UNC Report No.
TR-76-103
December 1976

DIVIDE AND CONQUER ALGORITHMS FOR CLOSEST POINT PROBLEMS IN MULTIDIMENSIONAL SPACE

by

Jon Louis Bentley

A dissertation submitted to the faculty of the University of North Carolina in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

December, 1976

Approved by:

[illegible]
Advisor

[illegible]
Reader

Peter Calingaert
Reader

This dissertation is dedicated

to the glory of God.

Acknowledgments

I would like to express sincere thanks to many friends who helped bring this dissertation into being. The technical contributions herein have profitted from fruitful discussions with Ray Finkel, Jerry Friedman, Mike Shamos, Don Stanat, and Andy and Frances Yao. The pain of preparing a dissertation was mitigated to a large degree by my advisor Don Stanat, my committee, and Anne Edwards, typist.

Portions of this research were conducted while I visited the Stanford Linear Accelerator Center and Carnegie-Mellon University. I was supported also by the National Science Foundation both as a Graduate Fellow and under Grant MCS75-19681.

On a personal level thanks are due firstly to my parents. In this endeavor, as always, they gave all the support I needed. It was Don Knuth who encouraged me by his words and his example to attempt to earn a Ph.D. in Computer Science. Pastor Dick Henderson and all the folks at his Friday Night Bible Study were a source of constant encouragement during my time at UNC. Marilyn Roper, who will soon be my wife, gave the abundant emotional and spiritual support that made the writing of this dissertation a joy.

JON LOUIS BENTLEY. Divide and Conquer Algorithms for Closest Point Problems in Multidimensional Space (Under the direction of DONALD F. STANAT.)

The contributions contained in this dissertation can be broadly classified as falling into three areas: multidimensional algorithms, the "divide and conquer" strategy, and principles of algorithm design.

Contributions to multidimensional algorithms are twofold: basic results and basic methods. The results deal with algorithms for determining properties of sets of N points in k dimensional space. Among other results it is shown that the closest pair among the N points can be found in time proportional to N log N and that the nearest neighbor to each point among the N can be found in time proportional to $N(\log N)^{k-1}$ (for $k \geq 2$). The basic methods include an algorithm schema applicable to many multidimensional problems and fundamental concepts for dealing with such problems.

The algorithms in this dissertation demonstrate the power of the divide and conquer strategy. The strategy is shown to be applicable to multidimensional problems. The basic technique is modified in many interesting ways to create faster algorithms.

The final area to which this dissertation contributes is algorithm design. Instead of merely presenting the results herein as finished products, they are arrived at through a detailed development process. This development is one of the few written records of the development of an asymptotically fast algorithm; as such it is a suitable basis for teaching algorithm design. The general principles of algorithm design employed are enumerated.

Table of Contents

1. Introduction

This introduction is designed to serve as a guide in reading this thesis. In this chapter we describe the overall content of the thesis and give a brief summary of each chapter. Such an overview should allow the reader to decide which chapters he should read.

The main topic of this thesis is the application of divide and conquer algorithms to multidimensional closest point problems. For the time being, one can think of a multidimensional closest point problem as one concerned with the proximity of points in a multidimensional vector space. The primary subtheme of this thesis concerns techniques for designing fast computer algorithms.

Chapter 2 is a review of previous work in multidimensional closest point problems and in divide and conquer algorithms. The reader already familiar with these areas ought to read only Section 2.1.1, which describes the notation used in the thesis.

The heart of this thesis is Chapter 3; there we give fast algorithms for the problems we are studying. The algorithms given there are the first algorithms with less than quadratic running times for general multidimensional problems. The techniques for divide and conquer algorithms described in that chapter are applicable to a broad class of problems. Finally, we derive lower bounds on the problems we study.

The presentation of the material of Chapter 3 is unusual. Instead of simply describing the completed work, the chapter reconstructs the actual development of the algorithms. The author believes that Polya's [1945] comments are just as applicable to solving programming problems as they are to solving mathematical problems: "Solving problems is a practical skill like, let us say, swimming. We acquire any practical skill by imitation and practice. Trying to swim, you imitate what other people do with their hands and feet to keep their heads above water, and, finally, you learn to swim by practicing swimming." The author knows of few written examples of the algorithm development process suitable for the imitation to which Polya refers; this presentation of Chapter 3 is the author's attempt to help fill that void. The material presented in that chapter covers a broad range; certainly parts of the chapter will prove too easy or too difficult (or both!) for any given reader. But just as no reader will go away from that chapter totally satisfied, the author believes that consideration of the algorithm development process in that chapter will be beneficial to anyone interested in learning more about how to design algorithms.

In Chapter 4 we extend the algorithms of Chapter 3 to other metrics and other (similar) problems. This chapter is not essential to a basic understanding of the results of this thesis, but extends the applicability of those results at a small cost.

Questions of how the algorithms of Chapter 3 could be efficiently implemented as computer programs are discussed in Chapter 5. Section 5.1 is concerned with implementing the algorithms given in Chapter 3. Section 5.2 discusses the problem of transforming those worst-case algorithms into fast average-case algorithms.

In Chapter 6 we enumerate some of the basic principles of algorithm design we employed in Chapters 3 through 5. Though the principles are grouped into four sections within the chapter, they are not further organized within the sections. Sections 6.1 through 6.3 deal primarily with principles employed in Chapter 3; Section 6.4 is based mostly on Chapter 5. Though it is not absolutely necessary to have read the earlier chapters to have some understanding of the principles given, a familiarity with them will enhance the reader's appreciation of the techniques described.

Areas for further work are described in Chapter 7, and in Chapter 8 we summarize the primary contributions of this thesis.

We will use standard mathematical notation throughout this thesis. The only atypical notation we will employ is the Θ notation due to Knuth [1976], which is similar to the common "big-oh" notation. By Knuth's definition $\Theta(f(n))$ denotes the set of all $g(n)$ such that there exist positive constants C, C', and n_0 with $Cf(n) \le g(n) \le C'f(n)$ for all $n \ge n_0$. The Θ notation provides both lower and upper bounds on the function while the "big-oh" notation gives only an upper bound. Thus $6N^2 = O(N^2)$ and $1.5N = O(N^2)$, while $6N^2 = \Theta(N^2)$ but $1.5N \ne \Theta(N^2)$.

2. Previous Work

In this section we briefly review previous work in two areas: multidimensional closest point problems and the divide and conquer technique. The reader familiar with these subjects need read only Section 2.1.1 which introduces notation for the problems with which we shall deal.

2.1 Multidimensional closest point problems

Much work has been done recently on multidimensional closest point problems. In Section 2.1.1 we define a specific subset of the problems with which we shall be interested. Section 2.1.2 gives examples of potential practical applications of solutions to these problems. We give a brief survey of previous work in Section 2.1.3.

2.1.1 Problem definitions

In order to describe accurately the problems we are to solve in this thesis, we must define the tools with which we will solve them. Throughout this thesis we assume that all problems are to be solved by an algorithm in the form of a computer program running on a Random Access Machine (RAM) or a Random Access Stored Program machine (RASP) as described in Aho, Hopcroft and Ullman [1974]. Since the problems deal essentially with real numbers, we will assume that each word of the RAM/RASP memory can contain one real number (which may be an integer) or instruction. (Thus we assume an "infinite precision" machine.) This abstract machine suitably models most high speed computers which would actually be used to solve the problems we will treat.

The closest point problems we deal with in this thesis all have as input N points in k dimensional space; the input is described by Nk real numbers (we often refer to the collection of N points as "the file"). Thus we will usually measure the complexity or cost of applying the algorithms as a function of N and k.

We will shortly describe a number of problems which we shall denote by names and abbreviations composed of one or more capital letters. For instance, ANN is the abbreviation for the "All nearest neighbors" problem. We will denote the worst case running time of the best possible worst case algorithm to solve problem ANN on a RAM/RASP machine by ANN(N,k); ANN(N,k) is often referred to as the minimax complexity of ANN. We will denote the minimean complexity of ANN by $\overline{\text{ANN}}$(N,k) (the minimean complexity is the average case time of the best possible average case algorithm). We will mention minimean complexities only rarely in this thesis. To speak precisely of mean times, one must have an accurate model of the probability of different inputs, and the well known unidimensional models (such as permutations for sorting) are not obviously extendible to multivariate cases. We will also determine the amount of storage required by the algorithms we discuss, though it will be clear that our algorithms use linear storage.

We now describe the multidimensional closest point problems which will be the center of this thesis. All of them deal with a set F of N points in k dimensional space and assume the existence of a distance function D on the space. (We discuss later what properties this function must have.)

Closest pair (CP)

Given N points in k dimensional space, determine which two points are closest together. That is, find the points W, X $\in$ F such that $D(W,X) = \min_{\substack{Y,Z \in F \\ Y \neq Z}} D(Y,Z)$. Ties may be broken arbitrarily.

All nearest neighbors (ANN)

Determine the nearest neighbor of each point X $\in$ F. The nearest neighbor of X is defined to be that point Y such that

$$D(X,Y) = \min_{\substack{Z \in F \\ Z \neq X}} D(X,Z).$$

Ties may be broken arbitrarily. The solution to this problem is N pairs of points.

Minimal spanning tree (MST)

Define the weighted graph induced by F to be an N vertex, complete undirected graph with a vertex at each point in F; the weight of each edge is the distance between the two corresponding points. The MST problem is to construct the minimal spanning tree of the weighted graph induced by F. The solution to this problem is a set of N-1 edges composing the minimal spanning tree, where each edge is specified by a pair of points. For more detail on this problem, see Bentley and Friedman [1975].

Fixed radius near neighbors (FR)

Given N points and a parameter δ, find all pairs of points in F within δ of each other in the space. The solution is a set of q pairs of points; where $0 \leq q \leq \binom{N}{2}$. The fact that the size of the output can be $\Theta(N^2)$ makes the problem very difficult to solve. We shall therefore restrict our attention to the special case of this problem given as the next problem. The general case is investigated by Bentley, Stanat and Williams [1976].

<u>Sparse fixed radius near neighbors (SFR)</u>

This problem is the same as problem FR given above with the addition of a constraint that the set F is sparse. The exact definition of sparsity that we will use is that there is some constant c such that no δ-ball (sphere of radius δ) in the multidimensional space contains more than c points of F. This guarantees that no point in the set will have more than c fixed radius near neighbors and therefore the output of the problem will be between zero and cN pairs of points.

<u>Nearest neighbor (NN)</u>

Given N points of F and an additional point X, which point of F is nearest to X? Preprocessing of the original N points is permitted. A generalization of this problem is the m nearest neighbor problem, in which the output is the set of the m closest points among F to X. We will use NNP(N,k) to denote the time required by the preprocessing phase of the NN algorithm.

These are the problems with which we shall deal. It is interesting to note that because the closest pair graph is a subgraph of the all nearest neighbors graph, which is in turn a subgraph of the minimal spanning tree, we can immediately deduce that

$$CP(N,k) \le ANN(N,k) \le MST(N,k).$$

2.1.2 Applications

In this section we will examine briefly some of the applications of multidimensional closest point problems. Knowing applications of the problems not only motivates the work, but also enables us to evaluate the reasonableness of our assumptions concerning the nature of the problems.

The closest pair is in a sense a contrived problem. The author knows of no application which actually requires the solution of this problem. Identification of the closest pair could be used as a signature of the size of a set which is invariant under translation. Its relevance to this work, however, lies in the fact that it is in many senses the "simplest" multidimensional problem; a solution to this problem is implicit in many more complex problems. One example of this is Kruskal's minimal spanning tree algorithm [1957]; its first step is to find the closest pair in the space. (An historic note here is perhaps appropriate. Professor M. I. Shamos reports [personal communication] that until late 1974 many people conjectured that $CP(N,2) = \Theta(N^2)$. Since this was a lower bound on most other closest point problems, no one looked for the existence of fast algorithms for the other problems. When it was shown that $CP(N,2) = \Theta(N \log N)$, many other fast algorithms were soon discovered. The author has noted similar occurrences in his work on multidimensional problems.)

We now turn to the all nearest neighbor problem. Given two multidimensional point sets, how can we tell if they were drawn from the same underlying probability function? A nonparametric test to determine this is described by Friedman, Steppel and Tukey [1973]. The basis of this test is the solution to a generalization of the all nearest neighbors problem in which all m nearest neighbors are sought for each point. Zahn [1971] describes how the all nearest neighbors graph can be used in cluster analysis, which is one basis for mathematical taxonomy.

There are many applications in the literature of the minimal spanning tree problem. The classical formulation is in finding the minimal cost communications network for cities on a map (Prim, [1957]). Its application in producing efficient breadboard wirings is described in Loberman and Weinberger [1957]. Zahn [1971] describes some very elegant algorithms which use minimal spanning trees for cluster analysis. An algorithm for mapping points in a high dimensional space into a lower dimensional space while preserving as much locality as possible is described by Lee, Slagle and Blum [1975]; the algorithm is based on the minimal spanning tree of the point set. The minimal spanning tree can be used to find a good approximate solution to the travelling salesman problem; this is discussed in Rosenkrantz, Stearns and Lewis [1974]. This is especially important in view of the proof by Garey, Graham and Johnson [1976] that the travelling salesman problem in the plane is NP-hard (which means that it is as difficult as the NP-complete problems, which many suspect are of exponential complexity); we are thus forced to use heuristic solutions for problems of even moderate size.

The fixed radius near neighbor problem arises whenever an agent has the capability of acting on another agent within a given distance. This arises in molecular graphics (Levinthal, [1966]) and gestalt clustering (Zahn, [1971]). We will see in this thesis that the fixed radius near neighbor problem arises naturally in the solution of other closest point problems. Sparsity is sometimes guaranteed by nature; we will later see how we can induce it.

Applications of nearest neighbor searching in document retrieval are described by Van Rijsbergen [1974]. It can be used in many pattern classification problems (Cover and Hart, [1967]), including speech recognition (Smith [1975]). It can also be used in estimating values of a probability distribution function from a sample of points drawn from the distribution (Loftsgaarden and Quesenberry, [1965]).

2.1.3 Previous approaches

This section is an overview of previous work on computer algorithms for multidimensional closest point problems. The descriptions of individual methods are quite brief; the interested reader is referred to either the original works or to Bentley's survey article [1975a] which examines these methods at a somewhat more detailed level.

The simplest procedures for solving these problems make no use of the geometric nature of the problem. For problems involving pairs of points, all $\binom{N}{2}$ pairs are examined. This approach leads to algorithms which are quadratic in N for CP, ANN, MST, FR and SFR and a linear algorithm for NN. Dijkstra [1959] describes such a "brute force" solution to MST.

The simplest method employing the geometry of the space is to project all the points onto a line and use linear sorting and searching algorithms to work with the projected set. This technique is referred to by Knuth [1973] as "inverted lists". A natural choice of the line used for projection is one of the coordinate axes. This technique was employed by Friedman, Baskett and Shustek [1975] to show that $\overline{\text{ANN}}(N,k) \leq \Theta(N^{2-1/k})$ for points drawn from a multivariate normal distribution. Lee, Chin and Chang [1975] discuss the idea of projecting onto lines other than coordinate axes. The projection technique can be used for many closest point problems. It seems, however, to yield algorithms which are asymptotically slow as well as being difficult to analyze.

A common approach for dealing with multidimensional closest point problems is to divide k-space into equally sized cells and store the points in the file in the corresponding cells. This usually involves representing the points in the cell as a set (typically by a linked list) and the cells by a multidimensional array. This idea was first described in the literature by Levinthal [1966]. Knuth [1973] suggests the idea of recursively subdividing a cell that is too crowded with points. Yuval pointed out [1975] that the cells need not be represented by a multidimensional array; a hash table or binary tree is more appropriate for sparse spaces. These cell approaches are described in Bentley's survey [1975a]. They are quite well suited to the fixed radius problem, but are inappropriate in other applications of closest point problems because they are not "locally adaptable". The fixed radius problem does arise in other closest point problems; both Yuval [1976] and Rabin [1976] have used cell techniques to solve the closest point problem. The cell technique is analyzed in detail by Bentley, Stanat and Williams [1976].

Much work has recently been done on closest point problems in the plane. Most of that work is due to Shamos [1975a,b] (see also Shamos and Hoey [1975]). He has shown that

$$CP(N,2) = ANN(N,2) = SFR(N,2) = MST(N,2) = \Theta(N \log N).$$

He has also shown that $NN(N,2) = \Theta(\log N)$ with $NNP(N,2) = \Theta(N^2)$ and quadratic space requirements and $NN(N,2) = \Theta(\log^2 N)$ with $NNP(N,2) = \Theta(N \log N)$ and linear space. In [1975b] he describes an algorithm due to Strong for finding closest pairs in the plane which inspired the algorithms in this thesis. Unfortunately, very few of Shamos's algorithms seem to be easily generalizable to k-space.

Tree structures have long been known to facilitate many unidimensional sorting and searching problems (see Knuth [1973]). The first tree structured approach to multidimensional problems is the quad tree of Finkel and Bentley [1974]. Its efficiency in solving a problem similar to fixed radius near neighbor searching is analyzed in the average case by Bentley and Stanat [1975] and in the worst case by Lee and Wong [1976]. The quad tree employs 2^k-way branching in k dimensional space, which is a serious drawback for most applications. A tree structured approach which overcomes this difficulty by employing binary trees is described by Bentley [1975b]. It employs a data structure called the "k-d tree". The worst case performance of the k-d tree seems very difficult to analyze; the only bounds attained so far (Lee and Wong [1976]) do not appear to be tight. There are strong heuristic arguments given by Friedman, Bentley and Finkel [1975] to indicate that use of the k-d tree gives

$$\overline{CP}(N,k) = \overline{ANN}(N,k) = \Theta(N \log N),$$

though these are not rigid arguments. The same paper suggests that $\overline{NN}(N,k) = \Theta(\log N + 2^k)$ where $NNP(N,k) = \Theta(kN \log N)$ and linear space is required. Bentley and Friedman [1975] discuss the application of k-d trees to the MST problem. They give heuristic arguments indicating that for many probability distributions $\overline{\mathrm{MST}}(N,k) = \Theta(N \log N)$ for fixed k. The basic idea underlying the k-d tree is very similar to the idea underlying the algorithms described in this thesis. In a certain sense one of the major benefits of the work described here is to give a theoretical explanation of the empirically observed good performance of k-d trees.

A problem area distinct from but conceptually related to the area studied in this thesis is retrieval from a file with multi-attribute records which assume discrete values. Such a problem is often characterized by a large number (say, over 30) of binary keys in each record. Rivest examines this area in [1974a]; further approaches are described by Burkhard [1976] and Bentley and Burkhard [1976]. Though these fields are relatively distinct, insights gained in one often yield application in the other.

A number of different approaches to closest point problems have been published recently. An algorithm by Elias for finding nearest neighbors is studied by Burkhard and Keller [1973] and Rivest [1974b]. Tree structured approaches based on clustering are described by Fukunaga and Narendra [1975], McNutt [1973] and Smith [1975]. A structure used by Shamos in the plane is generalized to k-space by Dobkin and Lipton [1976]. Those algorithms use a prohibitive ($\Theta(N^k)$) amount of storage for practical applications; in addition it is not clear that all the required preprocessing algorithms are known. Karp gives approximate algorithms for the travelling salesman problem in the plane in [1976].

A summary of the results mentioned in this section is given in Table 2.1.

