SLAC REPORT 206
STAN-CS-78-659
UC 32

# TOPICS IN COMPUTATIONAL GEOMETRY*

JOHN EDWARD ZOLNOWSKY
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

PREPARED FOR THE DEPARTMENT OF ENERGY
UNDER CONTRACT NO. EY-76-C-03-0515

February 1978

Printed in the United States of America. Available from National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22161. Price: Printed Copy $5.25; Microfiche $3.00.

---

*Ph.D. Dissertation

## ABSTRACT

We solve two problems in computational geometry. The first is to characterize the behavior of the nearest neighbor search algorithm based on the k-d tree data structure. The second is to derive an efficient algorithm for the construction of the intersection of a finite set of half-spaces in three dimensions.

The k-d tree is a data structure useful in classification and analysis of multidimensional data. Each nonterminal node of the k-d tree represents an ordinate axis and a partition hyperplane on that axis. The terminal nodes represent bins delimited by the partitions of its ancestor nodes. The entire k-d tree is a partition of Euclidean k-space.

If S is a finite set of points in Euclidean k-space, and p is a point in S, the nearest neighbor of p in S is a member of S, distinct from p which minimizes the distance from p. We find bounds for the time for finding the nearest neighbor of all points using the k-d data structure. In particular, we examine three different criteria for choosing the partition ordinate of each node in the k-d tree, based on the test point set. We obtain tight bounds for the best of these criteria, which generates the "square tree'.

The second problem is to efficiently construct the intersection of a finite set of half-spaces in three dimensions. If N is the number of half-spaces, our algorithm takes time proportional to N logN. Intuitively, the algorithm first constructs a northern and a southern cap, then intersects these two polyhedra to generate the result polyhedron. We can efficiently construct the caps from their constituent half-spaces by a divide-and-conquer technique. After constructing the caps, we vertically stratify space into slabs by passing horizontal planes through the vertices of the caps. By elementary geometry, we can examine the slab between two vertically consecutive vertices to determine whether the caps intersect within that slab. If not, we can determine whether any intersection must be above or below the slab. Thus using a binary search, we can find an edge of the final intersection, and need merely extend this edge around the polyhedron to complete the intersection.

# PREFACE

We solve two problems:

1. Find the worst case average search time for the nearest neighbor search using the k-d tree data structure, and

2. Determine efficiently the intersection of a finite set of half-spaces in three dimensions.

Both results are obtained using the technique of divide and conquer. The first result is of interest in pattern matching and data analysis, while the second result is better than the worst case of the simplex algorithm for three-variable problems.

## ACKNOWLEDGEMENTS

I am indebted to Andrew Yao for the planar form of the bounds on near neighbor search times using k-d trees. I would also like to thank Michael Shamos for suggesting the problem of intersecting half-spaces in three dimensions. A special note of thanks goes to Jon Bentley for stimulating my interest in computational geometry, and to Forest Baskett for being a patient and helpful advisor. And last but not least, deepest appreciations are due the large supporting cast which kept me going and made this possible.

This work was sponsored by the Computation Research Group of the Stanford Linear Accelerator Center, under contract with the United States Department of Energy.

TABLE OF CONTENTS

Appendix page

LIST OF FIGURES

## LIST OF TABLES

# Chapter I

## NEAREST NEIGHBOR SEARCH USING K-D TREES

### 1.1 GENERAL NOTATION

We shall be discussing finite sets of points in Euclidean k-space. If S is a set, |S| will denote the number of members of S. If p is a point, pi will denote the ith coordinate of p. We shall use the notation d(x,y) to denote the distance from object x to object y, and r(p,S) to be the distance to the nearest neighbor of point p in the set S. We shall also use the notation BALL(p,S) to represent the subset of Euclidean k-space within distance r(p,S) of point p. The term logN will mean the logarithm of N to the base 2.

We shall need some notation to describe order of magnitude calculations. We shall say $g(n) = O(f(n))$ if there exist a constant c and an integer M such that $g(n) \leq c\, f(n)$ for $n>M$. We shall say $g(n) = \Omega(f(n))$ if there exist a constant c and an integer M such that $g(n) \geq c\, f(n)$ for $n>M$. Lastly, we shall say $g(n) = \theta(f(n))$ if there exist constants c and d and an integer M such that $c\, f(n) \leq g(n) \leq d\, f(n)$ for $n>M$.

## 1.2 K-D TREES

### 1.2.1 History

Bentley[1] developed the concept of a multidimensional binary tree, and indicated that this data structure would be useful in answering nearest neighbor queries. Friedman, Bentley, and Finkel[2] used a modified form of the k-d tree to answer nearest neighbor enqueries.

### 1.2.2 Definition

A k-d tree is a binary tree with distinct types of nodes for the internal and leaf nodes. The internal nodes are called "partitions", the leaves are called "buckets". A bucket contains a list of points in k-dimensional Euclidean space. A partition is composed of four elements: a left subtree called the "lowson", a right subtree called the "highson", a partition axis called the "discriminant", and a partition value called the "position". The discriminant will in general be an integer between 1 and k, the position will be a real number.

A partition divides the points in the buckets below it. For example, suppose L is a partition which discriminates

---

[1] J. L. Bentley, "Multidimensional Binary Search Trees for Associative Searching", Communications of the ACM, 18(1975), pp. 509-517.

[2] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Time", Stanford Linear Accelerator Center Report SLAC-PUB-1549, February 1975.

along the ith axis. If p is a point in a bucket in the lowson subtree then pi ≤ position(L), likewise if q is a point in a bucket in the highson subtree then qi ≥ position(L).

With each node, we shall associate a set of bounds. There will be a lower and an upper bound along each axis. For the root of the tree, the lower bounds will be $-\infty$, and the upper bounds $+\infty$. If L is a partition in the tree with discriminant i, then lowson(l) has the same bounds as L, except that the upper bound along the ith axis is position(L). Likewise, highson has the same bounds except that the lower bound along the ith axis is position(L). Obviously, in order that the upper bound along an axis always be greater than or equal to the lower bound along that axis, the position of a partition must be between the upper and lower bounds of the discriminant.

Finally, we shall call the space delimited by the bounds set of a node the "bin" of that node. If q is in a bucket L, then BIN(q) will denote the bin associated with L. The sequence of partitions leading down to BIN(q) we will call the bin partitions of q.

### 1.2.3 Construction

Given a set S of N points in k-space, we wish to construct a k-d tree on these points. Construction of the k-d tree proceeds by topdown generation of the partitions. We shall use the term refinement to mean the selection of a

discriminant and position, and the division of the set of points between the lowson and highson.

As the root node, we first create a single large bucket which contains all the points. We then refine all bins until they contain no more than some constant number of points. The method of choosing the discriminant we shall leave until later. We shall always choose the position as the median coordinate along the discriminant axis, so that the lowson and highson subtrees will contain as equal a number of points as possible. This guarantees that the depth of the tree will be at most logN. In the illustration of our arguments below, we shall assume that the maximum number of points in a bucket is one. Appendix A gives a pseudo-ALGOL description of the refinement procedure.

### 1.2.4 Nearest Neighbor Searching

The nearest-neighbor of a point p in a point set S is a member of S different from p with minimal distance from p. The nearest-neighbor problem is to find the nearest-neighbor of a given point p. The brute force solution to this problem is to find the distance to all other points, and select the minimum.

Clearly, the time for finding the nearest neighbor of all points using the brute force method is proportional to the square of the number of points. Friedman, Baskett, and Shustek[3] developed an algorithm using projections which has

expected time strictly less than the brute force method. It is possible to find the nearest neighbor of a point even more efficiently if we have a k-d tree already constructed on that set.

We give a recursive definition of the search procedure using a k-d tree. For a bucket node, the procedure simply scans the point list, computing the distance from the prototype point p, and keeping track of the identity of the nearest one. For a partition node, it first searches the subtree which is in the same direction as p from the position along the discriminant axis. If any part of the bin of the opposite subtree lies within the distance to the current closest neighbor, that subtree must also be searched. Appendix B gives a pseudo-ALGOL description of the nearest neighbor search.

Dobkin and Lipton[4] developed an algorithm which can find the nearest neighbor of a point in O(N logN) time. Unfortunately, this quick search is possible only after a costly preprocessing step, generating a data structure of

$$\Phi(N^{2^k-1})$$

---

[3]J. H. Friedman, F. Baskett, and L. J. Shustek, "An Algorithm for Finding Nearest Neighbors", IEEE Transactions on Computers, October 1975, pp. 1000-1006.

[4]D. Dobkin and R. J. Lipton, "Multidimensional Searching Problems", Yale University Computer Science Research Report #34, October 1974.

elements. By comparison, the k-d tree data structure takes only O(N logN) time to generate. Thus in applications where preprocessing costs are important, the k-d tree may be an important method.

## 1.3 THE PROBLEM

Bentley suggests that one can construct a k-d tree in which the discriminant cycles among the axes as the level of refinement increases. We shall call such a tree "cyclic". In order to make use of the geometrical structure of the set of points in constructing the tree, it would seem to be appropriate to use the spread of a bin along an axis, that is, the difference between the largest and smallest values along that axis among the points in the bin. Friedman, Bentley, and Finkel recommend that the discriminant be chosen as the axis along which the points in the bin have the largest spread. We shall call this type of tree "spread". We propose a slight variation on this idea and choose that axis in which the separation of the bounds is currently greatest. We shall call such a k-d tree "square", since it tends to have equilateral bins.

We know that the search for the nearest neighbor of a point in the set using a k-d tree can be quite poor, indeed, we may be forced to look at all other points in the set. We might ask however, about the average search behavior. Friedman, Bentley, and Finkel empirically established that

the average search time for the spread tree is proportional to logN. Here we shall analyze the worst case average search time behavior. Our interpretation here is that we should find the worst case time for finding the neighbors of all the points in the tree, and then compute the worst case average by dividing by the number of points. Call the worst case total search time for all nearest neighbors in a k-d tree TStype(N,k), where type is either "cyclic" or "square", N is the number of points, and k is the dimension of the space.

