INFORMATION AND CONTROL **68**, 105–124 (1986)

New Upper Bounds for Neighbor Searching

B. CHAZELLE*

Brown University, Providence, Rhode Island 02912

R. COLE[†]

New York University, New York, New York 10012

F. P. PREPARATA[‡]

University of Illinois/Urbana–Champaign, Urbana, Illinois 61801

AND

C. YAP[†]

New York University, New York, New York 10012

This paper investigates the circular retrieval problem and the k-nearest neighbor problem, for sets of n points in the Euclidean plane. Two similar data structures each solve both problems. A deterministic structure uses space $O(n(\log n \log\log n)^2)$, and a probabilistic structure uses space $O(n \log^2 n)$. For both problems, these two structures answer a query that returns k points in $O(k + \log n)$ time. © 1986 Academic Press, Inc.

1. INTRODUCTION

Intersection search problems have a central place in computational geometry, and have been the object of intense study in recent years. These problems are formulated as follows: "Given a *file* S of n points in d-dimensional space E^d, and a *query*, represented by a domain D of E^d of chosen

* This research was supported in part by NSF Grants MCS 83-03925 and ONR and DARPA under contract N00014-83-K-0146 and ARPA Order 4786.

[†] This work was done under the auspices of the NYU/CIMS Laboratory for Robotics and Experimental Computer Science, which is supported by grants from Digital Equipment Corporation, The Sloan Foundation, and ONR Grant N00014-82-K-0381. Also supported in part by NSF Grant DCR-84-01633 and by an IBM Faculty Development Award.

[‡] This research was supported in part by Joint Services Electronics Program under Contract N00014-79-C-0424.

0019-9958/86 $3.00

type (defined by a fixed number of real parameters), report all the points of S contained in D." Searching occurs repeatedly, and the file S can be organized into a data structure particularly suited to achieve a fast query response. Thus—as is typical of all searching problems—there are three significant measures of performance: the storage $M(n)$ of the data structure, the query time $Q(n, k)$ (where k is the size of the retrieved set), and the preprocessing time $P(n)$ necessary to construct the data structure. The one-time cost $P(n)$ is usually assumed to be less important than the continuing costs $M(n)$ and $Q(n, k)$. Moreover one wishes that $Q(n, k)$ be of the form $O(f(n)+k)$, which exhibits two separate additive terms, one dependent upon the file size (search overhead), the other given by the size of the retrieved set. In such a case the pair $(M(n), f(n))$ is an appropriate measure of the performance of a searching technique.

The investigation of intersection searching problems has met with substantial success in the case of hyperrectangular ranges, where the search domain D is the Cartesian product of d one-dimensional intervals, each on a distinct coordinate axis (Bentley, 1975; Chazelle, 1983; and Willard, 1978). Here the task is facilitated by the fact that each one-dimensional interval identifies a "slice" of E^d, within which the original d-dimensional problem can be suitably transformed into a small set of $(d-1)$-dimensional problems. Unfortunately, no such property can be exploited for another important class of ranges, that of spherical ranges, where D is a sphere (in the L_2 metric), specified by its center q in E^d and a radius d. Since the (continuous) set of queries can be partitioned into (a finite number of) equivalence classes, each class being identified by the retrieved set (a subset of S), a possible approach to spherical range searching consists of classifying the query in its equivalence class and then accessing the subset of S associated with that class. Such an approach would exhibit the desired behavior $Q(n, k) = O(f(n)+k)$, but presumably $M(n)$ would be prohibitively large. Most of the known results concern the plane $(d=2)$, and we also refer to this case in this paper. Although our method is potentially extensible to higher dimensions, for that reason we speak of *circular range search.*

The original idea of the technique can be traced back to a paper by Bentley and Maurer (1979). Since circular range search involves proximity, they used higher order Voronoi diagrams as the structure supporting the search. Specifically, given a set S of points in the plane, and a nonempty subset T of S, $V(T)$ denotes the set of points of the plane that are closer to each member of T then to any member of $S-T$. The *kth order Voronoi diagram* of S, denoted $\mathrm{Vor}_k(S)$, for $k = 1,..., n-1$ is defined as

$$\mathrm{Vor}_k(S) = \bigcup_{|T| = k;\, T \subseteq S} V(T).$$

We say that k is the *scope* of $\mathrm{Vor}_k(S)$. Clearly, $\mathrm{Vor}_k(S)$ is a partition of the plane (a planar subdivision) and for some T the region $V(T)$ may be empty (only $O(n^3)$ of the $2^n - 2$ possible choices of T yield a nonempty $V(T)$ (Lee, 1982). Higher order Voronoi diagrams have been extensively studied (Lee, 1982; Preparata and Shamos, 1985) and we simply recall two of their salient properties ($k = 1, ..., n-1$):

(i) $\mathrm{Vor}_k(S)$ is a planar graph, whose vertices have degree $\geqslant 3$, with $O(k(n-k))$ vertices, edges, and faces;

(ii) each region of $\mathrm{Vor}_k(S)$ is a (possibly unbounded) convex polygon. As for any planar subdivision, a query point can be located in a region of $\mathrm{Vor}_k(S)$ in time $O(\log n)$, using a data structure stored in space $O(k(n-k))$ (Kirkpatrick, 1983; Lipton and Tarjan, 1980).

The technique of Bentley and Maurer uses the sequence of Voronoi diagrams ($\mathrm{Vor}_{2^i}(S)$: $i = 0, 1, ..., \lceil \log n \rceil - 1$) (as usual, logarithms are taken in base 2); associated with each region $V(T)$ of any of these diagrams is the list of the members of T, called the *neighbor list*. The circular range search proceeds as follows: Given a query (q, d), where q is a point (the center) and d is a positive real number (the length of the radius), q is successively located in $\mathrm{Vor}_{2^i}(S)$ for $i = 0, 1, 2, ...$, and the neighbor list is examined until a point is first encountered that lies further than d from q. At this stage, we know that the desired points all lie in the neighbor lists examined so far, and only the last one may contain undesired points.

The analysis of this method is both simple and revealing. First, we note that the storage requirement of $\mathrm{Vor}_{2^i}(S)$ (search data structure and neighbor lists) is $O(2^{2^i}(n - 2^i))$, and the global requirement, for $i = 0, ..., \lceil \log n \rceil - 1$, is $O(n^3)$. Second, if k denotes the size of the target set (i.e., the set within the query range), the total size of the examined neighbor lists is at most $1 + 2 + 4 + \cdots + 2^{\lfloor \log k \rfloor + 1} < 4k$; moreover, if $k < \log n \log \log n$ the search overhead (point location in $O(\log k)$ Voronoi diagrams) dominates the size of the retrieved set, while the opposite holds for $k \geqslant \log n \log \log n$. Thus $Q(n) = O(k + \log n \log \log n)$, resulting in an $O(n^3, \log n \log \log n)$ algorithm. The high storage requirements are clearly undesirable.