RESULT	STORAGE	REFERENCE	COMMENTS
$CP(N,2) = \Theta(N \log N)$	$\Theta(N)$	Shamos [1975b]	
$\overline{CP}(N,k) \le \Theta(N(\log N + 2^k))$	$\Theta(Nk)$	Friedman, Bentley, Finkel [1975]	Heuristic argument
$CP(N,k) = \Theta(f(k)N \log N)$	$\Theta(N)$	Yuval [1975]	
$\overline{CP}(N,k) = \Theta(f(k)N)$	$\Theta(N)$	Rabin [1976]	
$ANN(N,2) = \Theta(N \log N)$	$\Theta(N)$	Shamos [1975b]	
$\overline{ANN}(N,k) = \Theta(N^{2-(1/k)})$	$\Theta(Nk)$	Friedman, Baskett, Shustek [1975]	Normal distribution
$\overline{ANN}(N,k) = \Theta(N(\log N + 2^k))$	$\Theta(N)$	Friedman, Bentley, Finkel [1975]	Heuristic argument
$MST(N,2) = \Theta(N \log N)$	$\Theta(N)$	Shamos [1975b]	
$\overline{MST}(N,k) = \Theta(f(k)N \log N)$	$\Theta(N)$	Bentley, Friedman [1975]	Heuristic argument
$NN(N,2) \le \Theta(\log^2 N)$	$\Theta(N)$	Shamos [1975b]	$NNP(N,2) = \Theta(N \log N)$
$NN(N,2) = \Theta(\log N)$	$\Theta(N^2)$	Shamos [1975b]	$NNP(N,2) = \Theta(N^2)$
$NN(N,k) \le \Theta(k \log N)$	$\Theta(N^k)$	Dobkin, Lipton [1976]	NNP not defined
$NN(N,k) \le \Theta(2^k + \log N)$	$\Theta(N)$	Friedman, Bentley, Finkel [1975]	$NNP(N,k) = \Theta(kN \log N)$ Heuristic argument

Table 2.1. Summary of previous work

2.2 Divide and conquer

Divide and conquer is one of the most commonly used tools in the construction of algorithms. The basic idea underlying the technique is that to solve a given problem of a certain size, one divides the problem up into similar problems of smaller size, solves them, and then combines those answers to form an answer to the original problem. Most commonly, the same technique is applied to the smaller problems, making the procedure recursive. The recursion is terminated when the problem becomes small enough to solve using some straightforward method. An example of such a recursive divide and conquer algorithm is the MERGESORT algorithm described by Aho, Hopcroft and Ullman [1974]. To MERGESORT a set of N numbers, divide the set into two subsets of N/2 numbers each, sort those subsets recursively using MERGESORT, then merge those answers to form the sorted list desired. Such recursive divide and conquer algorithms are usually very easy to program and can often be analyzed by the use of recurrence relations. The technique of divide and conquer is nicely described by Aho, Hopcroft and Ullman [1974]; the reader unfamiliar with recurrence relations is referred to Liu [1968].

Divide and conquer has found application in many different areas. It is the philosophy underlying Cooley and Tukey's discrete Fast Fourier Transform algorithm [1965]. Strassen used the technique to reduce the time required to multiply two N by N matrices from $\Theta(N^3)$ to $\Theta\left(N^{\log_2 7}\right)$ [1969]. It was used by Blum, et al., [1972] to reduce the time required to find the median of N elements from the previous best known $\Theta(N \log N)$ to $\Theta(N)$. Many other applications of divide and conquer are found in Aho, Hopcroft and Ullman [1974] (listed in the index under "divide and conquer").

The divide and conquer technique has recently been applied to multidimensional problems. Warnock's algorithm [1969] for hidden line elimination uses the strategy in 2-space. Many of the algorithms described in Shamos's workbook [1975] use divide and conquer; as mentioned previously, the algorithms in this thesis are inspired by an especially elegant application of divide and conquer attributed therein to Strong. An algorithm by Kung, Luccio and Preparata [1975] for finding the maxima of a set of multidimensional vectors is of a flavor very similar to the algorithms described in this thesis.

3. The Algorithms

In this chapter we describe the algorithms that are the basis of this thesis. In Section 3.1 we develop a simple algorithm for the sparse fixed radius near neighbor problem and then modify it to reduce the running time. Section 3.2 is devoted to the closest pair problem; we show how the algorithm schema developed in Section 3.1 can be applied to the closest pair problem. The all nearest neighbors problem is the topic of Section 3.3; in that section we extend even further the schemata developed in the first two sections. In Section 3.4 we prove lower bounds on various problems discussed in Sections 3.1 through 3.3.

The sections in which we describe algorithms share a common theme. Instead of merely presenting the algorithms as finished products, we will start by "inventing" a very simple algorithm and develop more complex algorithms until we arrive at the final version, trying to reflect the development of the algorithm in our presentation. We will comment along the way on the tools and techniques of algorithm design employed. To make these easier to note we shall underline such comments. Hopefully the reader will go away from this section with both the understanding of some specific algorithms and insights into the general process of algorithm design. The principles which we employ in this section are discussed more fully in Chapter 6.

We will describe the algorithms in a very high level, abstract language. It should be obvious how such programs could be implemented on a RAM/RASP machine; when it is not so we will comment accordingly. Chapter 5 discusses questions of implementation in more detail. It will be necessary to adopt certain notations. We will denote the value of the m-th point in the i-th coordinate by $X_i(m)$. The distance measure used throughout this chapter is the L_∞ norm; that is

$$D(\ell,m) = \max_{1\le i\le k} X_i(\ell) - X_i(m).$$

3.1 Sparse fixed radius near neighbor algorithms

In this section we develop algorithms for the sparse fixed radius near neighbors problem. It is perhaps appropriate to review the problem briefly. By the definition of sparsity, we know that no δ-ball in the space contains more than some constant c points. For the L_∞ metric which we are using, this means that no hypercube of side 2δ in the space contains more than c points. Our problem is to enumerate all pairs of points which lie within δ of each other in the space.

A fundamental rule in algorithm design is to start with a simple problem. So instead of starting with the full k dimensional problem, let us restrict ourselves to a simple case. The simplest is the one dimensional case; we are given a sparse collection of real numbers and asked to enumerate all pairs within δ of one another. The solution that immediately leaps to mind is to sort the points into a sorted list, then proceed down the sorted list, checking ahead δ. The sorting of the first stage can be done in time of $\Theta(N \log N)$ by any one of a number of sorting algorithms (see Knuth [1973]). The scan of the second stage can be done in linear time; this is guaranteed by sparsity, which says that checking ahead δ will require examining at most c elements. Thus the second stage requires at most cN steps. We therefore have an algorithm that demonstrates $SFR(N,1) \leq \Theta(N \log N)$. Though this may appear to be but a trivial exercise, we will see later that this result is extremely useful. We therefore feel justified in stating _always start with the simplest case possible, even it if appears trivial at the time_.

We proceed now to the next case in order of decreasing simplicity: suppose the points lie in a plane. The first technique to which our search for a good algorithm leads us is iteration; we ask _is there a strategy which, when iterated for each point in the file, will solve the problem as a whole_? The answer is clearly yes; we examine all pairs of points. Unfortunately, this yields a $\Theta(N^2)$ algorithm. We could give up here, accept a quadratic algorithm, and try to prove a quadratic lower bound. However, we are motivated to search for a better algorithm by the fact that we know that we can do better than quadratic in the one dimensional case. Thus we see one important psychological effect of solving simpler problems first--motivation to keep looking for answers. The next technique of algorithm construction we consider is divide and conquer: _is there some way to divide the problem into smaller parts, such that the solution of those parts could be combined to form a solution to the whole_? As the principle of starting with the easiest dictates, we ought to look first for a divide and conquer algorithm that divides the problem into two parts (instead of more); we should also try to keep them of approximately the same size (that is, balanced; see Aho, Hopcroft and Ullman [1974] for details).

At this point our creativity (or whatever it is psychologists have left us with) suggests the following strategy, which is illustrated in Figure 3.1-1: Divide the file F into two point sets A and B by a vertical line ℓ such that both A and B contain N/2 points ($\lceil N/2 \rceil$ and $\lfloor N/2 \rfloor$ points for odd N). Solve the subproblems for A and B recursively by enumerating all fixed radius near neighbor pairs with both elements in A or both in B. Note now that to solve the problem after having done this, all we need to do is to enumerate all near neighbor pairs with one point in A and one point in B. To find these pairs we need only consider points in the slab of width 2δ with center ℓ, noted by S in Figure 3.1-1. Note that up to all N points could lie in region S.

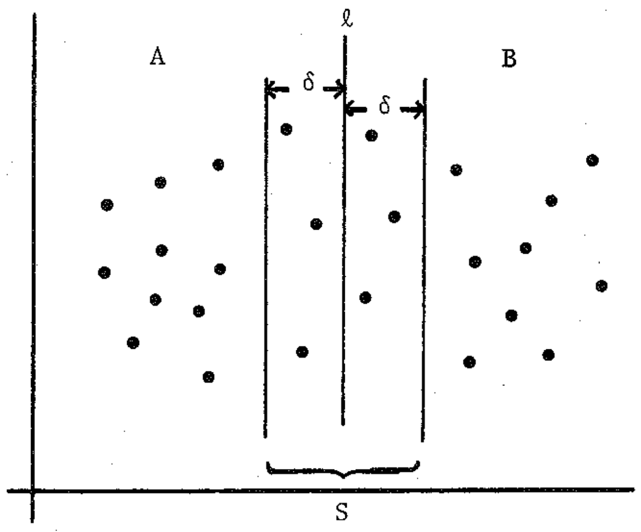

Figure 3.1-1: Illustration of Algorithm SPARSE2

How do we solve the reduced problem of enumerating all fixed radius pairs in S with members of the pair on different sides of ℓ?

Although this appears to be another problem in the plane, we can view this as an essentially one dimensional problem. That is, we can solve the reduced problem by projecting all points in S onto ℓ, and then solving the one dimensional sparse problem using our previously developed algorithm for one dimension. (Note that projection of points within δ onto ℓ preserves sparsity; any points in a δ-ball on the line were within the same δ-ball in the plane, thus there can be at most c within any δ-ball on the line.) We may, however, have to throw out some "false drops"--pairs which were on the same side of ℓ or greater than δ apart in the plane. First examining the simpler one dimensional case paid off both in giving us a mind set which allowed us to see a method of solution for the reduced problem, as well as providing the tool to solve the problem once we slightly modified it.

Thus we have the basic idea for a divide and conquer algorithm for the SFR problem in the plane. The next step in developing such an algorithm is to <u>describe it in an appropriate high level language, using notation as fits the problem</u>. Let the input to the problem be a set of points F. We will want to have available the algorithm for the one dimensional case described earlier; call it SPARSE1 (for reasons that will become obvious later). The procedure call SPARSE1(F) enumerates all fixed radius near neighbor pairs in the one dimensional file F. We now describe our algorithm (which we call SPARSE2) as Algorithm 3.1-1.

(It is appropriate to mention here that the cut line ℓ is "fuzzy" in the following sense: If many points lie on ℓ then they are arbitrarily assigned to A and B such that both A and B contain

N/2 points. All further cut lines we use in this thesis will be "fuzzy" in this sense.)

<u>Algorithm 3.1-1</u>

Procedure SPARSE2(set F)	$\|F\| = N$
1. If $\|F\| \leq 1$, then return.	c_1
2. Choose a vertical line ℓ partitioning F into two collections of N/2 points each, A and B. (Note that this can be accomplished by finding the median element of F in the X_1 direction; the line is defined by that X_1 value.)	c_2N
3. Solve the subproblems with the recursive calls SPARSE2(A) and SPARSE2(B).	$2T(N/2)$
4. Let S be the set of all points within δ of ℓ, projected onto ℓ. (Note that the projection preserves sparsity with the sparsity constant c unchanged.)	c_3N
5. Call SPARSE1(S) to solve the sparse problem on the line. Check the pairs enumerated there to insure (a) they are on different sides of ℓ and (b) they are within δ in the plane.	$SFR(N,1) + c_4N$

$$T(N) = 2T(N/2) + (c_2 + c_3 + c_4)N + c_1 + SFR(N,1)$$

$$= 2T(N/2) + \Theta(N \log N)$$

$$T(1) = T(0) = c_1$$

$$\therefore T(N) = \Theta(N \log^2 N)$$

The next step in the design process is to <u>analyze the resulting algorithm</u>. As with all of the algorithms developed in this thesis, we present a worst-case analysis with the algorithm itself, giving the running time in the Θ notation. We will go through the analysis of SPARSE2 now at a fairly close level of detail; later in this thesis we will only examine critical steps of the run time analysis in the text, presenting the analysis summarily with the algorithm itself. For each step we will count the number of RAM/RASP operations employed. Step 1 requires constant time. The median selection of Step 2 requires time linear in N using the selection algorithm of Blum, et al., [1972]. The recursive call of Step 3 will require two invocations of SPARSE2, each on a file of size approximately N/2 (the effect of approximation is discussed in Aho, Hopcroft and Ullman [1974]; it is negligible). The projection of Step 4 can be accomplished in linear time by processing each point individually. By our analysis of SPARSE1, the call in Step 5 could cost up to $SFR(N,1) = \Theta(N \log N)$ (since all N points could be in S). Since only up to cN pairs could be enumerated (by sparsity of the projection), we can check those in time linear in N for the stated conditions. (Note that we assume c to be a constant independent of N.) We thus arrive at the recurrence system given in the presentation of the algorithm, which is well known to have the stated solution. We have therefore shown that $SFR(N,2) \leq \Theta(N \log^2 N)$ (where we use $\log^k N$ as a shorthand for $(\log N)^k$).

We must now determine how much storage the algorithm requires (in addition to the storage of the N points). The median selection algorithm requires $\Theta(N)$ work storage. Up to $\Theta(N)$ storage could be required to hold the projected point set built in Step 4. The same storage areas for this data could be used by different invocations of the algorithm SPARSE2 (since it is not required to save any of the information contained there). Thus the total storage required by SPARSE2 is linear in N. All of the algorithms described in this thesis require linear storage; because the analyses are very straightforward we will omit them in future algorithms. A more exact analysis of the storage requirements is contained in Section 5.1.

Our development of SPARSE2 could now proceed in any one of three directions. In the first direction we would begin by describing the algorithm in a more detailed language. The resulting program is then analyzed in detail for its running time (many refer to this as the "Knuthian analysis"), and costly segments of code are fine-tuned. Though this step should certainly be made at some point in the algorithm design process, it would not be appropriate at this stage.

The second direction in which one could proceed is to see if any standard "speed-up tricks" can be applied to the algorithm. In our algorithm we note that a great deal of sorting is being done by calls on SPARSE1. It is fairly well known that repetitive sorting can often be avoided by "presorting" the file (in this case on the X_1 key); after this presorting the algorithm would have to maintain point sets in sorted order. Calls on SPARSE1, however, would take only linear time, and the resulting running time of SPARSE2 would be $\Theta(N \log N)$. This is the direction that the author actually took in the development of the algorithms in this thesis; we will not, however, proceed in this manner now. It turns out that the speed-up achieved by presorting can be achieved more elegantly in a different way, so the mention made thus far of presorting will suffice. We have, however, observed another general rule for algorithm development: <u>try to apply standard speed-up techniques, such as presorting</u>.

The third direction open to us, and that which we shall now take, is to <u>generalize the current algorithm</u>. A first step in the process of generalization is to <u>describe the algorithm in abstract terms</u>--terms which convey the essential idea of the algorithm without getting cluttered up in details. Such an abstraction of SPARSE2 might be "to solve the problem in the plane, solve two subproblems in the plane (each operating on half the points), and one subproblem on the line". The next step in the generalization process is to generalize the current algorithm to the next most simple case; for this problem we should now try three dimensions. The obvious generalization of the abstraction of SPARSE2 to the abstraction of SPARSE3 is "to solve the problem in 3-space, solve two subproblems in 3-space (each operating on half the points), and one subproblem in the plane". As we mentally sketch first versions of SPARSE3, we see that it is very similar to SPARSE2. Instead of choosing a cut line to divide the problem into smaller subproblems, we choose a cut plane. After solving the subproblems in 3-space, to solve the reduced problem we will project all points within δ of the plane onto the plane, and solve the sparse problem in the plane (for which we can use SPARSE2). It would be appropriate at this point to write down algorithm SPARSE3 in the same format we used for SPARSE2, and do a similar rough worst-case analysis (which would yield its running time as $\Theta(N \log^3 N)$).

We are now equipped to develop a fully general algorithm. We guess by induction from the cases k = 1, 2, 3 that the running time of algorithm SPARSEk will be $\Theta(N \log^k N)$. (Such guesses by induction can never hurt and are often helpful in finding the algorithm behind the running time.) Next we would generalize the abstraction to k-space. Finally we are ready to write down our generalization SPARSEk as Algorithm 3.1-2.

Algorithm 3.1-2

```
Procedure SPARSEk(set F)
1.  If $|F| \le 1$, then return.                                      $c_1$
2.  Choose a k-1 dimensional hyperplane P orthogonal                  $c_2N$
to the $X_1$ axis partitioning F into two collections
of N/2 points each, A and B.  (Note that this can
be accomplished by finding the median element of F
in the $X_1$ direction; the hyperplane is defined by
the median $X_1$ value.)
3.  Solve the subproblems with the recursive calls                    $2T(N/2,k)$
SPARSEk(A) and SPARSEk(B).
4.  Let S be the set of all points within $\delta$ of P               $c_3N$
projected onto P.  Note that the projection preserves
sparsity with the sparsity constant c unchanged.
5.  Call SPARSE(k-1) (S) to solve the sparse problem                  $T(N,k-1) + c_4N$
in the k-1 dimensional space.  Check the pairs
enumerated there to insure (a) they are on different
sides of P and (b) they are within $\delta$ of each other in
k-space.
```

$$T(1,k) = T(0,k) = c_1$$

$$T(N,1) = \Theta(N \log N) \quad \text{[Using SPARSE1]}$$

$$T(N,k) = 2T(N/2,k) + T(N,k-1) + \Theta(N)$$

$$\therefore T(N,k) = \Theta(N \log^k N)$$

In the analysis of SPARSEk given in Algorithm 3.1-2 we use the notation T(N,k) to denote the worst-case running time of SPARSEk on a collection of N points. It is important to note that our analysis is indeed of the worst case; we have assumed that all N points will lie in region S. By the existence of SPARSEk we have shown that $SFR(N,k) \le \Theta(N \log^k N)$.

At this point we do well to "meditate" on our algorithm. Its abstraction is "to solve a problem of N points in k-space, solve two problems of N/2 points in k-space, and one problem of up to N points in (k-1)-space". We might recall that we have seen such a schema before in the algorithm of Kung, Luccio and Preparata [1975] for finding the maxima of a set of N vectors in k space, and its running time of $\Theta(N \log^{k-2} N)$ is very similar to $\Theta(N \log^{k} N)$. We should now familiarize ourselves with the related work, to gain deeper insight into the problem and perhaps avoid reinventing a wheel (for reinvented wheels often have flat sides!).

We are now faced with a decision similar to one we faced previously about SPARSE2: should we try to prove a lower bound of $SFR(N,k) \geq \Theta(N \log^{k} N)$, or should we try to develop better algorithms? This decision is not so clear cut as before, and we should proceed down both paths simultaneously, with insights in one area helping our advance in the other until we eventually see on which path we should concentrate our attention. It turns out that the correct decision (that is, the profitable path) is to try to reduce the running time of the algorithms.

