) time in the worst case.

Proof

(a) The diameter of n real plane points can be found in O(nlogn) time [Shamos 75a]. This is done by first finding the convex hull of the set and then finding the diameter of the convex hull. Finding the convex hull takes O(nlogn) time. Finding the diameter of the convex hull requires O(n) time.

In the grid domain the diameter of n points can be found in O(n) time. Both finding the convex hull and finding the diameter of the hull are linear operations in the grid domain.

(b) Two finite plane point sets are said to be separable if and only if there exists a straight line l with the property that every point of one set lies on one side of l and every point of the second set lies on the opposite side of l. Shamos [Shamos 75a] notes that two plane sets are separable iff their convex hulls are disjoint. In the real plane the separability question for two point sets can be decided in O(nlogn) time. O(nlogn) time is sufficent to find the convex hull of each set and O(n) time is sufficent to determine whether these hulls intersect.

The separability question for two grid point sets can be decided in O(n) time. Finding the convex hulls which was the bottleneck can now be done in linear time.

(c) Shamos and Hoey [Shamos 75b] show that the smallest circle

enclosing a set of n real plane points can be found in O(nlogn) time. The diameter of the required circle is the same as the diameter of the point set.

In the grid domain finding the diameter of a set of n points takes O(n) time. Therefore the smallest circle enclosing a set a n grid points can be found in O(n) time in the worst case.

Simple Closed Polygonal Path Algorithm

Shamos [Shamos 75a] shows that in the real plane finding a simple closed polygonal path through n points must take Ω (nlogn) time in the worst case. If a simple closed polygonal path in the real plane could be found faster than in O(nlogn) time then the convex hull of n real plane points could also be found faster than O(nlogn) time.

A simple closed polygonal path through n grid points can be found in O(n) time in the worst case. Start by sorting the points by angle from the vertical about the leftmost extreme point. On the grid this takes O(n) time. Join the points in increasing order of angle. If several points lie at the same angle join them in increasing order of distance except if they lie at the maximum angle then join them in decreasing order of distance. None of these steps requires more than a linear amount of time.

Triangulations

Given n points in the plane a triangulation is formed by joining them by non-intersecting straight line segments so that every region interior to the convex hull is a triangle.[Shamos 75b] Triangulation is important in

numerical interpolation. It is necessary to construct a triangular grid to base the interpolation. There are various kinds of triangulations that have special properties. The minimum weight triangulation minimizes the sum of the edge lengths. The minimum weight triangulation has good numerical properties [Shamos 75b]. The Delaunay triangulation is the dual of the Voronoi diagram. Whenever the Voronoi polygons of two points share a common edge these two points are joined in the Delaunay triangulation.

On the real plane $\Omega(n\log n)$ is a lower bound on the time required to find any triangulation of n points. Shamos and Hoey [Shamos 75b] prove this by reducing comparison based sorting to triangulating.

The following algorithm which takes advantage of grid sorting capabilities enables one to triangulate n grid points in linear time. Figure 1 shows the algorithm in progress.

1. Find the point with the minimum x coordinate. Call it x_0.

2. Sort the points x_i by the angle x_0 -- x_i makes with the vertical line passing through x_0.

3. Find the point x_1 with the minimum angle and add the edge x_0 -- x_1 to the triangulation. Initialize a stack called BACK by pushing x_1 onto it. Set i to 1.

4. While i < n Do

- include edge x_0 -- x_{i+1} and x_i -- x_{i+1} in the triangulation.
- While (stack has two elements and (outer angle x_{i+1},top(BACK), second(BACK) < 180 degrees))

Do a) Include edge x_{i+1} -- 2nd(BACK) in the triangulation.

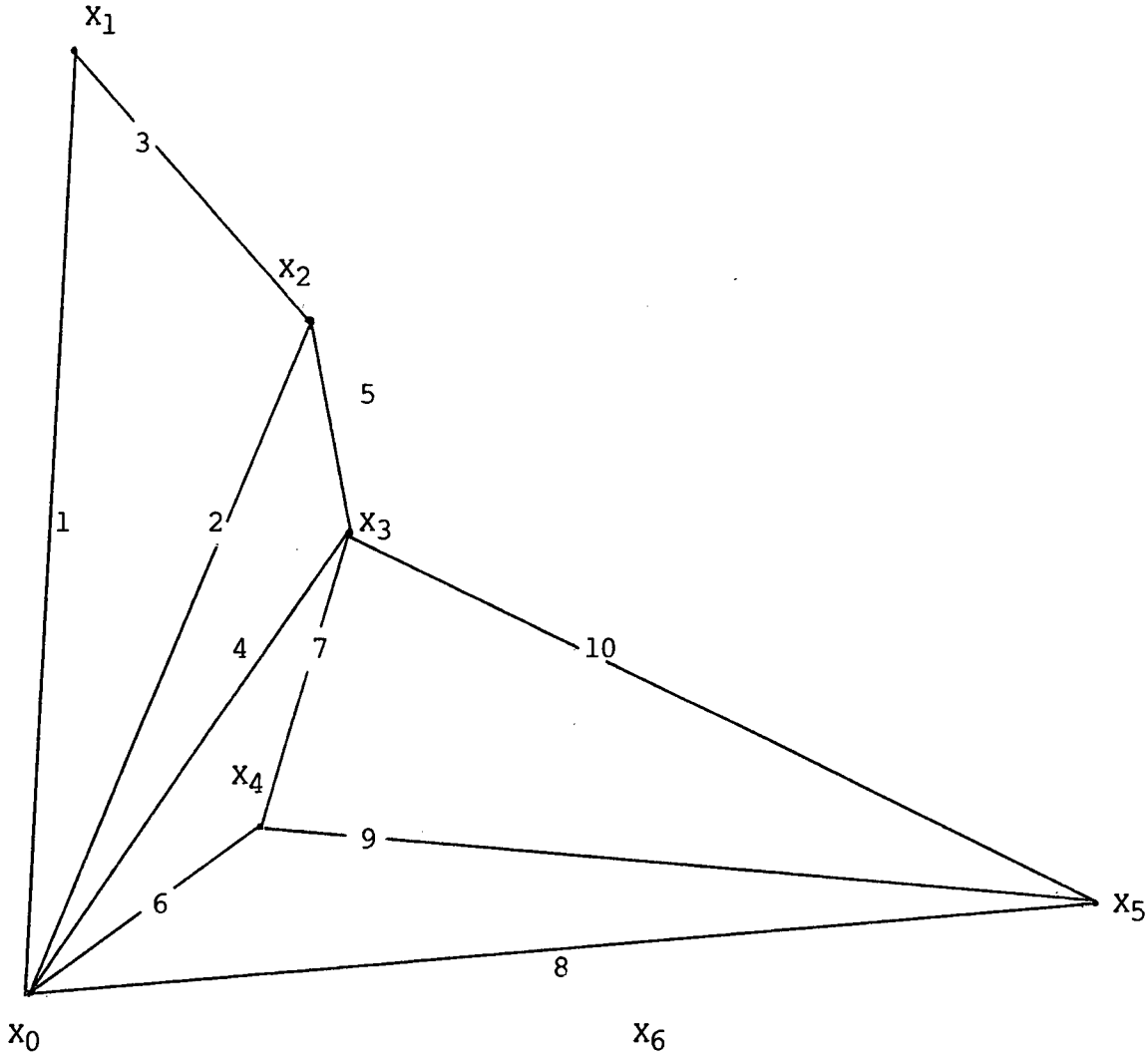
b) Pop stack

Partially Completed Triangulation

- edges are labelled in the order that they were inserted in the triangulation

- at this point the stack 'BACK' contains x_3 x_2 x_1 where x_3 is at the top of the stack

- algorithm is in inner while loop about to test the angle x_5 - x_3 - x_2 where x_3 is top(BACK) and x_2 is 2nd(BACK).



End inner while

- Push x_{i+1} onto stack
- i <- i + 1

END

Lemma 2.7: The above algorithm yields a triangulation of n grid points in O(n) time in the worst case.

Proof: In the triangulation algorithm steps 1 and 2 require O(n) time in the worst case. Step 3 takes a constant amount of time. In step 4 the outer while loop takes O(n) time. The inner while loop eliminates a point each time it is entered. It therefore can be entered at most n times. Step 4 and thus the entire algorithm run in O(n) time.

The triangulation found by the above algorithm was not created to have any particular properties. Lawson [Lawson 72] described an algorithm for forming the Delaunay triangulation from an arbitrary triangulation. A triangulation is a Delaunay triangulation if and only if in every strictly convex quadrilateral the replacement of the diagonal by the alternative diagonal does not increase the minimum of the six angles in the two triangles making up the quadrilateral [Sibson 78]. Lawson would start with an arbitrary triangulation and make exchanges of the diagonal in convex quadrilaterals until the Delaunay triangulation was formed. The complexity of this algorithm has not been determined.

Voronoi Polygon Algorithm

Shamos and Hoey [Shamos 75b] show that Ω (nlogn) time is required in

the worst case to construct the Voronoi polygon of a given real plane point with respect to n - 1 other real plane points. This is done by showing that a Voronoi polygon finding algorithm can sort.

One application of a Voronoi polygon finding algorithm is in the incremental formation of Voronoi diagrams. If the Voronoi diagram of a set of n points is given and another point is then added to the set, the Voronoi polygon of the new point with respect to the old ones must be found as part of the process to create the Voronoi diagram on the complete set of n + 1 points.

The Voronoi polygon about x_i with respect to a set of n - 1 points has the property that all plane points within the polygon are closer to x_i than to any x_j ($i \neq j$) within the given set. The Voronoi polygon about x_i is the mutual intersection of all half planes containing x_i defined by the perpendicular bisector of x_i and x_j for all $j \neq i$ in the given set. The Voronoi polygon is a convex polygon having at most n - 1 sides. Using Brown's [Brown 78] method of intersecting half planes and grid integer sorting capabilities a single Voronoi polygon can be constructed in O(n) time in the worst case.

To form the Voronoi polygon about a point P start by forming the lines that bisect the segments joining P with the other n - 1 grid points. These lines divide the plane into half planes which can be intersected using Brown's algorithm.

The half planes are divided into three sets: upper, lower, and vertical. A half plane is in the set upper if the line at its boundary is above the rest of the half plane. Lower and vertical are defined similarly. Brown's method divides the intersection problem into four parts: (1) the intersection of the upper half planes. (2) the intersection of the lower

half planes, (3) the intersection of the results of (1) and (2) , and (4) the handling of the vertical half planes.

Part (3) is accomplished in O(n) time even in the more general real plane case using Brown's ALGORITHM INTERSECTCHAINS.

To solve part (4) intersect the result of part (3) with the rightmost right vertical half plane and the leftmost left vertical half plane. Shamos [Shamos 75a] shows that the intersection of two convex n-gons takes O(n) time. This is done twice in this step. Part (4) is accomplished using O(n) time in the worst case.

Parts (1) and (2) remain. Consider part (1) the intersection of the upper half planes, part (2) can be done similarly. Some of the lines that define the half planes do not bound the final intersection . These are redundant lines . Brown's algorithm finds all such lines and throws them away. Then the remaining lines are sorted so that the top part of the final intersection is created.

The first step is to find the redundant lines. To do this Brown uses a transform that transforms points to lines and lines to points by the following formulas.

