Michael Ian Shamos and Jon Louis Bentley
Center for Computational Geometry
Departments of Computer Science and Mathematics
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
USA
(412) 621-2600 x265

OPTIMAL ALGORITHMS FOR STRUCTURING GEOGRAPHIC DATA

Abstract

This paper combines new results in computational geometry with a brief tutorial on the design and analysis of fast algorithms. We show a natural connection between the recursive behavior of an algorithm and the data structure that it operates on, and use this correspondence to produce new methods of data organization that facilitate rapid retrieval of geographic information. Algorithms for range searching, Thiessen polygons, triangulation, nearest-neighbor searching and various rectangle overlap problems are presented and analyzed. The method of reducibility is used to prove lower bounds on the computation time required to solve these problems and demonstrate that our algorithms are optimal.

1.1. Introduction

Analysis of algorithms has enjoyed a brief but interesting history. At one time, it was an accomplishment merely to be able to devise any algorithm at all for solving a problem on a computer. (It was even more of a challenge to get the resulting program working and tested.) Computer scientists demand more today: the algorithms must be efficient in terms of running time and storage used, designed so that they can be programmed readily, and should be provably optimal. That is, we must be able to demonstrate that no algorithm, regardless of how clever it might be, can possibly execute faster than the given one. It is easy to see that the game of algorithm design would never end, in the absence of such an optimality result -- the algorithm builder would never know when to stop trying to improve his program. What is considerably less obvious is how one might go about proving that an algorithm is the best one possible. In this paper we will present some current design and analysis techniques within the framework of problems in geography and spatial analysis. Hopefully, the reader will be able to tailor these methods to his own applications.

The problems and algorithms we shall discuss are part of a wider discipline known as *Computational Geometry*, which seeks to understand the interaction between geometry and computing. Its task is to isolate fundamental problems and develop optimal algorithms to solve them, building, so to speak, a set of computing *tools* that can be used in diverse applications. While we intend to delve into several algorithm design methods, by far the most important idea that we will discuss is that if *divide-and-conquer*, a process by which we are able to solve a problem faster by dividing it into smaller parts, solving the subproblems separately, then combining their solutions into a solution to the larger problem. That this

splitting can actually result in any saving of time is often a source of mystery to those who have not seen the method explained clearly. For this reason, we shall present many examples of the divide-and-conquer technique and show how it achieves its remarkable efficiency.

We believe that the user of algorithms should have some familiarity with current design technology as well as a knowledge of which algorithms are actually available. Development of a new algorithm and corresponding code can be, even if successful, a very expensive undertaking. We thus intend to acquaint the reader with as many new results as possible.

1.2. Analysis of Algorithms

The primary subject matter of the field of "Analysis of Algorithms" is obvious from its name. Other aspects of the field are indicated by such other names as "Applied Computational Complexity" and "Design of Algorithms". Since our purpose in this paper is to describe the application of this field to topological data structures, we will profit by briefly reviewing some of its goals and methods.

1.2.1 Goals

The immediate task of Analysis of Algorithms is to describe the difficulty of computing the answer to a given problem on a given machine. As we further investigate this seemingly simple goal, we will see just how complex it is. For example, when referring to "difficulty", do we mean the amount of time used or the amount of storage required? (In some contexts, a more relevant measure might be the number of man-weeks needed to produce working code.) In practice, most algorithms are analyzed for time and space

requirements, but should this be for an HP 65 or a CDC 7600? Most current work in computer science theory ignores differences among machines for several good reasons: First, an analytical result is intended to have wide applicability, and an analysis tailored to the characteristics of any one machine would neither be very useful nor would reveal much about the structure of the problem. Another reason is that counting instructions at the machine-language level is rarely profitable, because clever reprogramming can render the analysis useless. In view of this, we concentrate instead on ***asymptotic analysis***, giving only the order of magnitude of the number of operations used. Such results are applicable over a wide range of computer models and are often quite easy to obtain. We use the notation O(f(N)) to stand for the set of all functions g(N) such that there exist positive constants C and N_0 with $|g(N)| \leq Cf(N)$ for all $N \geq N_0$. We will adhere to the common terminological abuse which permits the writing of such statements as $3N^2 - 6N + 5 = O(N^2)$. (The error is that $O(N^2)$ denotes a set, not a function.) Special notation has been devised to describe lower bounds, in which one wants to say that g(N) is *at least as large as* some function f(N) (See Knuth [1976]), but its use does not seem necessary in this overview.

Another important distinction in analyzing an algorithm is whether we are describing its average (expected-time) or worst-case behavior. For many applications we are concerned only with the average and care little if the running time is occasionally longer. On the other hand, there are some applications (Air Traffic Control is the most oft-cited example) in which one particularly time-consuming operation could bring about disastrous results.

A most difficult concept to grasp initially is that the goal of Analysis of Algorithms is not to study the complexity of a given algorithm but of a given ***problem***. This distinction often appears blurred because the analysis of at least one algorithm is part of the analysis of the entire problem (to obtain an upper bound on running time).We can clarify this distinction by means of an example: Consider the problem of sorting a set of N numbers by pairwise

comparisons of elements of the set. In the next section we give an algorithm (named MERGESORT) which sorts N numbers in O(N lg N) comparisons (both in the worst case and the average case). Thus far we have spoken about the problem by speaking about the algorithm: by the existence of MERGESORT we know that sorting requires no more than O(N lg N) comparisons. One can, however, speak directly about the problem itself. Suppose we knew of no specific algorithm for sorting. It would still make sense to try to prove that *any* algorithm must take some definite minimum time, and that no faster procedure could possibly accomplish the task correctly. Such a result is known as a ***lower bound***. For the sorting problem, one can use arguments from information theory to show that any algorithm must use at least O(N lg N) comparisons to sort N numbers (in both the worst and average-case senses). In brief, an upper bound is normally demonstrated by exhibiting an algorithm with the desired time behavior, a lower bound often requires more sophisticated methods ***since it must apply to all conceivable algorithms***. Since the upper and lower bounds on MERGESORT are identical, we say that it is an ***optimal algorithm***. A common error is to show an algorithm which uses a sort step and to claim that this establishes a ***lower bound*** of O(N lg N) for the problem. This is not true unless one has shown the sort step to be ***necessary***.

In summary, then, the goal of Analysis of Algorithms is to analyze problems along the following dimensions:

Time required
Storage required
Worst case behavior
Expected behavior
Upper bounds
Lower bounds

These dimensions are by no means independent -- an algorithm that is allowed to use more storage may require less time. We will see many examples of such tradeoffs in later sections.

1.2.2 Methods

In the following sections we will see a number of particular algorithms, and the reader not familiar with their design and analysis might fail to note the analogies that exist between superficially dissimilar algorithms. In this section we will review a number of prominent methods used in algorithm design in a fairly abstract manner; the reader will see concrete examples of these methods in the algorithms we have chosen to present.

Let us start by reviewing two techniques for constructing algorithms: iteration and divide-and-conquer. Iteration is perhaps the most common algorithmic technique. It consists of the repetitive performance of a block of code under the control of a loop variable. Control structures for iteration are included in most programming languages and the resulting algorithms are usually simple to analyze: if a set of instructions requiring M steps is executed N times, then the total cost of the iteration is MN.

Divide-and-conquer is a special case of recursion, a more sophisticated and powerful technique. It takes advantage of the fact that solving small problems is often much more efficient, in relation to their size, than solving large problems. For example, to sort a list of N numbers, we might split them arbitrarily into two lists, sort each list separately, and then combine the two sorted lists into a single one in a merge procedure. In this case there are two subproblems, each of half the size of the original. The reason that this method is ***recursive*** is that the subproblems themselves are solved by splitting into further subproblems, etc., until a point is reached at which we only have to sort sets containing single elements, which is trivial. It remains to be seen how all of the overhead involved in splitting and merging results in an efficient algorithm.

The sorting procedure described above is known as MERGESORT and, as our first example of a divide-and-conquer algorithm, it deserves a more formal description:

Procedure MERGESORT (S):

1. If S contains 0 or 1 elements, then return S. (It is already sorted.)

2. Split S into two lists A and B, each containing N/2 elements.

3. Sort A and B by making recursive calls on MERGESORT (i.e. CALL MERGESORT(A) and MERGESORT(B).

4. Merge the sorted lists A and B together, and return the resulting sorted list.

If the lists are represented as vectors or linked lists, the required operations can be performed readily. The merge of step 4 can be accomplished by simple iteration: the next element of the sorted list is obtained by comparing the first elements of lists A and B, finding the smallest element, and deleting it. Finding each element except the last requires one comparison, so the total number of comparisons performed is one less than the number of elements in both lists.

The running time of MERGESORT on a set S of N elements is easy to analyze by the use of recurrence relations. Assume (without loss of generality, as it turns out) that N is a power of two; this allows us to always divide S into A's and B's of equal size. Denote the number of comparisons between list elements required by MERGESORT on a list of N elements by T(N); we now may obtain a recurrence relation for T(N) directly from the recursive structure of the algorithm. No comparisons between elements of

S are performed in steps 1 or 2. By the definition of T, each call in Step 3 will require time T(N/2), so the total cost of Step 3 is 2T(N/2). Step 4 makes exactly N-1 comparisons. Since no comparisons at all are needed to sort a list containing only a single element, the recurrence for T(N) is

$$T(N) = 2T(N/2) + N-1 , \qquad T(1) = 0 .$$

(The initial condition, T(1) = 0, is an essential part of the recurrence.)

The constant term "-1" somewhat complicates the solution, so we will replace the recurrence by T(N) = 2T(N/2) + N, whose solution is surely greater than or equal to that of the original. We now solve the modified equation by "iteration", which consists of repeated substitution for the unknown function T(N).

$$\begin{aligned} T(N) &= N + 2T(N/2) \\ &= N + 2[\, N/2 + 2T(N/4) \,] \\ &= N + N + 4T(N/4) \\ &= N + N + 4[\, N/4 + 2T(N/8) \,] \\ &= N + N + N + 8T(N/8) \ldots . \end{aligned}$$

Since N can be divided by two only lg N times before one is reached, this recurrence will iterate exactly lg N times, each step adding N operations to the total. Thus we see that T(N) = N lg N. Though this derivation is quite informal, there are many very powerful formal tools capable of handling such recurrences.

This example captures much of the flavor of divide-and-conquer algorithms. It consists of three steps: Divide, Solve Recursively, and Marry. The resulting algorithm is easily seen to be correct (by induction) and can be analyzed in a natural way by the use of recurrence relations.