A technique we have already seen for reducing the running time of an algorithm was presorting. When applied to SPARSE2, the running time was reduced from $\Theta(N \log^2 N)$ to $\Theta(N \log N)$. Unfortunately this method can be used in general to remove only one log N factor from the running time; it gives rise to a SPARSEk algorithm with $\Theta(N \log^{k-1} N)$ performance. We will therefore keep this technique in mind, but look elsewhere for other speedups, temporarily ignoring presorting to keep our vision clear for larger gains.

As we seek to reduce the running time of SPARSEk, the principle of "starting with the easiest" dictates that we first concentrate our attention on reducing the running time of SPARSE2. We set our goal as reducing the running time of SPARSE2 to $\Theta(N \log N)$. We have two reasons for choosing this goal. First, $\Theta(N \log N)$ was achievable for SPARSE1, and secondly, it was achievable for SPARSE2 with presorting. It is often helpful in algorithm design to <u>have in mind a specific performance bound as a goal</u>.

We should now investigate SPARSE2 to find out why it does not achieve $\Theta(N \log N)$; we should <u>ask why is it so expensive</u>? We see that the factor in the recurrence system of Algorithm 3.1-1 that raises the cost to $\Theta(N \log^2 N)$ is the call on SPARSE1 with up to N points, which costs $\Theta(N \log N)$. Thus we have identified the aspect of our algorithm which causes the increase in running time; we should now concentrate on reducing this cost.

There are two ways in which we can decrease the cost of the call on SPARSE1. The first is to make SPARSE1 faster; this is essentially the way in which presorting brought about a speedup. Apart from presorting, however, we can see no way to speed up SPARSE1. The second way to speed up our call on SPARSE1 is to reduce the cardinality of the set S which we pass to the procedure. We have assumed that all N points will be in S; let us study exactly what that situation entails by observing Figure 3.1-2. If all N points lie in a vertical slab of width 2δ, then the obvious strategy for reducing the cardinality of S is to choose as the cut line ℓ a horizontal line dividing F into two equally sized sets. The cardinality of S will then be at most c and the call on SPARSE1 will take only constant time. It turns out that not all cases are this easy, but examination of the most degenerate case has led us to an idea for reducing the size of S: sophisticated choice of cut lines. We thus note the importance of <u>examining the degenerate cases</u>.

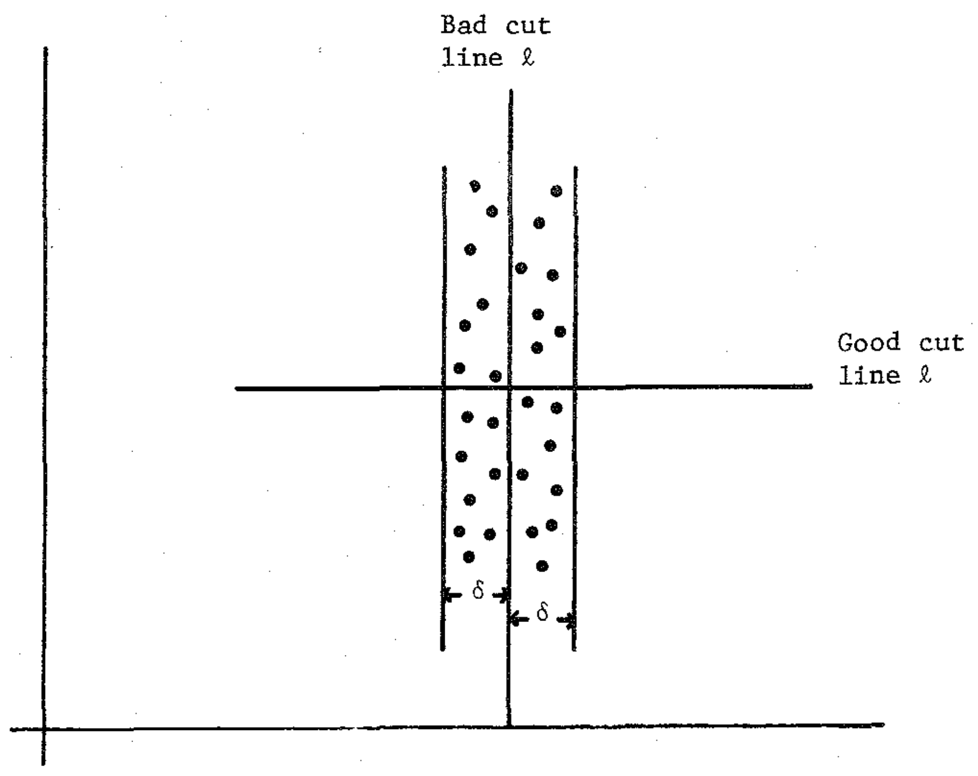

Figure 3.1-2: A degenerate case

We now look for a strategy for choosing cut lines. What we need is a strategy which keeps the subproblems A and B balanced, allows a cut line to be found in linear time, and gives a set S such that the call SPARSE1(S) takes at most linear time. If we assume that the running time of SPARSE1 cannot be reduced below $\Theta(N \log N)$, this implies that the cardinality of S will have to be $O(N/\log N)$ (note that for a $\Theta(M \log M)$ function to be $O(N)$, M must be $O(N/\log N)$). Because of our good fortune in the degenerate case we examined in Figure 3.1-2, we might guess that the strategy we should employ is to try a vertical cut line, and then if that fails, try a horizontal, hoping that at least one of the two will yield a set S of acceptable size. The counterexample depicted in Figure 3.1-3 shows that this strategy will not always work. Both the horizontal and vertical cut lines which partition F by finding median elements have resulting values of $|S| = N/2$, which is greater than $O(N/\log N)$. Hence we see that our strategy for choosing the cut line will have to become more sophisticated in one of two ways: by choosing a cut line which is not necessarily orthogonal to one of the axes, or by choosing a cut line which does not partition the set into subproblems of exactly equal size.

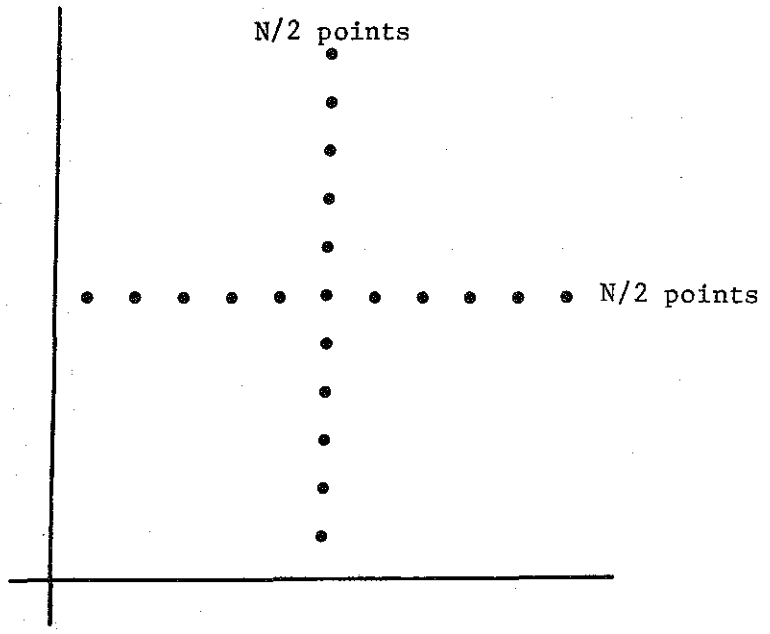

Figure 3.1-3: A second degeneracy

Though choosing a cut line which is not orthogonal to one of the axes is attractive at first glance, it seems to be a difficult strategy to implement. In Figure 3.1-3 it is obvious to the human eye to choose a cut line that makes a 45° angle with the axes and passes through the center of the set; it is not obvious that an algorithm could choose that as easily. After devoting much effort to this strategy, the author could make no progress.

We now concentrate our search for a cut line selection strategy on a cut line which is orthogonal to one of the coordinate axes but does not necessarily divide F exactly in two. A suitable strategy must insure three things about the cut line ℓ it chooses: First, ℓ must be locatable in linear time. Secondly, only O(N/log N) points can lie within δ of ℓ. Thirdly, the sets A and B induced by ℓ must be of almost the same size (that is, balanced). The definition we have used of balanced so far is that the cardinality of A and B differ by at most one. For many divide and conquer algorithms, however, such a strict definition of balanced is not required. A suitable definition of balanced, which preserves the asymptotic Θ(N log N) behavior but changes the constants, is that <u>the size of each subproblem is at least as large as some constant proportion p of the total problem size</u>. This is a generalization of the principle employed in the definition of balanced trees due to Adel'son-Vel'skii and Landis [1962]. A node in a tree which is balanced by their definition does not necessarily have the same number of left and right descendants, but each of the subtrees is known to contain at least $1 - \phi^{-1}$ (or approximately 38%) of the descendants of the node (see Knuth [1973] for a proof). This condition, which is weaker than demanding totally balanced (or complete) trees, is enough to guarantee logarithmic search behavior in balanced trees. Thus the third condition that our cut line selection strategy must meet is that both A and B partition the input set and each contain at least pN points, where p is some constant the strategy must choose.

Now that we have a fairly clear idea of what we are looking for, we must choose a value of p and show that such a cut line ℓ will exist. Once again we derive a benefit from having examined the degenerate case--we know from the degeneracy in Figure 3.1-3 that p must be less than 1/4. We thus have reduced our problem of algorithm development to a problem of constructing a geometric proof. We must now don hats of both algorithm designers and mathematicians and proceed to conjecture and prove the following theorem.

Theorem 3.1-1: (Existence of a cut line in the plane.) Given a sparse collection of N points in the plane (where N is greater than some constant N_2), there exists a cut line ℓ perpendicular to one of the original coordinate axes with the following properties:

1. No more than 7N/8 points are on either side of ℓ.
2. There are at most $2cN^{1/2}+1$ points within distance δ of ℓ.

Proof: To prove this theorem we will show that the assumption of its negation leads to a contradiction. To do this we demonstrate that a set without the properties described in the theorem must be both very dense and very sparse. Consider the points indexed in increasing order by x-coordinate (thus the point with the least x value has index 1, etc.; points with tied x values are indexed arbitrarily). Let us now restrict out discussion to M_x, the middle 3N/4 points in the indexing (only a cut line which passes through the restricted set will satisfy Condition 1 of the theorem). The assumption of the negation implies that every collection of

$2cN^{1/2}+1$ points contiguous in the indexing projects onto a segment of the x-axis less than 2δ in length (if a given collection projects onto a longer segment then the center of that segment could be used to define a cut line ℓ with the desired properties). The situation that we have described is depicted in Figure 3.1-4. The regions R_x and L_x contain the rightmost and leftmost N/8 points in the indexing, respectively. The region C_x is that to which we have restricted our discussion; it is the smallest vertical slab containing M_x. We now subdivide C_x into closed regions T_i by drawing vertical lines through the 1-st, $1(2cN^{1/2})+1$ -st, $2(2cN^{1/2})+1$ -st, $3(2cN^{1/2})+1$ -st, ... points of M_x (ordered by index); note that there are $2cN^{1/2}+1$ points in any region T_i. Because the boundary point (on the line) is in two regions T_i and T_{i+1}, there are only $2cN^{1/2}$ points uniquely associated with region T_i (associate the boundary point with T_i and not T_{i+1}). Since each T_i is defined by $2cN^{1/2}+1$ points contiguous in the x dimension, its width is bounded by 2δ. We can now bound the width of C_x by observing that C_x is comprised of $(3N/4)$ / $(2cN^{1/2})$ regions T_i, each of which is of width less than or equal to 2δ, so we have

$$\text{width}(C_x) \le \frac{(3N/4)}{2cN^{1/2}} \cdot 2\delta = \frac{3N^{1/2}\delta}{4c}.$$

We have thus far considered only C_x, the center region in the x dimension; similar arguments hold for C_y, the center region in the y dimension. Let us now examine C_{xy}, the intersection of C_x and C_y. Since at most N/4 points lie outside each of C_x and C_y, there must be at least N/2 points in C_{xy}. On the other hand, since the length of the sides of C_{xy} is bounded by $\frac{3N^{1/2}\delta}{4c}$, the total area of C_{xy} is bounded above by that length squared, or $\frac{9N\delta^2}{16c^2}$. By sparsity we know that the number of points in C_{xy} is therefore bounded above by $\frac{9\delta^2}{16c^2} \cdot \frac{c}{4\delta^2} = 9N/64c$ points (where $c \geq 1$). We have thus arrived at the contradiction that C_{xy} contains at least $N/2$ points, but at most $9N/64c$ points. Therefore the negation is false and we have proved our theorem. (Certain assumptions, such as C_x being composed of many collections of $2cN^{1/2}$ points, required "large enough" N. We included the phrase "$N > N_2$" in the statement of the theorem for that reason.) □

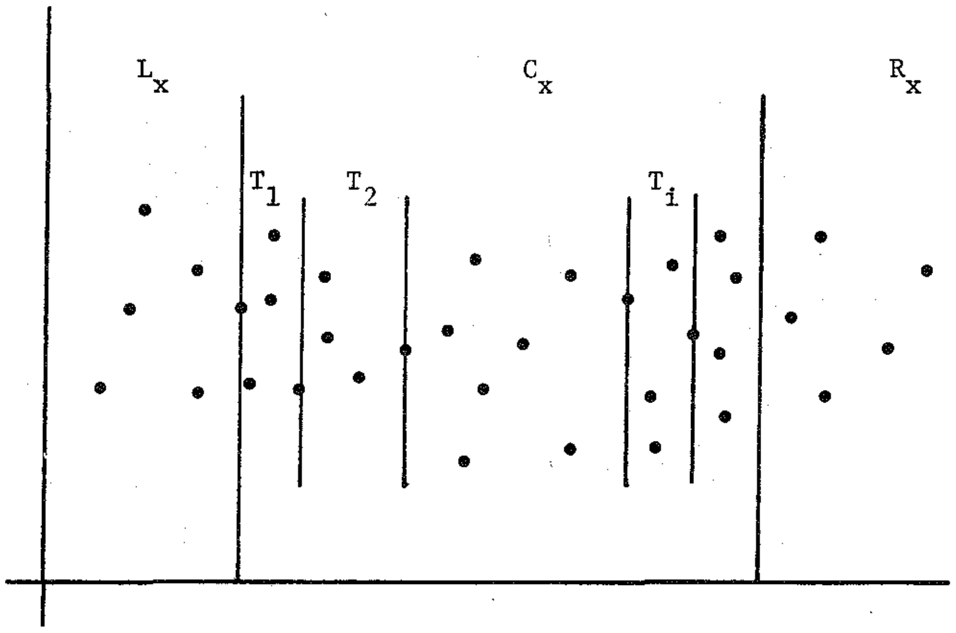

Figure 3.1-4: N points in the plane

Theorem 3.1-1 leads immediately to a $\Theta(N \log N)$ algorithm for the SFR problem in the plane. Before the first invocation of our faster recursive procedure (which we will call FSPARSE2), we presort the points on all coordinates to allow rapid sequential processing of the file. A recursive invocation of FSPARSE2 would then choose the cut line ℓ by scanning down both sorted lists until it found a gap 2δ wide which contained not more than $2cN^{1/2}+1$ points. It would use the middle of that gap for ℓ. The existence of such a gap is guaranteed by the theorem. After the resulting subcollections are recursively processed the algorithm would project all points in the slab to form set S; its cardinality would be $O(N^{1/2})$. Hence the call SPARSE1(S) would require O(N) time. The recurrence relation describing the worst case running time of FSPARSE2 would be

$$T(N) = T(N/8) + T(7N/8) + \Theta(N)$$

which has solution $T(N) = \Theta(N \log N)$.

In the actual algorithm design process one would at this point write down a description of FSPARSE2. We will skip that stage for brevity, and proceed to generalize to FSPARSEk. The generalization should be obvious. We start by proving Theorem 3.1-2, and then give FSPARSEk as Algorithm 3.1-3.

Theorem 3.1-2: (Existence of a cut plane in k-space.) Given a sparse collection of N points in k-space (where N is greater than N_k), there exists a cut plane P perpendicular to one of the original coordinate axes with the following properties:

1. No more than $N(1-1/4k)$ points are on either side of P.
2. There are at most $kcN^{1-1/k}+1$ points within distance δ of P.

Proof: The proof proceeds in the same manner as that of Theorem 3.1-1. Assuming the negation of the theorem leads us to the contradiction that the hypercube in k-space corresponding to

C_{xy} contains at least N/2 points, but at most $\left[\frac{\frac{1}{(1-2k)}}{kc}\right]^k \cdot cN$ points, which is less than N/2 for $k > 1$, $c \geq 1$. □

Algorithm 3.1-3

```
Procedure FSPARSEk(set F)
1. If $|F| \leq 1$, then return.                                   $c_1$
2. Choose a k-1 dimensional hyperplane P                           $c_2 kN$
   partitioning F into two point sets A and B in
   the following way. For each of the k dimensions,
   scan down the list of the points sorted by that
   dimension from the (N/4k)-th element to the
   (N-N/4k)-th element. In the scan keep two
   pointers active, one $kcN^{1-1/k}+1$ elements ahead
   of the other. For each pair examined, calculate
   the distance between them. When a distance
   greater than $2\delta$ is found, stop the scan and
   choose the center of that interval as the value
   defining the hyperplane. Theorem 3.1-2 guarantees
   that such an interval will be found.
3. Divide F into the two subsets A and B                           $c_2 N$
   determined by P. Maintain the sorted lists
   for each dimension for both A and B.
4. Solve the subproblems with the recursive                        $T[(1/4k)N,k]$ +
   calls FSPARSEk(A) and FSPARSEk(B).                              $T[(1-1/4k)N,k]$
5. Let S be the set of all points within $\delta$                 $c_4 N$
   of P projected onto P. Note that projection
   preserves sparsity. As in Step 3, maintain
   the orderings by dimension for set S.
6. Call FSPARSE(k-1) (S) to solve the sparse                       $T[ckN^{1-1/k}+1,k-1]$
   problem and check enumerated pairs for false
   drops.
```

$$T[1,k] = T[0,k] = c_1$$

$$T[N,1] = \Theta(N \log N)$$

$$T[N,k] = T[(1/4k)N,k] + T[(1-1/4k)N,k] + \Theta(kN) + T[\Theta(N^{1-1/k}),k-1]$$

$$\therefore T[N,k] = \Theta(kN \log N)$$

Algorithm FSPARSEk is easily understood as the reader keeps algorithm SPARSEk in mind. It is essential for the $\Theta(kN \log N)$

running time that the representation of a set of points in k-space include the point set sorted in order by each of the k dimensions. This can be easily accomplished by presorting the file on each of the k dimensions, and then keeping a linked list representation of each sorted list. When a set is partitioned into two subsets the list is "unzipped" to form two sorted lists (by an "inverse merge" procedure), and then merged back together again when the partitioning is no longer required.

The worst-case behavior of FSPARSEk occurs when the sets A and B are most unbalanced, hence we assume that condition in our worst-case analysis, giving the stated recurrence relation. We can easily prove by induction that the solution to the recurrence system is as stated. The thrust of the argument is that if we have shown that $T(N,k-1)$ is $\Theta((k-1)N \log N)$, and we invoke that on a set of $O(N/\log N)$ points, then the cost of the procedure is $O(kN)$. The rest of the analysis for the k dimensional case is standard, and FSPARSE1 = SPARSE1 gives the basis for the induction.