We immediately have the straightforward bound

$$TStype(N,k) = O(N^2),$$

since the worst that could happen is that we search the entire tree to find the nearest neighbor of each point. Bentley and Shamos[5] describe an algorithm which can solve the problem of finding the nearest neighbor of all points in

$$O(N \log^{k-1} N)$$

time, for k>1. We shall now develop upper and lower bounds on the maximum search time for the nearest neighbors of all points, first for cyclic trees, and then for square trees. We shall also demonstrate that the spread tree has worse worst case behavior.

---

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

## 1.4 BOUNDS FOR CYCLIC TREE SEARCHES

### 1.4.1 Upper Bound

The following theorem establishes an upper bound for the search time for finding the nearest neighbors of all points using a cyclic k-d tree.

THEOREM: $TScyclic(N,k) = O(N^{2-1/k} \log N)$.

PROOF: Since the k-d tree nearest neighbor search proceeds by first finding the nearest neighbor on the same side of the upper-most partition, and then if necessary, finding the nearest neighbor on the opposite side, we can say

$$TScyclic(2N,k) \leq 2TScyclic(N,k) + R(N,k), \qquad (1)$$

where R represents the cost used in looking for nearer neighbors on the opposite side.

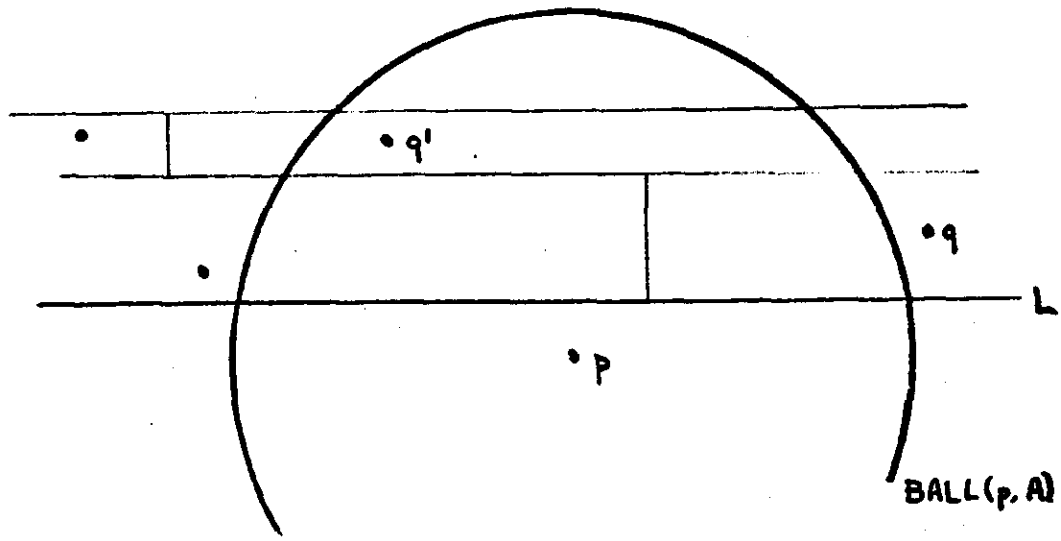
Let us investigate the size of the term R(N,k). If we say that the first partition hyper-plane L divides the set of points into subsets A and B, then we can write

$$R(N,k) \leq (|P(A,B)| + |P(B,A)|) \log N,$$

where (p,q) in P(A,B) iff p in A, q in B, and $d(p,BIN(q)) \leq r(p,A)$. The set P(A,B) contains the distance comparisons that must be done in crossing over from A to B. The factor of logN derives from the fact that it takes at most logN time to find q starting from the root of the tree.

Consider any pair (p,q) in P(A,B). Project BIN(q) onto L, the partitioning hyperplane. If some corner of the projected bin lies within BALL(p,A), then say that

(p,q) e C(A,B); assign all other pairs to C'(A,B) = P(A,B) - C(A,B). (See Figure 1.)




(p,q) in C(A,B), (p,q') in C'(A,B)

Figure 1: Bin Corner Inclusion Examples

Now let us see how many members are in each of C(A,B) and C'(A,B). If q is in B, then the projection of BIN(q) onto L has at most

$$2^{k-1}$$

corners. Also for any projection point u in L, at most some constant number ck of points p in A have r(p,A) ≥ d(p,u), so that BALL(p,A) includes u[6]. Thus

$$|C(A,B)| \leq |B|\, 2^{k-1}\, ck = O(N).$$

Suppose q is a point in B such that BALL(p,A) intersects BIN(q). If the projection of BIN(q) onto L has no corners within BALL(p,A), then clearly BIN(q) has no corners in BALL(p,A). We shall bound $|C'(A,B)|$ using this looser criterion.

We approach the question of how large $|C'(A,B)|$ is from the other side. That is, for each p in A, what is the maximum number of q in B such that BIN(q) is within distance r(p,A) of p, yet no corner of BIN(q) is within distance r(p,A) of p? With this question in mind, let us examine the construction of the k-d tree for B. We shall bound the number of bins whose corners lie outside the BALL(p,A), but such that they also overlap that ball.

As we start, note that all bins so far defined, namely all of B, satisfy the definition if BALL(p,A) intersects L, for there are no corners (unless k=1). If the first refinement does not create a corner, and the partition divides BALL(p,A), then we have two bins.

As we continue the refinement, we note that at each level, we may possibly double the number of bins, if the partitions are placed so that they divide the previous bins, but do not create corners within BALL(p,A). If a discrimi-

---

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

nant is chosen that must create a corner, choosing the position so that the partition does not overlap BALL(p,A) leaves the same number of bins overlapping that ball, without corners. As we cycle through the axes in further iterations of the refinement, at least one of the partitions must lie outside the ball, or else we divide our bin into subbins with corners, and all lower bins would have corners. Thus for each k levels, we can double the bin count at most k-1 times, and we can have at most

$$2^{\log N\ (k-1)/k} = N^{1-1/k}$$

bins. This is true for each p in A, thus

$$|C'(A,B)| \leq N^{2-1/k} = O(N^{2-1/k}).$$

We thus find

$$R(2N,k) = (O(N) + O(N^{2-1/k}))\ \log N = O(N^{2-1/k} \log N).$$

Substituting this in (1), and solving the recurrence relation, we finally obtain

$$TScyclic(N,k) = O(N^{2-1/k} \log N). \quad \bullet$$

### 1.4.2 Lower Bound

We can exhibit a particularly bad case for the nearest neighbor search using the cyclic k-d tree.

THEOREM: For k≥2, there exist point sets such that the total running time for the nearest neighbor search for all points using the cyclic tree is

$\Omega(N^{2-1/k})$.

PROOF: We proceed by construction. We first choose the main partition L, say x1=0. We want the set A to be strung out along the xk-axis, so we might choose the set A of points as (0, 0, 0, ..., 0, i), for $1 \le i \le N/2$. Thus for p in A, r(p,A)=1 and the overlap of BALL(p,A) with the other side of L is a complete hemisphere.

The points of B will be chosen arbitrarily, with only two restrictions. The first restriction is that all points of B must have all coordinates negative. Thus the k-d tree search will not find any new closer neighbors in B for points in A. The second restriction is that the points of B must be within distance 1/2 of the xk-axis. Those bins determined by these points B which also extend in the region of points xk>0 are long, spindly bins which pass within distance 1/2 of each of the points of A.

Now at each level, the number of bins of B entering the region xk>0 doubles, except when the discriminant axis is xk, in which case, the number remains the same. Thus there are

$$2^{\log N(1-1/k)} = N^{1-1/k}$$

bins of B entering the xk>0 region, and hence

$$N(N^{1-1/k}) = N^{2-1/k}$$

distance tests must be made by the k-d tree search. ∎

Figure 2 shows an example of this construction for k=2, N=32. For each of the sixteen points of A, the nearest neighbor search must investigate the four topmost bins of B.

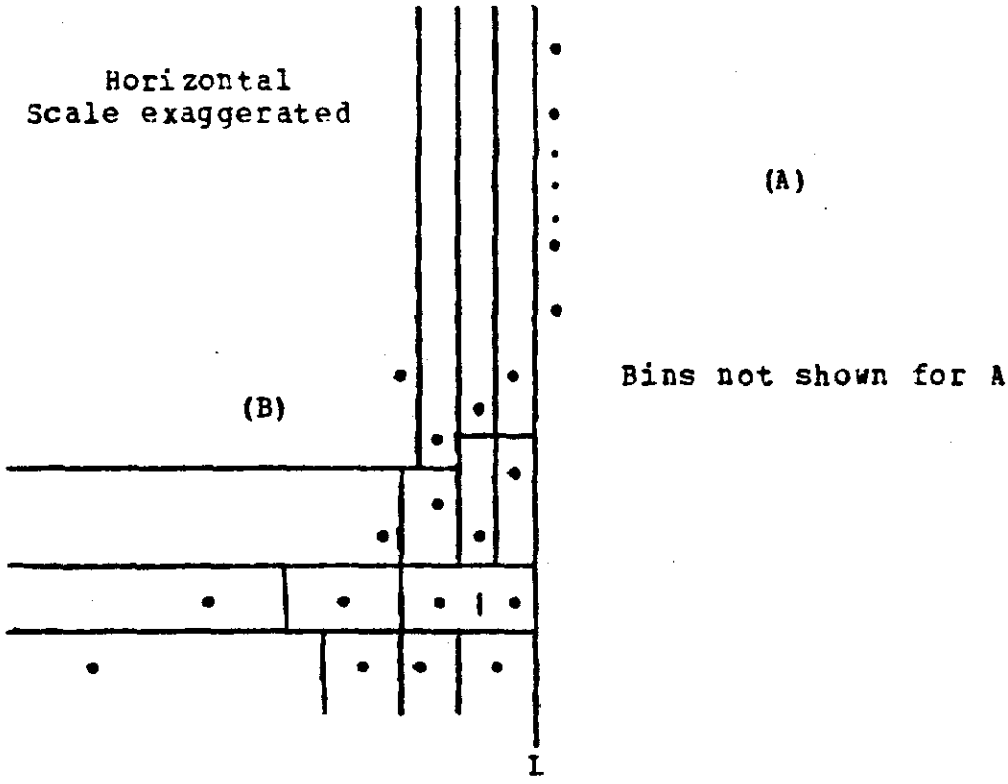



Figure 2: Worst Case Example of a Cyclic Tree

### 1.4.3 Summary

Combining the results of the previous two theorems, we know that the worst case time for finding the nearest neighbors of all points using a cyclic tree is bounded such that

$$\Omega(N^{2-1/k}) = TScyclic(N,k) = O(N^{2-1/k}\log N).$$

We do not attempt to find tighter bounds here for, as we shall see in the next section, the square tree exhibits significantly better characteristics.