Let us now turn our attention to the problem of structuring data, for the

performance of an algorithm is highly dependent on the manner in which its data is organized. The simplest example that illustrates this point is the problem of searching for an element in a vector. If the elements have been sorted in advance (structured), it is always possible to accomplish the search in O(lg n) comparisons. If no ordering has been done, as many as N-1 comparisons may be needed. Two structures which we will use repeatedly are linked lists and binary trees; the reader unfamiliar with these should refer to Knuth [1968]. As we present new structures that we have developed for topological problems we will often show how they are related (isomorphic, in a certain sense) to a given algorithm. In particular, we will see many examples in which a divide-and-conquer algorithm corresponds directly to a binary tree in the way it repeatedly splits its input. Data structures are at the heart of time-space tradeoffs -- allowing more space in which to store redundant information often reduces the time needed to search a database. Suppose, for example, we wanted to learn whether any english word contains the four-letter sequence "hipe". Certainly all the information necessary to answer this question is contained in an unabridged dictionary, but finding the answer will involve reading every word because ***the dictionary is not organized to facilitate this kind of search***. Its alphabetical order is useful for a different type of query. What is the alternative? Given sufficient foresight (and computer time) we could have produced a sorted list of all four-character strings that occur in english. This list would be much longer than the dictionary and would take some time to prepare, but would allow our type of query to be handled efficiently. (Incidentally, there are twelve words that contain the sequence "hipe", of which the only familiar one is "archipelago".)

There are many standard operations on sets which algorithm designers perform repeatedly, such as Insertion (put x in set S), Inclusion (is x in S?), Deletion (remove x from S), Minimum (what is the smallest element in S), etc. These operations have been thoroughly analyzed and algorithms for performing them are available as primitive tools for the algorithm designer

(often as language primitives). We will occasionally make use of some of these standard operations without specifying exactly how they can be implemented; Aho, Hopcroft and Ullman [1974] will serve as a reference.

Further tools which we will employ in the following sections are some basic lower bound techniques. Specifically, we will borrow previously established results through the method of "reducibility". The following example is taken from Shamos [1975]. The element uniqueness (EU) problem is to determine whether there are any duplicates in a set of N real numbers, and has been shown (by rather advanced methods) to require O(N lg N) comparisons. The closest pair problem (CP) is to find the difference between the two closest numbers in the set. We can show a lower bound of O(N lg N) for CP by reducibility. Suppose CP ran Suppose that there exists an algorithm for CP that runs in less than O(N log N) time. This algorithm can be used to solve EU in less than O(N lg N) time, which is impossible because of the known lower bound. Given a set of numbers, solve CP. If the difference between the two closest numbers is zero, there is some element that is repeated; otherwise, they are all distinct. In this way CP solves EU and must be at least as difficult. All of the lower bounds in this paper will be obtained in exactly the same manner.

1.3. Range searching

Suppose that we have a file representing a map of all the cities in the Continental United States. One search which might well arise is of the form "How many cities are between latitude lines c and d and also between meridians a and b?".(Such a search would be useful in counting the number of cities in Wyoming or Colorado!) This is an example of a ***range search***, in which we are given a set of points in the plane and wish to know how many lie in a given rectangle. In this section we will study several different algorithms for answering this type of query.

A search algorithm normally consists of two phases: the ***preprocessing***, during which the input data is organized into a structure that will facilitate the eventual query, and the search itself, based on the data structure selected. In analyzing the complexity of a search algorithm there are three components to discuss: the preprocessing time, the search time, and the amount of storage used by the data structure. If the original data can be modified between searches (for example, by adding new points or deleting old ones), then one must also describe the cost of such modifications. In this paper we will usually not treat the matter of updating because it is extremely involved and is the object of much current research.

In this section we will examine range searching in detail. In addition to the problem of finding the number of points in a given rectangle, we will also examine such related problems as listing all points in a given range and various ***weighted*** range queries. In some applications weights are associated with the points (population figures, for example) and we wish to know not just how many points lie in the range, but how much weight it contains. We proceed by first examining a few relatively simple approaches to the problem, and then investigate in detail two data structures particularly designed for range searching.

The simplest searching algorithm possible is often called, for obvious reasons, "brute force". In this search strategy no preprocessing time and no extra storage are required. The search must, however, investigate every item in the set, which takes O(N) time. This technique is appropriate if there are going to be very few searches and fast response time is not required. It may happen that a large expenditure in preprocessing time is not justified. For most applications, however, this approach is not acceptable.

A slightly more sophisticated technique for range searching is known as "projection" or, in data base terminology, the "inverted file". The

preprocessing consists of sorting the points by, say, the X coordinate (hence "projecting" the file onto the X-axis). A search then uses binary search to locate the particular X range requested then searches all the points in that X range to find those which are also in the desired Y range. The preprocessing requires O(N lg N) time for the sort (recall that we have upper and lower bounds for sorting) and O(N) memory locations. Unfortunately, n the worst case a search might look at the entire set without finding any points in the range, so the worst-case complexity of searching is O(N). The average search time is difficult to analyze; one has to know the probability distributions of both the points and the ranges that will be queried in order to make any precise statement. For most applications, however, this method also requires too much search time.

Two techniques which we will next investigate solve the problem of range searching by transforming it to an equivalent problem called "ECDF searching". Given a set S of N points in the X-Y plane, the point P is said to ***dominate*** Q iff $X_P \geq X_Q$ and $Y_P \geq Y_Q$. The empirical cumulative distribution function (ECDF) at a point R is defined as the number of points in S that R dominates. If we could design an algorithm to perform ECDF searching quickly it would allow range searching to be done quickly. (Refer to Figure 1.) Let ECDF(P) denote the ECDF of point P. We now observe that the number of points in rectangle ABCD is

$$\text{RANGE(ABCD)} = \text{ECDF(A)} - [\text{ECDF(B)} + \text{ECDF(D)}] + \text{ECDF(C)}.$$

Thus if the time required for an ECDF search is S(N), a range search can be performed in at most 4S(N) operations. (In addition to its use in range searching, the ECDF is also of interest in statistics.)

We will now examine a strategy for ECDF searching based on what can be called the "Locus Method". In this technique we attempt to characterize the locus of all points that have the same ECDF in such a way as to allow

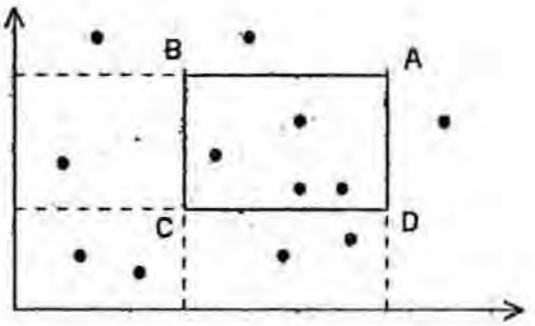


Figure 1: A range search as four ECDF queries.

fast ECDF searching. Imagine a point P that moves through the point set S horizontally and note that P's ECDF can change only when it assumes the X value of one of the points in S. This same phenomenon holds for vertical lines and Y-values. Thus, if we were to divide the plane into boxes defined by vertical lines and horizontal lines through every point in the set, then all points in any box would have the same ECDF. We illustrate this in Figure 2.

The number in each box is the ECDF for points within that box.

It is clear that the ECDF values can be stored as a matrix and searched quickly. To find the ECDF of a new point, we must first locate the vertical slab in which it lies by binary search, then find the particular box in that slab by another binary search. The cost of these two searches is O(lg N). Unfortunately, this method requires $O(N^2)$ preprocessing time and storage to create and store the boxes. Such a high cost (particularly of storage) usually prohibits the use of such a structure in practice. We have shown, however, that logarithmic ECDF searching (and therefore range searching) is possible, given that one is willing to pay a sufficiently high price.

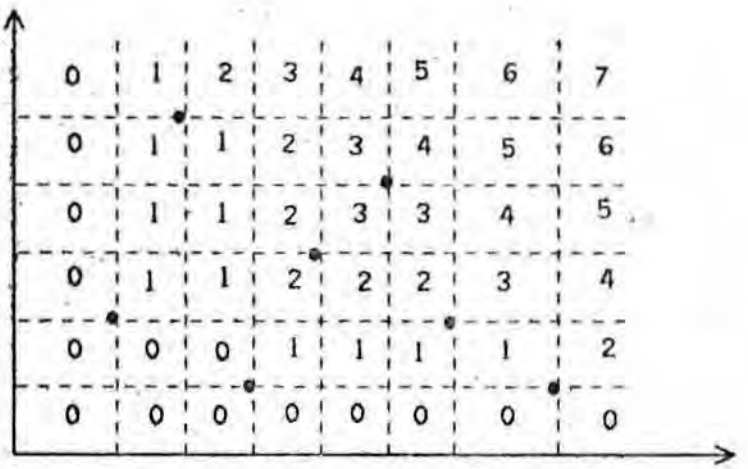


Figure 2: Regions within which the ECDF is constant.

None of the approaches we have described so far is really appropriate in practice. We turn now to investigate in detail two structures specifically designed for ECDF searching which will prove more attractive.

1.3.1 ECDF Trees

In this section we will examine a data structure designed by Bentley and Shamos [1977a] specifically to solve the problem of ECDF searching. It is a tree data structure, built by a divide and conquer algorithm, so we will describe it recursively.

Suppose that we draw a vertical line L through the point set S such that

half the points lie on each side of L. The line L divides the plane into two half-planes, the left half-plane A and the right half-plane B. What we have described so far is the root of a tree data structure representing the point set S. The points in A will be stored in A's right subtree and the points in B will be stored in the left subtree (and then these subtrees will be stored in the same way, etc., recursively). There are now two kinds of ECDF searches that we have to do: those for points to the left and right of L. We illustrate these cases in Figure 3.