Should we try to reduce the running time of FSPARSEk below $\Theta(kN \log N)$? It certainly seems that the factor of k cannot be decreased, for the amount of information describing each point increases linearly with k. The question of whether the $\Theta(N \log N)$ can be reduced seems to be very hard. We show in Section 3.4 that in general, $\Theta(N \log N)$ is a lower bound for $SFR(N,k)$ by examining the degenerate case of $\delta = 0$. For nonzero δ, however, Bentley, Stanat and Williams [1976] have shown that $\overline{SFR}(N,k) = \Theta(kN)$ and have also given an algorithm which shows $SFR(N,k) = \Theta(kN)$ if an arbitrary amount of random access storage is available. The author conjectures that $\Theta(kN \log N)$ is a lower bound on the minimax complexity of the SFR problem if only $\Theta(kN)$ work storage is permitted.

As we come to an end of our development of algorithms for the SFR problem, we should ask what we have gained by examining the problem. We have observed many general principles of algorithm design of which we should be conscious and employ in the development of further algorithms. The algorithms which we have developed are important tools for us to keep in our tool bag as we examine further closest point problems.

We have also gained insight into a particular class of divide and conquer algorithms for multidimensional space. There are three important themes in the algorithms in this section. The first theme is the abstract schema of solving a problem by dividing it into two smaller problems in the same space, and one problem in a space of lower dimensionality. The second theme is the use of sparsity, which we employed to limit output size as well as an invariant condition (of sorts) for our reduced sets. The third particular principle of multidimensional algorithm design we observed is that of prudent choice of cut planes. We will find these three techniques very useful in further problems which we study.

3.2 Closest pair algorithms

We now turn our attention to algorithms for the closest pair problem. In review, we are to develop a procedure which will tell what are the two closest together among N points in k-space. In this development we will use the results of Section 3.1 both as available tools and as a source of experience in multidimensional problem solving.

In developing a closest pair algorithm we will follow our well established rule and start with the simplest case. Finding the closest pair in one dimensional space can be easily accomplished by sorting the points and then scanning down the sorted list, looking for the closest point to each (which will be either its left or right neighbor) and remembering the pair with minimum separating distance. The sort will take $\Theta(N \log N)$ time and the scan $\Theta(N)$, so the running time of the algorithm as a whole will be $\Theta(N \log N)$. Thus we know that $CP(N,1) \leq \Theta(N \log N)$.

The next most simple case is that of the plane. The obvious iterative strategy that examines all $\binom{N}{2}$ pairs of points and finds the minimum distance requires quadratic time, and knowing that the complexity of the problem is $\Theta(N \log N)$ on the line makes us hesitant to settle for quadratic in the plane. We are therefore encouraged to look for faster algorithms in the plane and the first technique that occurs to us is divide and conquer.

The logical way to proceed in our attempt to apply divide and conquer to the closest pair problem in the plane is to <u>employ as much as possible of a strategy for solving a similar problem</u>. In this case the CP problem is similar to the SFR problem, so we will attempt to apply aspects of algorithm SPARSE2 to algorithm PAIR2 (which solves the CP problem in the plane). (Starting with the easiest dictates that we should not try to apply FSPARSE2.) A divide and conquer procedure typically has three identifiable stages: breaking the problem into subproblems, recursively solving the subproblems, and combining the solutions of the subproblems to yield a solution to the problem as a whole. In our development of PAIR2 we should try to use as many of these stages from SPARSE2 as possible.

We will now attempt to synthesize PAIR2 using components from SPARSE2. SPARSE2's first stage of dividing the file F into two point sets A and B by a vertical line ℓ seems to be applicable to the CP problem, so we will take that as the "divide" stage of PAIR2. For the "recursive" stage we will find the closest pairs among both A and B. Is there some way in which we can now combine these solutions to the subproblems to form a solution to the CP problem? In order to answer this question we should examine what properties are true in the plane after the subproblems have been solved; to investigate this we shall employ Figure 3.2-1. The letters, A, B, and ℓ all have their obvious meanings in the figure. We use δ_A to denote the L_∞ distance between the closest pair in A and similarly for δ_B.

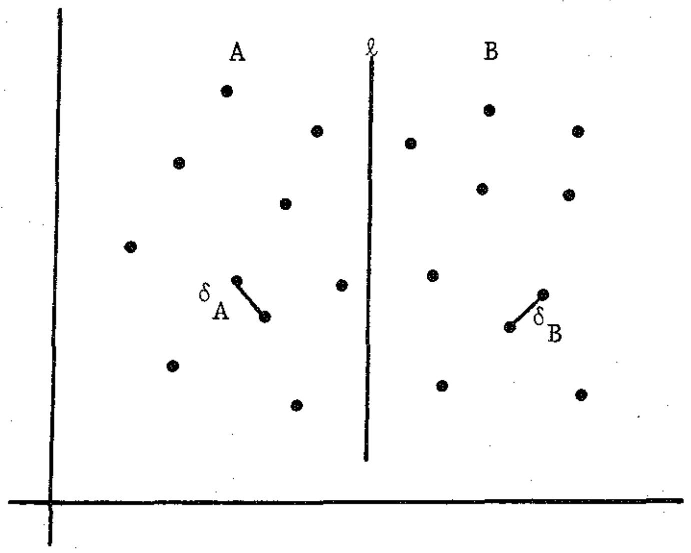

Figure 3.2-1: The plane with solved subproblems

The closest pair in the whole plane might very well be the closest pair in A or the closest pair in B. Let us use $\delta = \min(\delta_A, \delta_B)$; δ is the distance between the closest pair we have discovered so far. To combine the solution of the parts to form a solution to the whole we must find any pairs in the plane that are closer than δ to one another; the closest pair in the whole plane (if not δ_A or δ_B) will be among these.

We must make two observations before we can synthesize the third stage of PAIR2. First, if a pair is within δ in the plane, then the points of the pair must be on opposite sides of ℓ. Were they not, then the previous values of δ_A and δ_B must have been inaccurate. Secondly, we note that the set A is sparse with respect to radius δ_A. The sparsity constant $c = 9$ can be found by examining the worst-case configuration of points given in Figure 3.2-2 (nine points are in the δ_A ball with center x); no δ_A ball could contain more than 9 points, for if it did then two of the points would be closer than δ_A together, which denies that δ_A is the distance separating the closest pair in A. Region B is likewise sparse with respect to δ_B and $c = 9$. It is thus clear that the plane as a whole is sparse with respect to radius δ and $c = 2 \cdot 9 = 18$ (because no δ-ball contains more than 9 points from A or 9 points from B, or 18 points all together); the bound of 18 could be tightened.

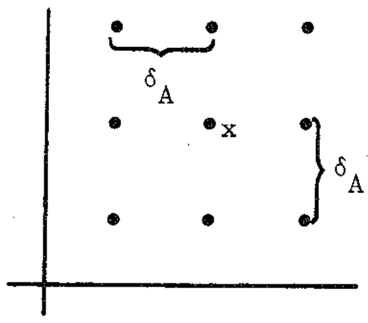

Figure 3.2-2: A worst-case configuration

We are now equipped to synthesize the "combining" stage of PAIR2. We must locate any pairs in the space within δ of each other and we have shown that the space is sparse; we need only solve the SFR problem in the plane. Our experience with SPARSE2 suggests that in doing that we might be doing more work than we need to do, and our first observation above (that the pair for which we are looking must have its points on opposite sides of ℓ) implies that we need only examine points in the slab of width 2δ and center ℓ, just as in SPARSE2. To do this we project all points within δ of ℓ onto ℓ, and then solve the resulting SFR problem on the line.

Having synthesized the third and final stage, we are ready to present the resulting divide and conquer algorithm PAIR2 as Algorithm 3.2-1 which returns the distance separating the closest pair in the space. (The bookkeeping required to give the points comprising the pair is obvious and giving it would only serve to obscure the structure of the algorithm.)

Algorithm 3.2-1

```
Procedure PAIR2(set F)
1.  If $|F| = 2$ then return the interpoint distance.              $c_1$
2.  Choose a vertical line $\ell$ partitioning F into two          $c_2N$
collections of about $N/2$ points each, A and B.
3.  Use the procedure recursively to find                          $2T(N/2)$
    $\delta_A \leftarrow$ PAIR2(A), and
    $\delta_B \leftarrow$ PAIR2(B).
4.  Set $\delta \leftarrow \min(\delta_A, \delta_B)$.              $c_3$
5.  Project all points within $\delta$ of $\ell$ onto $\ell$; call this    $c_4N$
set S.  (Note that this collection is sparse on the line
with sparsity constant $c = 18$.)
6.  Use SPARSE1(S) to enumerate all pairs within $\delta$ in S.    SFR(N,1)
Check for any pairs enumerated that are within $\delta$ in
the space and on different sides of $\ell$.  Let $\varepsilon$ be the
distance between the closest pair enumerated.
7.  Return $\min(\delta, \varepsilon)$.                            $c_5$
```

$$T(N) = 2T(N/2) + \Theta(N \log N)$$

$$\therefore T(N) = \Theta(N \log^2 N)$$

We should now inspect PAIR2 to see what principles of multidimensional problem solving we employed that might be valuable in our further development of closest pair algorithms. The technique of dividing a problem by a vertical cut line proved useful once again. One of the most interesting aspects of this problem is that sparsity, though not present in the original point set, was induced in the problem after the subproblems were solved. We were then able to use the induced sparsity to our advantage in putting together the subproblems to form a solution to the whole problem. The abstract description of PAIR2 is the same as for SPARSE2.

Our development of PAIR2 could now proceed in two directions, corresponding to the two directions in which we modified SPARSE2. The first is to generalize PAIR2 to PAIRk; the second is to speed up PAIR2 to yield FPAIR2. We see here an important principle of algorithm development: the development of new algorithms should follow as closely as possible the development paths of established algorithms. Following a development path similar to that of FSPARSEk helps make the decision of which path to follow now seem less important. We assume that we are aiming eventually toward an algorithm we will probably call FPAIRk; we will therefore probably have to follow both paths before we reach our goal. The order in which we pursue the paths does not at this point seem to be crucial.

Let us first generalize PAIR2 to higher dimensional spaces. In the actual development process we would try to develop an algorithm for 3-space at this point; for brevity, however, we will skip that stage and attempt now to develop PAIRk, which will find the closest pair in k-space. We should keep both SPARSEk and PAIR2 in mind as we develop PAIRk. The first stage of PAIRk is the same as the first stage of SPARSEk--choosing a (k-1)-dimensional hyperplane P dividing F into two almost equally sized sets A and B. The second stage is also fairly obvious, both from PAIR2 and SPARSEk--we should find δ_A and δ_B by recursive use of PAIR2. Note that we have now guaranteed sparsity in both A and B with constant $c = 3^k$, by an argument similar to that in our development of PAIR2. We can also show that if we let $\delta = \min(\delta_A, \delta_B)$, then the set as a whole is sparse with constant $c = 2 \cdot 3^k$. With this condition insured, the last stage of our divide and conquer algorithm becomes clear--project all points within δ of P onto P, then check that projection for pairs within δ using FSPARSE(k-1). Let ε be the distance in k-space between the closest pair in S, then return the minimum of δ and ε.

Algorithm PAIRk is so similar to PAIR2 that it is not necessary to describe it formally. The recurrence relation describing its running time is the same, and the running time is also $\Theta(N \log^2 N)$ for fixed k. The reason that the logarithmic term is only squared and not raised to the k-th power is that we used the faster algorithm FSPARSEk as a tool. (Though it would have been imprudent to use it as a model because it is so complicated, it is quite helpful to have it lying in our tool bag.) Algorithm PAIRk shows that for fixed k, $CP(N,k) \leq \Theta(N \log^2 N)$.

Should we be satisfied with the $\Theta(N \log^2 N)$ performance of PAIRk, or should we try to speed it up? Both our bound of $CP(N,1) \leq \Theta(N \log N)$ and our experience in building FSPARSEk suggest that we should try to reduce the running time. Should we now try to speed up the general algorithm PAIRk, or should we attempt to modify PAIR2 first? The author hopes that by now the principle of "starting with the easiest" is so firmly established that the reader's response is immediate--we start with PAIR2.

Before we attempt to modify PAIR2 we should <u>familiarize ourselves with our previous work</u> on FSPARSE2--though different in some ways, the problems share many similarities. As in our development of FSPARSE2, we should have in mind a specific performance bound as a goal; many signs point to $\Theta(N \log N)$ as a reasonable goal. As we analyze our failure in reaching that goal we see that it is the same that led to SPARSE2's $\Theta(N \log N)$ running time: poor choice of cut line could lead to all N points being within δ of ℓ. Because PAIR2 suffers from the same problem as SPARSE2, it is prudent for us now to seek the same cure, namely a strategy for intelligent choice of the cut line.

Our task now is to develop a cut plane selection strategy which will turn PAIR2 into FPAIR2. To fully employ what we learned in turning SPARSE2 into FSPARSE2 we should ask two questions about the CP and SFR problems: *How are the problems similar*? *How are the problems different*? The problems share many similarities. In both cases the goal is to find a cut line with $O(N/\log N)$ points near it. In both cases cases the collection of points is guaranteed to be sparse after the subproblems are solved. In both cases the sparsity constant c is known at the invocation of the procedure ($c = 2 \cdot 3^2 = 18$ for PAIR2). The crucial difference is that in SPARSE2 we knew the sparsity radius δ at the invocation of the procedure whereas with PAIR2 we learn δ only after the subproblems have been solved. This is indeed an important difference, for we must choose the cut line in order to divide the set into the subproblems to be solved.

We must develop, therefore, a cut line selection strategy that is promised a value of δ, but does not know that value at the time it executes. We should try to use as much of our selection strategy from SPARSE2 as possible. The essential step in that process (scanning until the distance between points $2cN^{1/2}+1$ apart on the list was greater than δ) is impossible without *a priori* knowledge of δ. At this point the author was forced to rely on intuition, and the following idea arose. Instead of scanning until we find an "interpoint span" (as we might call the distance between points $2cN^{1/2}+1$ apart on a projected list) of at least 2δ, and then stopping, why don't we just scan all lists, find the maximum "interpoint span", then use the center of that "interpoint span" as our cut line. As before, we insist that both A and B contain at least N/8 of the points, and we conjecture that when a value for δ is finally known, at most $2cN^{1/2}+1$ points will lie within δ of ℓ.

Once one conjectures the above, it is quite easy to prove. Assuming the nagative (that is, that more than $2cN^{1/2}+1$ points lie within δ of ℓ) leads to the same contradiction as in the proof of Theorem 3.1-2. Let us call the length of the maximum interpoint span m. Assuming that more than $2cN^{1/2}+1$ points lie within δ of ℓ implies that $m < 2\delta$; this is illustrated in Figure 3.2-3. Thus the length of the maximal interpoint span is less than 2δ. From this it immediately follows that every collection of $2cN^{1/2}+1$ points projects onto a length of at most 2δ; if it projected onto a greater length, then the stated span would not be maximal. We can therefore assert that every collection of $2cN^{1/2}+1$ points projects onto an interval of length at most 2δ. But this, along with sparsity, was the essential step in the proof of Theorem 3.1-1; the rest of that theorem follows from this. Thus we have shown that at most $2cN^{1/2}+1$ points will lie within δ of ℓ (after δ is found) when ℓ is chosen in the above way.

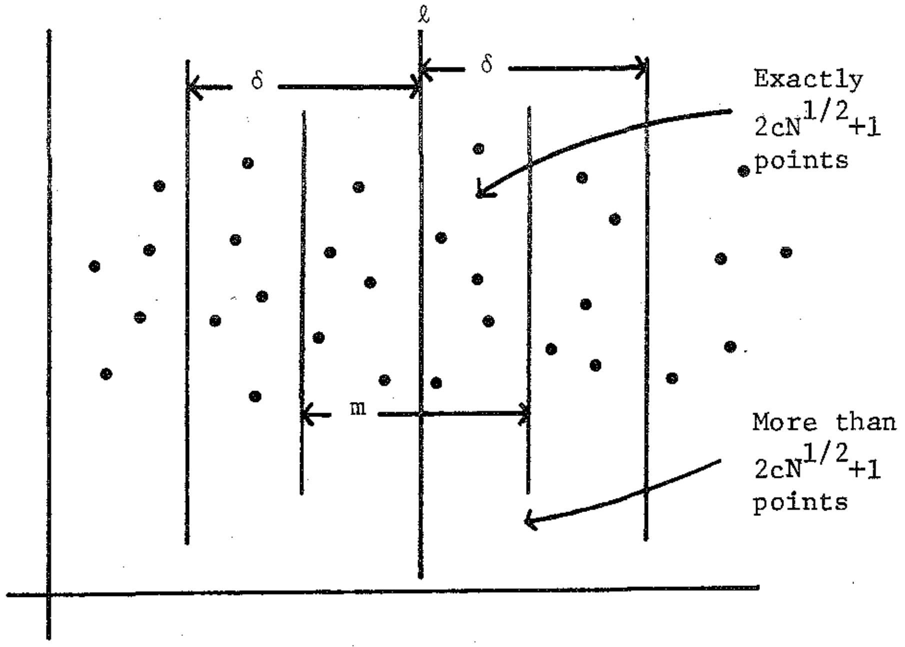

Figure 3.2-3: The maximum interpoint span

There is a subtlety about the above method that deserves study. It is interesting that we can choose our cut line before we know the value of δ. We observe here the value of following the development of FSPARSE2. We certainly could not have made the conjecture we did without the basis of our experience with FSPARSE2. With that experience, however, the conjecture was quite natural, and we were even able to "borrow" most of its proof.

We are now equipped to give a high level description of FPAIR2. For the sake of brevity, however, we will not actually give such a description in this thesis. FPAIR2 is the obvious mixing of PAIR2 and FSPARSE2; we will proceed now to develop and describe FPAIRk.

In our development of FPAIRk we will rely heavily on both FPAIR2 and PAIRk. Figure 3.2-4 shows how we can view those as two orthogonal directions in a "development vector space" which we now combine to yield a fast algorithm in k-space. Our task can be viewed as modifying PAIRk to employ a sophisticated cut plane selection strategy. Our experience with FPAIR2 and FSPARSEk immediately suggests a strategy: scan all dimensions, looking for the maximum interpoint separation of $kcN^{1-1/k}$ points, then choose the center of that interval for the cut plane (while, of course, guaranteeing that both A and B contain at least N/4k points). One can prove that such a selection strategy yields a cut plane within δ of which there are at most $kcN^{1-1/k}$ points after δ is known; the essential step in the proof is the same as that in the proof of FPAIR2.