## 1.5 BOUNDS FOR SQUARE TREE SEARCHES

### 1.5.1 Upper Bound

The following theorem establishes an upper bound for the search time for the nearest neighbors of all points using a square tree.

THEOREM: $TSsquare(N,k) = O(N \log^k N)$.

PROOF: In the case of TScyclic, we found that there were two types of bucket overlap in the search. Here, we shall first divide the computation on this basis. Thus, we write

$$TSsquare(N,k) = TCsquare(N,k) + TC'square(N,k), \quad (2)$$

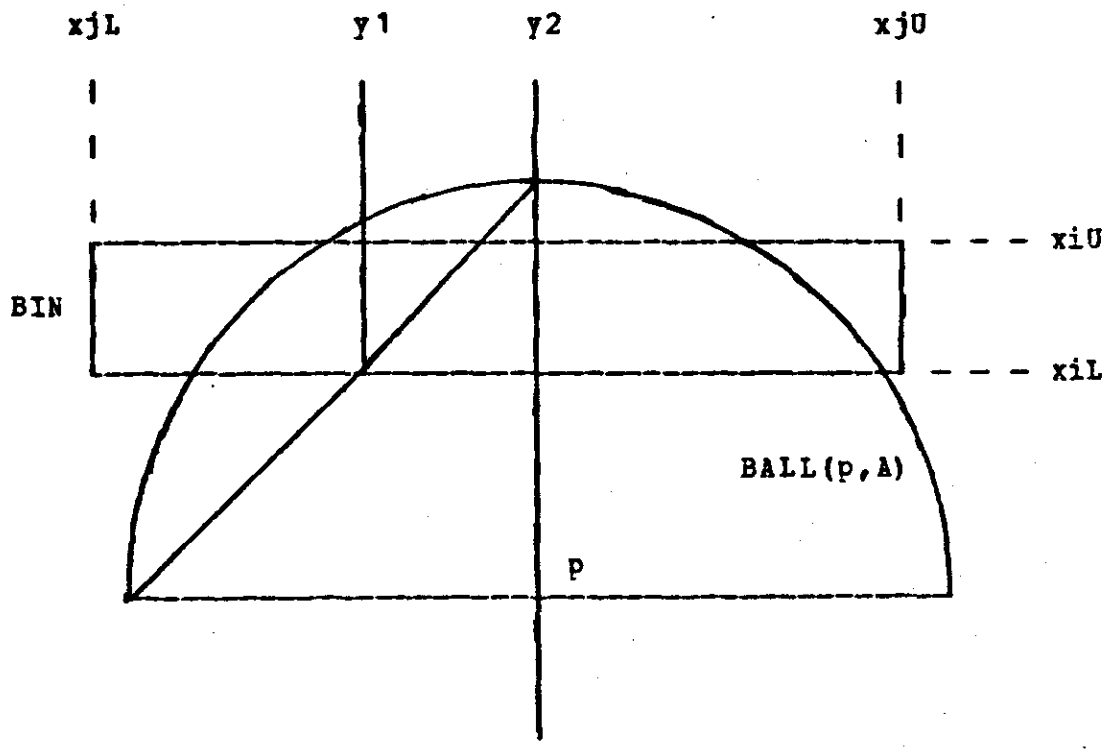
where TCsquare represents the search time involved with buckets which, when projected onto the separating partition, have a corner within the search ball, and TC'square represents the search time involved with buckets which overlap the search ball, but when projected onto the separating partition, have no corner within the search ball. Actually, we shall estimate TC'square by overlapping somewhat with TCsquare, and computing the search time involved with buckets which intersect the search ball, but have no corner in the search ball.

By the same divide and conquer reasoning as in the cyclic case, we readily find

$$TCsquare(N,k) \leq N\, 2^{k-1} ck \log^2 N = O(N \log^2 N).$$

We shall not use the divide and conquer technique to bound TC'square. Instead, we start by noting that for each

point p, the nearest neighbor search first finds the sequence of length logN of bin partitions of p. Suppose that L is a bin partition of p, separating the sets A and B, and that p in A. We then again ask the question: For any p in A, what is the maximum number of q in B such that $d(p,BIN(q)) \leq r(p,A)$ and no corner of BIN(q) lies within BALL(p,A)? As before, we answer this question by following the construction of the k-d tree on side B. We say that a bound of a bin intersects BALL(p,A) if the ball contains a segment of the face created by that bound.




$$xjU - xjL \geq 2(y2 - y1)$$

$$y2 - y1 \geq xiU - xiL$$

Figure 3: Projection onto xi,xj-axis plane

At the first level, the only bin is all of B with the boundary L and any preceding bin partitions of p, and it satisfies the criterion if L intersects BALL(p,A). Now consider the process of refinement of a bin at some later stage. If the bin has both upper and lower bounds along some axis which intersect BALL(p,A), then this axis cannot be chosen as the discriminant at some further stage, unless the bin so split has a corner which lies within BALL(p,A). This is because we always choose the axis with the largest spread between upper and lower bounds as the next discriminating axis. As can be seen in Figure 3, once a bin has upper and lower bounds along some axis which intersects the ball, then the spread along that axis is less that that along any axis for which neither upper nor lower bounds intersect the BALL(p,A).

Let us now examine how this limits the number of buckets which intersect BALL(p,A), but have no corner within BALL(p,A). Let F(h;k1,k0) denote the number of such buckets generated in h more refinement steps of a bin having k1 axes with a single bound intersecting BALL(p,A), and k0 axes with no bounds intersecting BALL(p,A). Then we have F(h;k1,k0) =

F(h-1;k1,k0), if we choose a discriminating axis and partition which does not intersect BALL(p,A), or

2 F(h-1;k1+1,k0-1), if we choose a discriminating axis from the second set, and a partition which intersects BALL(p,A), or

$F(h-1;k1-1,k0)+F(h-1;k1,k0)$, if we choose a discriminating axis from the first set, and a partition which intersects BALL(p,A).

Note that if $k0=0$, then we have a bin which must have a corner which projects onto L within BALL(p,A), and any bucket within this bin must also have this property. Hence we have the end conditions $F(h;k,0)=0$ and $F(0;k1,k0)=1$, for $k0\neq 0$.

We now ask, what is the maximum possible value of $F(h;k1,k0)$? The solution to the recurrence relation defined above is

$$F(h;k1,k0) = 2^{k0-1}\binom{h-k0+1}{k0+k1-1} = C(h^{k0+k1-1}).$$

Although the set B may be bounded by preceding bin partitions of p which intersect BALL(p,A), the relevant bin partitions are those which have the same discriminant as L, and which of course lie on the same side of p. If BALL(p,A) does not reach a relevant preceding bin partition of p, we shall call L a "terminating" partition. Then the number of bins in B to be searched is limited by

$$F(\log N;1,k-1) = O(\log^{k-1} N).$$

If BALL(p,A) does reach a relevant preceding bin partition of p, then the number of bins in B to be searched is limited by

$$F(\log N;0,k-1) = O(\log^{k-2} N).$$

A terminating bin partition is called that because the search ball for p will not intersect any preceding relevant bin partition of p. Thus along each axis direction, the search ball intersects at most

$$\log N \; O(\log^{k-2} N) + O(\log^{k-1} N) = O(\log^{k-1} N)$$

buckets. Considering all directions, all points p, and a factor of logN to find each bucket, we have

$$TC'square(N,k) \leq 2^{k+1} N \; O(\log^{k-1} N) \; \log N = O(N \log^{k} N).$$

Substituting this in (2), we have

$$TSsquare(N,k) = O(N \log^{k} N). \quad \blacksquare$$

### 1.5.2 Lower Bound

We can exhibit a particularly bad case for the square k-d tree.

THEOREM: For $k \geq 2$, there exist point sets for which the actual search time using a square tree is

$$\Omega(N \log^{k} N).$$

PROOF: The construction is similar to that for the cyclic tree. The set A is constructed strung out at unit intervals along the positive x1-axis. All members of B will have all non-positive coordinates, and most will lie close to the x1-axis. Some points of B will be chosen so as to force elongation of some of the bins.

We shall be building sets of points composed of translated sets of points. Because these sets of points are assembled in layers, we shall call our structures "plats". In order to recall how much building has been done in constructing a plat, we shall give it an order number, such as "k-plat" where k is a positive integer.

Let $z = 1/(2N)$. Define a 1-plat of size M to be the set of M points in k-space, $(nz, 0, \ldots, 0)$, where $0 \leq n < M$. If the contents of some subbin of B was a 1-plat, the k-d tree nearest neighbor search for a point in A would take $\log M$ time to find that there is no closer neighbor in that subbin.