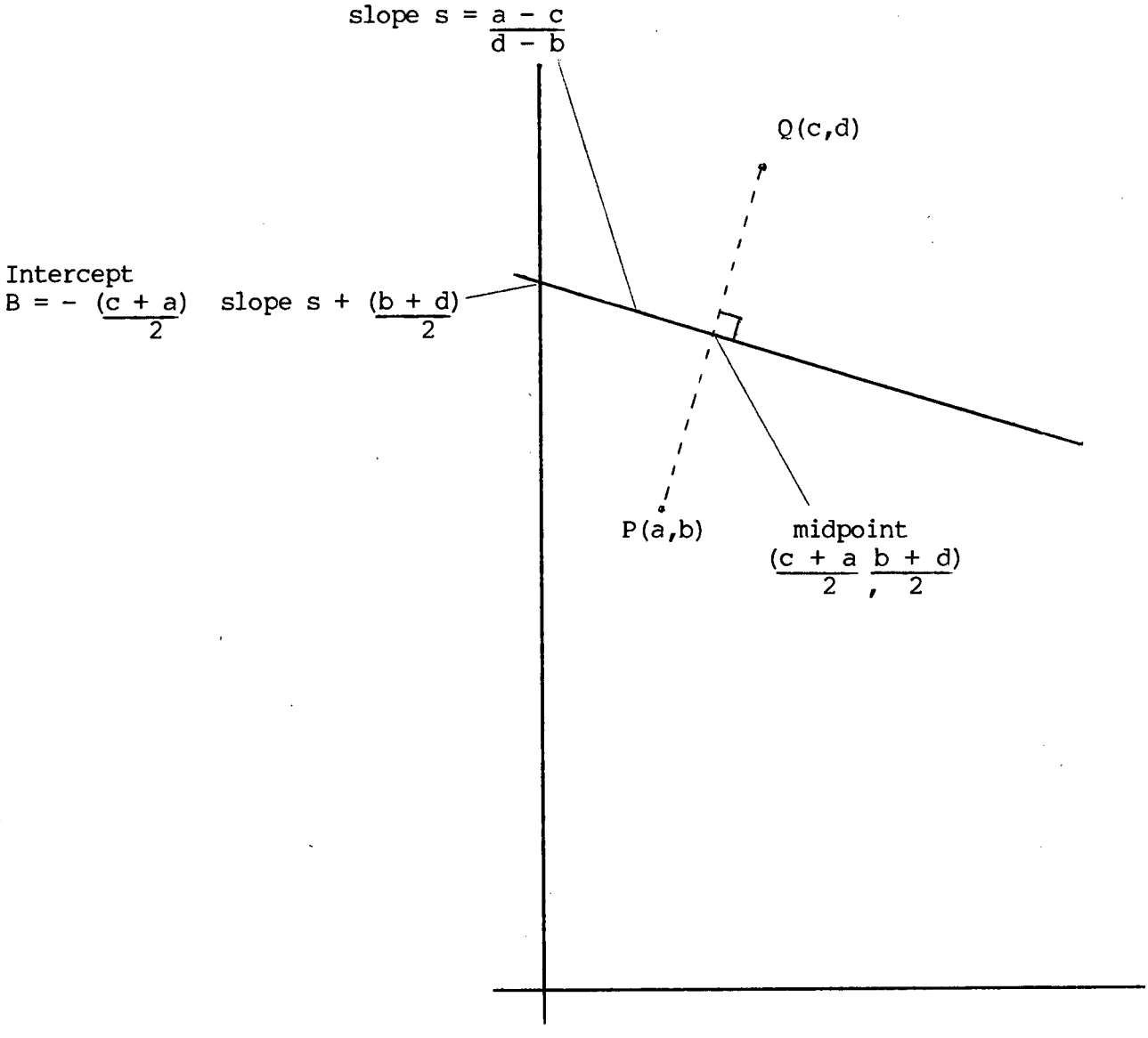
Y = slope · X + intercept -> (slope , intercept)

(x , y) -> intecept = -x · Slope + y

Brown transforms the lines which define the upper half planes to points. He then finds the lower convex hull of the resulting points. Those lines which do not correspond to points on the lower convex hull of of transformed points are redundant.

In the grid case start by expressing the lines defining the upper half planes in slope intercept form. In order to take advantage of the fast convex hull algorithm the slope and intercept of the bisectors between point

Fig 2 Brown's Transform



P and the other n - 1 points have to be converted to integer points on a grid finer than the original. The slope and the intecept have to be converted to integers preserving order.

This has already been done for the slopes in the discussion on grid angle sorting. $\lfloor n^{2k} \cdot \text{Slope} \rfloor$ will convert the slopes to integers preserving order. The intercept also needs to be converted.

As shown in fig 2 the bisector between point P (a,b) and point Q (c,d) has intercept

$$B = -x \cdot \text{Slope} + y = ((c + a)/2) \cdot \text{Slope} + ((b + d)/2)$$

The difference between two such intercepts is

$$B - B' = ((c + a)/2) \cdot ((a - c)/(d - b)) + ((b + d)/2) - (((c' + a')/2) \cdot ((a' - c')/(d' - b')) + ((b' + d')/2))$$

$$B - B' = ((c + a)(a - c)(d' - b') - (c' + a')(a - c)) / 2(d - b)(d' - b') + (b + d) / 2 - (b' + d') / 2$$

where $1 \leq a,b,c,d \leq n^k$. The smallest separation between two intercepts is then

$$1 / 2(d - b)(d' - b') \geq 1 / 2n^{2k}$$

Any transform used must separate intecepts that are separated by $1/2n^{2k}$. Intecepts that are separated by $1/2n^{2k}$ must be mapped to different integers. If the intercepts are converted by $\lfloor 2n^{2k} \cdot \text{Intercept} \rfloor$ they become integers and order is preserved.

Once the slopes and intercepts have been converted onto the finer grid the lower convex hull of the points is found in O(n) time. The finer grid is at most $2n^{2k}$ times as fine as the original grid. The size of the finer grid is still a polynomial function of n. "m" = n^{2k}. These points on the lower convex hull correspond to the bisectors in the original problem that

form part of the Voronoi polygon.

These bisectors are sorted in O(n) time using the grid angle sort. Part (1) has thus been completed in linear time.

Lemma 2.8: The grid Voronoi polygon algorithm runs in O(n) time in the worst case.

All four parts of the algorithm are done in linear time and thus the O(n) worst case time Voronoi polygon algorithm for grid points is complete.

Brown's method can also be used to find the intersection of n half planes in O(n) time if the lines defining the half planes are defined by two grid points. Brown's method can also be used to find the kernel of a polygon on the grid in O(n) time. Lee and Preparata [Lee 79] have an O(n) time kernel algorithm for the real plane. It is open whether the complete Voronoi diagram of n grid points can be found in linear time.

Chapter 3 Van Emde Boas Structure

When designing efficient algorithms for the manipulation of sets of points on the plane one encounters the problem of handling incompatible operations. Instructions for inserting or deleting points in sets requires a random access data structure. However instructions for finding the minimum element or the nearest neighbour of an element require an ordered representation. One example of a data structure which allows both random and ordered access is a priority queue which supports the instructions insert, delete and find minimum. Another example is a mergeable heap which supports the instructions insert,delete,form union and find minimum.[Van Emde Boas 77]

Algorithms which depend on both random and ordered access which work with real numbers and depend on comparisons to order numbers have so far shown a worst case processing time of O(nlogn) per instruction for a series of O(n) instructions on a n element universe. Aho, Hopcroft and Ullman [Aho 74] use 2-3 trees to implement priority queues and mergeable heaps so that n instructions can be processed in O(nlogn) time. Of course any sequence of n instructions which will sort n real numbers will require Ω (nlogn) time. If the domain is restricted to a bounded integer range this lower bound no longer applies. Van Emde Boas [77a][van Emde Boas 77b] presents a data structure which manipulates on-line a priority queue on the domain of integers from 1 to n with a worst case time of O(loglogn) per instruction. This data structure requires O(nloglogn) preprocessing time to create and O(n) space to store.

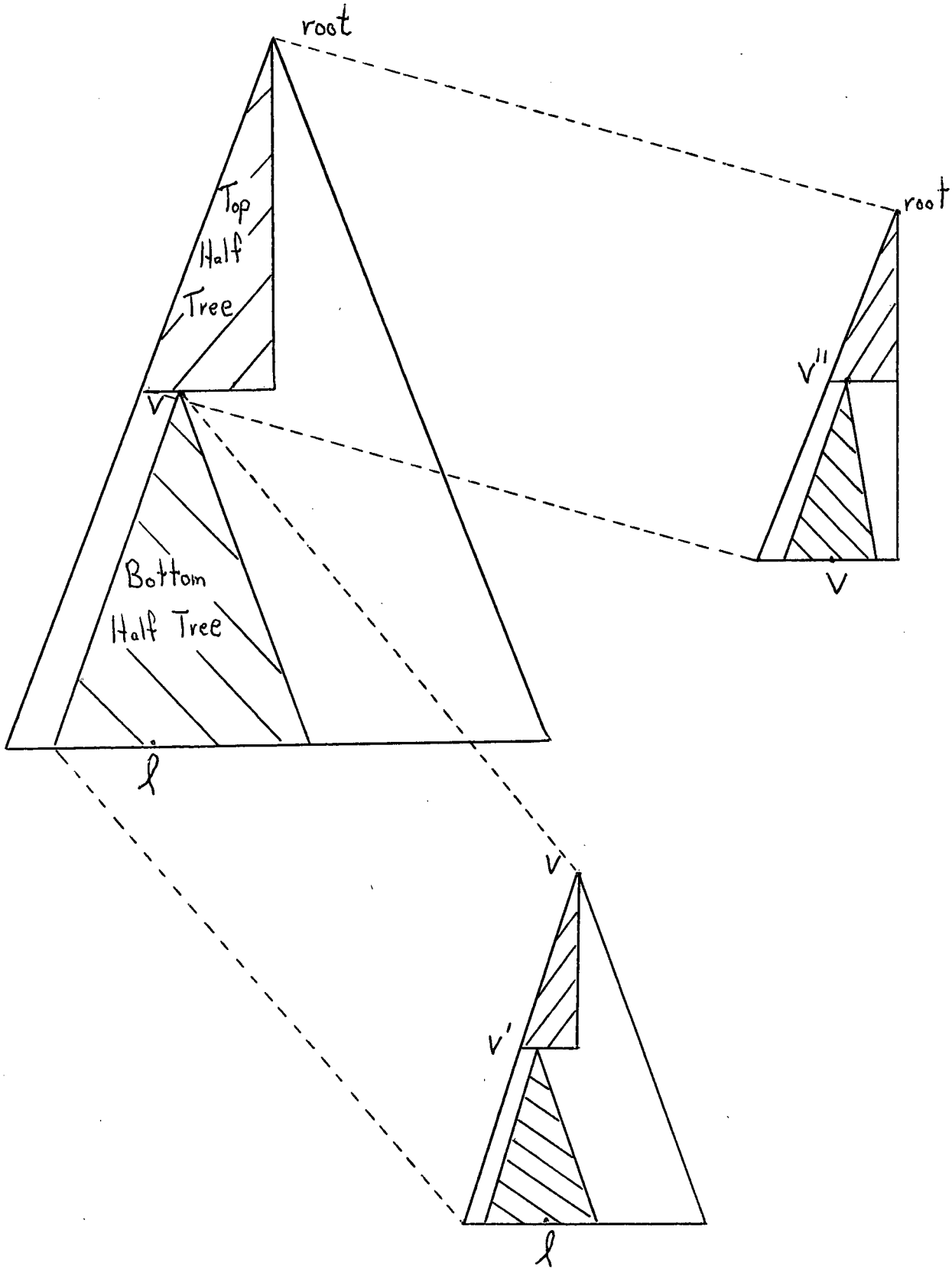
Van Emde Boas structure can actually support more instructions than a simple priority queue. On a universe consisting of integers in the range 1

to n van Emde Boas tree supports the instructions find minimum, find maximum, insert,delete, test for membership, find predecessor and find successor. Moreover, each of these instructions can be completed in O(loglogn) time in the worst case.