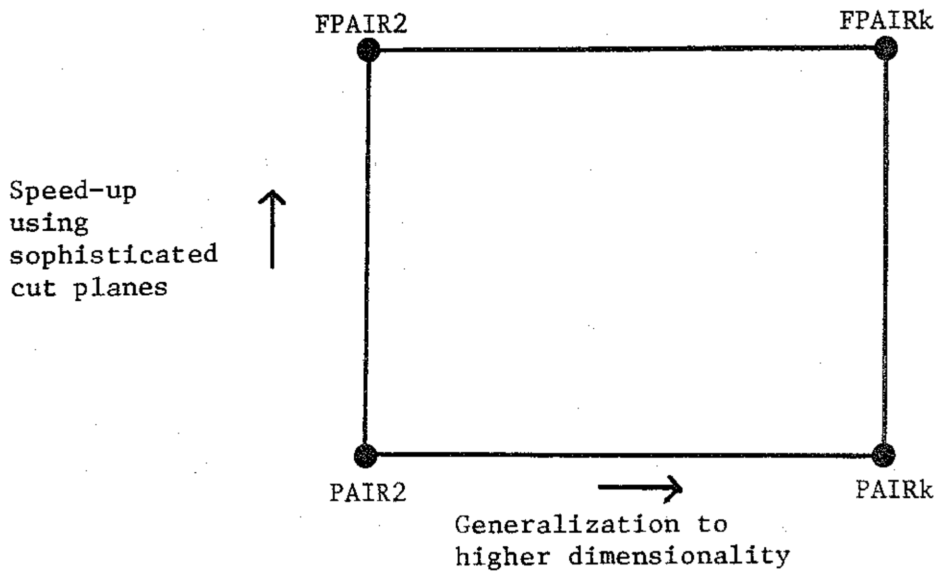
FPAIR2
FPAIRk
Speed-up using sophisticated cut planes
PAIR2
PAIRk
Generalization to higher dimensionality

Figure 3.2-4: An algorithm development vector space

We are now ready to describe FPAIRk as Algorithm 3.2-2. As with FSPARSEk, it assumes that the representation of point sets includes a sorted list of the projection of the set in each dimension. This can be accomplished by presorting at a cost of $\Theta(kN \log N)$ time and $\Theta(kN)$ storage used to represent the lists.

<u>Algorithm 3.2-2</u>

```
Procedure FPAIRk(set F)
1.  If $|F| = 2$ then return the interpoint distance.                $c_1$
2.  Choose a k-1 dimensional hyperplane P partitioning                $c_2kN$
    F into two point sets A and B in the following way.  For
    each of the k dimensions, scan down the list of the points
    sorted by that dimension from the (N/4k)-th element to the
    (N-N/4k)-th element.  In the scan keep two pointers active,
    one $kcN^{1-1/k}+1$ elements ahead of the other.  For each pair
    examined, calculate the distance between them.  Record the
    maximum interval so encountered, and use the center of that
    interval as the value defining the cut plane P.
3.  Use the procedure recursively to find                           T[N/4k]
       $\delta_A \leftarrow$ FPAIRk(A), and                         + T[N(1-1/4k)]
       $\delta_B \leftarrow$ FPAIRk(B).
    (Note that this requires maintaining the sorted lists.)
4.  Set $\delta \leftarrow \min(\delta_A, \delta_B)$.                $c_3$
5.  Project all points within $\delta$ of P onto P; call              $c_4N$
    this set S.  Note that $|S| \le kcN^{1-1/k}$ and that S is
    sparse for $c = 2 \cdot 3^k$.
6.  Use FSPARSE(k-1) (S) to enumerate all pairs                  SFR($\Theta(N^{1-1/k})$,k-1)
    within $\delta$ in S.  Check for any pairs enumerated
    that are within $\delta$ in the space and on different
    sides of P.  Let $\varepsilon$ be the distance between the
    closest pair enumerated.
7.  Return $\min(\delta, \varepsilon)$.                               $c_5$
```

$$T(2) = c_1$$

$$T(N) = T[N/4k] + T[N(1-1/4k)] + \Theta(kN) + SFR[\Theta(N^{1-1/k}),k-1]$$

$$= T[N/4k] + T[N(1-1/4k] + \Theta(kN)$$

$$\therefore T(N) = \Theta(kN \log N)$$

What lessons can we learn from our development of FPAIR2? An important principle of algorithm development which we employed was to <u>follow closely the development paths of algorithms for related problems</u>. We should adhere closely to this principle in our work on the ANN problem. The following observation is a tool for multi-dimensional algorithm design which allowed us to use our work on the SFR problem to help us in the CP problem: Although sparsity is not present in the original problem, it can be induced in sub-problems.

3.3 All nearest neighbor algorithms

The all nearest neighbor problem is the topic of this section. The output of an ANN algorithm is to be N pairs of points. The first element of the pairs will range over all N points, and the second element will be the closest point among the N to the first element (with ties broken arbitrarily).

By now it should be almost natural to start with the one dimensional case, because it is the most simple. After sorting the points to form a sorted list, the nearest neighbor of each point is either the right or left neighbor in the sorted list. The running time of such an algorithm is dominated by sorting; thus the algorithm described above shows $ANN(N,1) \leq \Theta(N \log N)$.

As we examine the planar problem it is obvious that the straight-forward iterative procedure requires $\Theta(N^2)$ time. Both our previous experience in this area and the fact that the problem is $\Theta(N \log N)$ on the line motivates us to look for a divide and conquer approach that will yield a faster algorithm. As we observed in Section 3.2, it will be very beneficial to our development to follow closely the development paths of our SFR and CP algorithms.

We will now attempt to synthesize an ANN algorithm for the plane, which we will call ALL2. In this synthesis we will use both insights and components from SPARSE2 and PAIR2. Our resulting divide and conquer algorithm will have three stages: break the problems into subproblems, solve the subproblems, then combine the solutions to the subproblems to yield a solution to the ANN problem. The first two stages should be fairly obvious to us by now. The divide stage will consist of choosing a vertical line ℓ which divides the points into two almost equally sized subcollections A and B. The second stage will consist of finding all nearest neighbors for all points in A and B (recursively).

Before we look for a way to combine these subsolutions to form a solution to the whole, we should ask what properties are true of the subsolutions? We will probably need to know certain crucial properties to be able to combine the answers, and observing the problem at this stage with an unbiased eye might help us to observe properties we would miss later on. From our previous experience we are most tempted to look for a property of sparsity. We have no fixed radius to work with as in the CP and SFR problems, but is some other type of sparsity present? We observe that in the CP and SFR problems, we had available a single distance that characterized sparsity for the whole space in the solution to the problem (namely δ for the SFR and the minimum interpoint distance for the CP problem). For the ANN, however, no such global distance is available; instead we have a local distance for each point, namely the distance to its nearest neighbor. Whereas before we described sparsity in terms of δ-balls, where δ was the same over all the space, for the ANN problem we should seek a definition of sparsity in terms of spheres of varying radii. It seems feasible to expect that the mathematical construct appropriate to describe the sparsity in the ANN problem is what we might call the NN-ball. We define the NN-ball for point X to be the closed ball of center X and radius equal to the distance to X's nearest neighbor.

We now look for a condition that can be predicated of the space in terms of NN-balls. To fully employ the similarity present in the problems, let us recall the definition of sparsity in SFR in terms of δ-balls. For the SFR problem we asserted that no δ-ball in the space contained more than some constant c points in the file (or "file points", as we call them to distinguish them from an arbitrary point in the space). Notice that this is equivalent to saying that no point in the space is contained in more than c δ-balls centered at file points. This viewpoint helps us to make the following conjecture: No point in the plane is contained in more than some constant c NN-balls. By analogy with the CP problem, we might also conjecture that $c = 9$.

Once we have conjectured the above it is easy to prove its truth. Recall that we are dealing with a collection of N points in the plane, and we know the nearest neighbor of each. To help the reader's insight one can view the NN-ball corresponding to each point in the file as the smallest rectilinearly oriented square centered at that point that contains another file point. A collection of points and their corresponding NN-balls are depicted in Figure 3.3-1. What we must prove now is that no point x in the file is contained in more than 9 such NN-balls. We will first show that if point x is contained in the NN-ball of point y, and point y is in quadrant 1 of x (this will hold true for any quadrant), then x is contained in no other NN-ball of a point in quadrant 1. We illustrate this in Figure 3.3-2. No other file point can be in y's NN-ball; if it were, then y's NN-ball was not accurately calculated. Thus any other NN-ball with center in quadrant 1 which contains x must have its center in the shaded area marked S. But any L_∞ ball centered in S and containing x must also contain y, in which case it cannot be an NN-ball. Thus we have shown that x can be contained in at most one NN-ball centered in each quadrant. It is clear that x can be contained in at most one NN-ball centered on each ray dividing the four quadrants, and finally x could be contained in an NN-ball centered at x itself. Thus there are at most nine positions in which points whose NN-balls contain x could lie: in the four quadrants, on the four rays dividing quadrants, and on x itself. That a point can actually be in nine NN-balls is illustrated in Figure 3.3-3; point x lies in the NN-balls centered at each of the nine points pictured. We state this observation formally as Theorem 3.3-1.

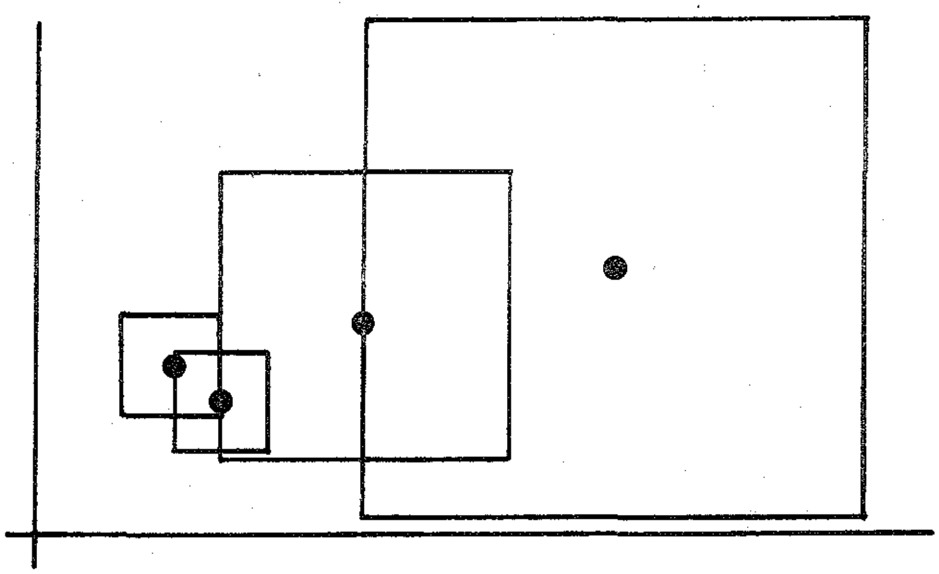

Figure 3.3-1: A collection of NN-balls

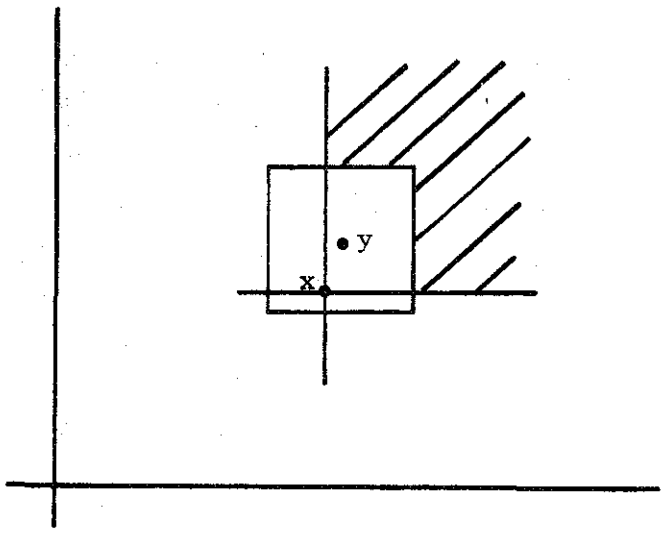

Figure 3.3-2: The NN-ball with center y is the only enclosing NN-ball in quadrant 1 of x

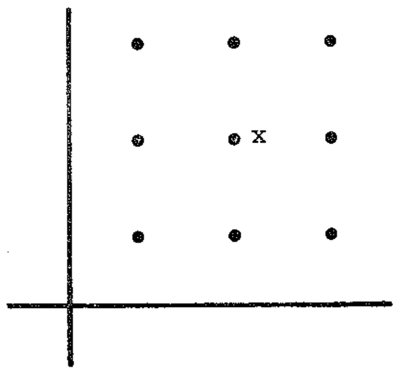

Figure 3.3-3: Point x is contained in 9 NN-balls

Theorem 3.3-1: Given a collection of N file points in the plane, no point in the plane lies in more than nine NN-balls. Furthermore, it is possible for some point in the plane to lie in nine NN-balls.

Equipped with this theorem we are prepared to synthesize the combining stage of Algorithm ALL2. Recall that the first two stages of ALL2 have divided the points in the file into two collections A and B and all nearest neighbor pairs in A and B have been found. Assume that for each point x in A we associate with x its nearest neighbor in A and likewise for all points in B. The recombining stage must check and see for all points x in A if there is any point y in B that is nearer to x than x's nearest neighbor in A (which has been recorded), and likewise for all points in B. This situation is illustrated in Figure 3.3-4. Points 1 and 2 in A and points 3 and 4 in B are shown with the NN-balls as determined in the "intraset" calculations. The only mistake that needs to be corrected is the fact that 3's nearest neighbor is 2, not 4. Thus we see that the recombining stage can be accomplished by first checking for every point x in A if there is any point y in B nearer to x than x's nearest neighbor in A, then doing the same for all points in B.

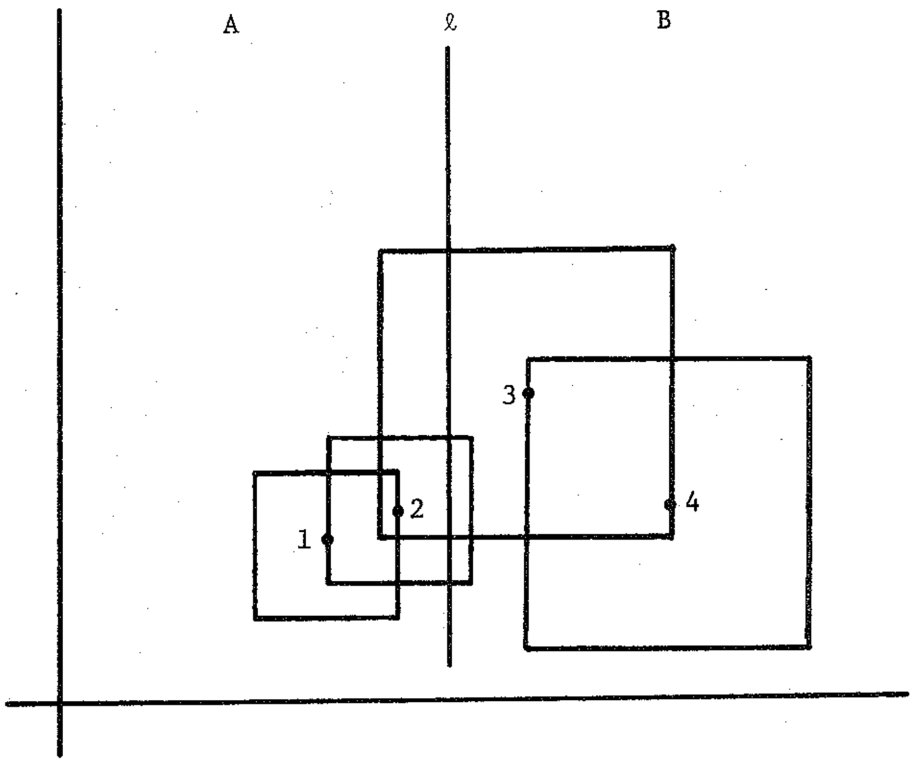

Figure 3.3-4: Two solved subproblems

Since these problems are obviously symmetric, we do well to concentrate our attention on only one of them; after we solve that we can apply its solution immediately to the other. We will concentrate on checking for all points in A if there is any point in B nearer to that point than its calculated nearest neighbor in A.

This situation is illustrated in Figure 3.3-5. Points 1 through 5 are in A and points 6 through 10 are in B. The NN-balls for the subsolution of A are shown. The mistakes in A that need to be corrected are that 6 is the nearest neighbor to 3, and 7 is the nearest neighbor to both 4 and 5 (because they lie in the sub-solution NN-balls). We can observe a number of things about the recombining process from Figure 3.3-5. It was not necessary to check for points in B lying in the NN-balls belonging to 1 and 2 because neither of their NN-balls intersected ℓ. In the projection of points in B onto ℓ, both points 6 and 7 lie in the NN-balls of point 3 and points 4 and 5, respectively.

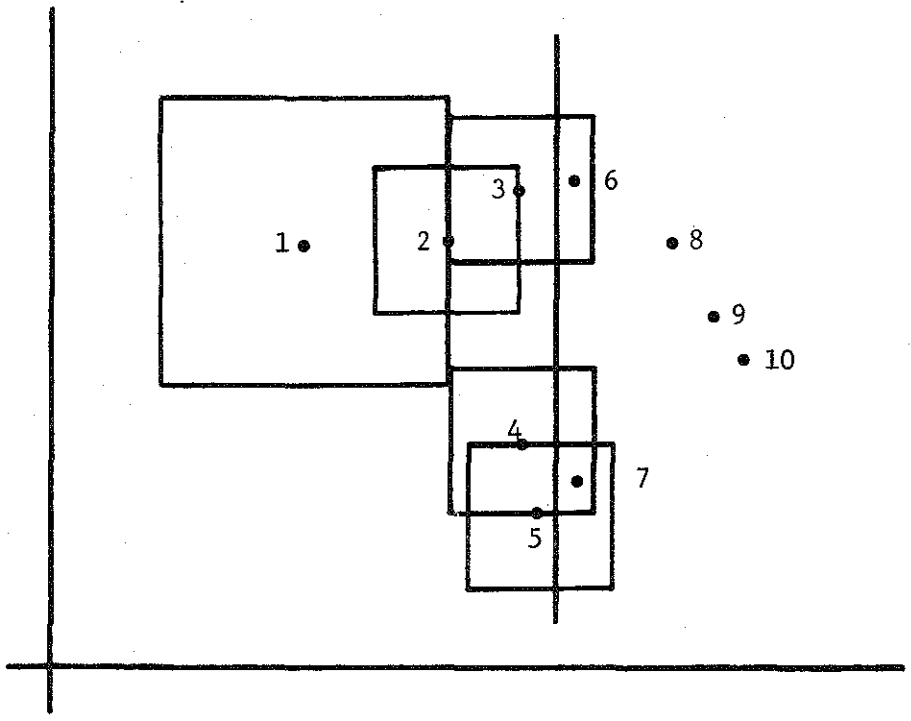

Figure 3.3-5: One solved subproblem

Having made these observations we are ready to construct an algorithm to combine the subsolutions. First project every point in B onto ℓ, then project all NN-balls in A which overlap ℓ onto ℓ. In the example, the NN-balls for points 3, 4 and 5 are projected onto ℓ. By Theorem 3.3-1 no point on ℓ was in more than nine NN-balls from A; this condition is not changed by the projection (because we project only NN-balls which intersect ℓ). Therefore none of the projected points in B lies in more than nine projected NN-balls. After having made the projections we sort the projection of the points in B and the end-points of the projections of the NN-balls in A. We then make a linear scan down the sorted lists, for each point in B having to check at most nine projected NN-balls in A. For every ball and point pair checked, determine if the unprojected point lies in the unprojected ball, and if so, modify the ball's nearest neighbor. Since for each point in B we have to check only at most a constant number of balls in A, the scan will take time linear in N (after sorting). Thus the total cost of the combining stage is $\Theta(N \log N)$.

Now that we have the three stages of the divide and conquer algorithm ALL2, we can describe it formally as Algorithm 3.3-1.