The technique displays—in an elementary form—an algorithmic concept recently fully developed by Chazelle (1983): the concept of *filtering search*. This notion prescribes to trade tradictional searching techniques for a two-step approach: scoop-and-filter. The idea is to collect (scoop) a set of $O(k)$ points that is guaranteed to include the k desired ones and, in a second stage, filter out the extraneous items. In this paper we show how a subtle and thorough exploitation of this approach leads to an $O(n(\log n \log \log n)^2, \log n)$ algorithm for circular range search. This represents the first attractive solution for a problem that for several years has eluded the development of an efficient algorithm. The only efficient solution for circular range search previously known uses linear space and allows us to answer any query in time $O(k+n^\alpha)$, for α slightly less than 1 (Yao, 1983). Before describing and analyzing the algorithm, in the next section we introduce a data structure which is crucial to the efficiency of the technique.

We also show that this algorithm (with minor changes) solves the k-nearest neighbor problem, with the same complexity. In this problem, the query consists of a pair (q, k) and the answer is the set of k points closest to q (if there are ties for the kth nearest point, we return any one—we will use this convention throughout). Finally, we describe a probabilistic method for building the underlying data structure that gives our algorithm a complexity of $O(n \log^2 n, \log n)$.

2. The kth Neighbor Diagram

Given an n point set $S = \{p_1, ..., p_n\}$ in the plane and an arbitrary point q, the *kth neighbor* of q in S is some $p_j \in S$ such that $\text{dist}(p_j, q)$ is the kth term in the ascending sequence $\{\text{dist}(p_i, q)\colon i = 1, ..., n\}$. The kth neighbor diagram for S, denoted $\text{near}_k(S)$, is the partition of the plane into the regions of points that have identical kth neighbors. We now elucidate the structure of $\text{near}_k(S)$.

Consider first the diagram $\text{Vor}_{k-1}(S)$. The reader is referred to Lee (1982) and Preparata and Shamos (1985) for a thorough treatment of higher order Voronoi diagrams. Referring to Fig. 1, each region of $\text{Vor}_{k-1}(S)$ (a convex polygon) is associated with a subset T of S of cardinality $k-1$. To construct $\text{Vor}_k(S)$ we partition polygon $V(T)$ in

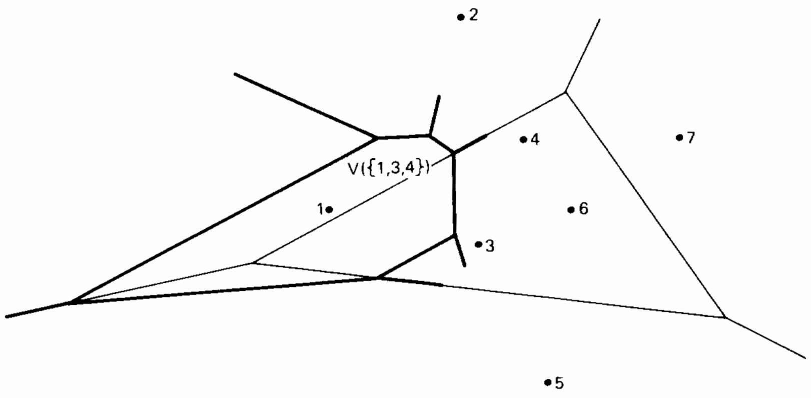

Figure 1

$\mathrm{Vor}_{k-1}(S)$ by intersecting in with $\mathrm{Vor}_1(S-T)$: in this manner $V(T)$ is partitioned into a collection of convex polygons, each belonging to $V(T \cup \{p_j\})$ for some $p_j \in S-T$. Assuming that $V(T \cup \{p_j\}) \neq \emptyset$, consider now the polygon $Q_j = V(T \cup \{p_j\}) \cap V(T)$ (a convex polygon, being the intersection of two convex polygons). For any $q \in Q_j$, T is the set of the $(k-1)$ points of S closest to q, while $T \cup \{p_j\}$ is the set of k closest points: it follows trivially that p_j is the kth neighbor of q, according to the earlier definition, or equivalently, that Q_j belongs to the region associated with p_j in $\mathrm{near}_k(S)$. This discussion also shows

$$\mathrm{near}_k(S) = \mathrm{Vor}_{k-1}(S) \cup \mathrm{Vor}_k(S).$$

This simple relation has the following interesting consequences:

1. If the points of S are in general position (no four are cocircular), each vertex of $\mathrm{near}_k(S)$ has either degree three or degree six: the degree-6 vertices are exactly those that are Voronoi vertices both in $\mathrm{Vor}_{k-1}(S)$ and $\mathrm{Vor}_k(S)$, whereas the degree-3 vertices are Voronoi vertices either in $\mathrm{Vor}_{k-2}(S)$ and $\mathrm{Vor}_{k-1}(S)$ or in $\mathrm{Vor}_k(S)$ and $\mathrm{Vor}_{k+1}(S)$;

2. $\mathrm{near}_k(S)$ is a planar graph with $O(k(n-k))$ vertices, edges, and faces. Inded, the sets of the vertices and edges of $\mathrm{near}_k(S)$ are the unions of the homologous sets of $\mathrm{Vor}_{k-1}(S)$ and $\mathrm{Vor}_k(S)$, and all the latter have cardinalities $O(k(n-k))$;

3. Point location in $\mathrm{near}_k(S)$ can be done in $O(\log n)$ time, using a data structure stored in $O(k(n-k))$ space (Kirkpatrick, 1983; Lipton and Tarjan, 1980).

We also observe that, since $\mathrm{near}_k(S)$ has $O(k(n-k))$ faces, the regions R_j of $\mathrm{near}_k(S)$ associated with $p_j \in S$ consists of $O(k)$ polygons on the average. It is also relatively easy to show that the polygons of R_j form a chain, two consecutive terms of which share a vertex (a degree-6 vertex of $\mathrm{near}_k(S)$). An instance of $\mathrm{near}_k(S)$, for $|S| = 7$ and $k = 4$, is shown in Fig. 2. The search algorithm described in the next section will require that with each face Q of $\mathrm{near}_k(S)$ we associate the neighbor list of the region of $\mathrm{Vor}_k(S)$ containing Q. Our next objective is to exhibit a representations of $\mathrm{near}_k(S)$ and its neighbor lists that can be stored in only $O(k(n-k))$ storage.