The structure of the van Emde Boas tree is quite complex. Some of this structure is static not depending on the set being represented. Assume that $n = 2^h$. The skeleton of the van Emde Boas structure will be a binary tree of height h. The n leaves of this tree will represent the numbers 1 to n in order from left to right. The leaves represent the potential members of the set. Each node of the tree also contains a number of labels and static pointers. The level of a node is the length of the path from the leaves to the node. Each node contains a series of father pointers which point upwards in the tree. These are used to perform a binary search on the levels of the tree. For example, a leaf will contain father pointers pointing to the half, quarter, eighth and so on positions on the path from the leaf to the root of the tree. A node at a level 1/4 of the way from the leaves to the root will have father pointers pointing to the nodes at the 3/8, 5/16 and so on levels on the path from a leaf to the root passing through that node. Each node will have O(loglogn) of these father pointers. These static pointers exist for every node and do not depend on the set being represented in the data structure.

There is also dynamic information stored in the tree that indicates which integers are members of the set and indicates the ordering between these members. At the leaves the dynamic information consists of the pointers successor and predecessor and the flag present. The pointers are part of a doubly linked list that includes the members of the set. The present flag is set if the leaf is a member of the set. Dynamic information

Fig 3 Van Emde Boas Tree



is stored at internal nodes only if strictly necessary to represent the relationships between members of the set. If dynamic information is stored at an internal node then the node is said to be active. For example, if a new element is to be inserted into the set various nodes will have to be marked present. First the leaf will be marked present. The root is then tested to see if it is present. If it is not present then this element must be the first element in the set. The root is marked present and a dynamic pointer is set to point to the only present leaf. If the root is present then a father pointer is followed from the leaf to a node half way up the tree. If this node is not present we have reduced the problem to inserting this node in the top half of the tree. If the node is present we have reduced the problem to inserting the leaf in the bottom half of the tree and a father pointer is followed to a point one quarter of the way up the tree. This process continues so that the presence of a leaf is indicated as high as possible in the tree.

To insert an integer into the tree first mark the leaf corresponding to that integer present. Then follow father pointers up to the nearest present node marking all active internal nodes present. This is a branchpoint. At the branchpoint one side will contain the newly inserted integer and the other will contain either the successor or predecessor of the new integer. Given this neighbour the doubly linked successor predecessor list at the leaf level can be updated to include the newly inserted integer. Using the binary search on the levels strategy that was used to mark the active nodes the branchpoint is found and thus an insert is performed in O(loglogn) time. Using a similar process a delete can be performed in O(loglogn) time. Testing for membership, finding the successor or the predecessor can be done in a constant amount of time.

To initialize the structure it is necessary to create the static structure . This consists of the binary tree skeleton, the level and other labels and the father pointers. There are O(n) nodes in this tree. Each node contains O(loglogn) father pointers and a constant number of labels. It follows that O(nloglogn) space and preprocessing time are required to initialize the tree. Later van Emde Boas cuts down on the number of pointers to achieve an O(n) worst case space structure [van Emde Boas 77b].

Van Emde Boas tree manipulates subsets of the set of integers ranging from 1 to n. On the grid integers can range from 1 to n^k. We would like to extend van Emde Boas tree structure so that it could function on the larger universe.

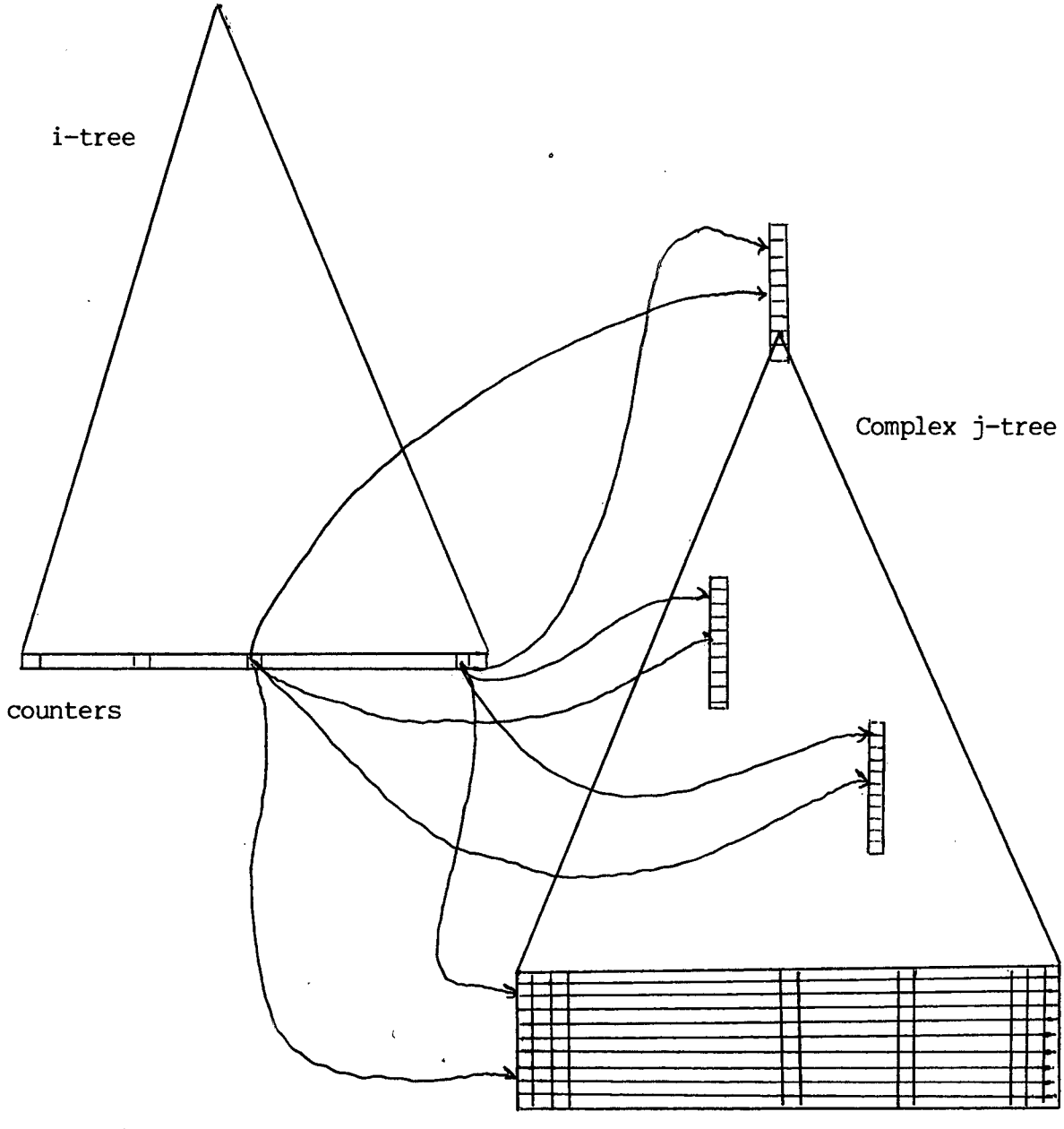
Extended van Emde Boas Tree

The following extension of van Emde Boas structure allows the processing of n instructions, when the universe is restricted to the domain of integers in the range 1 to n^k, in O(kloglogn) time per instruction in the worst case. The extended structure requires $O(n^k)$ space and O(knloglogn) preprocessing time.

$O(n^2)$ Extended van Emde Boas Tree

First consider the case where k = 2. The sets to be manipulated are subsets of the integers ranging from 1 to n^2. We can express any such integer in two digit n-ary notation (i,j) where i and j range from 0 to n - 1. A priority queue on the domain of integers ranging from 1 to n^2 can be represented by n priority queues on the domain of integers ranging from 1 to

Fig 4 $O(n^2)$ Van Emde Boas Tree Structure



n. We use one van Emde Boas structure hereafter called the i-tree to select the i coordinate . Then we use the j coordinate in one of some other van Emde Boas trees (j-trees) to finally locate the desired integer. The i coordinate selects which tree to look at and the j coordinate positions you within that tree. In order to make this extended structure efficient we separate the static and dynamic information in the nodes. In the n j-trees the static information is stored only once while each tree must have separate dynamic information. Each node of the complex j-tree will consist of an array N where N(i) represents that node in the ith tree. The static father pointers, level labels and other static information are stored only once in each node of the complex tree. This complex tree consists of n superimposed simple trees.

To handle the first coordinate i a separate simple van Emde Boas tree is built. The leaves of this tree act as a multiset containing counters indicating the number of points in the overall structure with the given i coordinate.

Lemma 3.1: This structure requires O(loglogn) time to process an instruction in the worst case.

Proof: O(loglogn) time is required to handle the i coordinate in the simple i tree. A constant amount of time is used to select the appropriate j-tree in the complex j-tree and O(loglogn) time is required to process an instruction in the j-tree.

Lemma 3.2: This structure requires $O(n^2)$ space.

Proof: There are O(n) nodes in the complex j-tree and each node requires O(n) space for the dynamic information. The simple i-tree requires only O(n) space.

Lemma 3.3: Initializing the structure requires only O(nloglogn) time

in the worst case.

Proof: We use this time to set up the static information in the i-tree and the complex j-tree. None of the dynamic information need be initialized before use. We use $O(n^2)$ space without intializing it by Lemma 1.2.

The extended van Emde Boas structure supports the operations insert, delete, test for membership, find predecessor and find successor. We insert an integer into the structure by first expressing it in n-ary notation with the coordinates i and j. We then test the i-th leaf in the i-tree. If the counter there registers zero we do a regular insert into the i-tree. If the counter registers non-zero then we merely increment the counter. The ith tree in the complex j-tree is selected. We insert the j coordinate of the number into this tree. Deletion is analogous to insertion. Membership testing can be done in constant time.

We will want to find the successor of an integer expressed in n-ary noatation with the coordinates (i,j). We begin the search by finding the successor of j in the ith tree of the complex j-tree. If one exists the successor of (i,j) is (i,succ(j)). If no successor of j exists it means that j is the maximum element of the ith part of the complex j-tree. In this case we find the successor of i in the simple i-tree. Next we find the minimum element in the succ(i) part of the complex j-tree. The successor of (i,j) is then (succ(i),min(j)). Finding the predecessor of an integer is analogous to finding the successor.

$O(n^k)$ Extended van Emde Boas Tree

We can extend van Emde Boas tree structure so that it can handle subsets of the set of integers in the range from 1 to n^k. We can express

any integer in this range in k digit n-ary notation $(i_1,i_2,\ldots i_{k-1},j)$ where $i_1,i_2 \ldots i_{k-1},j$ are integers ranging from 0 to n - 1. We use one simple van Emde Boas tree to handle i_1 just like the i-tree in the n^2 case. This reduces the problem to creating a data structure to handle the n^{k-1} case. The process of creating i-trees can be continued until the n^2 case is reached. The n^2 case uses an i-tree and a complex j-tree as before. In the n^k structure the i_1 coordinate selects the coordinate in the i_1 dimension. The complex j-tree will have k - 1 dimensional hypercubes at each node compared to the linear array in the n^2 case. As in the n^2 case we separate the static and dynamic information in the complex j-tree. The dynamic information is stored in the k - 1 dimensional hypercubes and the static information is stored separately at only the top level. The complex j-tree is really n^{k-1} superimposed simple van Emde Boas trees.

Lemma 3.4: This structure requires O(kloglogn) time to process an instruction in the worst case.

Proof: O(loglogn) time is required in each i-tree. There are k - 1 i trees so that the total time is O((k - 1)loglogn). Also O(loglogn) time is required in the chosen part of the complex j-tree. Altogether O(kloglogn) time is required.

Lemma 3.5: This structure uses $O(n^k)$ space.

Proof: Each i-tree uses O(n) space for a total of O((k - 1)n). Each of the O(n) nodes of the complex j-tree contains a k -1 dimensional hypercube of size n. The complex j-tree therefore uses $O(n^k)$ space.