Algorithm 3.3-1

```
Procedure ALL2(set F)
1. If $|F| = 1$, then assign as the nearest neighbor to the        $c_1$
   point in F a "dummy" point infinitely far from all points.
2. Choose a vertical line $\ell$ partitioning F into two            $c_2N$
   collections of N/2 points each, A and B.
3. Solve the subproblems by the calls ALL2(A) and                  $2T(N/2)$
   ALL2(B). Now we have noted for every point x in A the
   closest point among A to x, and likewise for all points
   in B.
4. Repeat steps 5 through 7 twice, the first time just
   as they are written, and the second time interchanging
   the roles of A and B.
5. Project every point in B onto $\ell$. Project every              $c_3N$
   NN-ball in A that overlaps $\ell$ onto $\ell$. (Such a ball is
   represented by its two end points.)
6. Sort the above projections.                                     $\Theta(N \log N)$
7. Scan down the lists simultaneously. As each new                 $c_4N$
   point in B is scanned, keep track of what balls are
   either left or entered. (This will require constant
   time, because at most nine could be left or entered
   each step.) Then for every point y in B and every
   ball containing y, check to see if y is closer to the
   center of that ball than its nearest neighbor so far.
   If so, make y the new nearest neighbor to that point.
8. Now that any mistakes made have been "patched up",              $c_5$
   return. For every point x in F we have recorded its
   nearest neighbor in F.
```

$$T(1) = c_1$$

$$T(N) = 2T(N/2) + \Theta(N \log N)$$

$$\therefore T(N) = \Theta(N \log^2 N)$$

What general methods for multidimensional algorithm design did we employ in our construction of ALL2? The general schema we used was the same as that for SPARSE2 and PAIR2: solve a problem of N points in the plane by dividing it into two subproblems of N/2 points each in the plane and one subproblem of up to N points on the line. The subproblem on the line that we solved was of a type we have not seen previously; it might be called a "sparse local fixed radius" search. Each point had its own search radius, but a sparsity of sorts was guaranteed. We should keep this problem in mind; we should look for it in a generalized form as we develop the ANN algorithm for higher dimensions.

Before we generalize ALL2 to higher dimensions, we should note that the asymptotic running time of the procedure could be decreased by the use of presorting. It is obvious that the presorting of the points in B in Step 6 would make unnecessary one sort of Step 7. Likewise the sort of Step 7 for the balls would be unnecessary. Because at most nine balls can overlap any point, the farthest the end points of the balls would have to "sift down" in an already sorted list is nine elements. Therefore the presorted list of centers of the NN-balls could be transformed in linear time into a sorted list of end points. Thus both sorts of Step 7 are made unnecessary by presorting and the additive term in the recurrence relation becomes linear, yielding a $\Theta(N \log N)$ algorithm. Thus we have shown that $ANN(N,2) \leq \Theta(N \log N)$.

We are now ready to attempt to develop ALL3, an algorithm for the ANN problem in 3-space. The first two stages of the divide and conquer algorithm are obvious. The divide stage chooses a cut plane P which divides the set F into two equally sized points sets A and B. The recursive stage will then find all nearest neighbors among A and likewise for B. At this point we ask what property can be predicated of the subsolutions, and we arrive at a generalization of Theorem 3.3-1, which we express as Theorem 3.3-2.

Theorem 3.3-2: Given a collection of N file points in 3-space, no point in 3-space lies in more than 27 NN-balls. Furthermore, it is possible for some point in 3-space to lie in 27 NN-balls.

Proof: The proof proceeds in a way similar to that of Theorem 3.3-1. One can show that at most one NN-ball centered in each octant defined by a point can overlap that point, and likewise for the planes separating the octants. A final NN-ball could have its center on the point itself, thus showing the upper bound of 27. That 27 is attainable is shown by considering points on a rectangular lattice in 3-space. □

With this theorem we are ready to synthesize the combining stage of ALL3. We must first locate any points in B that are in NN-balls of A, and then do the same, switching A and B. Since the problems are symmetric, let us consider only the first. The obvious way to proceed to solve this problem is to project all points in B onto P and then project all NN-balls that overlap P onto P. After this projection we can locate all points in NN-balls and see if the inclusion holds in 3-space, and if so modify the appropriate nearest neighbors. (Note that since each point can be in at most 27 balls, the amount of "patching up" to be done is at most linear.)

Let us now focus our attention on the reduced problem in the plane. We are given N/2 points and up to N/2 balls. We know that any point in the plane is in at most 27 balls. We are asked to enumerate all points that lie within a ball and tell in which balls they lie. Dr. Gideon Yuval has suggested that this be called the "Territorial Waters" problem due to the similarity to the international situation of each country claiming a different radius as its territorial limits. We will refer to this subproblem as the "STW" for "Sparse Territorial Waters". We will find it tractable for the parameter N of the STW problem to have the significance that the number of points plus the number of balls in the problem is N; then having exactly N/2 points and N/2 balls becomes a special case. We further assume that each ball is described by a point and a radius.

We will call the algorithm for the STW problem in the plane TER2. The first stage of TER2 will choose a cut line ℓ such that a total of N/2 points and centers of balls are to the left of ℓ and a total of N/2 points and centers are to the right of ℓ. (Note that in this stage we do not distinguish between a point in the space and a point which defines the center of a ball.) The recursive stage of TER2 will solve the subproblems for A and B. The combining stage must enumerate all points in B which are in a ball in A and likewise for balls in B and points in A. But note that this combining problem is exactly the combining stage we faced in ALL2! Hence we can use the same process we used there.

We now describe ALL3 and TER2 formally as Algorithms 3.3-1 and 3.3-2.

Algorithm 3.3-2

```
Procedure ALL3(set F)
1. If $|F| = 1$, then assign as the nearest neighbor to the point in F a "dummy" point infinitely far from all points.    $c_1$
2. Choose a cut plane P partitioning F into two collections of N/2 points each, A and B.    $c_2N$
3. Solve the subproblems by the calls ALL3(A) and ALL3(B). Now we have noted for every point x in A the closest point among A to x, and likewise for all points in B.    $2T(N/2)$
4. Repeat Steps 5 through 7 twice, the first time just as they are written, and the second time interchanging the roles of A and B.
5. Project every point in B onto P. Project every NN-ball in A that overlaps P onto P (such a projection is represented by a point and a radius). Note that the balls have the property that any point in the plane is in at most 27 balls.    $c_3N$
6. Let S be the collection of points and balls projected in Step 5. Call TER2(S) to enumerate all points which lie within balls (note that there will be at most 27N, because each point can lie in at most 27 balls).    $STW(N,2)$
7. For each point and ball enumerated in Step 6 see if the point is in the corresponding ball in 3 space. If so, modify that point's recorded nearest neighbor.    $c_4N$
8. Return.
```

$$T(1) = c_1$$

$$T(N) = 2T(N/2) + \Theta(N) + STW(N,2)$$

$$= 2T(N/2) + \Theta(N \log^3 N)$$

$$\therefore T(N) = \Theta(N \log^2 N)$$

Algorithm 3.3-3

```
Procedure TER2(set F)
1. If $|F| = 1$, then return.                                  $c_1$
2. Choose a vertical line $\ell$ partitioning F                $c_2N$
   into two collections of N/2 points and
   balls each, A and B.
3. Solve the subproblems by the calls TER2(A)                 $2T(N/2)$
   and TER2(B).
4. Locate points in balls which have their                    $\Theta(N \log N)$
   centers on the other side of $\ell$ using Steps 4
   through 7 of Algorithm 3.3-1. The only change
   necessary is to replace "nine" by "27"
   (and instead of "patching up" discovered
   near neighbors, merely enumerate them; they
   will be "patched up" by ALL3).
5. Return.                                                     $c_3$
```

$T(1) = c_1$

$T(N) = 2T(N/2) + \Theta(N \log N)$

$\therefore T(N) = \Theta(N \log^2 N)$

Because we used the components from ALL2 in our construction of TER2, we can apply the same speedup of presorting. Presorting will reduce the combining overhead of TER2 from $\Theta(N \log N)$ to $\Theta(N)$, and thus reduce the total running time from $\Theta(N \log^2 N)$ to $\Theta(N \log N)$. If we use this modified version of TER2, the running time of ALL3 is decreased to $\Theta(N \log^2 N)$. Thus we have shown that $\text{ANN}(N,3) \leq \Theta(N \log^2 N)$.

Examination of the schema employed in the construction of ALL3 shows that it is similar to all of our other algorithms: To solve a problem of N points in 3-space, solve two problems of N/2 points in 3-space, then two problems of up to N points in 2-space. The subproblems in the plane are of a new type, which we have called the "Sparse Territorial Waters" problem. Notice that the subproblem we encountered in the combining stages of ALL2 and TER2 is merely a one dimensional STW problem. We therefore make the important observation that the way the TER2 algorithm works is to solve two STW problems of N/2 points in the plane and then two STW problems of up to N points on the line.

We are now prepared to build Algorithm ALLk, using as examples ALL2, ALL3, and PAIRk. The first stage of ALLk will partition the file into two subsets A and B by a k-1 dimensional hyperplane P. The second stage will find all nearest neighbor pairs for A and B recursively. At this point the k dimensional analog of Theorem 3.3-2 will hold, namely that no point in the space is contained in more than 3^k NN-balls from either A or B. The combining stage of ALLk will consist of two calls on TER(k-1), which will solve the STW problem of the subsets projected onto P.

Algorithm TERk is the obvious generalization of TER2. The first stage divides the collection into two almost equal sized sets (of points and balls). The second stage solves the subproblems recursively, and the third stage makes two recursive calls on TER(k-1) to solve the subproblems. The recurrence relation describing this algorithm is identical to that describing SPARSEk, so the running time of TERk is $\Theta(N \log^k N)$. If the speedup trick of presorting is applied, the running time becomes $\Theta(N \log^{k-1} N)$ for $k \geq 2$. Using this as a tool for ALLk, the running time of ALLk is $\Theta(N \log^{k-1} N)$. Algorithms TERk and ALLk are so similar to Algorithms TER2 and ALL3 that it is unnecessary to present them formally.

The existence of Algorithm TERk with speedup by presorting shows that $\text{ANN}(N,k) \leq \Theta(N \log^{k-1} N)$ for $k \geq 2$. The divide and conquer schema used is the same as for the CP problem in k-space. The subproblem to be solved in the dividing hyperplane is the STW problem, a new closest point problem.

The next step in the development of an ANN algorithm is to attempt to reduce the running time of ALLk even further. Both the performance of $\Theta(N \log N)$ in the plane and the empirically observed average case performance of $\Theta(N \log N)$ of the algorithm of Friedman, Bentley, and Finkel [1975] would make us guess that $\text{ANN}(N,k) \leq \Theta(N \log N)$. From our experience with FSPARSEk and FPAIRk we would guess that the appropriate next step in the development is to try to find a strategy which uses intelligent choice of cut planes. Even with all of this help, though, the author is still unable to develop a faster algorithm for the ANN problem in k-space. This discussion of the ANN problem must therefore end here, and further development is left as an exercise for the reader.

3.4 Lower bounds

How does an algorithm designer know when he has finally "solved" the problem on which he is working? One suitable definition of a solved problem is that one has given matching lower and upper bounds on the complexity of the problem. In this section we develop some lower bounds for the problems we have examined in this chapter.

There are many techniques for the construction of lower bounds. Two of the most popular are "oracles" (or "adversaries") and information theoretic arguments. Reingold's survey [1972] mentions a number of other techniques for lower bound construction. Perhaps the most valuable technique, however, of lower bound construction for the practicing algorithm designer is "reducibility". A typical reducibility argument runs something along the following lines: It is known that F(N) is a lower bound on problem X. If one could solve problem Y in less than G(N) time then one could use that algorithm to solve problem X in less than F(N) time. Since it is known that one can not solve problem X in less than F(N) time, however, G(N) must be a lower bound on the amount of time required to solve problem Y. We will see this technique applied shortly.

The specific lower bound which we will employ in our arguments was derived by Dobkin and Lipton [1975]. They showed that in the worst case the problem of determining if all the elements in an ordered set of N elements are unique must require $\Theta(N \log N)$ comparisons, and therefore $\Theta(N \log N)$ time on a RAM/RASP. Shamos and Hoey [1975] observed that this implies that the problem of determining the two closest among N points on a line requires $\Theta(N \log N)$ time in the worst case (for if the two closest elements are distance zero apart, then the elements are not unique).

We will now use reducibility to show a lower bound of $\Theta(N \log N)$ on CP(N,k). Assume $CP(N,k) < \Theta(N \log N)$. To find the two closest among N points on a line, imbed the line in k-space then call the fast CP algorithm. Imbedding requires only linear time, so the whole algorithm would run in $\Theta(\max\{CP(N,k),\Theta(N)\})$ time which by assumption is less than $\Theta(N \log N)$. But such an algorithm is impossible by the lower bound of Shamos and Hoey, so we have shown a lower bound of $\Theta(N \log N)$ for the CP problem. We can use this bound immediately to show a lower bound of $\Theta(N \log N)$ on the ANN problem; if we had a faster algorithm for ANN then we could just scan all N near neighbors and find the closest, giving a faster CP algorithm which is impossible. It is important to remember that these are lower bounds on the worst-case complexities of the problems; indeed, Rabin [1976] has shown that $\overline{CP}(N,k) = \Theta(N)$. Our lower bound on CP(N,k) together with Algorithm FPAIRk establishes the fact that $CP(N,k) = \Theta(N \log N)$ for fixed k. We are still unable to close the gap $\Theta(N \log N) \le ANN(N,k) \le \Theta(N \log^{k-1} N)$.

We turn our attention now to the SFR problem. It seems to be difficult to establish lower bounds for arbitrary values of δ, but the degenerate case of $\delta = 0$ is easier. We thus observe that <u>examining degenerate cases</u> is important in proving lower bounds. If we let $\delta = 0$ and use the sparsity constant $c=1$, then an SFR algorithm can solve the element uniqueness problem as follows. The algorithm first imbeds the line containing the elements in k-space and then calls the SFR algorithm. If the elements are unique, then the algorithm will return in SFR(N,k) time with no pairs within δ observed. If the elements are not unique, however, it might take longer since sparsity (which the algorithm assumes) was violated. What one must do, therefore, is monitor the steps of the algorithm, and if it is taking longer than expected, halt it and return the answer that the elements are not unique. The monitoring can be accomplished by adding at most some constant cost c at each step, so all together the monitored algorithm would take at most c·SFR(N,k) time, which is Θ(SFR(N,k)). Since we can solve element uniqueness in Θ(SFR(N,k)) time, we have shown that SFR(N,k) $\geq \Theta$(N log N), and therefore Algorithm FSPARSEk is optimal. (At least for the case of $\delta = 0$, and therefore among all truly general SFR algorithms.)

We have seen here some important general techniques for the use of lower bounds in algorithm design. The first is the method of reduction for proving lower bounds. We also employed the trick of imbedding a one dimensional problem in a higher dimensional space. Finally, the lower bounds which we derived showed that some of our algorithms are optimal and that further attempts to speed up their asymptotic running times must prove vain.

4. Extensions

In this chapter we describe how the algorithms given in Chapter 3 can be extended to solve a broader class of problems. In Section 4.1 we investigate extending the algorithms to employ other metrics. Section 4.2 deals with the application of the algorithms of Chapter 3 to generalizations of the problems studied in that chapter.

4.1 Different metrics

All of the algorithms developed in Chapter 3 were based on the L_∞ metric; in this section we show that the algorithms are applicable for other metrics as well. We give three criteria which the metric must meet for the algorithms of Chapter 3 to be valid, and then show that the criteria are met for the L_2 metric. The criteria seem to be applicable for any L_p metric, though proving applicability for a particular p can be laborious.

The first criterion which the metric must meet is that projection preserves sparsity. Specifically we must show that if k-space is sparse and if all points within distance δ of a k-1 dimensional hyperplane P are projected onto P, then the projection will be sparse. This was a crucial factor in Steps 4 of Algorithms 3.1-1, 3.1-2, and 3.1-3. For the L_∞ metric the sparsity constant c was preserved in the projection; for the L_2 metric it may increase. Let us examine the planar case using the illustration in Figure 4.1-1. All points that will be projected onto the one dimensional

δ-ball (which is the 2δ segment of the line ℓ contained in the square S) are necessarily contained in S. But the four δ-balls also contain S, and since none of the four contain more than c points (by sparsity), S as a whole can not contain more than 4c points. Therefore the projection of S onto the line will contain at most 4c points, so sparsity is preserved although the sparsity constant is increased. This same method of proof will work for any higher dimensional space, since fixed radius spheres can be used to cover any hyper-solid in k-space (in this case a hyper-cylinder).

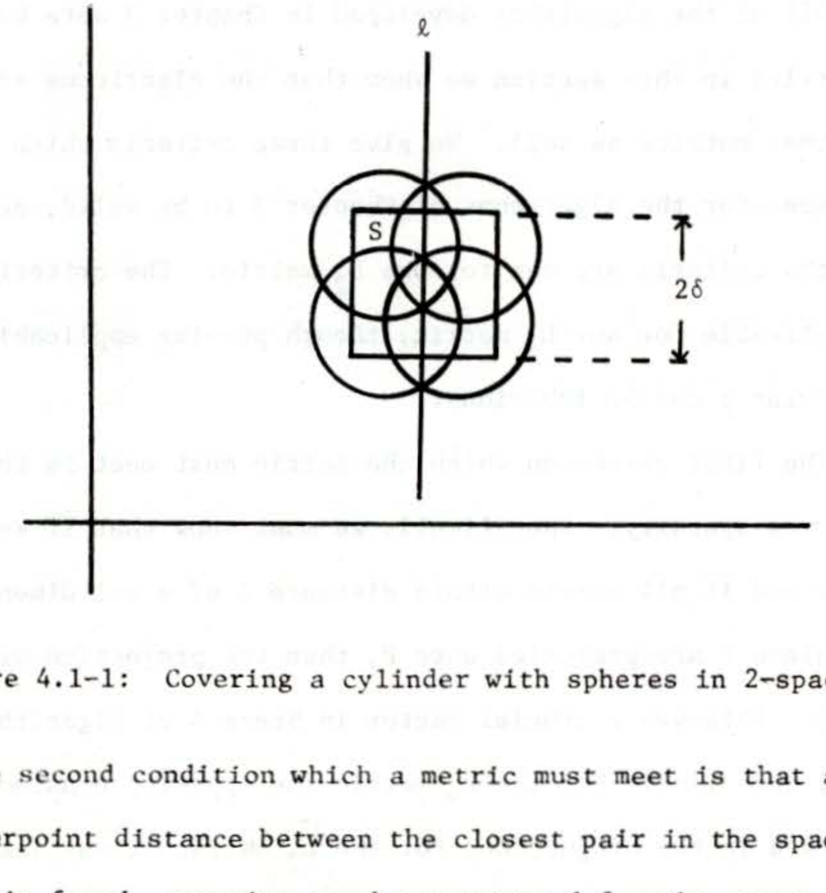

Figure 4.1-1: Covering a cylinder with spheres in 2-space

The second condition which a metric must meet is that after the interpoint distance between the closest pair in the space (say δ) is found, sparsity can be guaranteed for the space. This was necessary for Steps 5 of Algorithms 3.2-1 and 3.2-2. Let us assume for brevity that $\delta = 2$; then we know that no two points in the space are closer than distance 2 together. We must now show that no 2-ball in k-space can contain more than some constant S_k points (and we will further choose S_k so that it is attainable).