We denote by $\mathrm{Del}_k(S)$—the *Delaunay* graph of order k on S_n (Lee, 1982; Preparata and Shamos, 1985—the well-known dual graph of $\mathrm{Vor}_k(S)$, where vertices of the former and faces of the latter are in one-to-one correspondence, and adjacent vertices on $\mathrm{Del}_k(S)$ correspond to adjacent faces in $\mathrm{Vor}_k(S)$ (faces are adjacent if they share an edge and not just a vertex). We have already observed that if no four points in S are co-circular, the vertices of $\mathrm{Vor}_k(S)$ have degree three, therefore $\mathrm{Del}_k(S)$ is a

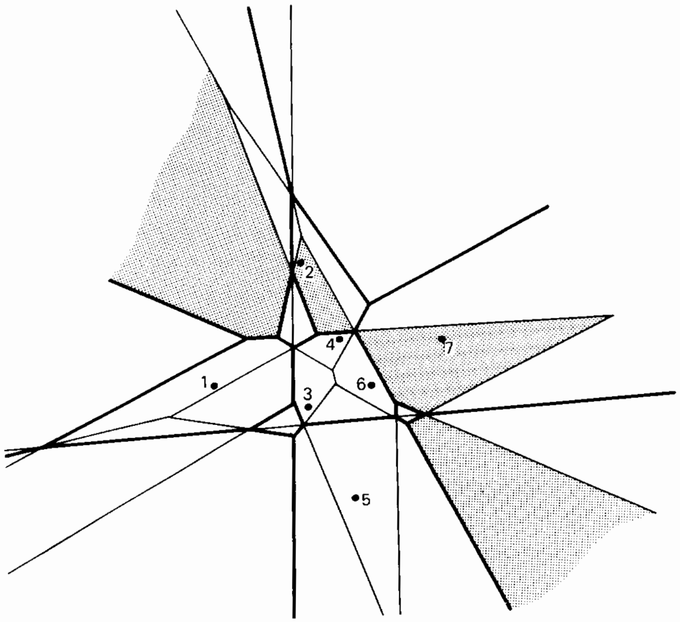

FIGURE 2

triangulation. In all cases, $\mathrm{Del}_k(S)$ is connected, which makes it possible to define a spanning tree of $\mathrm{Del}_k(S)$, denoted T_k. For the sake of simplicity, we transform T_k into a binary tree T^* (i.e., a tree with all degrees at most three), by reducing high degrees if necessary. To do so, assume that v is a vertex of T_k of degree $m > 3$ and let $v_1, \ldots, v_m$ be its adjacent vertices in clockwise order. We replace v by $m-2$ vertices $w_1, \ldots, w_{m-2}$, defined as follows: w_1 is adjacent to v_1, v_2, w_2, and w_{m-2} to w_{m-3}, v_{m-1}, v_m; each other vertex w_i is adjacent to w_{i-1}, v_{i+1}, w_{i+1} (cf. Fig. 3).

This transformation of T_k at most doubles the original number of vertices of $\mathrm{Del}_k(S)$: indeed, let μ denote the number of vertices of $\mathrm{Del}_k(S)$ and let v_i be the number of vertices of degree i in T_k. If δ is the maximum degree in T_k, we have $\sum_{1 \leqslant i \leqslant \delta} v_i = \mu$ and $\sum_{1 \leqslant i \leqslant \delta} i v_i = 2(\mu - 1)$ (since T_k is a tree). Let $|T^*|$ denote the number of vertices of T^*. Since a vertex of T_k of degree $i \geqslant 3$ is replaced by $i-2$ vertices in T^*, the size of T^* is given by

$$|T^*| = v_1 + v_2 + \sum_{3 \leqslant i \leqslant \delta} v_i(i-2) = \sum_{1 \leqslant i \leqslant \delta} i v_i - v_2 - 2 \sum_{3 \leqslant i \leqslant \delta} v_i < 2\mu - 2.$$

The following is a folk theorem; thus given without proof.

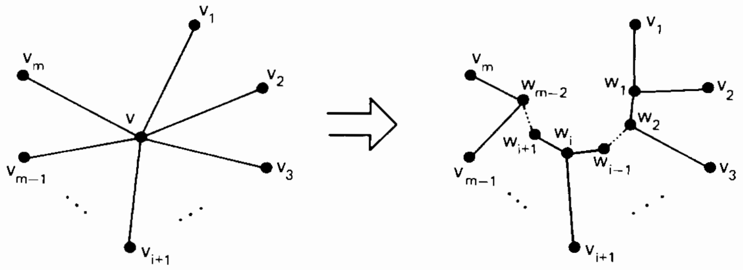

FIGURE 3

FACT. *Let T be a binary tree with m vertices. It is possible to find, in $O(m)$ operations, an edge of T whose removal leaves two (connected) subtrees $T^{(1)}$ and $T^{(2)}$, with ar most $2m/3$ vertices each.*

The decomposition process embodied by this fact can be applied recursively on the tree T^*, until we achieve a decomposition of the original tree into connected subtrees $T_1^*, ..., T_l^*$, each one having between k and $3k$ vertices. This is always possible since in each splitting step each component has at least one third as many vertices as its parent. Note also that a given face of $\mathrm{Vor}_k(S)$ (mapped to a vertex of $\mathrm{Del}_k(S)$) may be represented as a vertex in several of the subtrees $T_1^*, ..., T_l^*$ due to the node splitting incurred in the transformation of T_k to T^*. We will, however, allow only one instance to be accessible in the search process.

Let us now refer to one such subtree T_i^* and consider the neighbor lists associated with each of its vertices. We argue that these neighbor-lists cannot be substantially different from each other. Let $\sigma(u)$ be the neighbor list of the face of $\mathrm{Vor}_k(S)$ that corresponds to the vertex u of T_i^*. Two adjacent vertices $u, v \in T_i^*$ correspond either to the same face of $\mathrm{Vor}_k(S)$ or to two adjacent faces. For this reason, $\sigma(u)$ and $\sigma(v)$ differ in at most one element (i.e., the symmetric difference of $\sigma(u)$ and $\sigma(v)$ has cardinality either 0 or 2). This allows us to set up an implicit representation of neighbor-lists within each subtree T_i^*. The simplest solution consists of merging all the neighbor lists pertaining to T_i^* into a superset $S_i = \bigcup_{u \in T_i^*} \sigma(u)$ for $i = 1, ..., l$. Since T_i^* does not have more than $3k$ vertices and each neighbor-list has exactly k elements, we have the relation $|S_i| < 4k$. Since on the other hand, $|T_i^*| \geqslant k$, we have $l \leqslant |T^*|/k$, from which we readily derive $\sum_{1 \leqslant i \leqslant l} |S_i| \leqslant l \max |S_i| < (|T^*|/k)\ 4k = 4|T^*|$. The inequality $|T^*| < 2\mu - 2$ derived earlier implies that $\sum_{1 \leqslant i \leqslant l} |S_i| < 8\mu$.

We can now give a complete specification of the search data structure for given S and k, denoted $N_k(S)$. The diagram $\mathrm{near}_k(S)$ is preprocessed and organized for efficient planar point location (Kirkpatrick, 1983; Lipton and

Tarjan, 1980). With each face f of $\text{near}_k(S)$ we associate the index i of a subtree T_i^* containing the dual vertex of f (in case several vertices qualify as duals of f the tie is broken arbitrarily); with this index we associate, via a pointer, the list of the members of set S_i defined above. Since the planar point location structure can be stored in space proportional to the number $O(k(n-k))$ of vertices of $\text{near}_k(S)$, and the total storage requirement of $\{S_i \mid i=1,..., l\}$ is bounded by $8\mu = O(k(n-k))$, our claim is established. We note that using $N_k(S)$ we can find the k nearest neighbors of a point in time $O(k+\log n)$.