B shall now be defined as a structure composed of many translated 1-plats, such that the square k-d tree construction algorithm generates subbins coincident with the plats. The structure will be composed of repetitions of substructures, which will be composed of repetitions of substructures, and so on, down to 1-plats.

For $j>1$, define a j-plat of size M to be $\log M$ translated (j-1)-plats of decreasing size, and a single extra point. The ith subplat will be a (j-1)-plat of size

$$\frac{M}{2^i}$$

translated by the subtraction of $(i-1)z$ from from the jth coordinate of each point. The extra point's only non-zero coordinate will be the jth, which will be $-jN$. Thus the j-

plat's widest spread is along the jth axis. The square k-d tree construction procedure will partition between the largest (j-1)-plat and the rest of the subplats. This remainder will in turn be partitioned between the second largest (j-1)-plat and the remainder, and so on, down to the single extra point. It is easy to verify that the nearest neighbor search time for the j-plat would be at least

$$\binom{\log M}{j}.$$

We finally define B to a k-plat of size N/2. Considering that for each point in A, the nearest neighbor search must make an unsuccessful search of B, we obtain

$$TSsquare(N,k) = N/2\ \Theta(\log N)\ \Theta(\log^{k-1} N) = \Theta(N \log^{k} N).$$ ▪

Figure 4 gives an example of this construction for k=2, N=32.

### 1.5.3 Summary

Combining the preceding two theorems, we know that the worst case time for finding the nearest neighbor of all points using a square tree is bounded

$$TSsquare(N,k) = \Theta(N \log^{k} N).$$

This is within a factor of logN of the behavior of the algorithm of Bentley and Shamos, which was designed to solve the problem of finding the nearest neighbor of all points only. Thus we have determined the worst case performance of

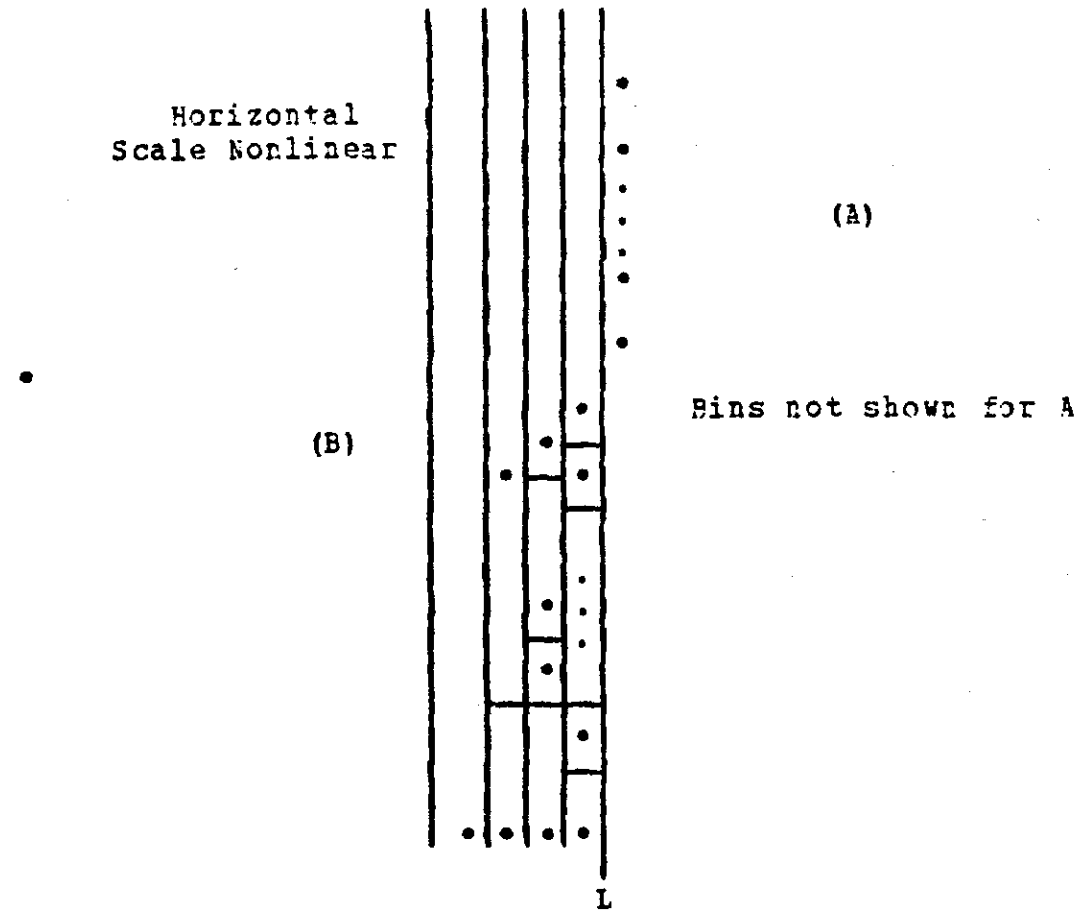



Figure 4: Worst Case Example of a Square Tree

the nearest neighbor search in a square k-d tree to within a constant factor. The above arguments can be extended with small changes to bucket sizes greater than one. The only effect is a change in the constant of proportionality.

## 1.6 SPREAD TREE NEAREST NEIGHBORING SEARCHES

Although Friedman, Bentley, and Finkel found that the expected behavior of the spread tree was quite good, and indeed was better than that of the cyclic tree, the worst case for the nearest neighbor search using the spread tree is worse.

We can construct examples for which time is used in the search for the nearest neighbor of all points is of the same order of magnitude as the brute force solution. We develop an example for k=2, in the x,y-plane. We choose the first N/2 points as (0,i), for $1 \le i \le N/2$. We choose the next N/2-1 points as (i/N,0), for $1 \le i \le N/2-1$. Lastly, we choose the Nth point as (N,0).

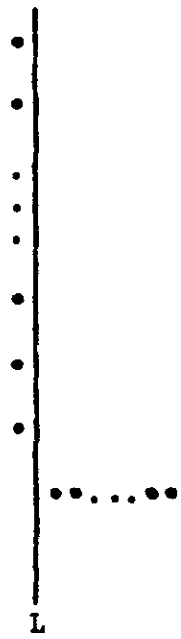



Figure 5: Worst Case Example of a Spread Tree

Now the spread in the x dimension is N, while along the y dimension it is N/2. Thus the first partition L will separate along the x-axis, between the points on the x-axis and those on the y-axis. Now the points on the x-axis have zero spread in the y dimension, so all partitions

of those points must have the x-axis as discriminant. Hence the search ball of each point on the y-axis must intersect the bin of each point on the x-axis. This forces the search time to be at least

$$(N/2)\ (N/2) = C(N^2).$$

Using a statistical definition of the spread still does not preclude the construction of such bad examples, as long as some fixed fraction of the points can be squeezed alongside the initial partition. This seems to indicate that in the creation of k-d trees, the geometry of the bins is more important than that of the point set. The choice of the median position seems to force the shape of the tree as much as is necessary.

## 1.7 CONCLUSION

### 1.7.1 Summary

Table 1 summarizes the known behavior of the k-d tree nearest neighbor search.

### 1.7.2 Further Research

We have not found the average behavior of the nearest neighbor search using a k-d tree. Despite the existence of strong empirical evidence, there is no analytic evidence that the k-d tree should have logarithmic expected search time. Our results hint that it should be possible to find an analytic logarithm-squared bound for the expected search time.

Table 1.

Search Time Behavior of K-d Trees

| TREE-TYPE | WORST CASE LOWER BOUND | WORST CASE UPPER BOUND | EXPECTED EMPIRICAL |
|---|---|---|---|
| CYCLIC | $\Omega(N^{2-1/k})$ | $O(N^{2-1/k}\log^2 N)$ | $O(N \log N)$ |
| SPREAD | $\Omega(N^2)$ | $O(N^2)$ | $O(N \log N)$ |
| SQUARE | $\Omega(N \log^k N)$ | $O(N \log^k N)$ | $O(N \log N)$ |

## Chapter II

## INTERSECTION OF A FINITE SET OF HALF-SPACES

Consider a set S of N half-spaces in three dimensions. The problem is to construct the polyhedron which represents the intersection of the half-spaces as quickly as possible. The lower bound for this problem is O(N logN) time, since any algorithm which can solve this problem could solve the corresponding problem in two dimensions, where the lower bound is O(N logN). Recent work by Preparata and Muller[1], and Brown[2] has led to algorithms which can solve this problem in O(N logN) time. We shall demonstrate here our algorithm for the solution of this problem in O(N logN) time. This would enable us to solve three-variable linear programming problems in time less than the worst case of the simplex algorithm.

---

[1] F. P. Preparata and D. E. Muller, "Finding the Intersection of a Set of n Half-spaces in Time O(nlogn)", University of Illinois Coordinated Science Laboratory Technical Report #R-803(ACT-7):UILU-ENG77-2250, December, 1977.

[2] K. Q. Brown, "Fast Intersection of Half Spaces", Draft, Carnegie-Mellon University, November, 1977.

## 2.1 TERMS

Each half-space is determined by a plane called the supporting plane and an orientation. The orientation specifies which of the two half-spaces defined by the supporting plane is in the set. For each half-space, consider the normal to the supporting plane pointing away from the half-space. If we follow a vector parallel to this normal from the center of the unit sphere at the origin, it intersects the sphere at a unique point. This point we shall call the corresponding sphere point. If x is a half-space in the set S, we shall refer to the supporting plane as the plane x, and the corresponding sphere point as the sphere point x. (See Figure 6).

The role to be played by the sphere points in the algorithm might be best illustrated by considering an extreme case of the problem. Suppose that all the half-spaces include a unit sphere, and that the supporting planes are all tangent to that sphere. Then the sphere points are the points of tangency, while proximate sphere points correspond to neighboring faces in the intersection polyhedron.

We shall use the term corner to mean the vertex of a polygon, and the term vertex to mean the vertex of a polyhedron.