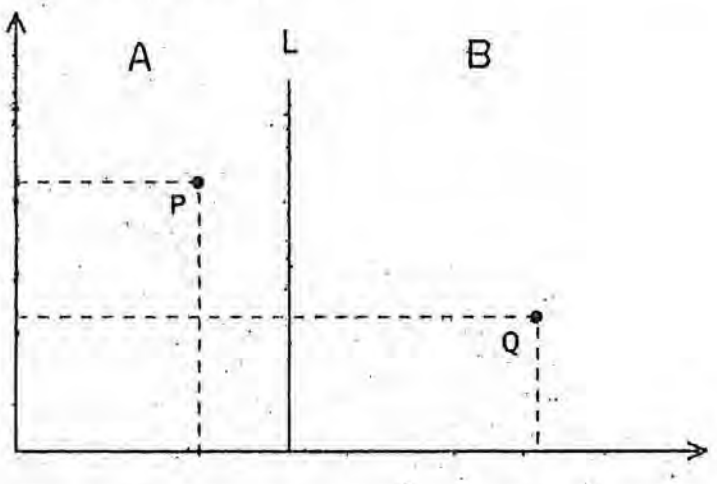


Figure 3: A Divide-and-Conquer ECDF Query

A query which falls in A (such as P) is easy to handle. Since P can dominate no points in B, the ECDF of P is just its ECDF among the points in A. That value can be found by recursively doing an ECDF search in the left subtree of the root. The case represented by Q is more difficult to handle, but is made much easier as we note that the total ECDF of Q is the sum of the number of points it dominates in A and in B. We can find the number of

points dominated in B recursively by searching the tree. To find the number of points in A which Q dominates we observe that Q's X-value is greater than the X-value of every point in A. Therefore Q dominates a point in A iff Q's Y-value is greater than that point's Y-value. We will store in the binary tree node representing S all the points of A, sorted by Y-value. To find how many points in A the point Q dominates, we find its Y-value in that set by binary search, and note how many points have a smaller Y-value. In Figure 4 we illustrate the root node of an ECDF tree. The horizontal lines from each point to L represent the sorted list (by Y) of the points in A. To find the ECDF of Q we do a binary search in the sorted list to count the points in A it dominates then recursively search B to find the number of points dominated there.

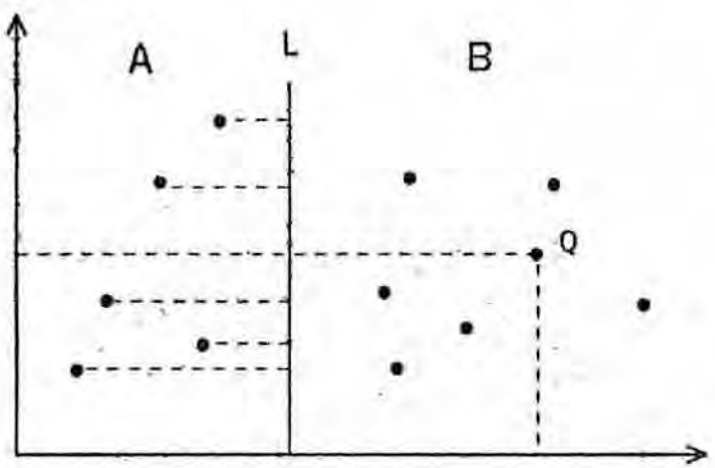


Figure 4: Root Node of the ECDF Tree

Now that we know how a search will be performed, we can describe precisely the data structure required. The point set S will be represented

by a binary tree. At each node in the tree we store the X-value defining L, pointers to the left son (the subtree which represents A) and the right son (represents B), and an array of the points Y-values of the points in A sorted into increasing order. We now define SEARCH, a procedure which returns the ECDF of P.

Procedure SEARCH (point P, tree T):

1. If T contains only one point, return zero or one according to whether or not P dominates that point.

2. If P is to the right of the L value of T, return SEARCH(P, left son of T).

3. (We know that P is to the left of L). Perform a binary search in the sorted array associated with T, and let P's rank in that set be R. Return R + SEARCH(P,right son of P).

The worst-case analysis of SEARCH is made simple by the use of recurrence relations. Let S(N) be the worst-case cost of searching an ECDF tree of N nodes. If N = 1, then SEARCH will take only two comparisons to see if the point is dominated, giving S(1)=2. For larger N the worst case occurs when P is to the right of L, forcing us to do a binary search (of cost lg N) and then recur down the subtree (costing S(N/2)). The resulting recurrence is

$$S(N) = S(N/2) + \lg N \ , \quad S(1) = 2 \ .$$

This recurrence is well known to have solution $S(N) = O((\lg N)^2)$.

The ECDF tree is built recursively. To construct the node representing the set S we find the median X-value in S and use that as L. We then sort all the points in A by Y-coordinate and associate the resulting sorted vector with the node. Then we recursively create subtrees by applying the same algorithm to sets A and B (a divide-and-conquer algorithm). Analysis of this preprocessing shows that it requires O(N lg N) time and the tree it produces

needs only O(N lg N) space. The recursive nature of the algorithm makes it unnecessary for us to consider more than one level of the tree at a time, which is very helpful conceptually. A finished ECDF tree is shown in Figure 5.

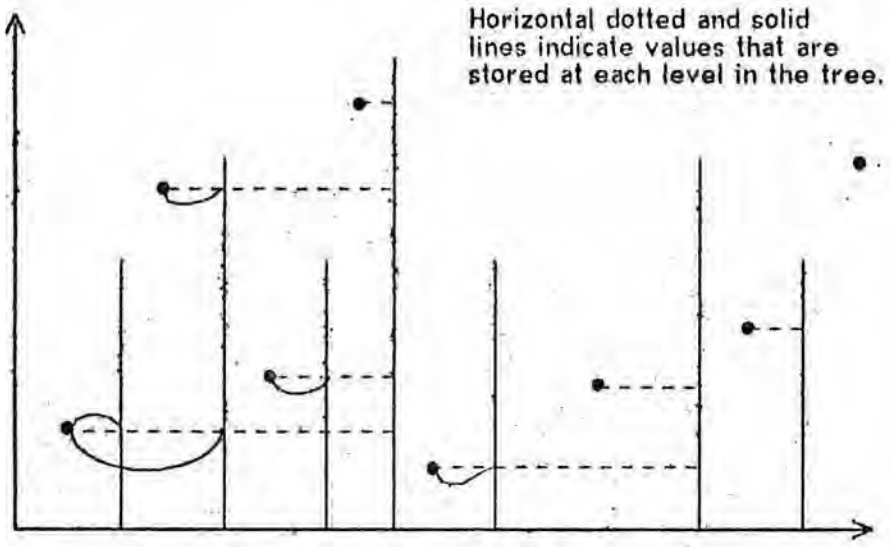


Figure 5: A Complete ECDF Tree

The basic ECDF tree can be extended in certain ways by assigning to each point a weight. We then ask for the sum of the weights that a new point P dominates. If the points were cities on a map, an appropriate weight might be the population of each city. A range search based on such an ECDF algorithm could then tell the population of a given range. A different weighting scheme would allow the existence of different classes of points (say, relics dating before or after a certain time period) and a range search could tell how many of each class of points were in a given range.

In summary, the ECDF tree allows ECDF searching (and therefore range searching) to be performed in $O((\lg N)^2)$ time after $O(N \lg N)$ preprocessing. The storage cost is $O(N \lg N)$ and weighted problems can be solved by a straightforward generalization.

1.3.2 k-d Trees

The multidimensional binary search tree (abbreviated k-d tree) was introduced by Bentley [1975]. It is a data structure suitable for use in many multidimensional problems; in this discussion we will only mention its planar applications. A k-d tree is a binary tree structurally similar to ECDF trees. To build a k-d tree representing a set S of points one proceeds by first drawing a vertical line L that partitions S into two sets A and B of equal size. (This step is exactly like that for ECDF trees.) In the next step one partitions A (and likewise B) into equal-sized subsets, except this time using a vertical line. This partitioning continues, simultaneously partitioning the space and constructing a tree. The root of the tree represents the whole set, its left subtree represents the points in A, and its right subtree represents the points in B. In Figure 6 we show a k-d tree, represented on the left by a binary tree and on the right by a partitioning of the plane.

A range search in a k-d tree is easily described recursively. Start by searching the root. At each node, test to see if the range intersects the regions of both sons. If so, then the search must proceed down both nodes; otherwise the search visits only the intersecting son.

The preprocessing necessary to build a k-d tree is $O(N \lg N)$ and the structure requires $O(N)$ storage. The search algorithm is rather difficult to analyze because it is highly dependent on the size of the range. Lee and Wong [1976] showed a highly pessimistic worst-case bound of $O(N^{.5})$ for

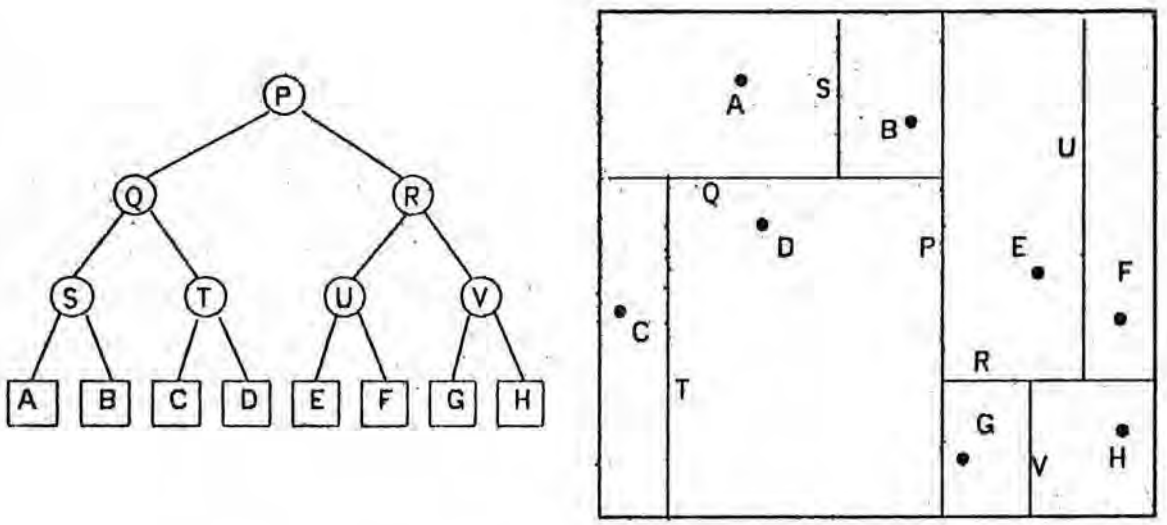


Figure 6: A k-d Tree on Eight Points.

range searching. Bentley and Stanat [1975] analyzed the average time required for searching when the points were drawn from a uniform distribution and showed that the average search time was much less than predicted by the worst-case analysis.

In addition to counting the number of points in the region, the k-d tree also solves weighted problems (as did the ECDF tree) and actually listing the points in the desired region (which is not possible with the ECDF tree). Also of interest is the fact that the k-d tree facilitates other operations, such as finding nearest neighbors (see Friedman, Bentley and Finkel [1975]).

In this section we have seen four primary strategies for range searching. Our analyses of these algorithms are summarized in

Algorithm	Search Time	Preprocessing Time	Storage
Brute Force	$O(N)$	$\emptyset$	$O(N)$
k-d trees	$O(N^{.5})$	$O(N \lg N)$	$O(N)$
ECDF trees	$O((\lg N)^2)$	$O(N \lg N)$	$O(N \lg N)$
Locus	$O(\lg N)$	$O(N^2)$	$O(N^2)$

This table makes clear the tradeoffs in choosing a particular algorithm over the others. Notice that just as the column describing search time is decreasing, so the columns describing preprocessing times and storage requirements are increasing. Evidently, if one wants faster searches, then he must pay for them with increased storage and preprocessing costs.