3. The Circular Range Search Problem

We begin with an informal description of the approach. The main drawback of Bentley and Maurer's method is to use higher order Voronoi diagrams with prohibilively large scopes. We can circumvent this difficulty by turning to the fundamental principle of filtering search: *the larger the output, the more naive the search.* The idea is to estimate increasingly tighter lower bounds on the number of points to be reported and use these bounds as credits to pay for less efficient searching.

An essential ingredient of the method is an efficient device for handling queries which generate small output. Let h be an arbitrary positive integer (h determines the scope). Let T be any subset of S; we define $L_h(T)$ as the set of structures

$$L_h(T) = \{N_{2^i \lfloor \log n \rfloor}(T) \mid 0 \leqslant i \leqslant \max(0, \lceil \log(h/\lfloor \log n \rfloor) \rceil)\}.$$

For consistency, we define $N_k(T) = T$ if $k \geqslant |T|$. In this way, $L_h(T)$ is always well defined, even for very small subsets T. Returning to the original problem, we show how to use $L_h(S)$ to answer any query (q, d), provided that the output size k does not exceed h. To do so, visit the structures $\{N_{2^i \lfloor \log n \rfloor}(S)\}$ in the order $i = 0, 1, 2,...,$. Using $N_{2^i \lfloor \log n \rfloor}(S)$ find the $2^i \lfloor \log n \rfloor$th neighbor p of q. If p is found to lie at a distance from q greater than d (or if all the structures in $L_h(S)$ have been examined), then retrieve the $2^i \lfloor \log n \rfloor$ nearest neighbors of q and stop; otherwise proceed to the next structure. Let j be the value of i upon termination. Note that since the largest scope in $L_h(S)$ is at least h, the set of neighbors reported from the structure $N_{2^j \lfloor \log n \rfloor}(S)$ includes the k desired points. Filtering the set completes the query answering in $O(k)$ time. It is immediate to see that the cost of all the planar point locations (except for the first one) is of the order of the number of points reported, therefore the running time of the algorithm is $O(k+\log n)$. It easily follows from Section 2 that the space required to store $L_h(S)$ is $O(hn)$.

We are now ready to describe the data structure for the general case. The main (*primary*) component is a complete binary tree T whose leaves are in one-to-one correspondence with the points of S. In this section we do not make any assumption on the nature of this correspondence. Later in this paper, we show how particular assignments can be used to our advantage (see probabilistic algorithm in Sect. 6). A search, prompted by a query (q, d), will be viewed as the visit of a subtree $T_{q,d}$ of T, denoted T_q when d is understood, where the term "subtree" refers here to any connected subgraph of T that contains the root. With each node $v \in T$, we associate a set of points $S(v) \subseteq S$ and a scope $k(v) > 0$. $S(v)$ is defined as the set of points whose corresponding leaves have v as a common ancestor, and $k(v)$ is for the time being left unspecified. $k(v)$ can be viewed as a parameter to the algorithm in the sense that it conditions its efficiency but not its correctness; any assignment of scopes to the nodes of T will provide a correct algorithm. The *secondary* structure at node v is simply defined as the pair $(\text{near}_{k(v)}(S(v)), L_{k(v)}(S(v)))$.

From this description of the data structure, we can outline the algorithm for answering a query (q, d). Starting at $v = \text{root}$, the algorithm operates recursively as follows: retrieve the $k(v)$th neighbor of q by searching $\text{near}_{k(v)}(S(v))$. If this point lies further than d from q, the output size must be within $k(v)$, so searching $L_{k(v)}(S(v))$ will complete the computation. Otherwise, we must pursue the exploration of T; to do so we distinguish between two cases. If v is a leaf of T, we report the unique point associated with it; in this case $L_{k(v)}(S(v)) = S(v)$ is a singleton. Otherwise no reporting takes place; instead, we iterate on the same process with respect to the two children of v.

The algorithm is trivially correct, so we need only investigate its running time $Q(n, k)$. As mentioned earlier, the computation involves the visit of a subtree T_q. Let $D_{q,d}$ (or D_q if d is understood) denote the tree obtained by removing from T_q each of its leaves (i.e., nodes with outdegree 0 with respect to T_q). Note that D_q may contain internal (i.e., non-leaf) nodes with outdegree 0. The following lemmas are central to the ensuing analysis.

LEMMA 1. $Q(n, k) = O(k + (1 + |D_q|) \log n)$.

Proof. Let v be an arbitrary node of T_q. If v is a leaf of T_q, the k' points of $S(v)$ which lie within a distance d of q are reported at a cost of $O(k' + \log n)$ steps, as derived earlier in the discussion of the structure $L_h(T)$. Otherwise the only cost incurred at v is that for a planar point location, i.e., $O(\log n)$. This shows that $Q(n, k) = O(k + |T_q| \log n)$, from which our claim is easily derived. ▌

Our next task is to provide a judicious assignment of scopes to nodes so that

$$|D_q| \log n = O(k). \tag{1}$$

From Lemma 1, this will ensure that $Q(n, k) = O(k + \log n)$. Suppose that for each node $v \in T$, we have $k(v) = \lceil \log^2 n \rceil$. Let m be the number of leaves of D_q. Each of these m nodes yields at least $\lceil \log^2 n \rceil$ points in $S(v)$ within a distance d from q, and no point appears in more than one set $S(v)$, therefore $k \geqslant m\lceil \log^2 n \rceil$. Since on the other hand $|D_q| \leqslant m\lceil \log n \rceil$ (indeed, the depth of D_q is bounded by $\lceil \log n \rceil$), we have $|D_q| \log n \leqslant m\lceil \log^2 n \rceil \leqslant k$, which satisfies (1) and therefore guarantees a running time $Q(n, k) = O(k + \log n)$. The amount of storage, $M(n)$, required by the data structure is easily shown to be $O(n \log^3 n)$.

A more careful assignment of scopes to the nodes of T can lead to a substantial reduction in space, as we proceed to show next. Let the *level* of node $v \in T$, denoted $l(v)$, be the number of ancestors of v (including itself). The smallest level is 1 (the root) and the maximum $\lceil \log n \rceil + 1$. We define $z(v)$ as the maximum number of trailing 0's (i.e., consecutive 0's starting from the right) in the binary representation of $l(v)$. We easily check that for any $v \in T$ and $n \geqslant 4$,

$$0 \leqslant z(v) \leqslant \lfloor \log(1 + \lceil \log n \rceil) \rfloor \leqslant \log \log n + 1. \tag{2}$$

Our most efficient algorithm for the circular range search problem will be attained for the following scope assignment:

$$k(v) = 2^{z(v)} \log n \log \log n. \tag{3}$$

After a trivial lemma, we will be ready to establish the major result of this section.