On the plane, the convex closure of a set of points can be defined as the intersection of all half-planes which contain all points in the set, and the convex hull as those

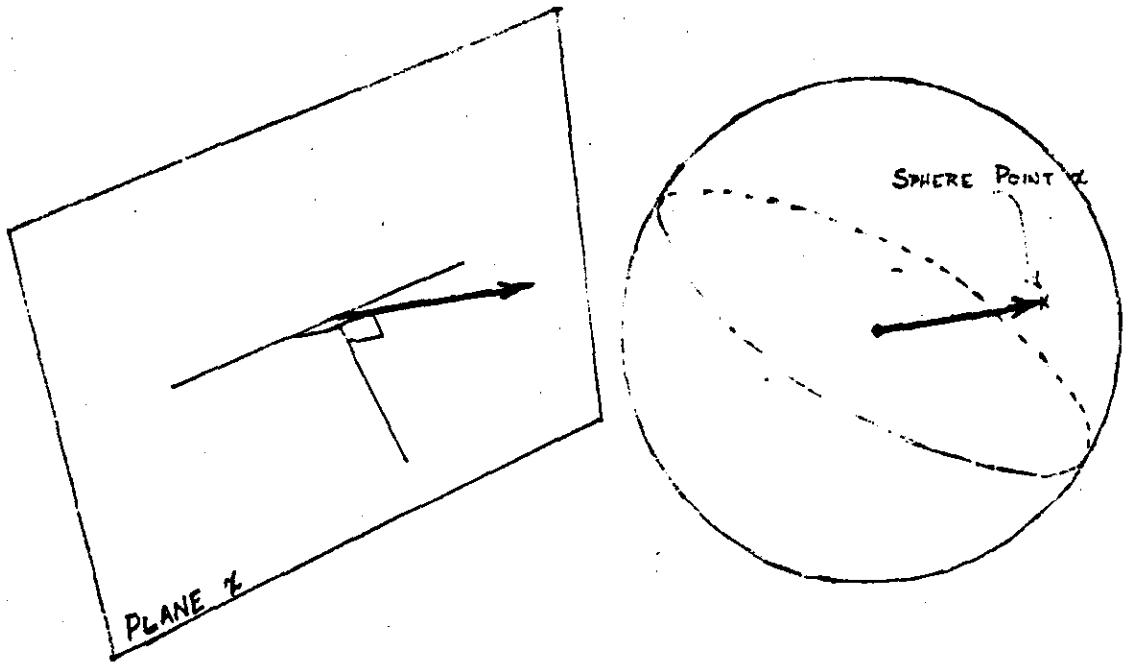



Figure 6: Connections between Half-space, Plane, and Sphere Pcint

points in the set on the boundary of the convex closure. On the surface of the sphere, we similarly define the convex closure of a set of points as the intersection of all hemispheres which contain all points in the set, and the convex hull as those points in the set on the boundary of the convex closure. For some point sets, such as the vertices of an inscribed regular tetrahedron, these structures do not exist. If a sphere point is in the convex hull, and does not lie between two other points on the convex hull, we shall say that it is independently in the convex hull. Points that are the in the convex hull, but are not inde-

pendently in the convex hull, would thus lie on the short segment of the great circle joining some pair of other sphere points; we shall say that these points are dependently in the convex hull.

By constructing the convex hull of a set, we mean the creation of a linked list of the points on the convex hull in order. By constructing the intersection polyhedron, we mean the creation of a data structure which encodes the planes which are faces of the intersection polyhedron and the relationship of "neighboring face". We shall call a face of a polyhedron an infinite face if it contains an infinite ray.

## 2.2 THE ALGORITHM

Our algorithm proceeds in three stages. In the first stage, the set of planes is partitioned into two parts. In the second stage, a divide and conquer method is used to construct the intersection of the half-spaces in each part. The method here is similar to the construction of the Voronoi diagram in the plane[3]. In the third stage, the polyhedra found in the second stage are intersected. It is not until this last stage that we shall discover whether the total intersection is finite or even null.

---

[3]M. I. Shamos, "Geometric Complexity", Conference Record of Seventh Annual ACM Symposium on Theory of Computing, (1975), pp. 224-233.

### 2.2.1 First Stage

We first partition the set S of N half-spaces into two parts. We do this by picking an arbitrary hemisphere of the unit sphere, then assigning a half-space to the first part if its sphere point lies in the hemisphere, and to the second part if the sphere point lies in the opposite hemisphere.

We shall find the following lemma useful.

LEMMA: Let H be a nonempty set of half-spaces whose corresponding sphere points lie within a hemisphere.

a. The set of sphere points of the members of H has a convex hull on the surface of the sphere.

b. If the supporting plane of a half-space appears as an infinite face of the intersection polyhedron of the half-spaces then the corresponding sphere point appears on the convex hull of the sphere points. If a sphere point is independently in the convex hull, the corresponding supporting plane appears as an infinite face of the intersection polyhedron. If a sphere point is dependently in the convex hull, the corresponding supporting plane either does not appear in the intersection polyhedron, or appears as an infinite face.

PROOF:

a. By definition, there is at least one hemisphere which contains all points in the set, and thus the intersection of all such hemispheres is well-defined.

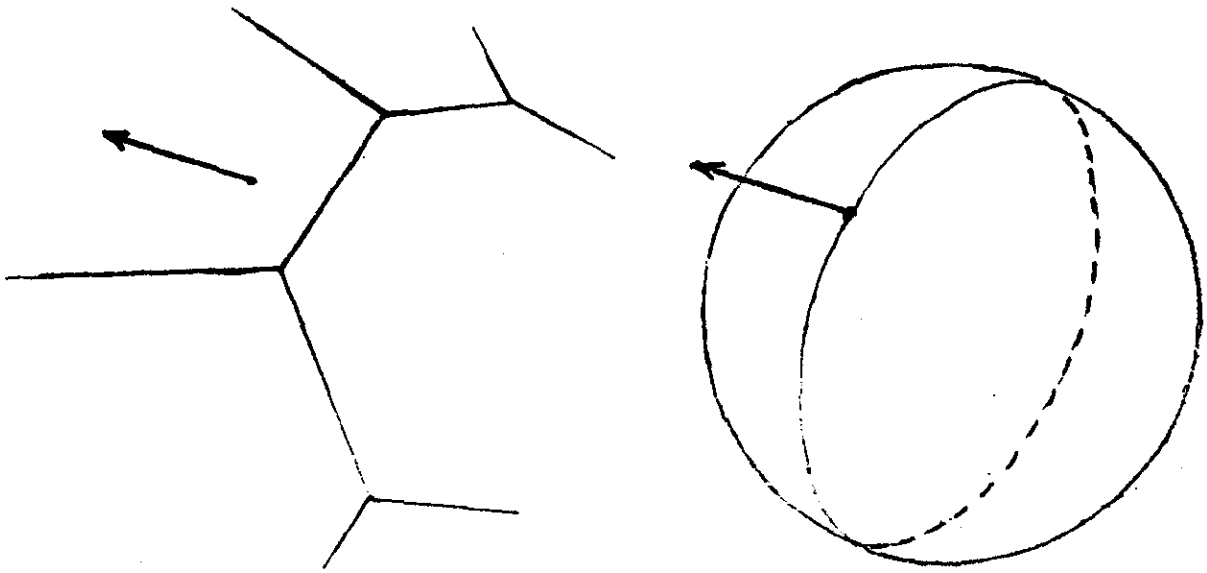


Figure 7: Relationship between Convex Hull and Infinite Faces

b. If a supporting plane x appears as an infinite face, by definition there is an infinite ray in that face. Consider the great circle passing through sphere point x, and orthogonal to the infinite ray (See Figure 7). The hemisphere generated by the great circle in the direction opposite the ray contains all sphere points of the set. For suppose there is some sphere point y not in this hemisphere. Then planes x and y must intersect such that y cuts off a terminal segment of the ray, which is contrary to the ray's being infinite. Since sphere point x is on the boundary of this hemisphere, it must also be on the convex hull.

Conversely, if sphere point x is on the convex hull between points y and z, where x is not on the great circle joining y and z, then there is a hemisphere with x on the boundary and all other sphere points in the interior. Thus the remaining planes can only cut off initial portions of the ray in the plane x which is orthogonal to the great circle and oriented in the direction opposite the hemisphere. Since there are only finitely many other planes, an infinite segment of that ray must remain in face x, and thus x is an infinite face. If x lies on the great circle be-

tween y and z, then face x may or may not be present; if it is present, it has infinite parallel edges perpendicular to the plane of the great circle yxz. ∎

2.2.1.1 Timing

Since we only need to examine each half-space once to assign it to a part, the first stage takes O(N) time.

## 2.2.2 Second Stage

### 2.2.2.1 Divide and Conquer

This stage makes use of a recursive divide and conquer procedure.

The procedure takes as an argument a set of half-spaces whose corresponding sphere points all lie in the same hemisphere. The procedure first partitions this set into two parts, applies itself recursively to each part to produce the convex hull of the sphere points and the intersection polyhedron of the half-spaces. It combines the two convex hulls to form the convex hull of the whole set. Lastly, it intersects the two partial polyhedra, to produce the intersection of all the half-spaces in the input set.

To separate the half-spaces, the procedure chooses a great circle which partitions the set of sphere points into two equinumerous parts. For example, if we had sorted all the points in a hemisphere by angle around some axis lying

in the diameter plane of the sphere, then we can choose the median of the angular coordinates to separate the set. The choice of a great circle as a partition ensures that the convex closures of the two parts are disjoint.