Lower bounds are known for many of the problems in this section, and can be found in the original references. It is easy to show a lower bound of $O(\lg N)$ on the range searching problem, so the Locus Method is optimal. The preprocessing algorithms for ECDF trees and k-d trees have also been shown to be optimal.

1.4. The Thiessen Diagram

A great many problems involve the notion of proximity, or "closeness" of points. Some of these are single-shot, such as finding a tree of shortest total length on a given finite set. Others involve searching, such as the nearest-neighbor problem: "Given N points in the plane, preprocess them so that the point closest to a new given point can be found quickly." The

problem that we will concentrate on in this section involves both single-shot processing and searching, and affords us the opportunity of introducing a geographic structure of great theoretical and algorithmic importance, the Thiessen diagram. This problem is *triangulation*: Given N points in the plane, connect them with non-intersecting straight-line segments so that each region interior to the convex hull is a triangle. Being a planar graph, a triangulation on N vertices has at most 3N-6 edges. A solution to the problem consists of a list of these edges. A triangulation is shown in Figure 7.

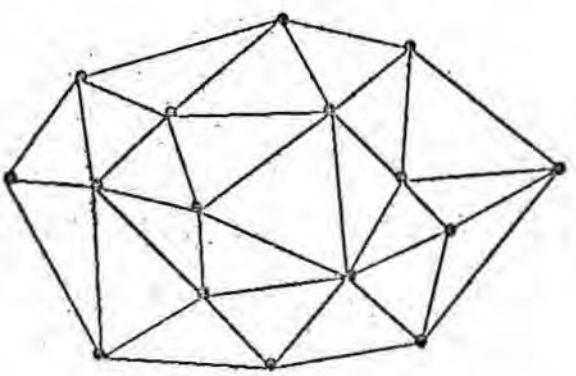


Figure 7: Triangulation of a Point Set

This problem arises in contouring and in numerical interpolation of bivariate data when function values are available at N irregularly-spaced data points (x_i, y_i) and an approximation to the function at a new point (x, y) is desired. One method of doing this is by piecewise linear interpolation, in which the function surface is represented by a network of planar triangular facets. The projection of each point (x, y) lies in a unique facet and the function value $f(x; y)$ is obtained by interpolating a plane through

the three facet vertices. Triangulation is the process of selecting triples that will define the facets. Many criteria have beeen proposed as to what constitutes a "good" triangulation for numerical purposes, some of which involve maximizing the smallest angle or minimizing the total edge length. These conditions are chosen because they lead to convenient proofs of error bounds on the interpolant, not because they necessarily result in the best triangulation. Later we will propose a new method of triangulation and show that it can be found as rapidly as *any* triangulation on N points. Meanwhile, we will content ourselves with another lower bound: by showing that O(N log N) comparisons are necessary to triangulate N points in the plane, even if no restriction is placed on the properties the triangulation must possess. We will prove that any triangulation algorithm can be used to sort. Consider the set of N points x_i pictured in Figure 8, which consists of N-1 collinear points and another not on the same line. This set possesses only one triangulation, the one shown in the figure. The edge list produced by a triangulation can be used to sort the x_i in O(N) additional operations by simple list manipulation, so O(N log N) comparisons must have been made. Now that we have a lower bound, how can we go about constructing a triangulation?

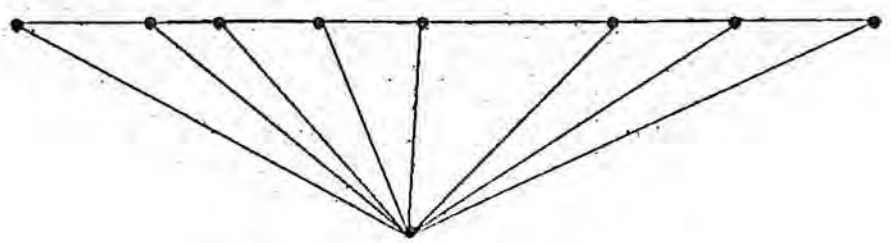


Figure 8: Triangulation Lower Bound

A valuable heuristic for designing geometric algorithms is to look at the

defining loci and try to organize them into a data structure. In this case we are given N points in the plane and want to find the locus of points (x, y) in the plane that are closer to p_i than to any other point?

If we knew these loci we would be able to solve the nearest neighbor problem directly, since determining the closest point to (x, y) is the same as asking which locus it lies in. Given two points, p_i and p_j, the set of points closer to p_i than to p_j is just the half-plane containing p_i that is defined by the perpendicular bisector of p_i and p_j. Let us denote this half-plane by $H(p_i,p_j)$. The locus of points closer to p_i than to any other point, which we denote by V(i), is an intersection of half-planes. This means that V(i) is a convex polygonal region having no more than N-1 sides, defined by

$$V(i) = \bigcap_{i \neq j} H(p_i, p_j) \qquad (1.1)$$

V(i) is called the Thiessen polygon associated with p_i. A Thiessen polygon is shown in Figure 9. (These polygons are also called Dirichlet regions, Voronoi polygons, or Wigner-Seitz cells. Dan Hoey has suggested the more descriptive term, "proximal polygon".)

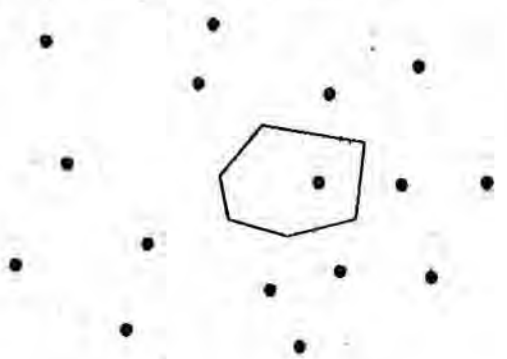


Figure 9: A Thiessen Polygon.

These N regions partition the plane into a convex net which we shall refer to as the *Thiessen diagram*, which is shown in Figure 10. The vertices of the diagram are *Thiessen points*, and its line segments are *Thiessen edges*.

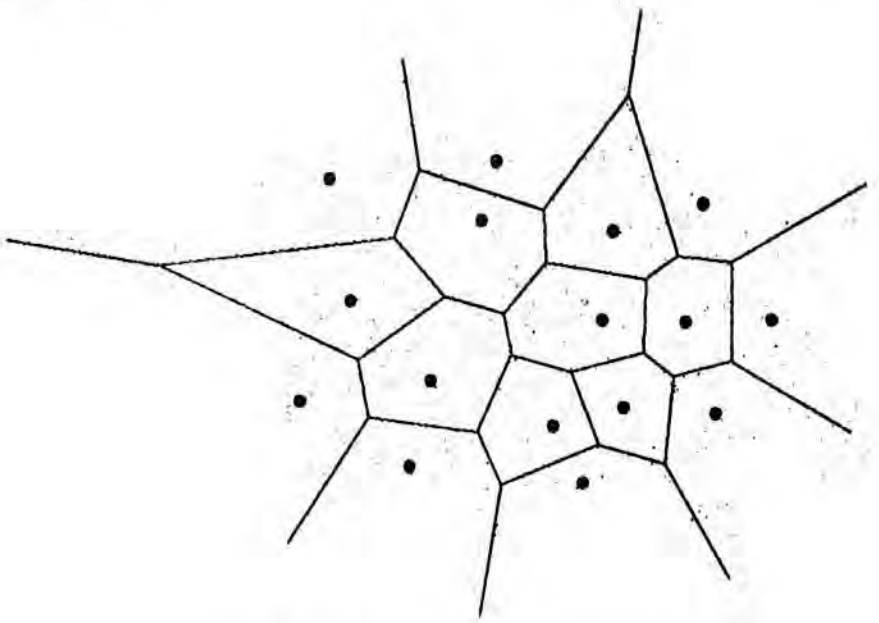


Figure 10: The Thiessen Diagram.

Each of the original N points belongs to a unique Thiessen polygon, thus if $(x,y) \in V(i)$, then p_i is a nearest neighbor of (x, y). The Thiessen diagram contains, in a powerful sense, all of the proximity information defined by the given set.

We now list a number of important properties of the Thiessen diagram. We will assume throughout that no four points of the original set are cocircular. If this is not true, inconsequential but lengthy details must be added to the treatment below. Eventually, we will want to use the Thiessen diagram to solve a number of other construction and search problems. This will only be successful if it can be constructed rapidly. A trivial lower bound on the time necessary to do this is the total number of Thiessen points and edges that are present. At first glance the diagram seems very complicated, but the number of elements it contains turns out to be small, as we now show.

Every edge of the Thiessen diagram is a segment of the perpendicular bisector of a pair of points and is thus common to exactly two polygons. We may form the ***straight-line*** dual of the Thiessen diagram by adding a straight line segment between each pair of points whose Thiessen polygons share an edge. The result is a graph on the original N points. (Figure 11.)