LEMMA 2. *Let i, j be two positive integers with $i < j$. The sequence $\{i, i+1, \ldots, j-1, j\}$ must contain at least one integer p ($p \geqslant (i+j)/2$) whose binary representation contains at least* $\log(j - i + 1) - 2$ *trailing* 0*'s.*

Proof. Any sequence of 2^u consecutive integers must contain an integer with at least u trailing 0's, therefore among the $\lceil (j-i+1)/2 \rceil$ candidate integers, at least one will have at least $\lfloor \log \lceil (j-i+1)/2 \rceil \rfloor$ trailing 0's. ▮

THEOREM 1. *It is possible to solve the* circular range search *problem in $O(k + \log n)$ time, using $O(n(\log n \log\log n)^2)$ storage; k denotes the number of points to be reported. The preprocessing time is $O(n \log^5 n (\log \log n)^2)$.*

Proof. Evaluating the running time is the most delicate part of the complexity analysis, so we will start with the analysis of space and preprocessing time. Consider the sequence of consecutive levels in T, $\{1, 2, ..., l\}$ with $l = 1 + \lceil \log n \rceil$. Exactly $l_0 = \lceil l/2 \rceil$ have no trailing 0's, exactly $l_1 = \lceil (l-1)/4 \rceil$ have one trailing zero, and more generally, exactly $l_i = \lceil (l - 2^i + 1)/2^{i+1} \rceil$ have i trailing 0's. This implies that at most $\lceil l/2^{i+1} \rceil$ levels have exactly i trailing 0's. We evaluate the storage by adding up upper bounds to the contributions of all levels with exactly 0, 1, 2,... trailing 0's. From (2) and (4) it follows that $M(n) = O(\sum_{0 \leqslant i \leqslant \log\log n + 1} (l/2^{i+1}) 2^i n \log n \log\log n)$, hence

$$M(n) = O(n(\log n \log\log n)^2). \tag{4}$$

To evaluate the preprocessing time, we use Lee's (1982) algorithm for constructing both $\text{near}_k(S)$ and $\text{Vor}_k(S)$ in time $O(k^2 |S| \log |S|)$. We easily derive that the construction time is

$$O\left(\sum_{0 \leqslant i \leqslant \log\log n + 1} (l/2^{i+1})\, 2^{2i} (\log n \log\log n)^2\, n \log n \right),$$

that is, $O(n \log^5 n (\log\log n)^2)$.

Turning now to the time complexity of the algorithm, we wish to show that condition (1) is satisfied. To do so, we partition D_q into a collection of node-disjoint paths, $P_1, ..., P_m$, with m again being the number of leaves of D_q. P_1 is the longest path from the root of T to a leaf of D_q. This path breaks off D_q into a forest of trees each of them rooted at a child of a node of P_1. Remove P_1 from consideration at this point, and let P_2 be the longest path in the forest. The parent of the root from which P_2 emanates is a node of P_1, called the *parent* of P_2. Iterating on this process yields $P_3, ..., P_m$ (Fig. 4).

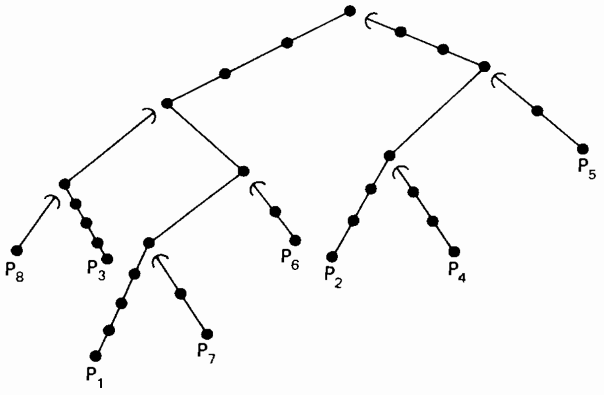

FIGURE 4

Since from the discussion of the structure $L_h(T)$ we know that each leaf of T_q "pays for itself," the computational costs to be accounted for are those of he paths $P_1, ..., P_m$ resulting from the decomposition of D_q. Specifically, path P_i has a search cost $|P_i| \log n$. This cost must now be charged to the search "payoff," i.e., to points of S retrieved at the leaves of T_q. Let $P_i = \{v_1, ..., v_t\}$, with $l(v_{j+1}) = l(v_j) + 1$ for each j $(1 \leqslant j < t)$. From Lemma 2, we know that for some p such that $(t+1)/2 \leqslant p \leqslant t$, we have $z(v_p) \geqslant \log t - 2$, therefore $k(v_p) \geqslant (t/4) \log n \log\log n$. The node v_p is called the *creditor* of P_i and is denoted w_i. Note that if $t = 1$, we have $w_i = v_1$. Observe that, since $w_i \in D_q$, each point in $S(w_i)$ will appear in the retrieved set; a difficulty—to be confronted shortly—is that a reported point p may belong to more than one set associated with a creditor node.

The charging scheme assigns the cost of P_i uniformly to the points of $S(w_i)$. Since $|S(w_i)| \geqslant (|P_i|/4) \log n \log\log n$, each $p \in S(w_i)$ is charged at most $4/\log\log n$ by P_i. We now evaluate the maximum number of times a point p may appear in sets $S(w_1), ..., S(w_m)$. Suppose that p is retrieved at a leaf of T_q whose parent is a node of path P_i. With reference to Fig. 5, we ascend towards the root and traverse nodes of $P_{i_1}, P_{i_2}, ..., P_{i_h}$ in this order. Assume now that $p \in S(w_{i_a})$ and $p \in S(w_{i_b})$, for $a < b$, and that for each $a < c < b$, $p \notin S(w_{i_c})$. The condition $p \in S(w_{i_a})$ and $p \in S(w_{i_b})$ implies that w_{i_b} is an ancestor of all nodes of P_{i_a}. The mechanism that constructs the path decomposition of D_q ensures that $|P_{i_a}|$ is less than the path length from w_{i_b} to the leaf of P_{i_b}; moreover, since by construction w_{i_b} belongs to the lower half od P_{i_b}, we conclude that $2|P_{i_a}| \leqslant |P_{i_b}|$. It immediately follows that p can be shared by at most $\log\log n$ creditor sets. Recalling that each

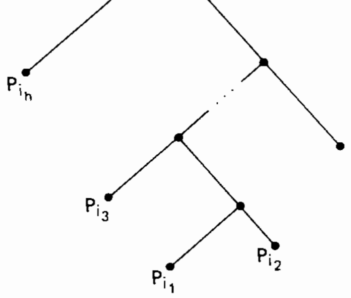

FIGURE 5

creditor set charges at most $4/\log\log n$ units to a point we conclude that each point is given a charge $O(1)$, thereby collectively absorbing the search cost. As a final comment we observe that the total cost of visiting D_q, i.e., $|D_q| \log n$ is upper bounded by the total payoff of the creditor nodes, $\sum_{1 \leqslant i \leqslant m} k(w_i)$, multiplied by $4/\log\log n$; but $\sum_{1 \leqslant i \leqslant m} k(w_i) \leqslant k \log\log n$, whence

$$|D_q| \log n \leqslant 4k. \tag{5}$$

The proof is now complete. ▌

4. The k-Nearest Neighbor Problem

We show in this section how a local transformation of the algorithm for the circular range search problem gives an efficient solution to the k-nearest neighbor problem. The query is now a pair of the form (q, k) and the output is the set of k points in S closest to q. Let δ be the distance from q to its kth neighbor. The problem can be solved by applying the previous algorithm to the query (q, δ). Since unfortunately δ is unknown, we must find a guiding criterion which can work as a substitute for the knowledge of δ. The algorithm will be described in two stages.