Lemma 3.6: Initializing the structure requires only O(nloglogn) time in the worst case.

Proof: Now that the space requirement is up to $O(n^k)$ the initialization trick used in the n^2 case becomes vital to preprocessing time. We use this

time to set up the static information in the i-trees and the complex j-trees. As before, none of the dynamic information need be initialized.

The n^k extended van Emde Boas tree supports all the instructions that the simple tree does. To insert an integer into the structure we insert the i_1 coordinate into the multiset storage of the i_1 i-tree in the same way as we did in the i-tree of the n^2 case. Next insert the i_j coordinate into the multiset storage of the i_j i-tree. We then insert j into the appropiate part of the complex j-tree. Again deletion is analogous to insertion. Membership testing is a constant operation.

Finding the successor or predeccessor of an integer in the n^k structure is an obvious extension of doing the same in the n^2 structure.

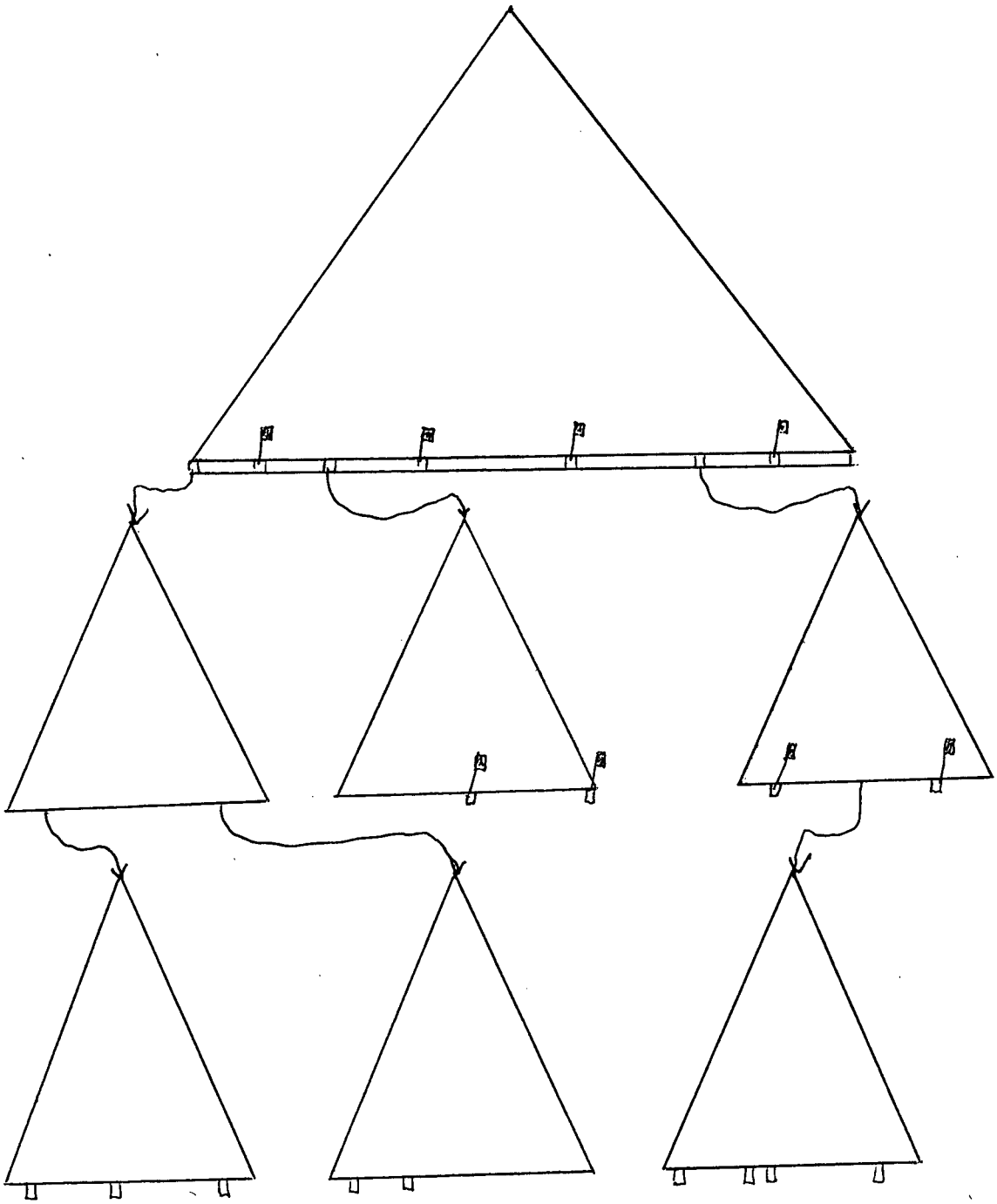
Dense Extended van Emde Boas Structure

If we only wish to represent a set of grid points with O(n) members in the n^k van Emde Boas structure there is much space that is not used. If we know that the integers we will place in the van Emde Boas tree are selected from a known subset of size O(n) of the integers ranging from 1 to n^k we can decrease the space required. We are able to further modify van Emde Boas structure so that we can process n instructions, when the universe is a known subset S of size O(n) of the integers in the range 1 to n^k, using only $o(n^2)$ space and O(kloglogn) time per instruction . This dense extended structure requires O(knloglogn) preprocessing time.

$O(n^3)$ Dense Extended van Emde Boas Structure

First consider the case where k = 3. The sets to be manipulated are subsets of a known set S of size O(n) whose members are integers ranging from 1 to n^3. We can express any such integer in three digit n-ary notation (i,j,k) where i,j, and k range from 0 to n - 1. We will use three levels of simple size n van Emde Boas trees to store these integers. At the top level there is one i-tree which we use to store the i coordinate of the integers. Descendant from some of the leaves of the i-tree are j-trees which we use to store the j coordinate of some of the integers. Descendant from some of the leaves of the j-trees are k-trees which we use to store the k coordinates of some of the integers. In order to be able to use this structure on members of subsets of set S we first must create and initialize the structure by inserting all of the universe set S into the structure and then deleting the members of S. We begin in the i-tree. All members of S are inserted into the i-tree by their i coordinates. If only one member of the set has a given i coordinate there is no need to store anything below that leaf. The number is stored at the leaf and a flag is set to show that there is nothing stored below. If more than one member of the set has the same i coordinate i_i we grow a j-tree below. At the i_ith leaf we set a counter to the number of elements with i coordinate equal to i_i. Also we initialize a pointer to point to a simple van Emde Boas tree which we will use as a j-tree. We insert the j coordinate of the elements with i coordinate i_i into the j-tree. This is done for all j-trees created. The process of inserting numbers into j-trees is analogous to inserting numbers into the i-tree. If two numbers in the same j-tree have the same j coordinate a k-tree is grown. For each k-tree grown there will be a set of two or more numbers with the same i and j coordinates. This set is inserted into the k-tree. There will not be any two numbers with the same k coordinate in the same k-tree. When

Fig 5 Dense $O(n^3)$ Extended van Emde Boas Structure



all the elements of the set S have been inserted we delete them all.

After this process is complete we are left with a structure consisting of an i-tree having descendant j-trees at some of its leaves. A j-tree may have descendant k-trees at some of its leaves. In the various trees some leaves will not have been touched and will be empty, some leaves will contain a counter and a pointer to a descendant tree and some leaves will contain a number and a flag indicating there is no structure below.

This dense extended van Emde Boas structure supports the operations insert, delete, test for membership, find predeccessor and find successor. We insert an integer into the structure by first expressing the integer in n-ary notation with coordinates i,j and k. We start in the i-tree. If the ith leaf has no structure below it we do a regular insert of i into the i-tree and we are done. Otherwise we test the counter at the ith leaf. If the counter there registers non-zero then we merely increment the counter. If the counter there registers zero we do a regular insert of i into the i-tree. We now insert the j coordinate into the descendant j-tree. If the jth leaf has a k-tree below it we also insert k in the descendant k-tree. Membership testing and deletion are analogous to insertion.

We will want to find the successor of an integer N expressed in n-ary notation with coordinates (i,j,k). We begin by finding the leaf that represents the integer N. This leaf may be in the i-tree if N is the only integer in the universe set with i coordinate i or the leaf may be in a j-tree or a k-tree. In the most general case the leaf representing N was in a k-tree. The successor of N is then the successor of N in the k-tree (i,j,succ(k)). If N is the maximum element in the k-tree it is necessary to look in the j-tree. The successor of N is the successor of j in the j-tree with the minimum k coordinate (i,succ(j),min(k)). If N was the maximum

element in the j-tree the successor of N is the successor of N in the i-tree with the minimum j and k coordinates (succ(i),min(j),min(k)). Finding the predecessor of an integer is analogous to finding the successor.

Let us now consider the complexity of this structure.

Lemma: 3.7 Only O(n) of the various i,j and k trees are necessary.

Proof: There will always be one i-tree used. Descendant j-trees are formed only if two or more elements of the set S have the same i coordinate. There can be at most n/2 j-trees. Regardless of how many j-trees there are a k-tree is formed only if two or more elements of the set S have the same j coordinate in the same j-tree. This can happen at most n/2 times. There are at most n/2 k-trees and therefore there are at most O(n) simple van Emde Boas trees in the structure.

$O(n^k)$ Dense van Emde Boas Structure

This dense extended van Emde Boas structure can also handle integers that are selected from a known set S of size O(n) whose members are integers ranging from 1 to n^k. We can express any such integer in k digit n-ary notation $(i_1,i_2 \ . \ . \ . \ i_k)$ where $i_1,i_2 \ . \ . \ . \ i_k$ are integers ranging from 0 to n - 1. We will use k levels of simple size n van Emde Boas trees to store these integers. At the top level we use one i_1 tree to store the i_1 coordinate of the integers. Descendant from some of the leaves of the i_1 tree are i_2 trees which we use to store the i_2 coordinate of the integers and so on. As in the case where k = 3 we create and initialize the structure by inserting all members of S into the structure. Handling the instructions insertion, deletion, membership testing, finding successor and finding predecessor in the n^k case is a simple generalization of handling

the instructions in the case where k = 3.

Lemma 3.8: Only O(kn) of the various $i_1, i_2 \ldots i_k$ trees are necessary.

Proof: There will be one i_1 tree. Descendant i_j trees are formed only if two or more elements of S have the same i_j coordinate in the same i_{j-1} tree. There can be at most n/2 i_j-trees. Therefore there can be at most 1 + (k - 1)n/2 or O(kn) trees altogether.

In the sparse extended van Emde Boas tree structure we stored the static pointers of a number of simple van Emde Boas trees only once in the complex j-tree. We can do the same in the dense structure. We can begin with O(kn) superimposed simple size n van Emde Boas trees. The static pointers are stored only once. Each node of the complex tree will consist of an array N where N(i) represents the node in the ith tree.

Lemma 3.9: The dense extended van Emde Boas structure requires O(kloglogn) time to process an instruction in the worst case.

Proof: O(loglogn) time is required in each i_j-tree. At most O(k) of the i_j trees have to be considered so that the total time is O(kloglogn).

Lemma 3.10: The dense extended van Emde Boas structure requires $O(n^2)$ space in the worst case.

Proof: The i_1 tree uses O(n) space. Each of the other trees uses O(n) space because they use the static pointers of the i_1 tree. By lemma 3.8 there are at most O(kn) of the simple van Emde Boas trees. $O(kn^2)$ space is required in the worst case.

Lemma 3.11: The dense extended van Emde Boas structure to hold integers ranging from 1 to n^k that are selected from a known set S of size O(n) requires O(knloglogn) time to set up in the worst case.