The dual may appear to be unusual at first glance, since an edge and its dual many not even intersect (look at the edges joining consecutive hull vertices.) It importance is largely due to the following theorem of Delaunay, which states that the straight-line dual of the Thiessen diagram is a triangulation. (An account of this result may be found in Rogers, ***Packing and Covering***.) This means that the Thiessen diagram can be used to construct a triangulation, but the theorem has a much more significant consequence: We will prove that the Thiessen diagram on N points has at most 2N-4 vertices and 3N-6 edges. Each edge in the straight-line dual corresponds to a unique Thiessen edge. Being a triangulation, the dual is a planar graph on N points, and thus has at most 3N-6 edges. Therefore, the number of Thiessen edges is at most 3N-6. Since it is the dual of a planar graph, which we shall call the ***Delaunay graph***, the Thiessen diagram is itself a planar graph, and can be stored in only linear space. The makes possible

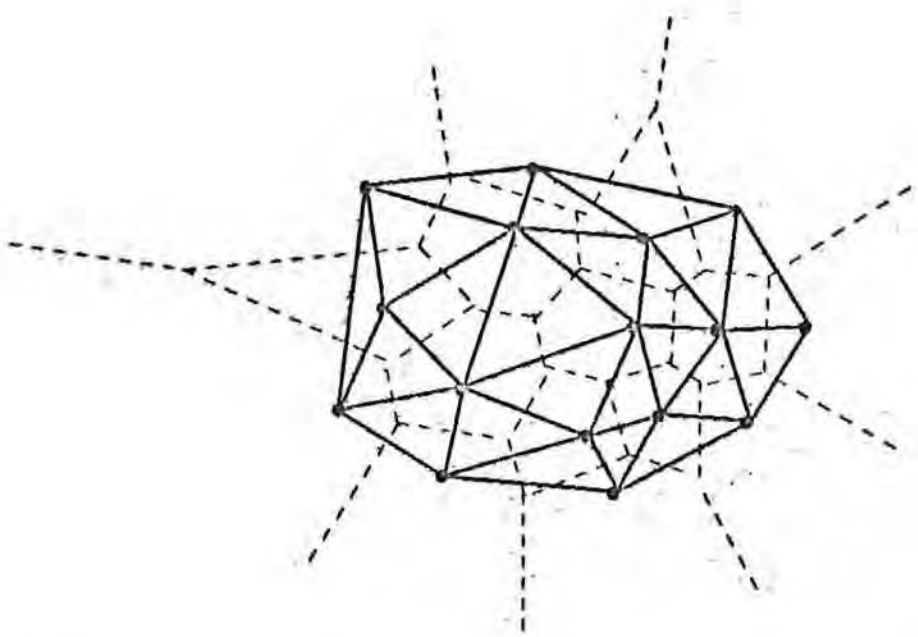


Figure 11: The Straight-Line Dual of the Thiessen Diagram.

an extremely compact representation of the proximity data. While any given Thiessen polygon may have as many as N-1 edges, there are at most 3N-6 edges overall, each of which is shared by exactly two polygons. This means that the ***average*** number of edges in a Thiessen polygon does not exceed six.

The Thiessen points are vertices shared by three Thiessen polygons, and hence are equidistant from three of the original N points. They are thus ***circumcenters of the Delaunay triangles***, which explains why the Thiessen diagram is regular of degree three. (In graph-theoretical parlance, a graph

is regular if all its vertices have the same degree.) A crucial property for purposes of numerical interpolation is that ***the circumcircle of a Delaunay triangle contains no other points of the set.*** (For proof, see Shamos [1977].) This is illustrated in Figure 12.

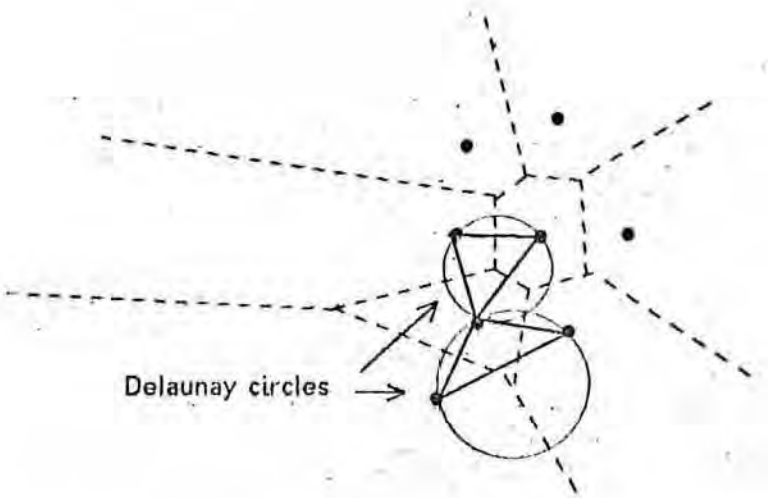


Figure 12: The Circumcircle of a Delaunay Triangle is Empty.

This means that each point interior to the convex hull lies in a triangle composed of the three nearest points that surround it (as distinguished, of course, from its three nearest neighbors, which may or may not enclose it).

Notice that certain of the polygons are unbounded. These correspond to

points on the convex hull of S. The semi-infinite rays of the diagram are defined by pairs of adjacent hull vertices.

Even though we will be using it for other purposes, it is well to note that construction of Thiessen diagrams is an end in itself in a number of fields. In ecology, the survival of an organism depends on the number of neighbors it must compete with for food and light, and the Thiessen diagram of forest species and territorial animals is used to investigate the effect of overcrowding. The structure of a molecule is determined by the combined influence of electrical and short-range forces, which have been probed by constructing elaborate Thiessen diagrams.

Since each Thiessen polygon is an intersection of N-1 half-planes, it can be constructed in O(N log N) time by an algorithm due to Shamos [1977]. (This is optimal for producing any *single* polygon.) There are N polygons to be formed, so the entire construction can be accomplished in $O(N^2 \log N)$ time. While we will be able to improve this result enormously in a moment, let us first obtain a lower bound in order to have a clearer view of the eventual goal. We have already shown that the element uniqueness problem can be solved if we find the two closest points of a set, and that both of these problems require O(N lg N) operations. It is a simple matter to show that that the Thiessen diagram can be used to find the two closest points in O(N) time (once the diagram has been constructed). This pair is joined by an edge of the dual and the dual contains at most 3N-6 edges. We need only examine each dual edge once, looking for the shortest. This implies that constructing a Thiessen diagram on N points in the plane must take O(N lg N) operations, in the worst case, or we would be able to find the closest pair of points faster. In the next section we show that this lower bound can be achieved, which means that constructing the entire diagram is no more difficult than finding a single one of its polygons!

1.4.1 A Fast Algorithm for Thiessen Diagrams

Even though the Thiessen diagram appears to be a complex object, it is eminently suited to attack by divide-and-conquer. The method we employ depends for its success on various structural properties of the diagram that enables us to merge subproblems in linear time.

By "finding" the Thiessen diagram of a set of points we shall mean obtaining all of the following data:

1. The coordinates of the Thiessen points.

2. The Thiessen edges (pairs of Thiessen points) incident with each Thiessen point.

3. The two original points that determine each Thiessen edge.

4. A list of the edges of each polygon in cyclic order.

Let us suppose that we have divided a set S, containing N points, into two subsets, L and R, by a vertical median line M. This means that every point in L lies to the left of every point in R, and every point of R lies to the right of every point in L. (Unless, of course, two or more points lie on the median line, in which case we assign them all to set L arbitrarily.) Let us now find the Thiessen diagrams V(L) and V(R) of each subset recursively. If these can be merged somehow in linear time to form the Thiessen diagram V(S) of the entire set, we will have an O(N log N) algorithm. But what reason is there to believe that V(L) and V(R) bear any relation to V(S)?

Consider the locus P of points that are equidistant from a point of L and a

point of R. This is just the set of edges of V(S) that are shared between polygons V(i) and V(j), with $p_i \in L$ and $p_j \in R$. Thus, P is a polygonal line. (It is a union of segments and is obviously connected.) This line P has the property that any point to the left of P is closest to some point of L and any point to the right is closest to some point of R. (Figure 13.)

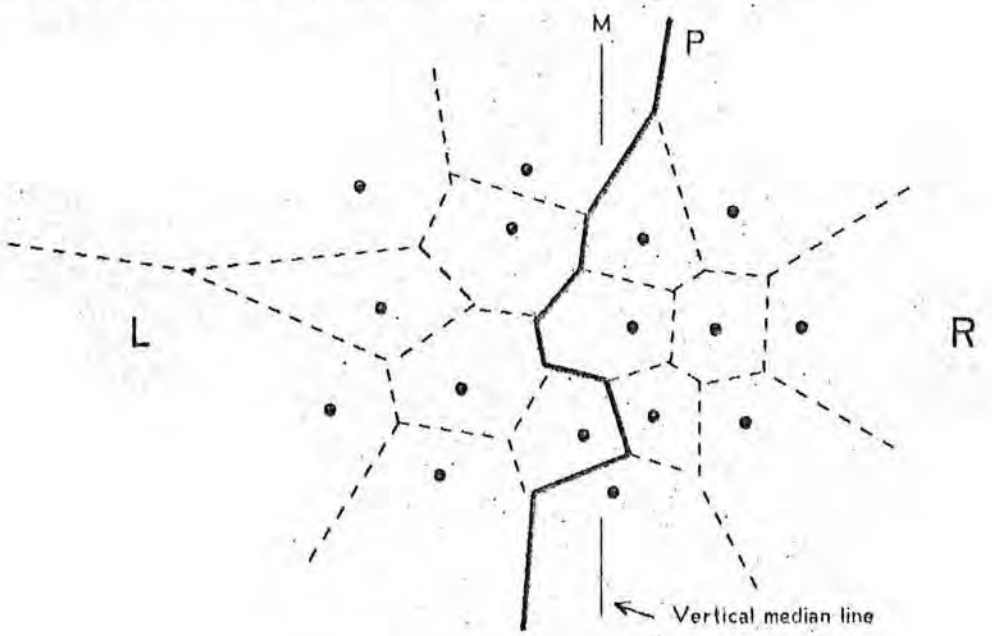


Figure 13: The Locus of Points Equidistant from L and R.

V(L) and V(R) are shown separately in Figures 14 and 15. V(L), V(R) and P are superimposed in Figure 16.

Let z be any point to the left of P, so it is closest to some point of L.

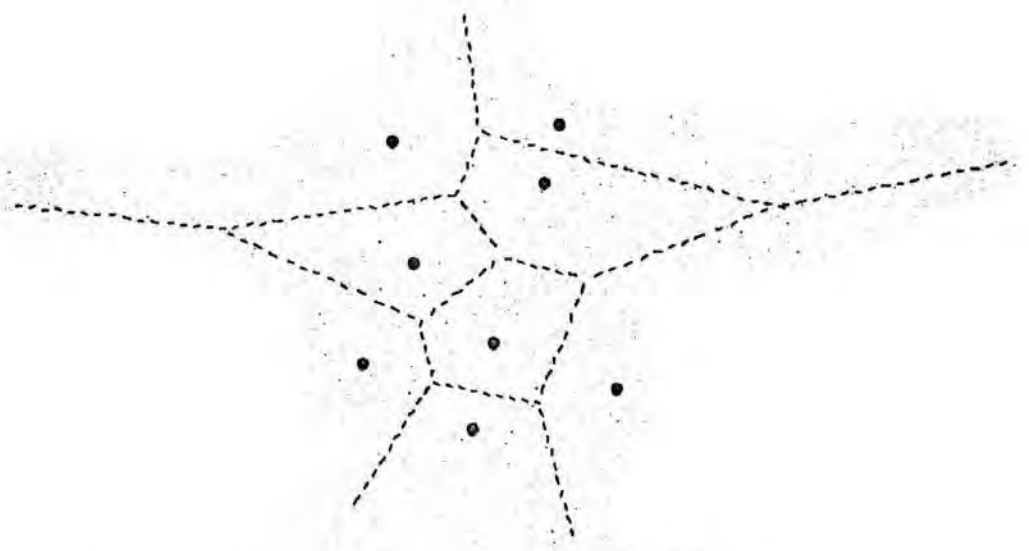


Figure 14: The Thiessen Diagram of the Left Set.