Let T_q and D_q be the subtrees of T defined as in the previous section (with respect to the pair (q, δ)). Let $n_q(S(v))$ be the $k(v)$th nearest neighbor of q in $S(v)$, and let $d(p, q)$ denote the Euclidean distance between p and q. Visiting T_q involves growing a subtree of T, denoted T^*, in the following manner: the current leaf v of T^* with the *minimum* value $d(q, n_q(S(v)))$ is chosen and its two children (if any) are added to T^*. When the tree T^* reaches a certain threshold size, i.e., when its number of nodes equals a predetermined value α, or when no more nodes in T are available, whichever happens first, we stop the process and retrieve the k-nearest neighbors of q from the sets $\{S(v) \mid v \text{ is a leaf of } T^*\}$. This terminates the first stage. We implement these operations by maintaining a priority queue with the values $d(q, n_q(S(v)))$. Every time nodes are added, we search the relevant $\text{near}_{k(v)}(S(v))$ structures to update the priority queue.

We must choose a threshold α, so small that the visited tree T^* is guaranteed to contain T_q as a subtree. We claim that $\alpha = \lceil 1 + 8k/\log n \rceil$ is an adequate choice. Let T^* be the tree obtained by the process outlined above, with $|T^*| = \alpha$. Since the visit of each node of T^* costs time $O(\log n)$, the running time of the completion of this stage is $O(k + \log n)$. From (5) we derive that $|T_q| \leqslant 2|D_q| + 1 \leqslant \alpha = |T^*|$, which verifies the necessary condition $|T^*| \geqslant |T_q|$. We now show that T_q is a subtree of T^*. For each leaf v of T_q that is neither the root nor a leaf of T, we have $d(q, n_q(S(v))) > \delta$ and

$d(q, n_q(S(w))) \leqslant \delta$, where w is the parent of v. The priority queue mechanism ensures that the children of a leaf of T_q cannot be added to T^* before all the leaves of T_q have been added. Since $\alpha \geqslant |T_q|$, our claim is now obviously true. This shows that the set of structures $\{L_{k(v)}(S(v)) \mid v$ is a leaf of $T^*\}$ contains all the necessary information for us to retrieve the k-nearest neighbors of q. Recall that $L_{k(v)}(S(v))$ is a set of preprocessed Voronoi diagrams with geometrically increasing scopes; $L_{k(v)}(S(v)) = \{L_0(v), L_1(v), ...\}$ with $L_i(v) = N_{2^i \lfloor \log n \rfloor}(S(v))$.

We now begin the second stage of the algorithm, the search of $\{L_{k(v)}(S(v)) \mid v$ is a leaf of $T^*\}$. To aid the intuition, let us figuratively attach to each leaf v of T^* a chain of nodes, each storing one of the structures $L_0(v)$, $L_1(v)$,..., in this order. This representation allows us to restart the computation in a similar fashion. We continue growing the tree T^* by visiting the nodes of these added chains, according to essentially the same criterion as before. For the sake of consistency, we refer to the child of a chain-node as its successor. Adding to T^* the ith node in the chain from v involves searching $L_i(v)$, examining the $2^i \lfloor \log n \rfloor$ nearest neighbors of q in $S(v)$ and computing the maximum distance from q to one of these points. This value is then inserted in the priority queue. At the beginning of this second stage of the algorithm, we assume that the priority queue contains the distance from q to its ($\log n$)th neighbor in $S(v)$, for each leaf v of T^*. At the generic step, the node corresponding to the top of the queue is selected, and its child (if any) is added to T^*; of course its corresponding entry is deleted from the priority queue, while the child's entry is inserted. As usual, as soon as the tree T^* grows over a certain threshold, β, we stop the process. The measure of growth will be slightly different here, however. Let $N(w, q)$, or simply $N(w)$ if q is understood, be the set of neighbors associated with a node w of some chain; if w is the ith node of the chain, we have $|N(w)| = 2^i \lfloor \log n \rfloor$. The process terminates as soon as no new node is available or the sum of all quantities $|N(w)|$ for all leaves of T^* exceeds β; this sum is denoted $C(T^*)$. At this point we form the set W, defined as the union of all the sets $N(w)$ for each leaf w of T^*. Finally, using a linear selection algorithm (Blum, Floyd, Pratt, Rivest, and Tarjan, 1973), we retrieve the k nearest neighbors of q from the set W.

Of course, it is crucial for the correctness of the algorithm to ensure that the set W does indeed contain the k nearest neighbors of q. To do so, we set $\beta = 8k + 2 \log n$. As in the previous case, the priority queue guarantees that as long as T^* has a leaf w with a child in its chain, yet with all the points in $N(w)$ within δ from q, the child of a leaf w' with a point in $N(w')$ further than δ from q will never be added to T^*. Let z be any node of a chain and let z' be its child. Since $N(z')$ is twice as large as $N(z)$, it seems that setting $\beta = 2k$ should be suficient. Unfortunately, it is easy to see that in the worst case, the leaves of T^* in its final stage will be the first nodes of all the chains, except for one of them. Let t be the number of leaves of T^*; in order to allow for the consideration of all relevant neighbors, we must ensure that $C(T^*)$ can reach the value $t\lfloor \log n \rfloor + 2k$. Since $t \leqslant |T_q| \leqslant \alpha = \lceil 1 + 8k/\log n \rceil$, β does not have to exceed $10k + 2 \log n$. The time cost of this second phase of the algorithm is clearly proportional to β, so here again, the running time is $O(k + \log n)$. We will note that the threshold values α and β could be fine-tuned to improve the algorithm by constant factors. As in Theorem 1, we easily derive that the time to construct the data structure in $O(n \log^5 n (\log\log n)^2)$.

THEOREM 2. *It is possible to solve the* k-nearest neighbor *problem in* $O(k + \log n)$ *time, using* $O(n(\log n \log\log n)^2)$ *storage;* k *denotes the size of the output. The preprocessing time is* $O(n \log^5 n(\log\log n)^2)$.

5. TRADING TIME FOR SPACE

It is possible to lower the space requirements of the previous algorithms at the price of some increase in the query time. We will present an $O(n \log n)$-space data structure that allows a query to be answered in time $O(k \log^2 n)$. Let T be a complete binary tree defined over the n points of S; each leaf of T corresponds to a distinct point, with the n points appearing sorted by ascending abscissa x from left to right. Each node v of T spans a subset $S(v)$ of S consisting of the points stored at the leaves of the subtree rooted at v. The preprocessing involves computing the (order 1) Voronoi diagram of each subset $S(v)$. Each Voronoi diagram is preprocessd for efficient planar point location. Using divide and conquer, the data structure can be computed in time $O(n \log n)$ (Preparata and Shamos, 1985).