A crucial step in our proof is a result from the "Sphere Touching" problem. That problem asks how many unit spheres can be made to touch a given unit sphere in k-space with no spheres overlapping (the spheres are defined by the L_2 metric). We will call the answer to that question T_k (for the number of touchings possible in k-space); lower and upper bounds on T_k are contained in Leach and Sloane [1971]. We assert that $S_k = T_k + 1$.

To show that $S_k \geq T_k + 1$ it suffices to observe a collection of T_k spheres touching a given sphere in k-space. Notice that the centers are all at least distance 2 apart; were they not, the spheres would overlap. Notice also that all the $T_k + 1$ centers are within distance 2 of (i.e., they are exactly distance 2 from) the point which is the center of the given sphere. Given a touching of T_k spheres we have generated $T_k + 1$ points meeting the desired criteria (i.e., they are all within a 2-ball and no two are closer than 2 together). Thus a sphere touching implies a pessimal arrangement of points (that is, one in which the sparsity constant c is realized).

We will now use sphere touching to show that $T_k + 1$ is an upper bound on the number of points that can lie in any 2-ball in a space in which all points are at least 2 apart. We will consider the case in which the center of the 2-ball we are examining is a file point and show that no more than T_k file points can lie within the ball; the other case is similar. Assume that m points lie within the ball; because they are separated from the center point by 2, they must be on the surface of the ball. Notice that this arrangement implies a sphere touching of m spheres by letting the given sphere have as its center the given point and centering the other spheres about the other m points. Since all points are separated by 2, no spheres overlap. Thus the maximum value of m is T_k; otherwise we have generated a "better" sphere touching. When we include the one point at the center of the ball, we have shown that no more than $T_k + 1$ points can lie in any 2-ball in the space. Thus we have shown that $T_k + 1$ is a suitable sparsity constant and that it is attainable.

As an example the worst case arrangement of points in the plane is illustrated in Figure 4.1-2. The value of T_2 is six; note that seven points are within the 2-ball centered at point x.

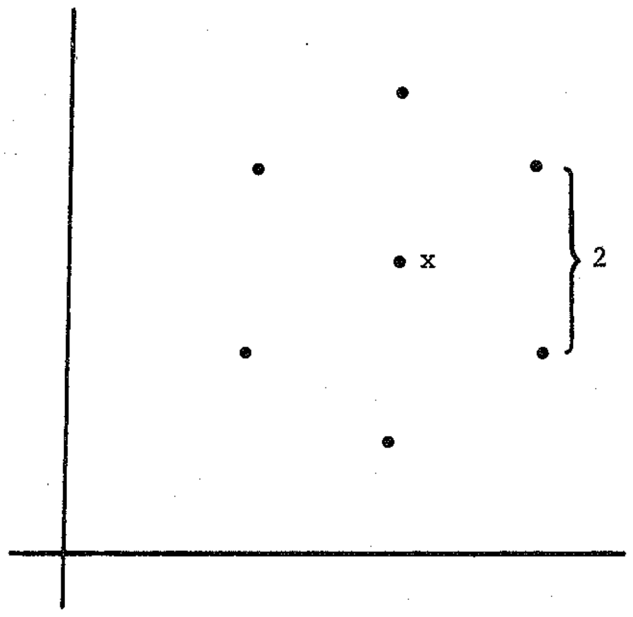

Figure 4.1-2: A worst-case arrangement of points in the plane

The final criterion which a metric must meet is the analog of the generalization of Theorem 3.3-2; that theorem states that no point in k-space is contained in more than some constant c NN-balls. To show this for the L_2 metric we will use sphere touching and employ $c = T_k + 1$ as our sparsity constant. We will give a construction showing that to every arrangement of m file points whose NN-balls overlap a given point x (which is not a file point), there exists a corresponding m-touching of spheres. Since T_k is an upper bound on sphere touching, we know that $m \le T_k$; therefore x can lie in at most T_k NN-balls (or $T_k + 1$ if x is a file point).

The touching is constructed by placing the center of the "touched" sphere at x, and placing the centers of the touching spheres at distance 2 from x, on the rays from x to each file point. To prove that the touching spheres do not overlap, we will demonstrate that the angle between any two rays is at least $\pi/3$ radians. Thus points at distance 2 from x must be at least distance 2 apart in space, showing that the spheres can not overlap. (Coxeter [1961] also used this angular definition of the problem.)

We have called the overlapped point x; let p and q be two file points. Consider the plane P in E^k which is defined by p, q and x (the argument is trivial if the points are collinear; assume they are not). The plane P is depicted in Figure 4.1-3. Let ℓ be the perpendicular bisector of the line segment px. Note that q can not be on p's side of ℓ; if it were, it would be closer to p than it is to x, and thus could not contain x in its NN-ball (since $D(q,p) < D(q,x)$). Let c be the intersection of p's NN-ball and the plane P. By definition of NN-ball, q cannot lie in the interior of c. Thus we have excluded q from both c and p's side of ℓ, so q must lie in the shaded region S.

Consider now the ray $\overrightarrow{xq}$; its angle α with xp is minimized when q is on the intersection of c and ℓ. Given this configuration, α is minimized when x is on c. At that point, $\alpha = \pi/3$ radians. Therefore we have shown that $\alpha \geq \pi/3$ radians.

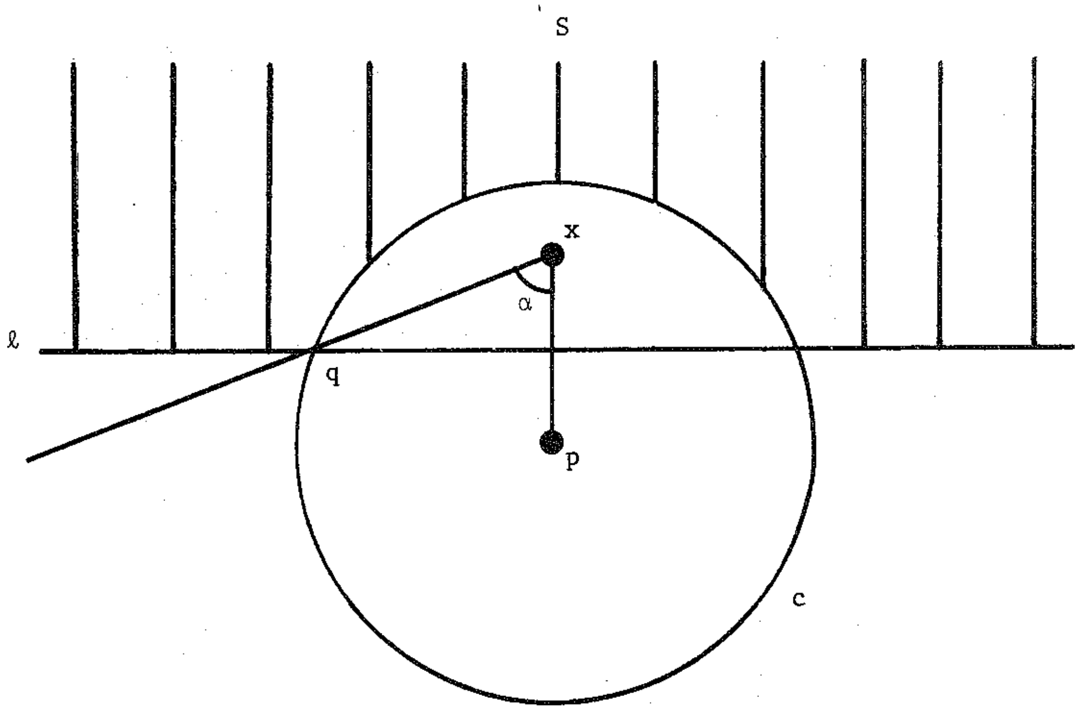

Figure 4.1-3: The plane containing p, q and x

Note that the bound of $T_k + 1$ is attainable. The pessimal arrangement for this criterion is the same as that for the second criterion of sparsity after the CP problem is solved. The pessimal planar arrangement is illustrated in Figure 4.1-2. Note also that this third criterion implies the second; we have demonstrated both here to make the proofs easier to follow.

4.2 Different problems

In this section we suggest straightforward extensions of the algorithms given in Chapter 3 to different problems. These problems are minor modifications of the problems studied in Chapter 3. We will not give complete algorithms, therefore, but rather sketch how the algorithms of that chapter might be modified to solve similar problems.

An obvious extension of the CP problem is to ask for the three points (or in general, m points) in the space with least maximal interpoint separation. Algorithm FPAIRk could be modified to find such points easily. The crucial step in the development of such an algorithm would be in showing that sparsity holds in the subsolutions.

A similar generalization of the ANN problem asks for the m nearest neighbors to each point to be enumerated. In such an algorithm the "m nearest neighbor ball" (the ball centered at a point with radius equal to the distance to its m-th nearest neighbor) would take the place of the NN-balls of Algorithm ALLk. A crucial step in the proof of the algorithm would be to show that no point in the space is contained in more than some constant number of "m-NN-balls".

Different definitions of sparsity might arise in practice from the one which we employed in our work on the SFR problem. If such definitions arose, it would not be difficult to use an alternative definition of sparsity to arrive at an algorithm similar to FSPARSEk.

In the STW problem we were given a sparse collection of hypercubes; a similar problem occurs if we are given a sparse collection of hyper-rectangles. In addition to the sparse hyper-rectangles, we are given a collection of points and asked to enumerate all pairs of points within rectangles. The algorithm TERk can be trivially modified to accomplish this. The crucial insight is to view a hyper-rectangle as a center and k "dimensional radii" (the radius in each dimension is the same for a hypercube, but differs for a hyper-rectangle).

5. Issues of implementation

Thus far in this thesis we have concerned ourselves only with asymptotic running times; in this chapter we will consider how the algorithms developed in Chapter 3 can be implemented on computers. The efficient implementation of the worst-case algorithms given in Chapter 3 is the subject of Section 5.1. In Section 5.2 we discuss how the worst-case algorithms can be modified to yield faster average-case algorithms which are more suited to many applications.

5.1 Worst-case algorithms

In this section we will describe how the algorithms of Chapter 3 could be efficiently implemented. We assume that we are dealing with an ALGOL-like language (we will take any language constructs we require from ALGOLW; see Kieburtz [1975]), but our comments should be applicable to almost any algorithmic language. We will focus our attention primarily on the ANN algorithm ALLk because it is most general and in a sense subsumes the other algorithms.

The implementation of the basic constructs used in the algorithms is quite straightforward. The set of N points in k dimension could be represented by <u>real array</u> POINTS(1::N,1::k); POINTS(i,j) contains the j-th coordinate of the i-th point. Using this scheme a point can be referred to by an integer between 1 and N (instead of having to be described by k reals). For ALLk we must tell the nearest neighbor to each point; we can hold this information in integer array NN(1::N). If the nearest neighbor to point i is point j, then NN(i) = j. We can use the array NN to keep track of the nearest neighbors found in each subsolution; at the time the entire problem has been solved NN would contain the final answer. It would probably also be expedient to maintain the real array NN_DIST(1::N) with the condition that NN_DIST(i) = DISTANCE(i,NN(i)) (that is, at termination NN_DIST(i) is the distance from i to its nearest neighbor). The array NN_DIST can serve as the radii of the NN-balls during the invocation of TERk.

The sorted projections required by the algorithms can be easily maintained through the use of linked lists. The list for each dimension would require 2N words of storage (or pointers): one word for the identification of the point and one word for a "next pointer". Thus the total storage requirement for the linked list scheme is 2kN words of storage. The storage requirement can be reduced to kN words of storage by using integer array PROJ(1::N, 1::k). The rows PROJ(*,m) will be pointers to the POINTS sorted by the m-th coordinate (formally, POINTS(PROJ(i,m),m) ≤ POINTS(PROJ(j,m),m) iff i ≤ j). The partitioning into subproblems is quite easy using linked lists: traverse down each dimension's list appending each point to one of two sorted sublists for that dimension (one sublist for each subproblem). After solving the subproblems the lists can be merged together again. The same basic strategy is used for the array technique; to break a problem into subproblems of size m and N-m, partition PROJ such that the subarrays PROJ(1::m,*) and PROJ(N-m+1::N,*) have the condition for the subproblems. For the array scheme, the work of partitioning and remerging is a bit more difficult, but the storage savings could make that worthwhile.

The task of projection onto a lower dimensionality subspace could be accomplished by making a copy of the point set reduced by one dimension, but that is wasteful of both time and storage. A more elegant solution is to keep a global <u>integer array</u> ACTIVE_DIMENSIONS(1::k) such that if the algorithm is currently dealing with m dimensions then those dimensions will be found in ACTIVE_DIMENSIONS(1::m). All references to a point would then be made by accessing this array to turn a "virtual" dimension into an actual dimension. Projection from the current m dimensional space onto the m-1 dimensional hyperplane defined by a value in the ACTIVE_DIMENSIONS(j) dimension could be accomplished by swapping the j-th and m-th elements of ACTIVE_DIMENSIONS.

The algorithms of Chapter 3 solved a subproblem by a direct solution for very small problem size (typically, $N \le 2$). In practice it would be more efficient to solve larger subproblems by "brute force" techniques rather than by divide and conquer. To this end, some value N_0 should be chosen for each of the algorithms such that if the file size N is less than N_0, then the problem is solved by brute force (for ease of presentation, we chose $N_0 = 1$ or $N_0 = 2$ in Chapter 3).

5.2 Average-case algorithms

The algorithms presented in Chapter 3 were developed for the purpose of displaying good worst-case behavior. In many applications, however, one is willing to tolerate the possibility of poor worst-case behavior if that sacrifice leads to faster running times on the average. In this section we will suggest certain techniques that one could use to transform the worst-case algorithms into faster average-case algorithms with the risk of poor worst-case performance.

Many of the algorithms we developed depended on the ability of finding medians in linear time. For that task we proposed to use the selection algorithm of Blum, _et al_., [1972] which has linear worst-case time. A selection algorithm due to Hoare (modified by Floyd and Rivest [1975] to employ sampling) has a much faster average-case time but quadratic worst-case time. Using such an algorithm for selection would lead to faster average-case closest point algorithms. Instead of merely borrowing their algorithm, however, we could go even further and borrow the idea underlying the algorithm. Instead of taking the time to find the true median of the points, why don't we just sample a subset of the points and use the median of the sample? In the worst case that could give us very unbalanced subproblems, but it would not hurt much on the average.

Our fast algorithms went to great lengths to choose cut planes with good worst-case properties. A fast average-cast strategy might look instead for a cut plane which it expects to exhibit good properties. One such strategy would replace scanning a sorted list in each dimension by performing a similar scan on a sample of the points from that dimension, and then choosing the dimension in which to cut and the plane by which to cut from those samples. A more heuristic strategy might choose the dimension in which to cut as that exhibiting the maximum variance (variance chosen as an estimator of dispersion, which suggests dimensional sparsity). The value defining the cut plane might then be chosen by sampling.

Many isomorphisms exist between recursive divide and conquer procedures and binary trees. One well known example is the isomorphism between QUICKSORT and randomly built binary search trees; this is mentioned by both Knuth [1973] and Sedgewick [1975]. When we view a divide and conquer algorithm it is often helpful to think of the tree corresponding to it. This insight allows us to see many similarities between the divide and conquer algorithms of this thesis and the multidimensional binary search trees due to Bentley [1975b]. The resemblances are even more noticeable in the version of the k-d trees described by Friedman, Bentley and Finkel [1975]. The average-case running time empirically observed for their k-d tree algorithm in k dimensions for the ANN problem is $\Theta(N \log N)$, which is certainly superior to ALLk's time of $\Theta(N \log^k N)$. One reason for this superiority is that the tree structure of the k-d trees allows the subsolutions to be maintained, whereas in the ALLk algorithm, TERk makes no use of the work done by the previous stages of the algorithm. Had not k-d trees existed before the algorithms in this thesis were developed, we probably would have developed them as a result of this work. As it was, the intuition provided by the k-d tree algorithms proved quite useful to the author in developing the algorithms presented here.

6. Principles of algorithm construction

In this chapter we enumerate some of the general principles of algorithm construction that we have employed thus far in our work. We must limit ourselves to enumeration; a more complete treatment of even this small set of principles is beyond the scope of this thesis. We therefore divide the chapter into four sections, in each of which we shall investigate a particular class of algorithm construction techniques.

6.1 Principles of algorithm development

General strategies. In our work in Chapter 3 we noted two specific strategies for dealing with sets of elements. The technique of iteration solved a problem by obtaining a subsolution for each element in the set then combining those to form a solution to the whole; in our work it led to quadratic algorithms. The second general strategy we employed was divide and conquer, which is discussed in detail elsewhere in this thesis.

High level description of algorithms. All of the algorithms we described in Chapter 3 were described at a very high level. This freed us from the cumbersome and unenlightening chores of bookkeeping to concentrate our attention on algorithm design. Though we had to have some idea as to how we were going to implement the constructs we employed, we could postpone the implementation details until we had developed algorithms with good asymptotic behavior.

Process of generalization. Our work in Chapter 3 was in many ways a study of the psychological process of generalization. One might view the algorithms we developed in that chapter as points in a three dimensional space: the first dimension is the problem being solved (SFR, CP, ANN, STW); the second is the dimensionality of the space (1, 2, 3, k); and the third is the speed of the algorithm ($\Theta(N \log^k N)$, $\Theta(N \log N)$). We often made use of Polya's [1954, p. 194] frank advice, "try the simplest thing first", even when the simplest appeared trivial. After that we proceeded to the next most simple algorithm, always moving to an adjacent point in the algorithm development space. When we moved to a different "plane" in the space (from SFR to CP or from CP to ANN) we tried to choose our path on that plane to resemble as closely as possible our path on the previous plane. By doing so we found that we were able to employ many parts of previously developed algorithms. Polya [1945, p. 42] refers to this process as using "both the method and the result".

Abstract description. We saw that a very high level description of an algorithm (i.e., solve two problems of N/2 points in the plane and one problem of up to N points on the line), giving its structure but not its task, is very helpful in understanding the algorithm and generalizing it.

Identify expensive parts. A main point of Knuth's work [1971] is that program optimization should usually be concerned with only one relatively small part of the program. We saw that the same principle applies to creating faster asymptotic algorithms. In trying to reduce the running time of SPARSE2, for instance, we first found where the major cost was incurred, and then we modified that step.

Relationship between worst-case and average-case ("heuristic") algorithms. In Chapter 5 we noticed an important relationship between worst-case and heuristic algorithms: worst-case algorithms often suggest faster average-case algorithms, and, conversely, experience with heuristic algorithms can yield valuable insight in developing worst-case algorithms.

Lower bounds. The construction of lower bounds gives the algorithm designer both a target at which to aim and a good reason for stopping his work. We saw three important techniques in dealing with lower bounds: reduction allowed us to use previous results; examining degenerate cases (i.e., $\delta = 0$) allowed us to make general assertions about the complexity of the problem (for any correct algorithm must work for the case of $\delta = 0$); and imbedding allowed us to apply results in one dimensionality to problems in a higher dimension.

Standard speed-up tricks. Presorting a set of numbers and maintaining a sorted list of those numbers is a fairly standard technique in algorithm design. The algorithm designer should be familiar with such useful techniques.

Specific performance goals. In developing an algorithm it is helpful to have a specific performance bound in mind.