Those segments of V(R) that lie to the *left* of P play no role in discriminating proximity between points of L since they pertain only to R. This means that the portion of V(S) that lies to the left of P is precisely the subset of V(L) that lies to the left of P. Similarly, V(S) to the right of P is the same as V(R) to the right of P.

Here is a sketch of the emerging algorithm:

1. Divide S into two subsets L and R by median x-coordinate. This can be done in linear time.

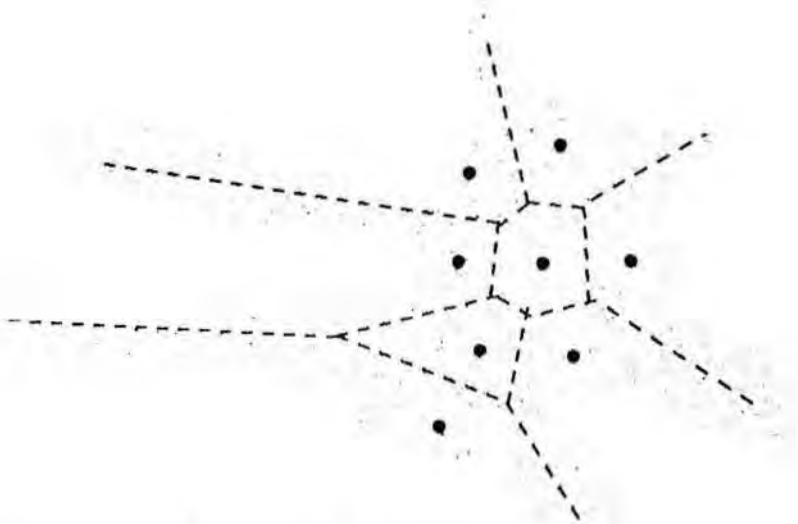


Figure 15: Thiessen Diagram of the Right Set.

2. Find V(L) and V(R) recursively. Time: 2T(N/2).

3. Construct P, the locus equidistant from a point in L and a point in R.

4. Discard all segments of V(R) that lie to the left of P, and all segments of V(L) that lie to the right of P. The resulting is V(S), the Thiessen diagram of the entire set.

The success of this procedure depends on how rapidly we are able to find the dividing line P.

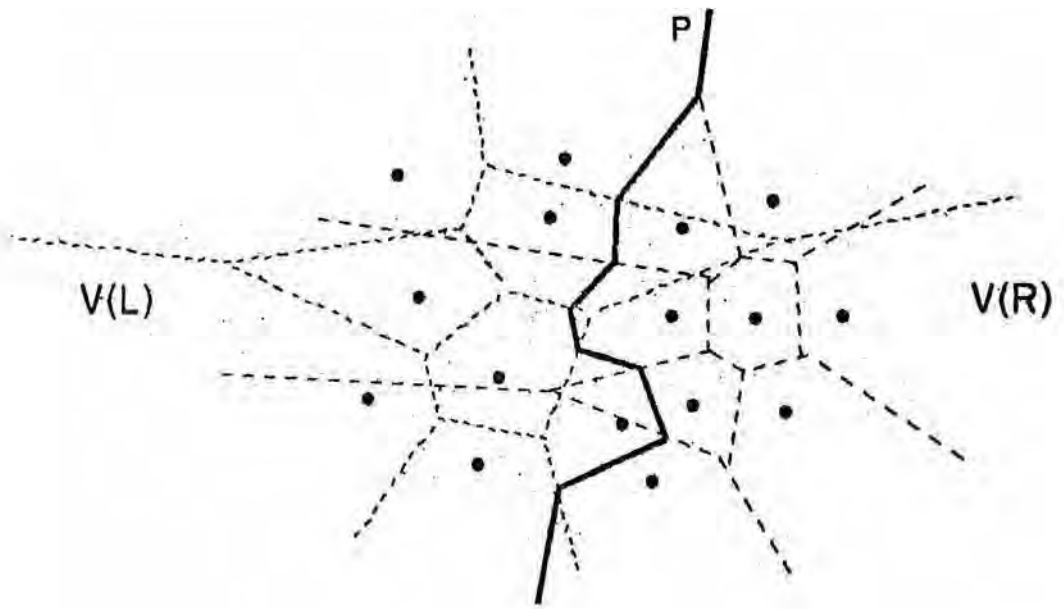


Figure 16: V(L), V(R) and P superimposed.

In Shamos [1977] a linear-time algorithm is given for producing P. This means that T(N), the time needed to produce the Thiessen diagram, is given by $T(N) = 2T(N/2) + O(N) = O(N \lg N)$.

In nearest-neighbor searching, we are given a set of points, we wish to preprocess them so that given a new point *z*, its nearest neighbor can be

found quickly. However, finding the nearest neighbor of z is equivalent to finding the Thiessen polygon in which it lies. The preprocessing just consists of creating the Thiessen diagram ! The diagram is a planar graph whose edges are straight-line segments. Two different methods for searching such graphs are given in Shamos [1977], and they imply the following results:

1. Nearest-neighbor search can be performed in O(log N) time, using $O(N^2)$ storage and $O(N^2)$ preprocessing time.

2. Nearest-neighbor search can be performed in $O(\log^2 N)$ time, using O(N) storage and O(N log N) preprocessing time.

(The second search algorithm is due to Lee and Preparata [1976].)

Thus we see that the Thiessen diagram is a useful object indeed.

1.5. Generalization of the Thiessen Diagram

The Thiessen diagram, while very powerful, has no means of dealing with farthest points, k-closest points and other distance relationships. The difficulty is that we have been working with the Thiessen polygon associated with a single point but such a restriction is not necessary and it will be useful to speak of the *generalized Thiessen polygon* $V^*(T)$ of a subset T of points, defined by

$$V^*(T) = \{x: \forall_{y \in T} \forall_{z \in S-T} \; d(x,y) < d(x,z)\} \qquad (1.2)$$

That is, $V^*(T)$ is the locus of points p such that all points of T are nearer to p than is any point not in T. An equivalent definition is

$$V^*(T) = \cap H(i,j) , \ i \in T, \ j \in S-T \ , \qquad (1.3)$$

where H(i,j) is the half plane containing i that is defined by the perpendicular bisector of i and j. This shows that a generalized Thiessen polygon is still convex. It may, of course, happen that $V^*(T)$ is empty. In Figure 17, for example, there is no point with the property that its two nearest neighbors are 5 and 7. Even though a set S of N points has 2^N subsets, Shamos and Hoey [1975] have shown that no more than $O(N^3)$ of these possess non-empty Thiessen polygons.

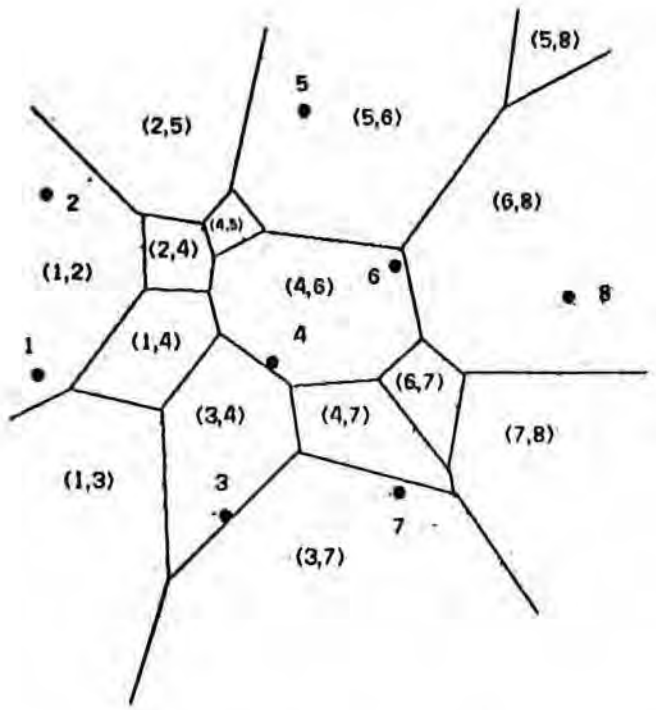
Let us define the *Thiessen diagram of order k*, denoted $V_k(S)$ as the collection of all generalized Thiessen polygons of k-subsets of S, so

$$V_k(S) = \cup V^*(T) \ , \ T \subset S, |T| = k \ ; \qquad (1.4)$$

In this notation, the ordinary Thiessen diagram is just $V_1(S)$. It is proper to speak of $V_k(S)$ as a "diagram" because its polygons partition the plane. Given $V_k(S)$, the k points closest to a new given point z can be determined by finding the polygon in which z lies. Figure 17 shows a Thiessen diagram of order two, the set of loci of nearest pairs of points.

In order to obtain bounds on the time and space required to perform k-nearest-neighbor searching, we must compute the number of edges in the order k diagram. The number of regions in $V_k(S)$ is O(k(N-k)). This result appears in Shamos and Hoey [1975]. A complete proof may be found in Lee [1976]. Because each vertex is of degree three, this means that the number of Thiessen edges is also O(k(N-k)). The union of the Thiessen polygons of all orders is precisely the set of perpendicular bisectors of pairs of points of S.

By starting with the order one diagram and successively updating it through orders 2,3,...,k, Lee has been able to prove that the order k



Some Voronoi polygons of order two are empty. For example, there is no (5,7) polygon.

For a set of N points there are N(N-1)/2 possible polygons. Here, N=8 but only 15 out of the 28 polygons are non-empty.

Figure 17: A Thiessen Diagram of Order Two.

Thiessen diagram on a set of N points can be obtained in $O(k^2N \log N)$ time, using $O(k^2(N-k)$ storage. The next two results follow from our earlier results on planar graph searching:

1. The k nearest out of N neighbors of a point can be found in $O(\max(k,\log kN)$ search time and $O(k^2(N-k)^2)$ storage, after $O(k^2(N-k)^2)$ preprocessing. (Note that the search always requires at least O(k) time since k objects are being retrieved.)

2. The k nearest out of N neighbors of a point can be found in $O(\max(k,(\log k(N-k))^2))$ time and $O(k^2(N-k))$ storage, after $O(k^2N \log N)$ preprocessing.

The generalized Thiessen diagram unifies closest- and farthest-point problems since the locus of points whose k nearest neighbors are the set T is also the locus of points whose N-k ***farthest*** neighbors are the set S - T. Thus, the order k closest-point diagram is exactly the order N-k farthest-point diagram. Let us examine one of these more closely, the order N-1 closest-point diagram, or the order 1 farthest-point diagram (Figure 18.)