Suppose now that the intersection of the half-planes and the convex hulls of the sphere points have been found for each part by using the procedure recursively. We first combine the convex hulls to find the convex hull of all the points. Since the two convex closures are disjoint, from any sphere point on the first part we can find a link of the convex hull of the second part such that the point of the first part is known to be on the exterior side of that face of the convex closure. Likewise we can find a link of the first part which faces out to some point in the second part. We then delete these links, and link the two hulls together. Finally we iteratively delete concave vertices until no more exist. The worst case time to construct the convex hull of the entire set is proportional to the number of points on the convex hulls.

There exists in the combined convex hull a link from the first part to the second and another link back. These links are between points which by the above lemma correspond to planes which are infinite faces of the intersection polyhedron of all the half-spaces, and the intersection of these planes appears as an edge on that polyhedron. We can easily check for dependent presence on the convex hull and the loss of faces in linear time.

Applying the edge extension procedure described below by starting at the known infinite edge, we can generate the set of edges of the polyhedron determined by the intersection of the partial polyhedra. During the extension, some faces and edges of the partial polyhedra may be "cut off" and thus must be deleted. The edges and faces of the intersection polyhedron are those created by the extension procedure together with those in the partial intersection polyhedra which were not "cut off".

2.2.2.2 The Edge Extension Procedure

The edge extension procedure is an important part of the algorithm. This procedure starts from an edge that is known to be created by the intersection of two polyhedra, extends that edge in a specified direction, and generates all edges formed by the intersection along that direction. The polyhedra must have a special relation, namely that the convex hulls of the sphere points of the face planes are disjoint.

For example, say that the known edge e1 is the intersection of a face x of the first polyhedron and a face y of the second polyhedron. We extend this edge until it first intersects an existing edge of x or y, say edge e2 of y (See Figure 8). This edge is the intersection of y with a face z in the second polyhedron. We terminate e1 at the point of intersection with e2, and now begin the extension of the

edge e3 defined by the intersection of x and z. If e1 does not strike any opposite edge of x or y, then x and y are infinite faces, and the extension procedure terminates.

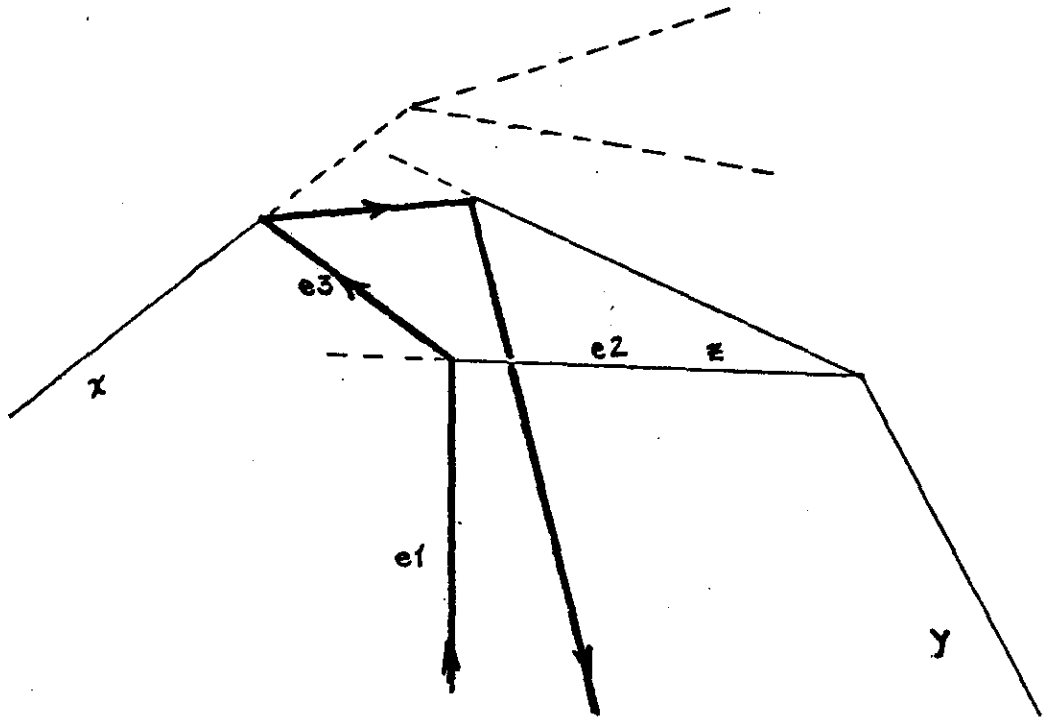



Figure 8: Operation of the Edge Extension Procedure

Since the search for the next edge of face x to be intersected by e3 can commence at that edge intersected by e1, we can design the search for the first intersecting edge such that the extension procedure takes time proportional to the number of edges.

The use of a great circle to separate the sphere points of the parts ensures that the partial polyhedra will have

the special relation specified above. If we were to project the intersection polyhedron onto the plane determined by the great circle, the intersection edges form the convex hull of the projection polyhedron. Thus we see that the relation forces a connected sequence of new edges and the extension procedure will generate all new intersection edges.

#### 2.2.2.3 Timing

Thus we can construct the convex hull of the corresponding sphere points and the intersection polyhedron of our set of half-spaces. The divide and conquer procedure is applied to both parts constructed in Stage 1. The combining of the convex hulls and application of the edge extension procedure each take C(N) time. Thus like the time for the construction of the Voronoi diagram in the plane, the execution time of Stage 2 is O(N logN).

### 2.2.3 Third Stage

Let us examine our current situation. We have reduced our N half-spaces to two polyhedra, both with infinite extent. To illustrate their relative orientation, we might say that these polyhedra are two flowers facing in exactly opposite directions. The relative position of these flowers is still unknown however. They may be facing each other, back to back, or displaced laterally.

If we can now find a pair of faces, one from each polyhedron, whose intersection appears as an edge in the polyhedron which is the intersection of all half-spaces, we can again apply the edge extension procedure to generate the final intersection. Most of the remainder of the algorithm is concerned with finding just such a pair of faces.

Let us examine just how we might find such a pair of intersecting faces. For descriptive purposes, we assume that the diameter plane used in Stage 1 is a horizontal plane, the first polyhedron is below all its faces, and the second is above all its faces. We shall call the first polyhedron down-facing, the second up-facing. We construct a list of the vertices of the polyhedra, augmented by the addition of virtual vertices at the zenith and nadir, and ordered by the vertical or z-coordinate.

Consider any horizontal plane passing through one of the real vertices in our list. Such a horizontal plane would intersect a fixed set of faces of the polyhedra, determined by the chosen vertex. The intersection of each polyhedron with the horizontal plane may be a convex polygon lying in the plane, may be null, or if the chosen vertex is virtual, may be the entire plane.

#### 2.2.3.1 Binary Search

Our global strategy in the search for a pair of intersecting edges will be a binary search. Our search investi-

gates the region between a pair of vertices in our sorted list. We use the horizontal planes passing through the vertices, and generate the polygons which are the intersections of the planes with the polyhedra. Thus we can construct the intersection of each polyhedron with the slab delimited by the horizontal planes passing through our chosen vertices. We call these intersection polyhedra slices, and observe that their vertices all lie in one or the other of the limiting horizontal planes and that the non-horizontal faces, or "sides", have at most four edges and vertices.

We now look for an intersection edge of the two slices. We first project the top and bottom polygons of each slice into a single horizontal plane*. We then examine the intersection relationship between these four polygons in that plane.

If we intersect two convex polygons A and B in the plane, we have four possible outcomes. First, an edge of A may cross an edge of B; we shall call this an "actual" intersection. Secondly, we may find that A is within B, or thirdly, we may likewise find that B is within A. The fourth possibility is that the intersection is null since A

---

*For brevity, we shall assign two character names to each polygon. The first character will be U or D, if the polyhedron is the up-facing or down-facing, respectively. The second character will be T or B, if the intersection is with the top or bottom limiting plane, respectively. Thus UT is the projection of the intersection of the up-facing polyhedron with the top limiting plane. We note that UB is within UT and DT is within DB.

Table 2.

Decision Tree of Polygon Intersections

| | POLYGONS | ACTUAL | INTERSECTION RESULT A within B | B within A | NULL |
|---|---|---|---|---|---|
| 1 | UT-DB | 2 | 3 | 4 | DISJOINT |
| 2 | UT-DT | FOUND! | CAN'T BE | 1* | 5 |
| 3 | UT-DT | FOUND! | 3* | 2* | 2* |
| 4 | UB-DB | FOUND! | 1* | 4* | 1* |
| 5 | UB-DB | FOUND! | 2* | CAN'T BE | HILL CLIMB |

1* DT is within UT. Any edge of DT appears in the intersection polyhedron and even intersects a side face of the up-facing slice.

2* UB is within DB. Any edge of UB appears in the intersection polyhedron and even intersects a side face of the down-facing slice.

3* The up-facing slice is within the down-facing slice.

4* The down-facing slice is within the up-facing slice.

FOUND We have found a pair of intersecting edges, which are in the intersecting faces we wanted.

DISJOINT We have established that the slices do not intersect, and have a separating plane, which is the vertical extension of the separating line between the polygons UT and DB.

CAN'T BE This outcome will not arise.

HILL CLIMB We must "hill climb" to find a possible intersection.

and B are disjoint. Table 2 represents a decision tree of polygon intersections to be tested. We always start at 1 by finding the intersection of UT and DB.

2.2.3.2 The Hill Climb

Decision table 2 resolves the question of the intersection of the two slices, with the exception of the hill-climbing outcome. Let us examine the hill climb in detail.

The slices may yet have an intersection. If so, its projection lies within the intersection of UT and DB. For each point x in the horizontal plane, let US(x) denote the point on the surface of the up-facing polyhedron which projects onto x; similarly define DS(x) for the down-facing polyhedron. We define the function U(x) where x is in the intersection of UT and DB to be the z-coordinate of the point US(x). Thus U(x) is a piece-wise linear convex function on its domain. Similarly, we define D(x) as the z-coordinate of the point DS(x), and observe that it is a piece-wise linear concave function on its domain.