Consider now the k-nearest neighbor problem. We answer a query (q, k) by first computing the nearest neighbor of q in S, using the structure $\text{Vor}_1(S(\text{root}))$, where $S(\text{root}) = S$. Next we visit the two offsprings of the roots and proceed as in the method described in the preceding section; in the present case the priority queue yields the neighbors of q in order of increasing distance. There are a few obvious modifications, suggested by the special nature of the problem: let p be the point just extracted from the top of the queue, and let v be the corresponding node in T. It is easy to see that p will "drag" the computation all the way down to the leaf, w, where it is stored. Once this leaf has been reached, we will delete p from the queue and iterate. Note that it is useless to search the structures $\text{Vor}_1(S(z))$ encountered on the path from v to w, since this will always produce the same answer, i.e., p. Instead, we shall just visit the siblings of the nodes on this path, thereby cutting to a half the computational search work. Thus, since the number of nodes of T visited by the search is $O(k \log n)$ (actually,

$O(k \log(n/k))$, as can be easily shown) and each visit has a cost of $O(\log n)$, the running time is $O(k \log^2 n)$. The same technique applies to the *circular range search* problem as well (discarding the priority queue for which we have no use).

THEOREM 3. *It is possible to solve the* circular range search *problem and the k*-nearest neighbor *problem in $O(k \log^2 n)$ time, using $O(n \log n)$ space; in both cases, n (resp. k) denotes the input (resp. output) size. The preprocessing time is $O(n \log n)$.*

6. THE PROBABILISTIC METHOD

We say this method is probabilistic because the algorithm to build the data structure is probabilistic. However, once the data structure is built the searches are wholly deterministic. We use the data structure described in Section 3. The scope at node v is $k(v) = c \log|S(v)|$, if $|S(v)| \geqslant n_0$, and is $|S(v)|$ otherwise, c and n_0 being constants chosen below. We define $\text{Nearest}_k(X, x)$ to be the k points in X closest to x. We assign the points S to the leaves of T in such a way that for each $v \in T$, with $|S(v)| \geqslant n_0$, if w is a child of v, then there is a constant d such that for any location x:

$$|\text{Nearest}_{k(w)}(S(w), x) - \text{Nearest}_{k(v)}(S(v), x)| \geqslant d \log|S(v)|. \qquad (6)$$

In the Appendix we show that such an assignment can be found by a probabilistic algorithm.

It is clear the space used by this data structure is $O(n \log^2 n)$. Next, we show that a query runs quickly, assuming we have assigned the points of S in the manner described above.

LEMMA 3. $Q(n, k) = O(k + \log n)$.

Proof. The following abbreviation is useful:

$$N(A, q) = \text{Nearest}_{c \log|A|}(A, q).$$

Define $\text{New}(w, q) = N(S(w), q) - N(S(v), q)$, where v is the parent of w in T.

Consider a query (q, d); let T_q and D_q be as in Section 3. Let w be an arbitrary node of T_q. If w is a leaf to T_q, the k' points of $S(w)$ that lie within a distance d of q are reported at a cost of $O(k' + \log|S(w)|)$ steps, as derived earlier. Otherwise, if w is not a leaf of T_q, the only cost incurred at w is that for a planar point location, i.e., $O(\log|S(w)|)$; at each such vertex w (except the root) there are $\theta(\log|S(w)|)$ points in $\text{New}(w, q)$, each one within a distance d of q. So the cost incurred at nodes in D_q is bounded by $O(\log n + \sum_{w \in D_q - \{\text{root}\}} |\text{New}(w, q)|)$. We next show that the sets $\text{New}(w, q)$ are disjoint. Let v_1 and v_2 be the children of v; since $\text{New}(v_i, q) \neq \varnothing$, $i = 1, 2$, we deduce $N(S(v), q) \subseteq N(S(v_1), q) \cup N(S(v_2), q)$, where v_1 and v_2 are the children of v. This implies, for w an ancestor of v_i, that $N(S(w), q) \cap N(S(v_i), q) \subseteq N(S(v), q)$, $i = 1, 2$. Hence $\text{New}(w, q) \cap \text{New}(v_i, q) = \varnothing$ for w a proper ancestor of v_i, $i = 1, 2$. For w not a proper ancestor of v_i, since $S(w) \cap S(v_i) = \varnothing$ we have $\text{New}(w, q) \cap \text{New}(v_i, q) = \varnothing$. Hence the sets $\text{New}(w, q)$ are disjoint. Thus $\sum_{w \in D_q - \{\text{root}\}} |\text{New}(w, q)| = |\bigcup_{w \in D_q - \{\text{root}\}} \text{New}(w, q)| = O(k)$. So the cost incurrent at nodes in D_q is $O(k + \log n)$. The cost incurred at leaves of T_q is bounded by

$$O\left(k + \sum_{w \in T_q - D_q} \log |S(w)|\right) \leqslant O\left(k + 2 \sum_{v \in D_q - \{\text{root}\}} \log |S(v)| + \log n\right)$$

$$\leqslant O\left(k + \sum_{v \in D_q - \{\text{root}\}} |\text{New}(w, q)| + \log n\right) = O(k + \log n).$$

So the search time is bounded by $O(k + \log n)$. ∎

In light of our previous results, we can conclude.

THEOREM 4. *It is possible to solve the* circular range search *problem and the* k-nearest neighbor *problem in $O(k + \log n)$ time, using an $O(n \log^2 n)$ data structure computed by a probabilistic algorithm; k denotes the size of the output.*

7. CONCLUSIONS

The contribution of this work has been to propose economical methods for solving a number of neighbor problems in the Euclidean plane. In all the problems considered, the size of the output varies as a function of the input. This allowed us to use filtering search and, by doing so, save storage. None of our algorithms will be effective, however, if instead of asking for a list of points meeting the specifications of the query, one was to require, say, the cardinality of the set, or for that matter any single-valued function thereof. The existence of efficient algorithms for handling these cases is still an open question. As we already mentioned, we believe that our technique can be generalized to higher dimensions: carrying out the generalization in all its particulars, however, remains to be done.

8. Appendix

We show that we can assign the points of S in the manner claimed. It suffices to show that given a set X of n points, $n \geqslant n_0$, we can divide X into two disjoint sets A_1 and A_2, each of $n/2$ points, such that for each location x:

$$|N(A_i, x) - N(X, x)| \geqslant d \log |X|, \qquad i = 1, 2.$$

For the probabilistic construction we further require that at least half the possible pairs of sets (A_1, A_2) satisfy this condition.

As remarked in Section 1 there are $O(cn \log |X|)$ possible sets $N(X, x)$ (the neighbor lists corresponding to the faces of $\mathrm{Vor}_{c \log |X|}(X)$; let f be the constant of proportionality.

Claim (proved below). There is a choice of c such that given $fcn \log n$ sets $X_i \subseteq X$, each of size $c \log n$, at least one half of the disjoint pair of sets A_1, $A_2 \subseteq X$, each of size $n/2$, satisfy $|X_i \cap A_j| > g \log n$, for some constant $g > 1$, and for all $1 \leqslant i \leqslant fcn \log n$, $j = 1, 2$, and $n \geqslant n_0$.