Associated with each point p_i is a convex polygonal region $V_{N-1}(i)$ such that p_i is the farthest neighbor of every point in the region. This diagram is determined only by points on the convex hull, and there are no bounded regions. The farthest-point diagram can be constructed in O(N log N) time by a procedure analogous to the algorithm for the closest-point diagram: Having found the farthest-point diagrams of the left and right halves of the set, the polygonal dividing line is exactly the same as in the closest-point case. This time, however, we discard all segments of $V_{N-1}(L)$ that lie to the ***left*** of P, and also remove those segments of $V_{N-1}(R)$ that lie to the right. It is difficult not to marvel at the power of divide-and-conquer.

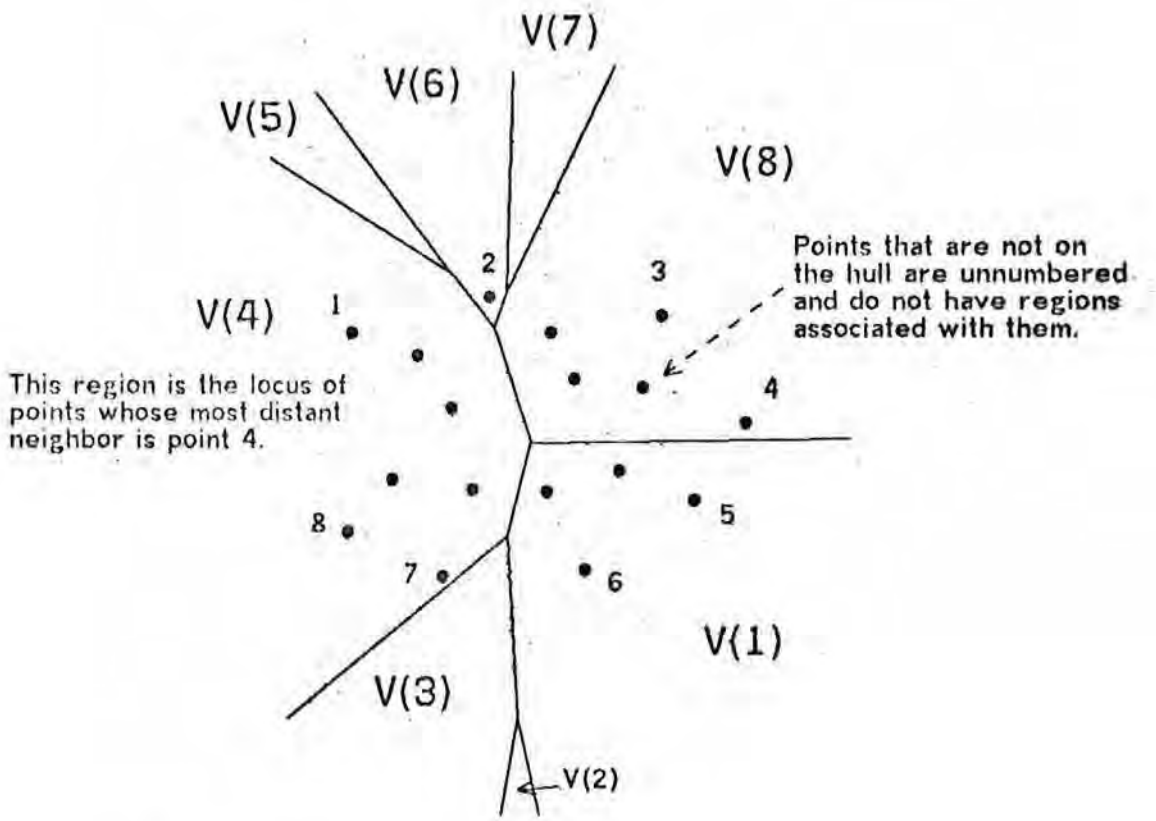


Figure 18: The Farthest-Point Thiessen Diagram.

1.6. Rectangle Problems

The problems that we have discussed so far have all dealt with points, and thus have the most simple geometric structure possible. In this section we deal with rectangles whose sides we assume to be parallel to the coordinate axes. There are many applications which immediately involve rectangles, such as problems dealing with map sectors and other rectangular geographic regions. In addition to these applications, it is fairly easy to approximate other geometric shapes by rectangles. In this section we will deal with two classes of rectangle problems: problems in which we are given a set of rectangles and asked to determine some property of the set such as its area, and problems involving searching.

1.6.1 Single-shot problems

Given N rectangles in the plane, what is the area of their union? This problem is an example of a task that humans can solve very easily (due to the power of visual perception), so it is difficult to see how a computer would have any difficulty. One way to attack this problem by machine is to make use of the principle of inclusion and exclusion, which, in this context, states that the area of the union is the sum of the areas of the individual rectangles ***minus*** the area covered by at least *two* rectangles ***plus*** the area covered by at least three rectangles, etc. The complete formula unfortunately involves 2^N terms, one for each possible subset of rectangles. A different, more successful, strategy is to view the set of overlapping rectangles as a planar graph; this leads to an $O(N^2)$ solution. In this section we will describe an O(N lg N) algorithm due to Bentley and Shamos [1977b].

The best way to visualize the working of our algorithm is to imagine a vertical line being swept from the leftmost rectangle through the set to the rightmost rectangle. At every instant during the sweep we keep track of the total length of line that is covered. If C inches of the line are covered for a distance of D inches during the sweep, an area of CD square inches has been passed over. By repeatedly finding the length that is covered we will be able to "integrate" in this manner to find the area of the union. Suppose we are given N rectangles, whose sides are specified by L_i, R_i, B_i, and T_i (for the coordinates of the Left, Right, Bottom, and Top). We will first give the algorithm formally, then describe it more casually below.

Procedure AreaOfUnion:

1. Sort the set $\{L_i,R_i , 1 \leq i \leq N\}$ and call it VerticalLines.

2. Sort the set $\{B_i,T_i , 1 \leq i \leq N\}$ and then build this set into Tree (a special data structure which we describe below).

3. Go through the set VerticalLines in increasing X-order. Take the last value of LengthCovered, multiply it by the distance between this vertical line and the last, and add that product to TotalArea. (LengthCovered tells how much of a vertical line passing through the set is currently covered by rectangles; we update it now.) If this element of VerticalLines is L_i (for some i), then we are currently entering a new rectangle, so insert L_i into the tree. Otherwise (the element is R_i and we are leaving a rectangle we previously visited), delete R_1 from the tree. Modify the Tree and update LengthCovered accordingly.

To explain this algorithm we show its progress through a set of rectangles in Figure 19. The vertical lines in the figure represent the set VerticalLines; below each we show the values of both LengthCovered and of TotalArea.

The only part of the algorithm remaining to be specified is the implementation of Tree. We must be able to insert segments into it and delete segments from it, being able to tell at any time the total length covered by the segments currently stored. To accomplish this we use a perfectly balanced binary tree in which each leaf node represents one of the horizontal slabs defined by the T_i and B_i. An internal node of the tree represents a contiguous set of such horizontal slabs. Each node contains

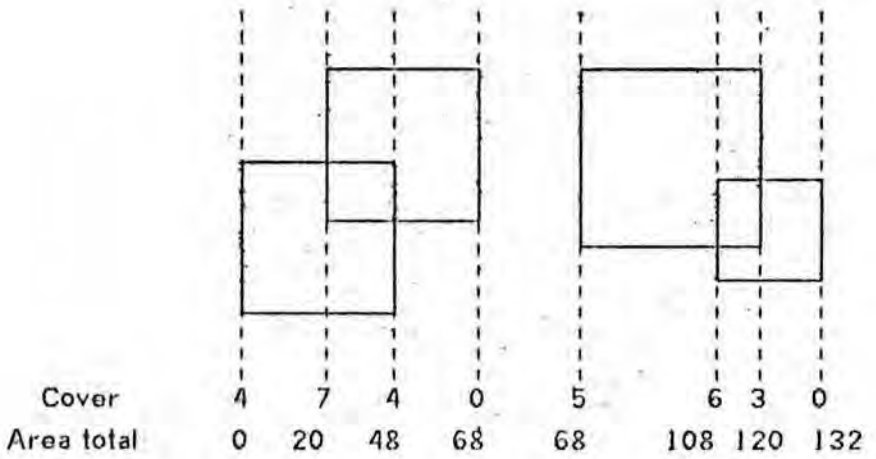


Figure 19: Computing the Area of Overlapping Rectangles.

three fields (aside from tree maintenance items): the number of segments wholly containing this slab, the total length of the slab, and the length of the slab currently occupied by rectangles. Using this structure it is possible to give a recursive algorithm which will accomplish each insertion or deletion in time O(lg N), worst-case.

Since each of the 2N insertions and deletions requires only O(lg N) time, the total running time of AreaOfUnion will be O(N lg N). We will see shortly that this is optimal. The algorithm uses two common techniques: iteration (in the left-to-right scan) and the perfectly balanced binary tree. It is interesting to note that although the union of the N rectangles could consist of $O(N^2)$ separate rectangles (consider N/2 thin vertical rectangles each intersecting N/2 narrow horizontal rectangles), the area of their union can be found in less than $O(N^2)$ time.

We turn our attention now to a different problem with rectangles: given N rectangles in the plane, do any two intersect? One way to determine this would be to compute the area of each and sum them, comparing the result with the area of the union. If the sums are equal, then none of the rectangles intersect (except possibly on a set of measure zero). Both problems require at least O(N lg N) time (Shamos and Hoey [1976]). In this section we will investigate a divide-and-conquer algorithm for the overlap problem, because it will suggest a data structure for a related search problem.

Let S be the set of N rectangles which we want to check for overlap. Draw a vertical line L such that exactly half of the rectangles are entirely to the right of L. Call the area to the left of the line A, and let B be the area to the right of L. The situation which we have described is illustrated in Figure 20.

If an intersection occurs in S, it will occur either between two rectangles in A, two in B, or one in A and one in B. We can check whether rectangles intersect in A and B recursively (these are the subproblems in divide-and-conquer). The marriage step is to see if any rectangle in A intersects any in B. We use a "skyline" to accomplish this. The skylines of A and B are shown in Figure 21 by heavy lines.

Given two skylines (stored in either an array or a linked list) we can see if they intersect in linear time. Also, given the (left and right) skylines of A and B, we can construct the (left and right) skylines of S in linear time. Thus the marriage step takes linear time, and the resulting divide and conquer algorithm checks for overlap in a total of O(N lg N) time and O(N) storage.