Proof: None of the dynamic information in any of the simple trees need

be initialized. The static pointers require O(nloglogn) time to set up. To create the structure each of the O(n) elements of S is inserted. Each insertion requires O(kloglogn) time in the worst case. The O(n) insertions will require O(knloglogn) time in the worst case. This dominates the preprocessing time.

We use of the capabilities of the dense extended van Emde Boas tree in chapter 4 when locating a point in a planar subdivision. We can also use the dense extended van Emde Boas tree to detect intersections among polygons in chapter 5.

Searching in Planar Subdivisions

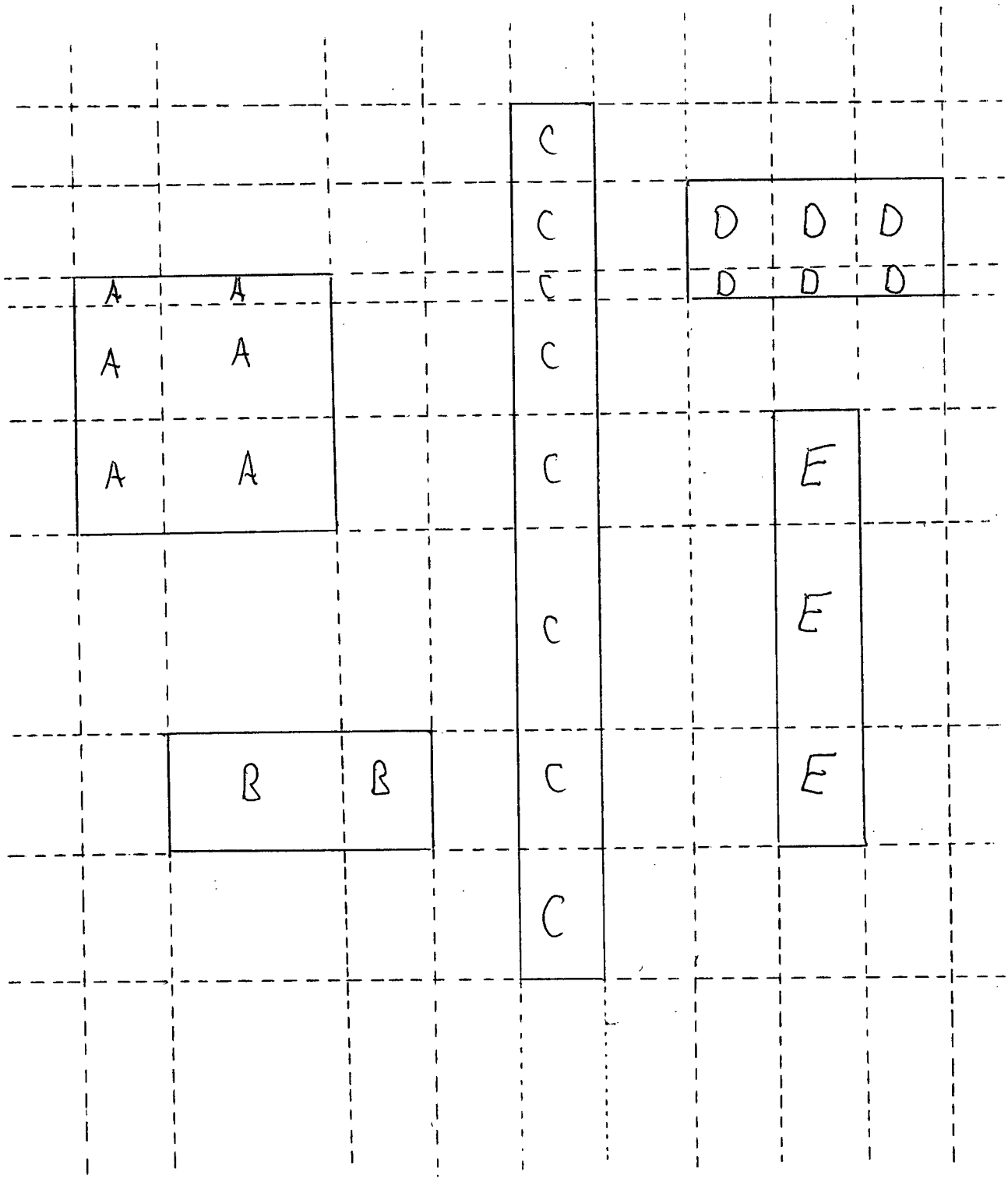
Rectangles

We present three algorithms for searching in a restricted planar subdivision. We work in the $O(n^k)$ size grid domain. Working on the grid enables us to employ the dense extended van Emde Boas tree and get better search times then are possible in the real plane. However the preprocessing time and space requirement are slightly worse than in the real plane case. Nonoverlapping rectangles is the first type of planar subdivision we will consider. Given n nonoverlapping rectangles whose corners lie on grid points, in which (if any) of the rectangles does a new test point lie?

Shamos and Bentley [Shamos 77] have given an O(nlogn) preprocessing time and space and O(logn) test time algorithm for the problem in the real plane. The test time can be cut to O(loglogn) on the grid using a dense extended van Emde Boas structure. However the preprocessing time goes up to $O(n^2)$ and the space required is $O(n^2)$.

We begin by taking all the sides of the rectangles and extending them to lines. Now the plane is divided into $O(n^2)$ rectangular regions determined by at most 2n vertical lines and at most 2n horizontal lines. The vertical lines are the vertical lines that pass through corners of rectangles. These are numbered from left to right. The horizontal lines are the horizontal lines that pass through corners of rectangles. These are numbered from bottom to top. A 2n x 2n matrix is used to indicate which original rectangle (if any) a given newly created region is in. For each original rectangle we determine the small regions determined by the lines that it covers. These smaller regions are labeled by two coordinates. The

Fig 6 Locating a Test Point among Rectangles



first is the number of the vertical line which bounds it on the right. The second is the number of the horizontal line that bounds it on top. If a region (i,j) is covered by rectangle "B" then "B" is the label placed at position a_{ij} of the matrix. See figure 5. In this way we initialize the matrix to contain the information about which small regions are covered by which original rectangles.

The next step is to set up a dense extended van Emde Boas tree to hold the O(n) x coordinates of the corners of the rectangles. Insert the x coordinate of the corners of the rectangles in the tree. Label these coordinates with the number of the vertical line that passes through them as determined above. This tree will serve to locate a test point with respect to the vertical lines passing through the corners of the rectangles. This tree will give the first coordinate of the position in the matrix where the answer is stored. Set up a second dense van Emde Boas tree for the y coordinates. Insert the y coordinate of the corners of the rectangles in the tree. Label these coordinates with the number of the horizontal line that passes through them. This tree will serve to locate a test point with respect to the horizontal lines.

Given a test point it is sufficient to determine which region (i,j) determined by the extended lines it is in. Once the coordinates of the region are known we merely look at position a_{ij} of the matrix set up in the preprocessing to determine in which original rectangle (if any) the test point lies. We can determine between which two vertical lines the test point lies in O(loglogn) time with the predecessor and successor instructions on the dense extended van Emde Boas tree. This gives the first coordinate i of the region. The same process works in the second dense extended van Emde Boas tree to determine the second coordinate j of the

region.

Lemma 4.1: $O(n^2)$ time is required for preprocessing in this algorithm.

Proof: There are $O(n^2)$ regions determined by the lines. It will therefore take $O(n^2)$ time to set up the answers corresponding to these regions in the matrix. The two dense van Emde Boas trees will take O(nloglogn) time to initialize. The total preprocessing time for this algorithm will be $O(n^2)$.

Lemma 4.2: The test time is only O(loglogn) once the preprocessing is complete.

Proof: It takes O(loglogn) time to use the dense extended van Emde Boas tree to determine each coordinate of a region.

Lemma 4.3: The algorithm requires $O(n^2)$ space.

Proof: The matrix takes $O(n^2)$ space. Each extended van Emde Boas tree requires $O(n^2)$ space. The total space requirement is $O(n^2)$.

Rectilinear Planar Subdivisions

A slightly more general problem is that of searching in a rectilinear planar subdivision. A rectilinear planar subdivision is a subdivision of the plane where all divisions between regions are vertical and horizontal line segments. We assume that no two line segments intersect except at their endpoints. We will again be working in the grid domain. This means that all the endpoints of these segments will be grid points. There are n of these endpoints or corners in the subdivision. In which region of a O(n) size grid base rectilinear subdivision does a new test point lie?

The algorithm and analysis of this problem are very similar to that of

the non-overlapping rectangle problem. We extend all vertical and horizontal line segments to lines. These lines are numbered as before. Initializing the matrix will be slightly more complicated. One way to do this would be to divide each original rectilinear area into rectangles and proceed as in the rectangle case. Although more care must be taken the preprocessing time remains $O(n^2)$. The preprocessing can be performed using a special case of the method described in the following section. The space required is again $O(n^2)$ and the test time O(loglogn).

Searching in a Restricted Angle Subdivision

Generalizing the problem still further we let the line segments separating regions in the subdivision lie at a fixed restricted number of angles. Again we assume that no two of these line segments intersect except at their endpoints. The subdivsion is made up of n line segments. No segment is unnecessary in so much as its removal will not alter the regions. The line segments are restricted to lie at z different angles. We need not use the rectangular grid in the following algorithm. We could use a grid created by overlaying z different series of evenly spaced lines. Each series would lie at one of z possible angles.

A restricted angle grid based planar subdivision is exactly the kind of subdivision that must be produced by most plotters. Nearly all digital plotting is based on the use of a Cartesian coordinate system in which successive data points are constrained to lie on nodes of a square grid [Freeman 79]. The square grid is popular because of the wide use of the Cartesian coordinate system for data representation and the simplicity of the hardware which is able to use independent systems for the two coordinate

positioning mechanisms. These devices are restricted to plotting lines that are parallel to the coordinate axes or at an angle of 45 degrees to them. Occasionally other systems are used so that multiples of 30 or 60 degrees are allowed.

The question is therefore: in which region of an O(n) size grid based subdivision where the angles of the line segments are restricted to z values does a new test point lie? Again the algorithm and analysis of the problem are similar to the non-overlapping rectangle problem. We extend each of the n line segments to lines. We number the lines of each orientation separately. There could be as many as O(n) lines at any given orientation. There are z different possible orientations. A z dimensional matrix of size n is required to allow for all possible regions created by the lines defined by the segments in the subdivision. However in any given subdivision there will only be $O(n^2)$ regions defined by the O(n) lines because of planarity. It is only in these $O(n^2)$ regions that we want to record the answer in the matrix.

If the original subdivision is given as a basic list of adjacencies as chosen by Lee and Preparata [Lee 77], it can be converted to Kirkpatrick's edge-ordered representation [Kirkpatrick 79] in linear time by taking advantage of grid sorting capabilities. Each original region has associated with it a list in clockwise order of the edges bounding that region. From this representation original regions can be considered one by one.

At each angle "a" lines orientated at angle "a" are placed in an extended van Emde Boas tree for "a" orientation. These lines are labeled with the numbers they would have if counted left to right.