Now consider the function D(x)-U(x). It is a piece-wise linear concave function, although there may be more than O(N) pieces. If it has a value less than zero at x0, then the line segment joining US(x0) to DS(x0) lies outside both polyhedra. If it has a value greater than zero at x0, then the line segment joining US(x0) and DS(x0) lies within both polyhedra. The function D(x)-U(x) attains the value zero at x0 if and only if the polyhedra intersect at US(x0)=DS(x0). Thus if we do an uphill search on the value of D(x)-U(x), we can determine whether the polyhedra intersect in the slab.

We first evaluate D(x)-U(x) around the perimeter of the intersection of UT and DB. We know for instance that, at a corner generated by this intersection, the function has a value which is the negative distance between the limiting planes. We only need evaluate at the corners of the intersection polygon, since D(x)-U(x) is linear between corners. If the polygon has infinite edges, it is an easy matter to check whether the function becomes positive along that edge. We choose a maximal corner, and proceed to the actual search, which follows along the projections of the side edges of the slices.

Consider the general case of our search. We find ourselves at the intersection of the projections of an up-facing edge and a down-facing edge. There are two other edges in the upper slice adjacent to our current edge, and similarly for the down-facing edge. Thus we have three lines, which do not intersect each other while intersecting each of three similar lines (See Figure 9). Thus a total of nine intersection points are defined, the center one of which is our current position. In constant bounded time, we can evaluate D(x)-U(x) at the other eight points, and move on to the highest. If all are lower, then we are at the maximum and no intersection occurs. The fact that some of the eight new points may not be defined because we are too close to the perimeter can be handled in constant bounded time also. Note that we don't have to follow any edges induced by the slicing, since we started at the maximal perimeter corner.

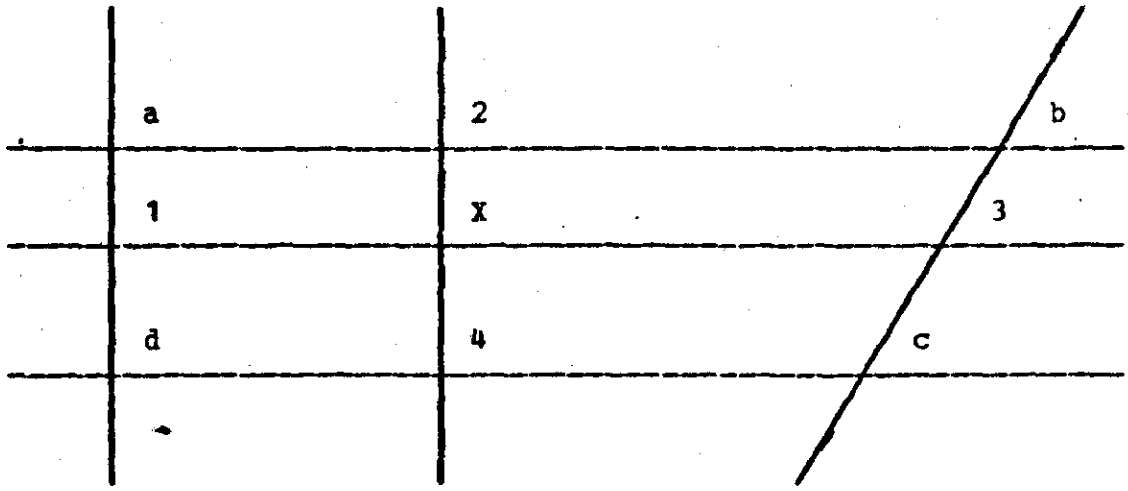



Figure 9: Typical Step Options

Of course, we want to know how many steps might be taken by the hill climb. As we noted above, within the intersection of UT and DB there are no intersections between up-facing edges, and none between down-facing edges. Thus the intersection pattern of the up-facing edges with the down-facing edges is like that of two families of parallel lines.

Now let us examine Figure 9. If the climbing procedure moves to one of the points adjacent to X, say 1, then we know that D(1)-U(1) is greater than each of D(a)-U(a), D(d)-U(d), and D(X)-U(X). But a1X and 1dX define bounding planes for the surface of the function D(x)-U(x), thus all points to the right of the line a1d evaluate to less than D(1)-U(1). Hence we can make steps to adjacent intersections at most N times in each direction, and thus at most 2N steps will be to adjacent intersections.

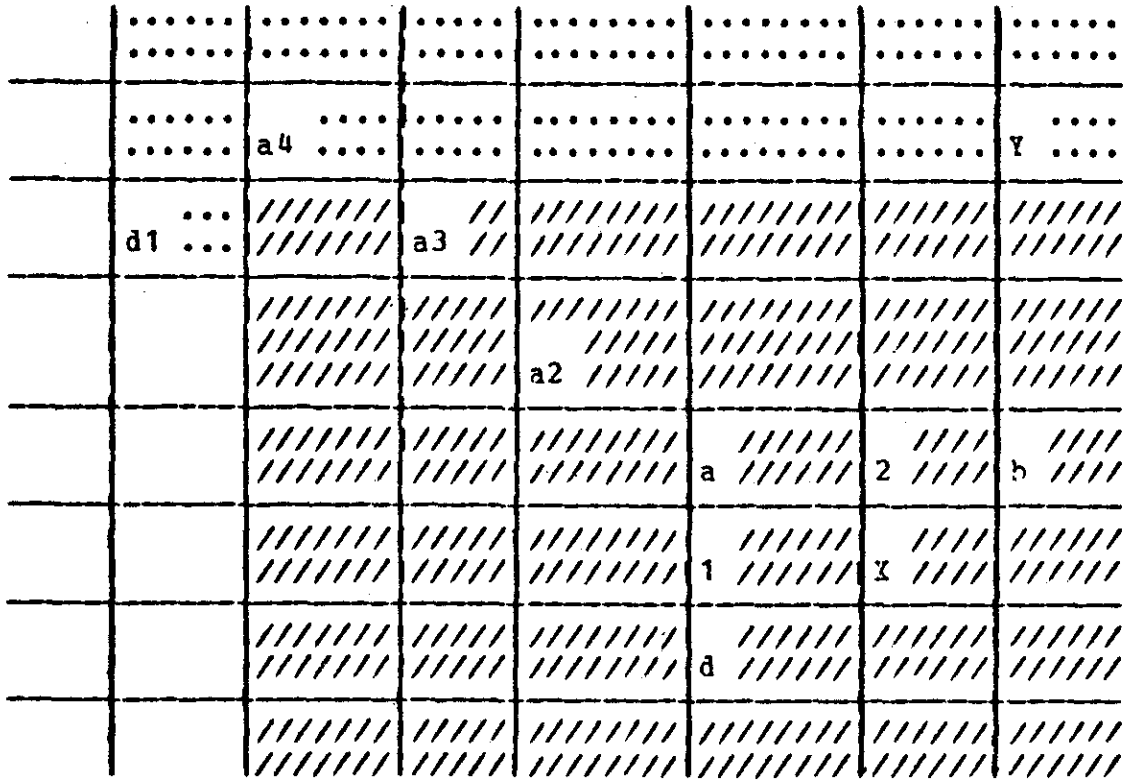



Figure 10: Example of Diagonal Stepping

Now consider a sequence of diagonal steps terminated by a step in some other direction, such as a,a2,a3,a4,d1 in Figure 10.

Since the function D(x)-U(x) has greater value at a than at 1, 2, or X, we see that the quadrant below the line a2b and right of a1d is now known to be lower. This is true for each a-type step, thus when we reach a4, we have dominated the slashed area. Then the step to d1 dominates the dotted quadrant also. But the total dominated area is the same as if we had started at Y and used 1-type steps to a4, then the single step to d1. Similarly we can replace any sequence of diagonal steps which is followed by a different

step with a sequence of adjacent steps of the same length. This sequence would also dominate the same set of intersection points. The only case left is a long sequence of the same steps, which as before can be at most length N. Thus any climb sequence has length at most 2N.

Of course, as we make the ascent, at each step we check whether we are crossing the threshold $D(x)-U(x)=0$. If we do cross it, we can stop, having found our intersecting faces. If the polygon is infinite, the search may terminate because there are no more intersection points in the direction of ascent. We can easily check whether the infinite faces which remain undominated do indeed intersect. If not, we find the maximum at the intersection of the projection of two edges. A plane parallel to both of these edges and lying between them is a separating plane for the slices.

### 2.2.3.3 Onward and Upward

We now know how the slices intersect. We have four possible conditions: the slices have a actual intersection, one slice is in the interior of the other, one or both of the slices is null, or the slices fail to intersect. In the case of a actual intersection, one of the edges of the intersection polyhedron is formed by the intersection of faces from each of the slices. This pair of faces is the pair we have been looking for, and we terminate our search.

If one slice is in the interior of the other or only one is null, our next investigation is in the direction of the polyhedron with the smaller or null slice. If both slices are null, we have a separating plane and hence the intersection of the two polyhedra must be null.

If the slices do not intersect, we need to find in which direction, up or down, we should move so that we might find a pair of intersecting faces. If one of the points is virtual, we are at one end of the list, so we know we must move in the other direction. Thus we may assume that both vertices of our pair are real. Each polyhedron is a subset of the larger polyhedron defined by the side faces of the slices. The intersection of these larger polyhedra is a convex polyhedron either strictly above or strictly below our slab. This intersection of the larger polyhedra contains the intersection of the original polyhedra, and thus both intersections must lie on the same side of the slab. While attempting to construct the intersection of the slices, we found no intersection, but we were able to construct a separating plane. This separating plane intersects the limiting planes in a pair of parallel lines. In each limiting plane, and for each polyhedron, find the corner of the intersection polygon closest to the parallel line lying in that limiting plane, and construct a parallel line through that point (See Figure 11). The two lines for each polyhedron define a plane which in fact is a bounding plane

for that polyhedron. The intersection of these planes is another parallel line outside the slab. We then move in the direction in which these planes intersect.