Examining degenerate cases. We saw this employed in establishing lower bounds, in gaining insight into why worst-case behavior occurs, and in determining the boundary conditions of an algorithm. Polya [1954, p. 23] refers to a similar process as "picking out an extreme special case".

How to solve it. Polya's [1945] work of this title gives valuable insight into solving programming problems. The book's leaning toward geometric examples and its emphasis on problem solving using analogy and generalization make it particularly applicable to the work in this thesis. His later work, Mathematics and Plausible Reasoning [1954], is also extremely valuable for the algorithm designer.

6.2 Principles for divide and conquer

Uneven balancing. In most applications of divide and conquer the original problem is divided into two subproblems, each containing N/2 points. We observed that the exact fraction (1/2)N is not necessary; guaranteeing that each subproblem contains at least some constant fraction p of the N points is suitable (in many situations) to insure balancing. This is similar to the idea behind the balanced binary trees due to Adelśon-Velśkii and Landis [1962].

Speed-up techniques. We made frequent use of the speed-up trick of presorting. This technique was also used in Shamos's [1975b] description of Strong's algorithm and in the balanced tree construction algorithm of Finkel and Bentley [1974].

Division into subproblems. The asymptotic running times of our algorithms were improved as we allowed our cut plane selection strategy the freedon to choose among many cut planes. We eventually chose a cut point that had good worst-case properties and was easy to find. This technique is similar to the way the selection algorithm of Blum, _et al_., [1972] chooses the partitioning element.

Reducing subproblem size below O(N/log N). If the size of a subproblem is reduced to O(N/log N), then the cost of applying a Θ(N log N) algorithm to that subproblem is at most linear. A similar usage of this technique was made in the Θ(N log log N) median algorithm of Blum (see Knuth [1973]) and in Rabin's CP algorithm [1976].

Multiple calls on the same procedure. The standard divide and conquer schema consists of three parts: (1) divide the problem into subproblems, (2) solve the subproblems, and (3) combine the subsolutions into a solution to the whole problem. In the typical application of divide and conquer Step 2 is recursive and Steps 1 and 3 are non-recursive. In our algorithms we usually used a recursive call for Step 3 (though we reduced the dimensionality by one). Other algorithms have deviated from the typical schema. Kung, Luccio and Preparata [1975] used a recursive Step 3 as well. In the linear median algorithm of Blum, _et al_., [1972] Step 1 was accomplished recursively. One might accurately view Rabin's CP algorithm [1976] as a divide and conquer algorithm in which all three steps were accomplished non-recursively.

6.3 Principles for multidimensional algorithms

Divide and conquer. We have developed a divide and conquer schema applicable to many multidimensional problems.

Sparsity. Sparsity was a key to many of our algorithms. We have seen three different types of sparsity. In some problems sparsity was given as a condition of the problem (SFR). In the CP problem we induced global sparsity on the point set. In the ANN problem we used a type of "local" sparsity.

Wise choice of cut planes. This reduced the asymptotic running times of our algorithms.

Imbedding. Imbedding was a valuable technique for applying results of lower bounds on unidimensional problems to multidimensional problems.

Varying metrics. We saw that though all of our work was done for one metric (the L_∞), it was applicable to others (i.e., the L_2). It is often convenient to work with the most tractable metric available, then attempt later to apply results to other metrics.

Value of one-dimensional analogs. One-dimensional analogs proved quite helpful to us in gaining insight in multidimensional problems. The many results in the plane due to Shamos [1975b] might prove invaluable to future workers in multidimensional algorithms.

6.4 Principles for turning algorithms into programs

Simulating work by representation. When our algorithms called for working with a projected point set, we might have been tempted to make a copy of the point set (reducing dimension by one). Instead we accomplished the task by referencing dimension through an array which represented the active dimensions.

Solving small problems. One often solves small problems most efficiently in a way that is not asymptotically optimal. Though a linear algorithm is asymptotically superior to a $\Theta(N \log N)$ algorithm, $N \log_2 N < 15N$ for $N < 32{,}000$.

Fast average-case components. One can often turn a good worst-case algorithm into a good average-case algorithm by substituting average-case components for worst-case components. In our work we substituted the fast average-case selection algorithm of Floyd and Rivest [1975] for the good worst-case algorithm of Blum, _et al_., [1972].

Sampling. Though it is usually necessary to examine all elements of a set to be able to guarantee a property of that set, one can often get a good expectation of that property by investigating a sample (which is more efficiently accomplished).

Heuristics simulating guaranteeable properties. In our work we suggested the heuristic of choosing the cut plane as the median element in the dimension of maximum variance as a simulation of a good worst-case cut plane.

Isomorphism between trees and divide and conquer. The natural isomorphism between divide and conquer algorithms and tree data structures often suggests one, given the other.

7. Further work

We have examined many different problem areas in this thesis. Though we have made some contribution to each area, there is much important work that still needs to be done. In this chapter we mention a few outstanding problems that particularly merit further research.

The first area that deserves attention is reduction of upper bounds on time complexities. The author conjectures that $ANN(N,k) = \Theta(N \log N)$; it is conceivable that Algorithm ALLk could be modified to achieve that bound by making wise use of cut planes. The author also conjectures that the bound $NN(N,k) = \Theta(\log N)$ can be obtained using storage linear in Nk. The way to approach this problem using divide and conquer is to investigate data structures isomorphic to our algorithms (see especially Friedman, Bentley and Finkel [1975]). A third bound the author believes can be reduced is the quadratic bound on the MST problem. The minimal spanning tree has global properties that make a subtree hard to compute; one way of using divide and conquer for this problem might be to find a supergraph of the minimal spanning tree than apply Yao's [1975] fast graph MST algorithm to that supergraph.

Shamos's notebook [1975b] is a rich source of planar problems and solutions; extending his work to multidimensional space is an important problem. Two especially attractive problems are finding the Voronoi diagram of points in k space and finding the diameter of a set in k space.

It is desirable to shorten the gap between the "theoretical" algorithms described in this thesis and the "practical" algorithms such as described by Friedman, Bentley and Finkel [1975] and Bentley and Friedman [1975]. One way of "shortening the gap" is to decrease the running time of the algorithms in the thesis to actually run more efficiently than the "practical" algorithms. Another way to shorten the gap is to show that these algorithms give a "theoretical" explanation of the running times of the fast algorithms.

Much further work needs to be done on the principles of algorithm construction. The typical way in which a computer science student is now taught how to build algorithms is to be shown twenty or thirty algorithms, then asked to go out and build some of his own. This was basically the way in which the author learned about algorithms; it was only during the writing of this thesis that he shared the experience of Descartes [1650]: "As a young man, when I heard about ingenious inventions, I tried to invent them by myself, even without reading the author. In doing so, I perceived, by degrees, that I was making use of certain rules". The author hopes that the reader is convinced that there are "certain rules" which an algorithm designer can use in his task.

It is important to discover what principles competent algorithm designers employ. One way of doing so is for algorithm designers to include with future algorithms the main principles employed in constructing them. Another way of discovering such principles is to attempt to systematically reinvent the algorithms as Descartes did ("without reading the author"), and see what principles are employed. After such principles are discovered, they ought to be collected together and systematized in some way.

In this thesis we have touched upon different areas that lend themselves to general principles. The first is algorithm development; much further work needs to be done in describing the general approach an algorithm designer should take in attacking a problem. A second area is the divide and conquer strategy. Further work might describe more techniques that are commonly used in divide and conquer. Other strategies ought also to be analyzed; among those are dynamic programming and depth-first search. A third area which merits investigation is that of turning algorithms into programs. This is especially important as one observes the great distance between many "theoretical" algorithms that are now presented and the "real-life" task of solving real problems on real computers.

8. Conclusions

In this chapter we will briefly review the major contributions contained in this thesis. The contributions can be broadly classified as falling into three areas: multidimensional algorithms, the divide and conquer strategy, and principles of algorithm design.

The contributions of this thesis to multidimensional algorithms have been twofold: basic methods and basic results. We have shown that divide and conquer is a fundamental tool which can be used in multidimensional algorithms; we developed one particular divide and conquer schema that was well suited to many problems. Another fundamental tool which we used often was the notion of sparsity. We saw three kinds of sparsity: given, induced and local. The basic results of this thesis can be summarized as follows:

$$\mathrm{CP}(N,k) = \Theta(N \log N),$$

$$\mathrm{SFR}(N,k) = \Theta(N \log N), \text{ and}$$

$$\Theta(N \log N) \le \mathrm{ANN}(N,k) \le \Theta(N \log^{k-1} N).$$

These appear to be the first less-than-quadratic upper bounds for multidimensional closest point problems.

The algorithms in this thesis demonstrate the power of the divide and conquer strategy. We saw that the strategy was applicable to a problem domain fundamentally different from any in which it had been previously used. We also observed many ways in which divide and conquer algorithms can be employed. Among techniques we used were uneven balancing, "double recursion" (in problem size and dimensionality), and reducing subproblem size by intelligent cut point strategies.

The final area to which this thesis contributes is algorithm design. Chapter 3 of this thesis is one of the few examples of which the author knows of a written presentation of the algorithm design process. The author feels that Chapter 3 might be a valuable tool in communicating the algorithm design process to novices; in particular it might be the basis for one or two weeks of classroom discussion in an algorithms course. In addition to examples, Chapter 6 provides a summary of principles used in the design process. Though the list of principles is small, it certainly contains some fundamental insights, and it is a suitable basis from which others can expand.

Bibliography

Adel'son-Vel'skii, G. M. and E. M. Landis [1962]. "An algorithm for the organization of information", Akademiia Nauk SSSR, Doklady, Matematika, vol. 146, no. 2, pp. 263-266. English translation in Soviet Mathematics, vol. 3, pp. 1259-1263.

Bentley, J. L. [1975a]. A survey of techniques for fixed radius near neighbor searching, Stanford Linear Accelerator Center Report SLAC-186, August 1975, 33 pp.

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

Bentley, J. L. and W. A. Burkhard [1976]. "Heuristics for partial match retrieval data base design", Information Processing Letters, vol. 4, no. 5, February 1976, pp. 132-135.

Bentley, J. L. and J. H. Friedman [1976]. Fast algorithms for constructing minimal spanning trees in coordinate spaces, Stanford Computer Science Department Report STAN-CS-75-529, January 1976, 29 pp.

Bentley, J. L. and M. I. Shamos [1976]. "Divide and conquer in multidimensional space", Proceedings of the Eighth Symposium on the Theory of Computing, ACM, May 1976, pp. 220-230.

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.

Bentley, J. L., D. F. Stanat, and E. H. Williams, Jr. [1976]. The complexity of near neighbor searching, in preparation.

Blum, M., et al. [1972]. "Time bounds for selection", Journal of Computer and System Sciences, vol. 7, no. 4, August 1973, pp. 448-461.

Burkhard, W. A. [1976]. "Hashing and trie algorithms for partial-match retrieval", ACM Transactions on Data Base Systems, vol. 1, no. 2, June 1976, pp. 175-187.

Burkhard, W. A. and R. M. Keller [1973]. "Some approaches to best match file searching", Communications of the ACM, vol. 16, no. 4, April 1973, pp. 230-236.

Cleary, J. G. [1975]. Finding nearest neighbors in Euclidean space, University of Canterbury Preprint, Christchurch, New Zealand, 1975, 18 pp.

Cooley, J. W. and J. W. Tukey [1965]. "An algorithm for the machine calculation of complex Fourier series", Mathematics of Computation, vol. 19, pp. 297-301.

Cover, T. M. and P. E. Hart [1967]. "Nearest neighbor pattern classification", IEEE Transactions on Information Theory, vol. IT-13, no. 1, January 1967, pp. 21-27.

Coxeter, H. S. M. [1961]. "An upper bound for the number of equal nonoverlapping spheres that can touch another sphere of the same size", in Proceedings of the Symposium in Pure Mathematics, vol. 7, (Convexity), American Mathematical Society, Providence, Rhode Island.

Descartes, R. [1650]. Rules for the Direction of the Mind. Quoted in Polya [1945, p. 93].

Dijkstra, E. W. [1959]. "A note on two problems in connexion with graphs", Numerische Mathmatik, vol. 1, no. 5, October 1959, pp. 269-271.

Dobkin, D. and R. Lipton [1975]. On the complexity of computation under varying sets of primitives, Yale University Computer Science Department Technical Report No. 42, 1975.

Dobkin, D. and R. J. Lipton [1976]. "Multidimensional search problems", SIAM Journal on Computing, vol. 5, no. 2, June 1976, pp. 181-186.

Finkel, R. A. and J. L. Bentley [1974]. "Quad trees--a data structure for retrieval on composite keys", Acta Informatica, vol. 4, no. 1, pp. 1-9.

Floyd, R. W. and R. L. Rivest [1975]. "Expected time bounds for selection", Communications of the ACM, vol. 18, no. 3, March 1975, pp. 165-172.

Friedman, J. H. [1975]. A variable metric decision rule for nonparametric classification, Stanford Linear Accelerator Center Report SLAC-PUB-1573, April 1975, 30 pp.

Friedman, J. H., F. Baskett, and L. J. Shustek [1975]. "An algorithm for finding nearest neighbors", IEEE Transactions on Computers, vol. C-24, no. 10, October 1975, pp. 1000-1006.

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.

Friedman, J. H., S. Steppel, and J. W. Tukey [1973]. A nonparametric procedure for comparing multivariate point sets, Stanford Linear Accelerator Center Computation Research Group Technical Memo No. 153, November 1973, 22 pp.

Fukanaga, K. and P. M. Narendra [1975]. "A branch and bound algorithm for computing k-nearest neighbors", IEEE Transactions on Computers, vol. C-24, no. 7, July 1975, pp. 750-753.

Garey, M. R., R. L. Graham, and D. S. Johnson [1976]. "Some NP-complete geometric algorithms", Proceedings of the Eighth Symposium in the Theory of Computing, ACM, May 1976, pp. 10-22.

Iverson, K. E. [1962]. A Programming Language, Wiley, New York.

Karp, R. M. [1976]. "Probabilistic analysis of heuristic search methods", in the proceedings of the Carnegie-Mellon University Symposium on New Directions and Recent Results in Algorithms and Complexity, to be published by Academic Press.

Kieburtz, R. B. [1975]. Structured Programming and Problem Solving with ALGOLW, Prentice-Hall, Englewood Cliffs, New Jersey.

Knuth, D. E. [1971]. "An empirical study of FORTRAN programs", Software--Practice and Experience, vol. 1, no. 2, April-June 1971, pp. 105-133.

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

Knuth, D. E. [1976]. "Big omicron and big omega and big theta", SIGACT News, vol. 8, no. 2, April-June 1976, pp. 18-24.

Kruskal, J. B. Jr. [1956]. "On the shortest spanning subtree of a graph and the travelling salesman problem", Proceedings of the American Mathematical Society, vol. 7, February 1956, pp. 48-50.

Kung, H. T., F. Luccio, and F. P. Preparata [1975]. "On finding the maxima of a set of vectors", Journal of the ACM, vol. 22, no. 4, October 1975, pp. 469-476.

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.

Lee, R. C. T., Y. H. Chin, and S. C. Chang [1975]. Application of principal component analysis to multi-key searching, National Tsing Hua University, Republic of China, 28 pp.

Lee, R. C. T., J. R. Slagle, and H. Blum [1975]. "A triangulation method for the sequential mapping of points from N-space to 2-space", Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures, May 14-16, 1975, IEEE, pp. 374ff.

Leech, J. and N. J. A. Sloane [1971]. "Sphere packings and error correcting codes", Canadian Journal of Mathematics, vol. 23, no. 4, pp. 718-745.

Levinthal, C. [1966]. "Molecular model building by computer", Scientific American 214, June 1966, pp. 42-52.

Liu, C. L. [1968]. Introduction to combinatorial mathematics, McGraw-Hill, New York.

Loberman, H. and A. Weinberger [1957]. "Formal procedures for connecting terminals with a minimum total wire length", Journal of the ACM, vol. 4, no. 4, October 1957, pp. 428-437.

Loftsgaarden, D. O. and C. P. Quesenberry [1965]. "A nonparametric density function", Annals of Mathematical Statistics, vol. 36, pp. 1049-1051.

McNutt, B. [1973]. Experiments by Bruce McNutt reported in Knuth [1973], sec. 6.5, p. 555.

Newman, W. M. and R. F. Sproull [1973]. Principles of interactive computer graphics, McGraw-Hill, New York.

Polya, G. [1945]. How to solve it, Princeton University Press, Princeton, New Jersey.

Polya, G. [1954]. Mathematics and plausible reasoning, 2 vols., Princeton University Press, Princeton, New Jersey.

Prim, R. C. [1957]. "Shortest connection networks and some generalizations", Bell Systems Technical Journal, vol. 36, no. 6, November 1957, pp. 1389-1401.

Rabin, M. O. [1976]. "Short expected time algorithms for the nearest pair", in the Carnegie-Mellon University Symposium on New Directions and Recent Results in Algorithms and Complexity, to be published by Academic Press.

Reingold, E. M. [1972]. "Establishing lower bounds on algorithms--a survey", Spring Joint Computer Conference 1972, AFIPS Press, pp. 471-481.

Rivest, R. L. [1974a]. Analysis of associative retrieval algorithms, Stanford Computer Science Department Report STAN-CS-74-415, 1974, 102 pp.

Rivest, R. L. [1974b]. "On the optimality of Elias' algorithm for performing best match searches", Proceedings IFIP Congress 74, Stockholm, Sweden, August 1974, pp. 678-681.

Rivest, R. L. [1976]. "Partial-match retrieval algorithms", SIAM Journal on Computing, vol. 5, no. 1, March 1976, pp. 19-50.

Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis [1974]. "Approximate algorithms for the travelling salesperson problem", Proceedings of the Fifteenth Symposium on Switching and Automata Theory, IEEE, October 1974.

Shamos, M. I. [1975a]. "Geometric complexity", Proceedings of the Seventh Symposium on the Theory of Computing, ACM, May 1975, pp. 224-233.

Shamos, M. I. [1975b]. Problems in computational geometry, unpublished manuscript.

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

Smith, A. R. [1975]. Nearest neighbor algorithms for speech, Carnegie-Mellon University preprint, November 1975, 10 pp.

Strassen, V. [1969]. "Gaussian elimination is not optimal", Numerische Mathmatik, vol. 13, no. 4, pp. 354-356.

Van Rijsbergen, C. J. [1974]. "The best-match problem in document retrieval", Communications of the ACM, vol. 17, no. 11, November 1974, pp. 648-649.

Warnock, J. E. [1969]. "A hidden-surface algorithm for computer generated half-tone pictures", University of Utah Computer Science Department Report TR4-15, 1969. Also in Newman and Sproul [1973, pp. 297-312].

Yao, A. C. [1975]. "An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees", Information Processing Letters, vol. 4, no. 1, September 1975, pp. 21-23.

Yao, A. C. and F. F. Yao [1974]. On computing the rank function of a set of vectors, University of Illinois Computer Science Department Report UIUCDCS-R-75-699, February 1975, 15 pp.

Yuval, G. [1975]. "Finding near neighbors in k-dimensional space", Information Processing Letters, vol. 3, no. 4, March 1975, pp. 113-114.

Yuval, G. [1976]. "Finding nearest neighbors", to appear in Information Processing Letters.

Zahn, C. T. [1971]. "Graph-theoretical methods for detecting and describing gestalt clusters", IEEE Transactions on Computers, vol. C-20, no. 1, January 1971, pp. 68-86.