Consider one original region, say region A, of the restricted angle subdivision. Region A will contain as many as $O(n^2)$ fine regions defined by

Fig 7 Touring a Region of a Restricted Angle Subdivision

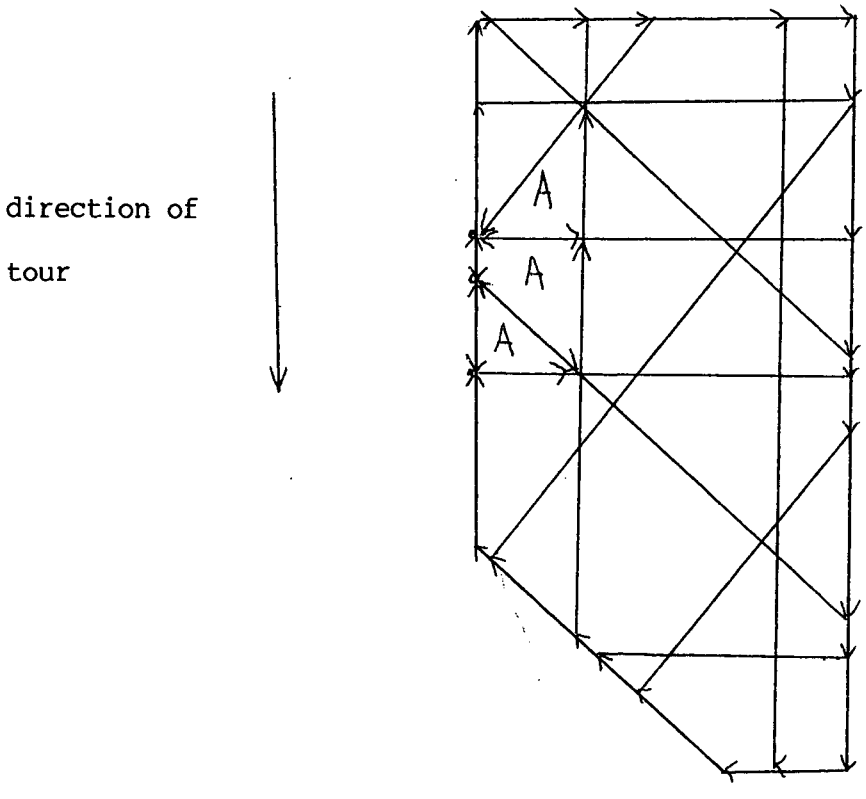
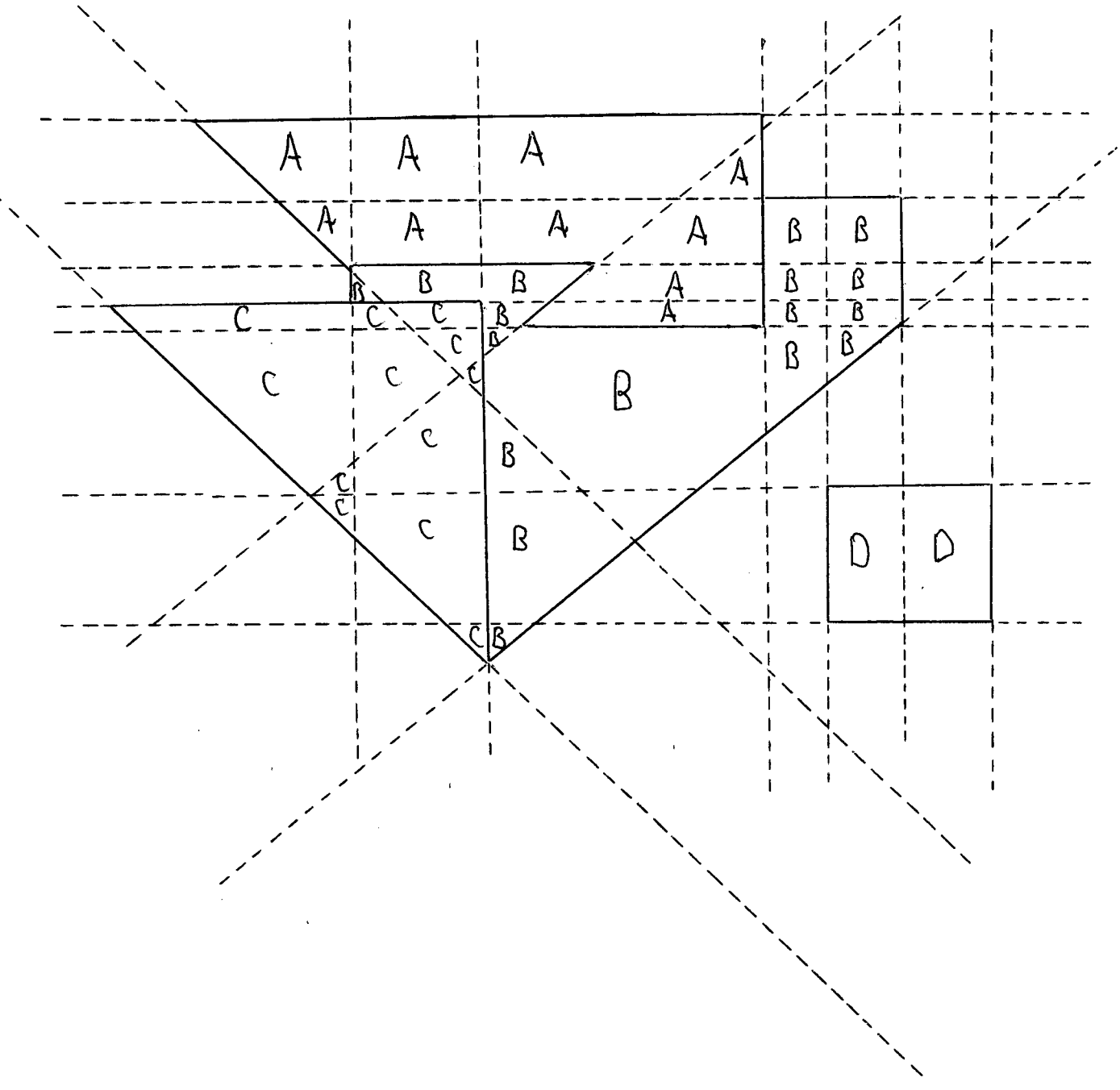


Fig 8 Locating a Point in a Restricted Angle Subdivision



segments of the lines extended from the original segments of region A. We need to store in the matrix the fact that each of these fine regions is in region A. The coordinates of a fine region in the matrix are the coordinates of a point in the region with respect to the lines that are stored in the dense extended van Emde Boas trees. For the purposes of initializing the matrix we also store the line segments in this matrix. A line segment is stored below region (i,j,k) in the matrix if the segment separates (i,j,k) from (i,j,k + 1). Associated with each line segment stored in the matrix are two flags which indicate whether the line segment has been traveled on in the up direction and the down direction. Associated with each orientation is an up direction and a down direction.

Initially all the line segments that make up the boundary of region A are marked in clockwise order. One of the direction flags is set for each segment on the boundary of region A. A corner point on the boundary is selected to begin a touring and shrinking process during which fine regions are marked "A" in the matrix and the size of the unmarked portion of region A decreases. See figure 7. From the starting point a segment is followed to a junction. This segment is marked in the opposite direction to that on which it was travelled. A right turn is made and another segment followed and marked in the reverse direction. This process continues until the starting point is reached. The fine region encircled is marked with an "A". This process is repeated until there are no untravelled line segments leaving the starting point. Next a segment that has been travelled in both directions is followed to advance the starting point to a new location. This process continues until all line segments in region A have been marked in both directions and all corner points have been used as starting points. At this time all fine regions within region A will have been marked. The

process is repeated for all original regions in the subdivision as shown in figure 8.

Initializing the matrix takes $O(n^2)$ time since there are $O(n^2)$ fine regions and each region is examined a constant number of times. The matrix however takes $O(n^z)$ space. Most of the space is never accessed and therefore need not be initialized by lemma 1.2.

A dense extended van Emde Boas tree is set up for each orientation. Given a test point we find one coordinate from each of the van Emde Boas trees. These z coordinates locate the position in the n^z matrix where the answer is stored.

Lemma 4.4: The total preprocessing time for this algorithm will be $O(n^2)$.

Proof: initializing the matrix requires $O(n^2)$ time. Initializing each of the z van Emde Boas trees requires O(nloglogn) time.

Lemma 4.5: O(zloglogn) time is required for each test.

Proof: O(loglogn) time is required to find each of the z coordinates of a test point.

Lemma 4.6: The space requirement for the algorithm is $O(n^z)$.

Proof: The matrix requires $O(n^z)$ space. Each dense extended van Emde Boas tree requires $O(n^2)$ space.

Intersection Problems

Shamos and Hoey [Shamos 76] studied a number of geometric intersection problems. They present an O(nlogn) time algorithm for detecting whether any two of n line segments intersect. Bentley and Ottman [Bentley 79b] present an O(nlogn + clogn) algorithm for reporting and counting all intersections among n line segments where c is the number of intersections.

Rectangle intersection problems have recently been investigated by a number of people [Bentley 79a] [Vaishnavi 79a] [Vaishnavi 79b] [Vaishnavi 79c] [Bentley 79b] [Shamos 77]. There are several related problems to consider. Shamos and Bentley [Shamos 77] present an O(nlogn) algorithm for detecting whether any two of n rectangles edge intersect. Bentley and Ottman present an O(nlogn + c) algorithm for reporting and counting all edge intersections among n rectangles. Bentley, Vaishnavi and Wood [Bentley 79a] [Vaishnavi 79a] [Vaishnavi 79c] have given O(nlogn + c) algorithms for reporting each time one rectangle encloses another.

Intersection of Rectangles

The problem we will consider is to detect whether any two of n rectangles whose corners are grid points intersect. The rectangles have sides that are parallel to the coordinate axis. By intersect we mean both edge intersection and enclosure. The following algorithm accomplished this in O(nloglogn) time using an extended van Emde Boas structure. However the space requirement is $O(n^2)$.

First we sort the left and right sides of the rectangles by x coordinate. We begin a left to right sweep of the sides starting with the side with minimum x coordinate. If we encounter a left side we insert the top left and bottom left y coordinates into an $O(n^k)$ size dense extended van Emde Boas tree structure. Mark the corners in the tree so that we can tell whether they are tops or bottoms. If we encounter a right side during the sweep we delete the top and bottom y coordinates from the tree.

Each time we make an insertion we perform some tests to detect intersections. An intersection has occured if either the successor or predecessor of a top is a top; also if either the successor or predecessor of a bottom is a bottom. With these successor and predecessor tests we can detect both edge intersection and enclosure among rectangles.

Clearly if the top of one rectangle is between the top and the bottom of another rectangle the two rectangles intersect. If there are a number of intersections among a set of rectangles the top two rectangles that intersect will have the property that their tops are consecutive and the botoom two rectangles that intersect will hav e the property that their bottoms ar consecutive.

Lemma 5.1: The algorithm requires $O(n^2)$ space.

Proof: The extended van Emde Boas tree requires $O(n^2)$ space. This is the dominant space requirement of the algorithm.

Lemma 5.2: The algorithm requires O(nloglogn) time in the worst case.

Proof: The initial sort of the sides of the rectangles requires O(n) time in the grid domain. O(nloglogn) time is required to initialize the van Emde Boas tree. Each insertion, deletion, find successor or find predecessor requires O(loglogn) time. There can be O(n) of these

operations. The total time requirement for the algorithm is O(nloglogn).