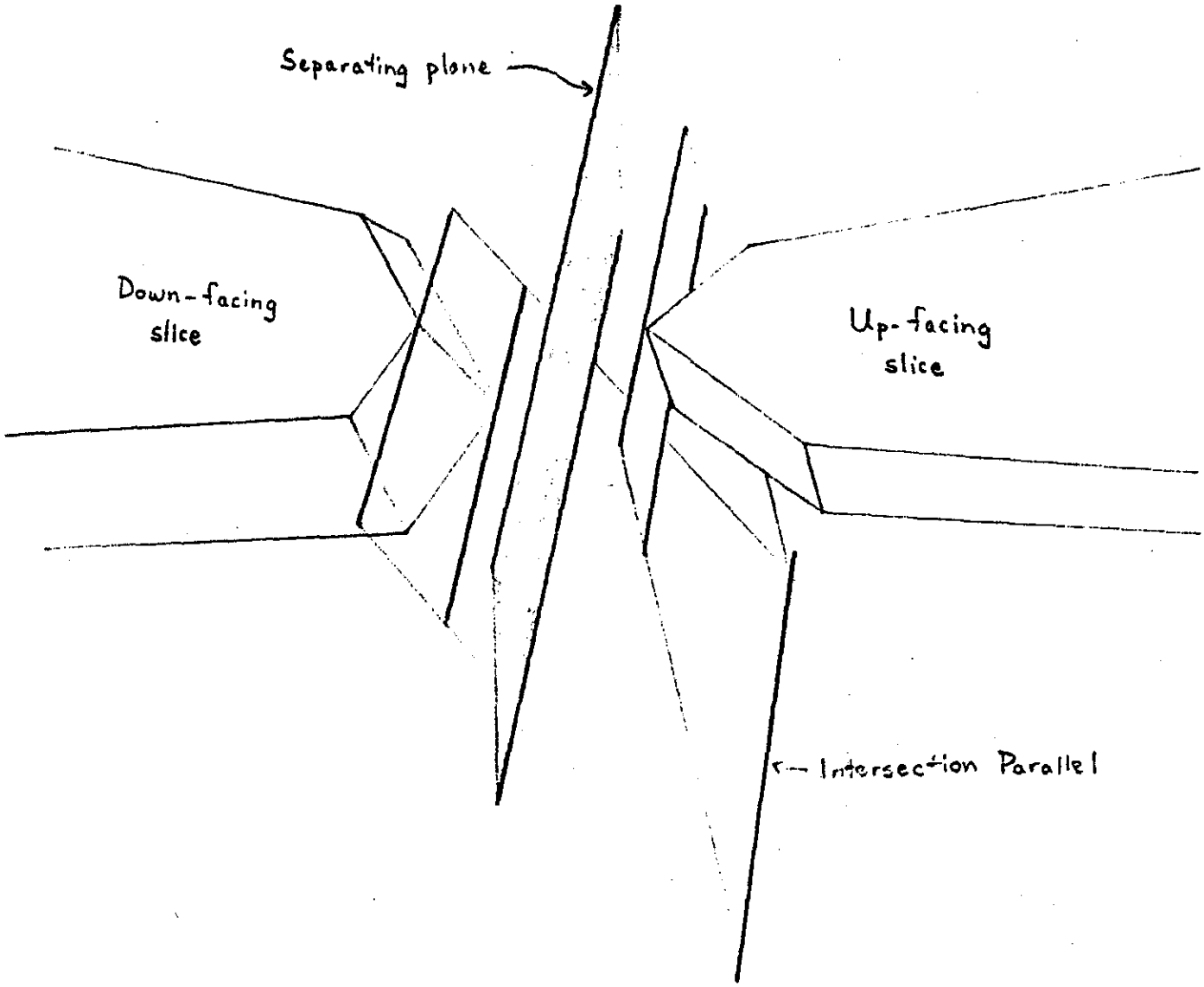



Figure 11: Choosing the Direction in the Binary Search

As long as we find no intersection and have not shown that one does not exist, we continue the search, until the

search directions indicate that the intersection must be both strictly above and below some vertex. This is impossible, so the intersection must be null.

#### 2.2.3.4 The Final Edge Extension

As mentioned above, once we have found an edge formed by the intersection of the up-facing and down-facing polyhedra, we can simply apply the edge extension procedure to generate the final intersection. Because we do not know that the final edge is infinite, we will need to apply the edge extension procedure in both directions. Also, because the intersection of the polyhedra may now be finite, we shall need to check for a loop, or more specifically, whether the current edge is between the same faces as the originally found edge.

#### 2.2.3.5 Timing

Each step in the search takes O(N) time: intersecting the horizontal planes with the polyhedron, doing the polygon intersection tests, climbing the hill, and choosing the direction of the next investigation. The binary search takes O(logN) steps, thus the total search time is O(N logN). The final edge extension will take O(N) time.

## 2.3 TIMING SUMMARY

Each of the Stages takes O(N logN) time to complete, thus the total running time of the algorithm is O(N logN).

## Appendix A

### The Bin Refinement Procedure

The following is a pseudo-Algol description of the bin refinement procedure.

```
PROCEDURE binRefine(NODE x);
   INTEGER axisChoice;
   REAL positionChoice;
   NODE lowx,highx;
COMMENT axisChoose(a,x) is a procedure to be specified
        elsewhere which selects the DISCRIMINANT to be used
        to partition the bucket x.  ;
axisChoose(axisChoice,x);
positionChoice = MEDIAN(x,axisChoice);
lowx = BUCKET({x|x[axisChoice]≤positionChoice});
highx = BUCKET({x|x[axisChoice]>positionChoice});
x = PARTITION(LOWSON=lowx,HIGHSON=highx,
         DISCRIMINANT=axisChoice,POSITION=positionChoice);
END binRefine;
```

## Appendix B

## The Nearest Neighbor Search

The following is a pseudo-Algol description of the nearest neighbor search.

```
PROCEDURE nearestNeighbor(NODE x, POINT test,bestYet);
   NODE xson;
IF x IS BUCKET
    THEN FOR i IN x
              IF distance(i,test)<distance(bestYet,test)
                   THEN bestYet = i;
    ELSE IF test[ DISCRIMINANT(x) ] ≤ POSITION(x)
              THEN xson = LOWSON(x);
              ELSE xson = HIGHSON(x);
         nearestNeighbor(xson, test, bestYet);
         IF test[ DISCRIMINANT(x) ] > POSITION(x)
              THEN xson = LOWSON(x);
              ELSE xson = HIGHSON(x);
         IF distance(test,bin(xson))≤distance(test,bestYet)
              THEN nearestNeighbor(xson, test, bestYet);
END nearestNeighbor;
```

## Appendix C

### Coconut Crunchies

| | | |
|---|---|---|
| 1 | 1/2 | cups butter |
| 1 | 1/2 | cups white sugar |
| 1 | 1/2 | cups brown sugar |
| 3 | | eggs |
| 1 | 1/2 | tsps vanilla extract |
| 3 | | cups flour |
| 1 | 1/2 | tsps double-acting baking powder |
| 3/4 | | tsps baking soda |
| 3 | | cups oatmeal (uncooked) |
| 3 | | cups cornflakes |
| 1 | | 7-ounce package shredded coconut |
| (1 | 1/2 | cups raisins) |
| (4 | | medium bananas) |

Melt butter, mix with sugars in large mixing bowl. Beat in eggs, and stir in vanilla. If using bananas, peel, mince and stir in.

Sift together flour, baking powder, and baking soda. Sift this mixture into the sugar mixture a cup at a time, then stir well. Similarly, add the oatmeal and cornflakes a cup at a time. Lastly, stir in the coconut, and raisins if desired.

Drop by large spoonfuls onto cookie sheet. Bake in 325° oven for 12 to 15 minutes. This recipe makes six to eight dozen cookies.

## BIBLIOGRAPHY


1. J. L. Bentley, "Multidimensional Binary Search Trees for Associative Searching", Communications of the ACM, 18(1975), pp. 509-517.

2. J. L. Bentley and M. I. Shamos, "Divide and Conquer in Multidimensional Space", Proceedings of the Eighth Symposium on the Theory of Computing, ACM, May 1976, pp. 220-230.

3. K. Q. Brown, "Fast Intersection of Half Spaces", Draft, Carnegie-Mellon University, November, 1977.

4. D. Dobkin and R. J. Lipton, "Multidimensional Searching Problems", Yale University Computer Science Research Report #34, October 1974.

5. J. H. Friedman, F. Baskett, and L. J. Shustek, "An Algorithm for Finding Nearest Neighbors", IEEE Transactions on Computers, October 1975, pp. 1000-1006.

6. J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Time", Stanford Linear Accelerator Center Report SLAC-PUB-1549, February 1975.

7. D. H. McLain, "Two dimensional interpolation from random data", Computer Journal, 19(1976), pp. 179-181.

8. F. P. Preparata and D. E. Muller, "Finding the Intersection of a Set of n Half-spaces in Time O(nlogn)", University of Illinois Coordinated Science Laboratory Technical Report #R-803(ACT-7);UILU-ENG77-2250, December, 1977.

9. M. I. Shamos, "Geometric Complexity", Conference Record of Seventh Annual ACM Symposium on Theory of Computing, (1975), pp. 224-233.

10. M. I. Shamos, "Geometrical Intersection Problems", Proceedings of the 16th Annual Symposium on Foundations of Computer Science, (1975), pp. 208-273.

11. T. P. Yunck, "A technique to identify nearest neighbors", IEEE Transactions on Systems, Man, and Cybernetics, 6(1976), pp. 678-683.

12. G. Yuval, "Finding nearest neighbors", Information Processing Letters, 5(1976), pp. 63-65.