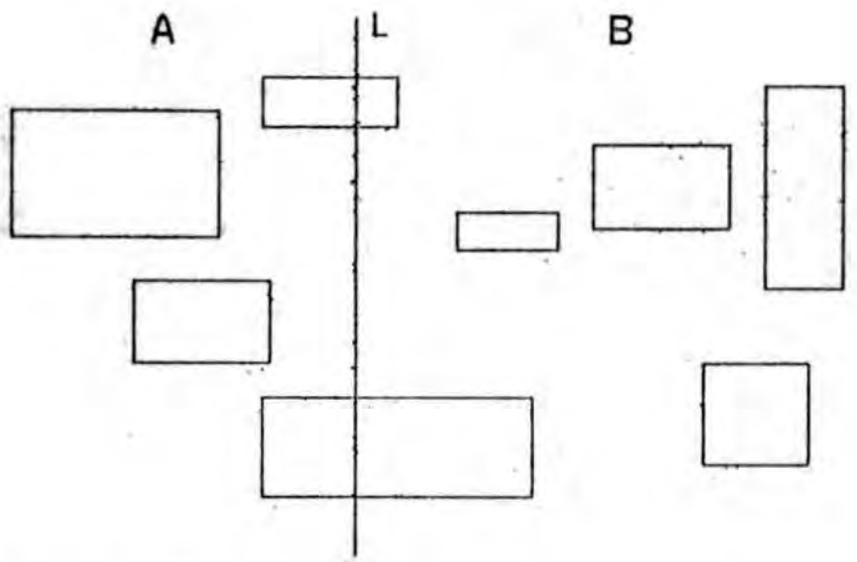


Figure 20: Determining Whether Rectangles Overlap.

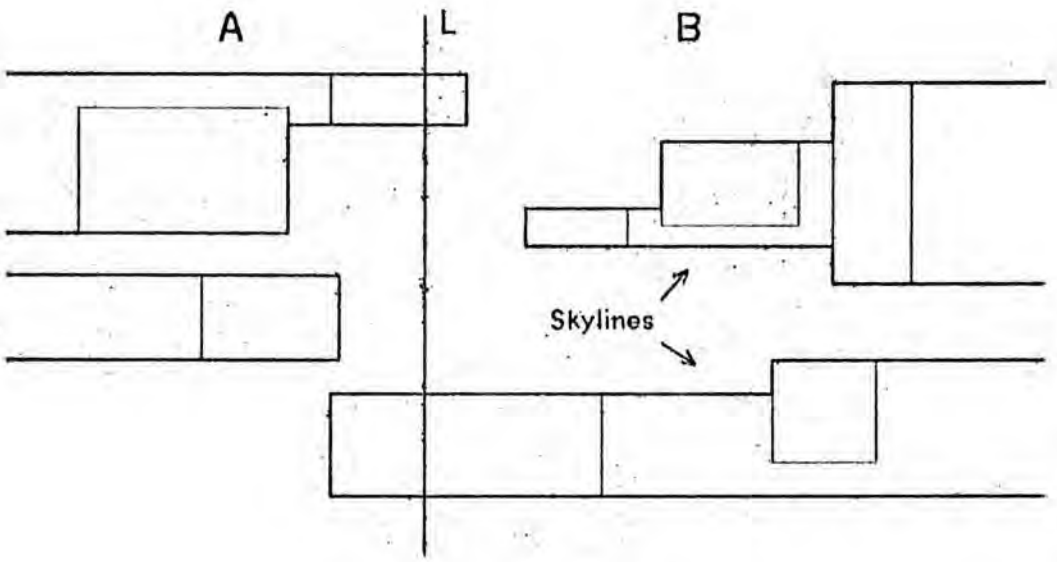


Figure 21: Skylines.

1.6.2 Searching Problems

In this section we will investigate a number of search problems related to rectangles. Our task is to preprocess a set S of N rectangles into an efficient data structure for searching. The first type of query we will investigate assumes that S consists of non-overlapping rectangles and asks in which (if any) of the rectangles a new point lies. To do this we will use a data structure suggested by an earlier algorithm. Refer again to Figure 20; if a point P is to the left of L, then it can only intersect one of the rectangles in A. If P is to the right of L, then it can intersect only a

rectangle in B or one of the rectangles in A which overlaps L. We may store the rectangles which overlap L as a sorted array, and then determine if P (on the right of L) intersects any rectangle in B by a single binary search. This approach (which was certainly suggested by the last algorithm) is very similar to the ECDF tree presented earlier. The analysis is identical, and shows that the structure requires O(N lg N) time to build, O($(\lg N)^2$) time to search, but only O(N) storage, which is achieved by storing each rectangle only once.

A more difficult searching problem asks us to store the N rectangles (which may now overlap) in such a way that we can quickly determine if a new rectangle intersects any of the others. We solve this in a way that is analogous to the algorithm for the point problem, but which uses a more complicated data structure. In the tree, instead of storing the rectangles which intersect L, we must store the left skyline of A and the right skyline of B so to allow us to quickly determine if a new rectangle overlaps either skyline. We do this by using a perfectly balanced tree, with information in the nodes very similar to the Tree in the AreaOfUnion algorithm. Such an approach costs O(N lg N) preprocessing time and storage, but permits searching in only O($(\lg N)^2$) time.

The problems that we have described arise in many applications, but we will focus on one particular application to see where our algorithms might be employed. Consider a computer program designed to place machines on the floor of a shop (those familiar with computer rooms can easily envision the machines as a set of rectangles!). To insure that we have not assigned two machines to the same location we can check to see whether any of the rectangles intersect. If we already have fixed positions for N of the machines, then we can store them in a data structure and decide very quickly if the potential placement of a new machine conflicts with any of the old. If we want to know which machine (if any) occupies a given spot, we can simply perform a point-in rectangle search.

A number of other problems with rectangles have been solved. For instance, given N points and N disjoint rectangles in the plane, we can tell in which rectangle every point lies in O(N lg N) time. Many of the algorithms which we have described are easily generalized to points in 3-space (or higher dimensional spaces).

1.7. Applications of Computational Geometry

Computational Geometry provides the theoretical and practical tools of automated cartography, computer graphics, image processing, and many other fields in which the data is inherently geometric. As such, it is a comprehensive discipline that is able to deal coherently with the automation of many computational problems. A good example is statistical geography. We begin with geometric data, are compelled to develop efficient algorithms for handling and summarizing it, and are left with a problem in computational statistics, to which the methods of this paper also apply! Suppose we wish to study the randomness of spatial point patterns. According to the criterion of Hopkins [1954], a pattern is random if the distribution of distances from a random point (x,y) to the nearest pattern point p_i] is identical to the distribution of nearest-neighbor distances among the p_i themselves. This can be determined by using the Thiessen diagram for finding the interpoint distances and for finding the nearest neighbor of a given point, then by using a fast statistical algorithm to analyze the results. Thus the techniques we propose are quite powerful and wide in scope. Other examples of the interplay between geometric and statistical problems occur in numerical taxonomy, anthropology and archaeology (see, for example, Hodder [1976]). Thiessen diagrams have been used to study the growth of technology in ancient cultures, the rise of trade centers, and the spread of disease, all of which are both statistical and geometric in nature.

1.8. Conclusions

While a large number of algorithms have been developed here, the reader should be able to see that very few techniques were involved. We have used divide-and-conquer, combined with various geometric ideas, to produce fast algorithms for diverse problems. It is often a useful exercise for the algorithm designer to *force* himself to use divide-and-conquer, even if there seems to be no obvious way of applying it. It is normally the merge step that causes difficulty, but great effort must be expended to make it run as quickly as possible, for only then will a truly efficient procedure result.

We stress again that it is essential to isolate computational problems that are truly fundamental and are common to many applications, and to concentrate on them, proving lower bounds and developing fast algorithms. Only then are they ready to be added to the designer's tool box.

References

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

Bentley, J. L. [1975]. "Multidimensional binary search trees used for associative searching", Communications of the ACM, vol. 18, no. 9, September 1975, pp. 509-517.

Bentley, J. L. [1976]. Divide and conquer algorithms for closest point problems in multidimensional space, unpublished Ph. D. Dissertation, University of North Carolina, Chapel Hill, North Carolina.

Bentley, J. L. and J. H. Friedman [1975]. Fast algorithms for constructing minimal spanning trees in coordinate spaces, Stanford Computer Science Department Report STAN-CS-75-529, January 1976, 29 pp. (To appear in IEEE Transactions on Computers.)

Bentley, J. L. and M. I. Shamos [1977a]. A problem in multivariate statistics: algorithm, data structure, and applications, submitted for publication.

Bentley, J. L. and M. I. Shamos [1977b]. Algorithms for Segments and Rectangles, in preparation.

Bentley, J. L. and D. F. Stanat [1975]. "Analysis of range searches in quad trees", Information Processing Letters, vol. 3, no. 6, July 1975, pp. 170-173.

Friedman, J. H., J. L. Bentley, and R. A. Finkel [1975]. An algorithm for finding best matches in logarithmic time, Stanford Linear Accelerator Center Report SLAC-PUB-1549, February 1975, 20 pp. (To appear in ACM Transactions on Mathematical Software.)

Hodder, I. and C. Orton [1976]. Spatial Analysis in Archaeology. Cambridge University Press. 270 pp.

Hopkins, B. and J. G. Skellam [1954]. A new method for determining the type of distribution of plant individuals. Ann. Bot. Lond. N. S. 18, pp. 213-227.

Knuth, D. E. [1968]. The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass.

Knuth, D. E. [1973]. The Art of Computer Programming, vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass.

Knuth, D. E. [1976]. Big omicron and big omega and big theta, SIGACT News 8,2.

Lee, D. T. [1976]. On finding k nearest neighbors in the plane. Technical Report, Department of Computer Science, University of Illinois.

Lee, D. T. and F. P. Preparata [1976]. Location of a point in a planar subdivision and its applications. Proceedings of the Eighth Annual ACM Symposium on Automata and Theory of Computation, pp. 231-235.

Lee, D. T. and C. K. Wong [1976]. Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees, IBM Watson Research Center preprint, 18 pp. [Shamos, M. I. [1975]. "Geometric complexity", Proceedings of the Seventh Symposium of the Theory of Computing, ACM, May 1975, pp. 224-233.

Shamos, M. I. [1976]. "Geometry and Statistics: Problems at the Interface", in Algorithms and Complexity: New Directions and Recent Results, ed. J. F. Traub, Academic Press, pp. 251-280.

Shamos, M. I. [1977]. Computational Geometry. Springer-Verlag, to appear.

Shamos, M. I. and D. J. Hoey [1975]. "Closest-point problems", Proceedings of the Sixteenth Symposium of Foundations of Computer Science, IEEE, October, 1975.