Intersection of Rectilinear Polygons

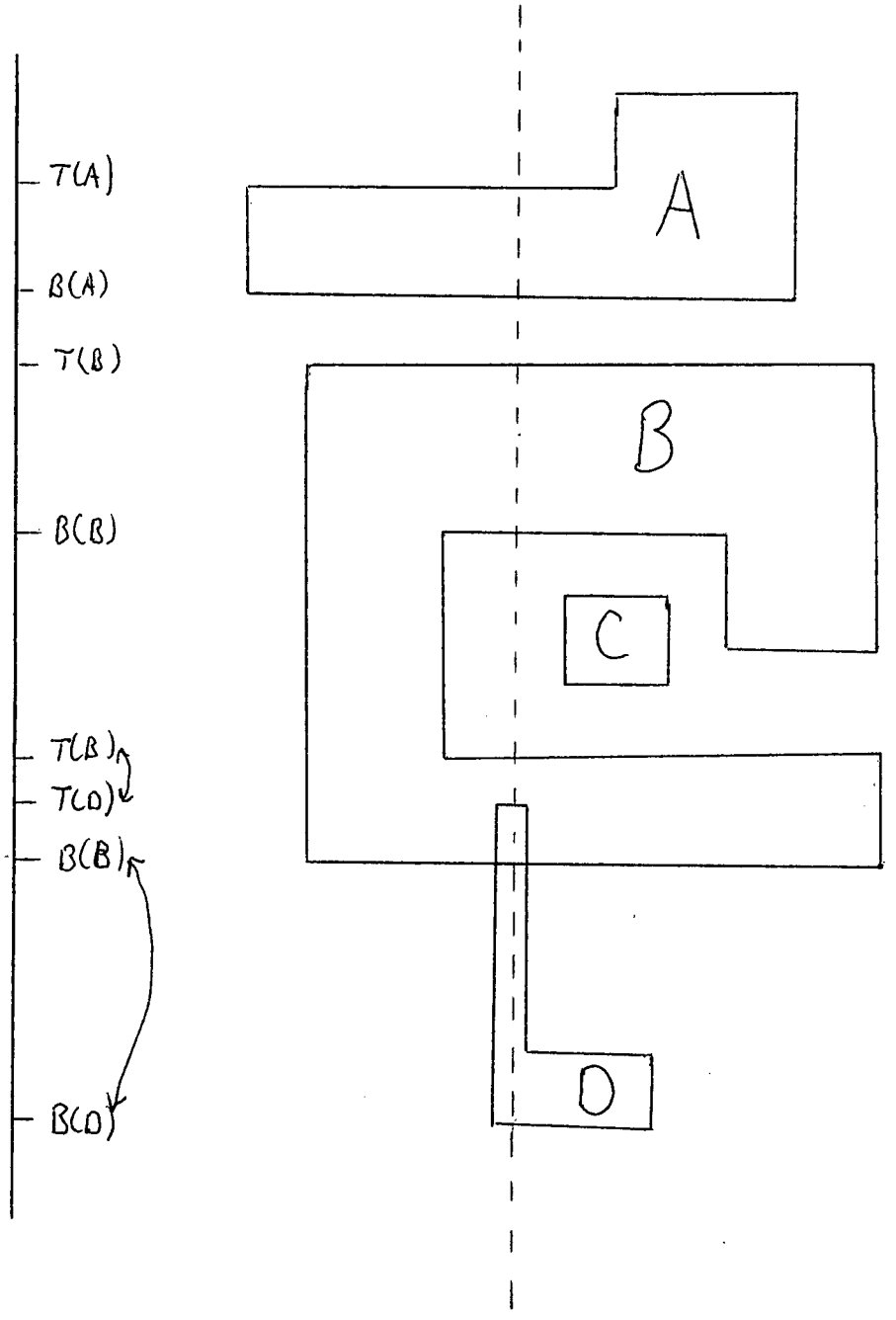
We can generalize this algorithm so that it will work with rectilinear polygons with sides parallel to the coordinate axes. We will detect whether any two of a set of rectilinear polygons which have n corners which lie on grid points intersect. The time and space requirements are the same as in the rectangle intersection algorithm.

We begin the algorithm by marking each vertical line segment as either left or right. A line segment is marked left if the space to its right is occupied by the interior of a polygon. Also mark each vertical line segment with the number of the polygon that it is a part of. Next sort the vertical line segments by x coordinate.

We begin a left to right sweep of the vertical line segments by starting with the segment with the minimum x coordinate. If we encounter a left side we insert the y coordinate of the top and bottom endpoints of the segment into the extended van Emde Boas tree. We mark these points either top or bottom and we also mark them with the number of the polygon that they are a part of. We then perform the successor and predecessor tests as in the rectangle case to check for intersection.

If we encounter a right side the procedure is more complex. Consider the top endpoint of the right side. If the top of a left side from the same polygon has the same y coordinate then delete the top of the left side from the tree. Otherwise insert the y coordinate of the top endpoint of the right side into the tree. Label it bottom. Find its successor in the tree. It will be the top of a left side of the same polygon. Perform analogous

Fig 9 Detecting Intersections among Rectilinear Polygons



operations with the bottom endpoint of the right side. All of this is the same as deleting the segment $B_l - T_l$ and inserting the two segments $T_r - T_l$ and $B_l - B_r$ if necessary. See figure 9.

Lemma 5.3: The algorithm requires $O(n^2)$ space for the van Emde Boas tree. Again there are O(n) tree operations each of which requires O(loglogn) time. The algorithm requires O(nloglogn) time.

Proof: as in the rectangle case.

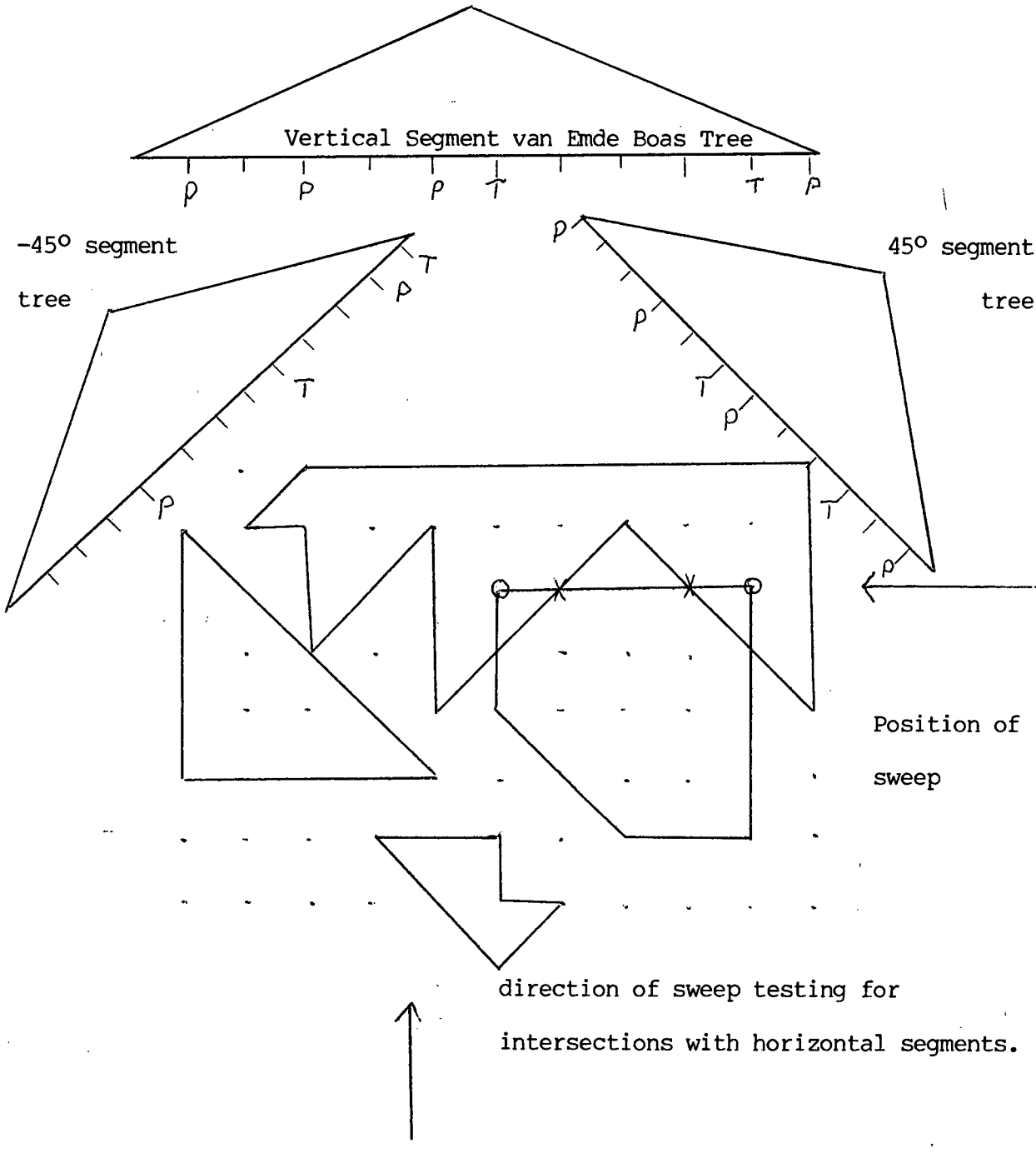
Intersection of Restricted Angle Polygons

We can detect intersection among still more general polygons. These polygons have sides that are restricted to lie at a fixed number of slopes. The set of polygons we will consider has n corners which lie at grid points. The problem is to detect whether any two of a set of polygons whose sides are restricted to lie at one of z possible angles intersect. We solve this problem by first presenting an algorithm which will detect any edge intersections among the polygons and then presenting an algorithm which will find any cases of one polygon enclosing another.

This algorithm for detecting edge intersections among restricted angle polygons requires $O(z^2 n \log\log n)$ time and $O(zn^2)$ space. We begin by dividing the sides into z different classes according to slope. This can be done in linear time. Select one slope I and sort the corners of the polygons according to an axis with a slope of angle I. We will perform a minimum to maximum sweep in the angle I direction. There are z - 1 extended van Emde Boas trees to hold polygon sides. One to hold each orientation of side except for angle I. During the sweep if we encounter the first endpoint of a segment that lies at an angle other than angle I the segment is inserted

Fig 10 Edge Intersections among Restricted Angle Polygons

- 'P' in a tree represents a present segment at the given position of the sweep. 'T' represents the temporary insertion of the two circled endpoints. A 'P' between the 'T's in any tree means an intersection has been found.



into The van Emde Boas tree for that angle. There is a grid size numbering of the possible segments of each orientation. A segment of a given non-I orientation is entered into the tree for its orientation simply by making the leaf corresponding to its number present. If we encounter the second endpoint of a segment that lies at an angle other than I the segment is deleted from its van Emde Boas tree. If we encounter an angle I segment (we will encounter both endpoints at the same time) we test for intersection by looking for a possible intersection in each of the z - 1 trees. We test for intersection between an angle I segment and segments of other orientations by temporarily inserting the two endpoints of the angle I segment into each of the z - 1 van Emde Boas trees. If the successor of the left endpoint of the angle I segment is not the right endpoint of the angle I segment an intersection has been found. This sweep will find intersections of angle I segments with any other segment. We empty all the van Emde Boas trees and repeat the algorithm letting each angle be angle I.

Lemma 5.4: The total space requirement for the algorithm is $O(zn^2)$.

Proof: Each of the z extended van Emde Boas trees will require $O(n^2)$ space.

Lemma 5.5: The algorithm requires $O(z^2 n \log\log n)$ time in the worst case.

Proof: $O(n\log\log n)$ time is required to initialize each of the z extended van Emde Boas trees. $O(zn\log\log n)$ time is required to perform each of the sweeps. Each of the n segments may require z intersection tests. The total time requirement for the algorithm is $O(z^2 n \log\log n)$.

The following algorithm will detect enclosures among grid based restricted angle polygons.

We begin by labeling each non-vertical segment as either top or bottom. We sort the endpoints lexicographically first by x coordinates and second by y coordinates. We will perform a left to right sweep. In the case of ties the sweep will move from top to bottom.

When the leftmost endpoint of a segment is reached we determine if it is a top. If it is a top we find the closest segment above it in various angle van Emde Boas trees that are each holding the segments of one orientation. If the closest segment is also a top then the top of one polygon is between the top and bottom of another so either an enclosure or possibly an edge intersection has been detected. Otherwise we insert the segment into the extended van Emde Boas tree for its orientation. When the right endpoint of a segment is reached the segment is deleted from its tree.

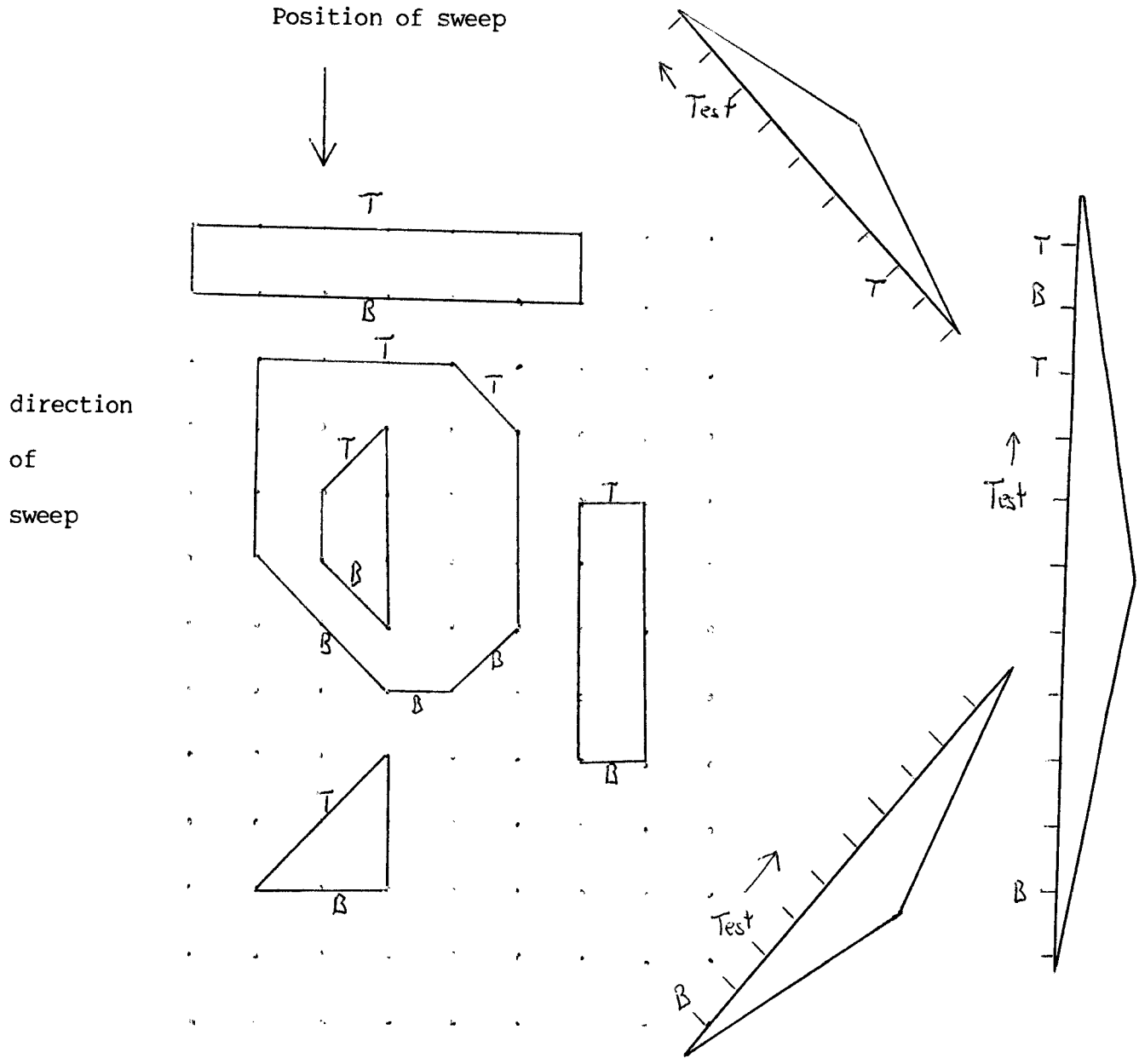
Lemma 5.6: $O(zn^2)$ space is required for the algorithm in the worst case.

Proof: $O(zn^2)$ space is required to store the z extended van Emde Boas trees.

Lemma 5.7: O(znloglogn) time in total is required for the algorithm.

Proof: Each time a top is inserted O(zloglogn) time is required to perform tests. There are O(n) tops.

Fig 11 Enclosure Detection among Restricted Angle Polygons



An enclosure has been found in the horizontal tree. There is a top directly above the test point.

Conclusions

In this thesis we have studied a number of geometric problems in the integer grid setting. By taking advantage of integer grid sorting capabilities we have been able to describe linear time algorithms for a number of grid based geometric problems. These include finding the convex hull of a set of grid points and solving related problems. These also include finding the diameter and finding a triangulation of a set of grid points. It is open whether or not the Delaunay triangulation or another triangulation with special properties for a set of n grid points can be found in linear time.

We have described an algorithm that will find the Voronoi polygon about one of n grid points in linear time. We have not determined whether or not the complete Voronoi diagram of a set of n grid points or even of the points of a grid based convex polygon can be found in less than O(nlogn) worst case time.

By extending van Emde Boas tree we have retained the fast searching and preprocessing times at the expense of space requirements. We have used the dense extended van Emde Boas tree to obtain fast algorithms for some searching in subdivision and detection of intersection problems. O(loglogn) search time algorithms are presented for searching in a set of rectangles, in a rectilinear planar subdivision and in a restricted angle planar subdivision. We have been able to detect intersections among rectangles, rectilinear polygons and restricted angle polygons in O(nloglogn) time.

The speed of all the algorithms we have described depends on the integer grid domain. Whether or not algorithms are practical depends on

whether or not the integer grid domain is a realistic domain for an application. If an integer grid is a realistic domain one can take advantage of the algorithms described in this thesis, otherwise more general algorithms must be used.

The Euclidean L_2 norm was used as a metric in this thesis. Some of the results, such as the Voronoi polygon algorithm, will carry over when other norms such as the L_1 and L_∞ norms are used.

We could use some of the integer algorithms presented in this thesis to find fast expected time algorithms for real problems if the points are selected from appropriate distributions. The real points would first be rounded to integers. The real points need be selected from a distribution with the property that not more than a constant number of the real points would round to the same integer. The problem would then be solved in the integer domain and the solution mapped back to the real domain perhaps after some corrections. This process could be used to produce fast expected time triangulation or convex hull algorithms for real plane point sets.

Many integer structures and algorithms use only the easily sortable property of integers. If we are presented with a bounded set that has already been sorted many integer methods can be used even if the numbers are real. In many of the geometric problems we considered sorting was the most time consuming step in the solution. Sorting can in many cases reduce real problems to integer problems.

Bibliography

[Aho 74] Aho,A.V., Hopcroft,J.E. And Ullman,J.D., The Design and Analysis of Computer Algorithms, Addison - Wesley (1974).

[Anderson 78] Anderson,K.R. "A Reevaluation of an efficient algorithm for determining the convex hull of a finite planar set" Information Processing Letters 7,1 (Jan 1978) pp53-55.

[Andrew 79] Andrew,A.M. "Another Efficient Algorithm for Convex Hulls in Two Dimensions" Information Processing Letters 9,5 (Dec. 1979) pp216-219

[Baird 78] Baird,H.S. "Fast algorithms for LSI artwork analysis" Design Automation and Fault-tolerant Computing (1978), pp179-209.

[Bentley 79a] Bentley,J.L. And Wood,D. "An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles" Technical Report McMaster University 79-CS-13.

[Bentley 79b] Bentley,J.L. And Ottman,Th. "Algorithm for reporting and counting geometric intersections" IEEE Transactions on Computers 26,8 (Sept 1979).

[Brown 78] Brown,K.Q., "Fast Intersection of Half Spaces" Carnegie - Mellon University (June 1978) Technical Report CMU-CS-78-129.

[Dobkin 76] Dobkin.D. And Lipton,R.J., "Multidimensional Searching Problems," SIAM J. Comput. 5,2 (June 1976), pp181-186.

[van Emde Boas 77] van Emde Boas,P., Kaas,R., Zijlstra,E., "Design and implementation of an efficient priority queue", Math. Systems Theory 10,2 (1977), pp99-127.

[van Emde Boas 77b] van Emde Boas,P., "Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space", Information Processing Letters 6,3 (June 1977) pp80-82.

[Fortune 78] Fortune,S., and Hopcraft, J. "A Note on Rabin's Nearest-Neighbor Algorithm" (1978) Technical Report Cornell University 78-340.

[Freeman 79] Freeman,H., "Algorithm for Generating a Digital Straight Line on a Triangular Grid" IEEE Trans. On Computers 28,2 (February 1979) pp150-152.

[Graham 72] Graham,R.L. "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set" Information Processing Letters 1 (1972) pp. 132-133.

[Kirkpatrick 79] Kirkpatrick,D.G. "Optimal Search in Planar Subdivisions". Detailed Abstract 1979.

[Kung 75] Kung,H.T., Luccio,F., and Preparata,F.P. "On finding the maxima of a set of vectors". Journal of the ACM 22,4 (Oct. 1975) pp. 469-476.

[Lauther 78] Lauther,U. "4 dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits", Design Automation and Fault-tolerant Computing (1978) pp241-247.

[Lawson 72] Lawson,C.L. "Generation of a triangular grid with applications to contour plotting", Cal. Inst. Of Tech. Jet Prop. Lab. Technical Memorandum 299 (1972).

[Lee 77] Lee,D.T. And Preparata,F.P., "Location of a Point in a Planar Subdivision and its Applications", SIAM J. Comput. 6,3 (Sept. 1977), pp594-606.

[Lee 79] Lee,D.T. And Preparata,F.P., "An Optimal Algorithm for Finding the Kernel of a Polygon" (August 1979) Journal of the ACM 26,3 pp415-421.

[Lipton 77] Lipton,R.J. And Tarjan,R.E., "Applications of a Planar Separator Theorem", Proc. 18th Annual IEEE Symposium on Foundations of Computer Science (Oct. 1977), pp162-170.

[Nagy 79a] Nagy,G. And Wagle,S.G. "Geographic Data Processing" ACM Computing Surveys 11,2 (June 1979) pp139-181.

[Nagy 79b] Nagy,G. And Wagle,S.G. "Approximation of Polygonal Maps by Cellular Maps" Comm. ACM 22,9 (Sept. 1979), pp518-525.

[Preparata 77] Preparata,F.P. "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm. Of the ACM 20,2 (Feb. 1977) , pp.87-93.

[Shamos 75a] Shamos,M.I. "Geometric Complexity", Proceedings of the Seventh Annual ACM Symposium on Theory of Computing, (May 1975), pp224-233.

[Shamos 75b] Shamos,M.I., and Hoey,D. "Closest Point Problems", Proc. Of the 16th Annual IEEE Symposium on Foundations of Computer Science, (Oct. 1975), ppl51-162.

[Shamos 76] Shamos,M.I., and Hoey,D. "Geometric Intersection Problems Proc. 17th Annual IEEE Symposium on Foundations of Computer Science, (1976) pp208-215.

[Shamos 77] Shamos,M.I., and Bentley,J.L., "Optimal Algorithms for Structuring Geographic Data", Proc. Of an Advanced Study Symposium on Topological Data Structures for Geographic Information Systems, Harvard Univ., 1977.

[Shamos 78] Shamos,M.I., "Computational Geometry", Doctoral Dissertation, Yale University, New Haven, CN, 1978.

[Sibson 78] Sibson,R. "Locally Equiangular Triangulations", 1978 The Computer Journal 21,3 pp243-245.

[Vaishnavi 79a] Vaishnavi,V., "Optimal worst-case algorithms for rectangle intersection and batched range searching problems", Technical Report McMaster University 79-CS-12

[Vaishnavi 79b] Vaishnavi,V., Kriegel,HP and Wood,D., "Space and Time Optimal Algorithms for a Class of Rectangle Intersection Problems", Technical Report McMaster University 79-CS-15

[Vaishnavi 79c] Vaishnavi,V. And Wood,D. "Data Structures for the Rectangle Containment and Enclosure Problems" Technical Report McMaster University 79-CS-19