Let the sets X_i in the claim be the $fcn \log n$ possible sets $N(X, x)$. We have

(i) $|X_i \cap A_2| > g \log n$

(ii) $|N(A_1, x) \cap N(X, x)| = c \log n - |X_i \cap A_2|$, where $X_i = N(X, x)$.

(iii) $|N(A_1, x)| = c \log(n/2)$.

So $|N(A_1, x) - N(X, x)| > c \log(n/2) - c \log n + g \log n = g \log n - c \geqslant d \log n = d \log |X|$, with $d = g - 1$, $|X| \geqslant 2$. As $g > 1$, $d > 0$. Thus a proof of the claim implies a probabilistic algorithm for assigning points to the leaves of T, in the manner described.

Proof of Claim. Let $c = 8/\log(27/16)$. We count the choices of A_1 which make one of the conditions fail, and show that with probability $\geqslant \frac{1}{2}$ a random choice of A_1 succeeds, for $n \geqslant n_0$, and some $g > 1$.

Number of choices of A_1 violating condition $|X_i \cap A_1| > g \log n$,

$$= \sum_{0 \leqslant i \leqslant g \log n} \binom{c \log n}{i} \binom{n - c \log n}{n/2 - i}.$$

Number of choices of A_1 violating some condition,

$$\leqslant 2fcn \log n \sum_{0 \leqslant i \leqslant g \log n} \binom{c \log n}{i} \binom{v - c \log n}{n/2 - i}.$$

Number of Choices of $A_1 = \binom{n}{n/2}$. Taking $g = c/4$, and for large enough n, the fraction F of choices violating some condition is given by

$$F \leqslant 2fcgn(\log n)^2 \binom{c \log n}{g \log n}\binom{n - c\log n}{n/2 - g\log n} \Big/ \binom{n}{n/2}. \tag{7}$$

By Stirling's approximation, for large enough n,

$$\binom{c \log n}{g \log n}$$

$$\leqslant \frac{2(c\log n)^{1/2}(c\log n)^{c\log n}}{(2\pi)^{1/2}(g\log n)^{1/2}((c-g)\log n)^{1/2}(g\log n)^{g\log n}[(c-g)\log n]^{(c-g)\log n}}$$

$$\leqslant \frac{8\cdot 4^{1/4c\log n}(4/3)^{3/4c\log n}}{(2\pi)^{1/2}(3c)^{1/2}} \leqslant \frac{8\cdot 4^{c\log n}}{(6c\pi)^{1/2}3^{3/4c\log n}} \tag{8}$$

and

$$\binom{n}{n/2} \geqslant \frac{n^{1/2}n^n}{2(2\pi)^{1/2}n/2(n/2)^n} \geqslant \frac{2^n}{(2\pi n)^{1/2}}. \tag{9}$$

Also

$$\binom{n - c\log n}{n/2 - g\log n}$$

$$\leqslant \frac{2(n-c\log n)^{1/2}(n-c\log n)^{n-c\log n}}{\begin{pmatrix}(2\pi)^{1/2}(n/2 - c/4\log n)^{1/2}(n/2 - 3/4c\log n)^{1/2} \\ \times (n/2 - c/4\log n)^{n/2 - c/4\log n}(n/2 - 3/4c\log n)^{n/2 - 3/4c\log n}\end{pmatrix}}$$

$$\leqslant \frac{4}{(2\pi)^{1/2}(n-2c\log n)^{1/2}}\left(\frac{n - c\log n}{n/2 - c/4\log n}\right)^{n/2 - c/4\log n}$$

$$\times \left(\frac{n - c\log n}{n/2 - 3/4c\log n}\right)^{n/2 - 3/4c\log n}$$

$$\leqslant \frac{8\cdot 2^{n - c\log n}}{(2\pi n)^{1/2}}\left(1 - \frac{c/4\log n}{n/2 - c/4\log n}\right)^{n/2 - c/4\log n}$$

$$\times \left(1 + \frac{c/4\log n}{n/2 - 3/4c\log n}\right)^{n/2 - 3/4c\log n}$$

$$\leqslant \frac{8\cdot 2^{n - c\log n}2\cdot e^{-1/4c\log n}e^{1/2c\log n}}{(2\pi n)^{1/2}} \qquad \text{for large enough } n$$

$$\leqslant \frac{16\cdot 2^{n - c\log n}}{(2\pi n)^{1/2}}. \tag{10}$$

By (7), (8), (9), and (10)

$$\begin{aligned} F &\leqslant \frac{2cfgn(\log n)^2 8 \cdot 4^{c \log n} 16 \cdot 2^{n - c \log n}(2\pi n)^{1/2}}{(6\pi c)^{1/2} 3^{3/4c \log n}(2\pi n)^{1/2} 2^n} \\ &\leqslant \frac{64fc^2b(\log n)^2 2^{c \log n}}{(6\pi c)^{1/2} 3^{3/4c \log n}} \\ &\leqslant 1/2n^2 \left(\frac{16}{27}\right)^{c/4 \log n} \qquad \text{for large enough } n, \\ &\leqslant 1/2 \frac{n^2}{n^{c/4 \log(27/16)}}. \end{aligned}$$

Since $c = 8/\log(\frac{27}{16})$, we have $F \leqslant \frac{1}{2}$, and $g > 1$. Take n_0 to be the smallest n satisfying all the conditions "for large enough n." The claim now follows.

RECEIVED: June 28, 1984; ACCEPTED: August 19, 1985

REFERENCES

BENTLEY, J. L. (1975), Multidimensional binary search trees used for associative searching, *Comm. ACM* **18**, 509–517.

BENTLEY, J. L., AND MAURER, H. A. (1979), A note on Euclidean near neighbor searching in the plane, *Inform. Process. Lett.* **8**, No. 3, 133–136.

BLUM, M., FLOYD, R. W., PRATT, V. R., RIVEST, R. L., AND TARJAN, R. E. (1973), Time bounds for selection, *J. Comput. System Sci.* **7**, No. 4, 448–461.

CHAZELLE, B. (1983), Filtering search: A new approach to query-answering, *in* "Proc. 24th Annu. Sympos. Found. of Comput. Sci., Los Angeles," pp. 122–132.

KIRKPATRICK, D. G. (1983), Optimal search in planar subdivisions, *SIAM J. Comput.* **12**, No. 1.

LEE, D. T. (1982), On *k*-nearest neighbor Voronoi diagrams in the plane, *IEEE Trans. Comput.* **C-31**, No. 6, 478–487.

LIPTON, R. J., AND TARJAN, R. E. (1980), Applications of a planar separator theorem, *SIAM J. Comput.* **9**, No. 3, 615–627.

PREPARATA, F. P., AND SHAMOS, M. I. (1985), "Computational Geometry: An Introduction," Springer-Verlag, New York.

WILLARD, D. E. (1978), "Predicate-oriented database search algorithms," Harvard Univ., Aiken Comput. Lab., Ph.D. thesis, Report TR-20-78.

YAO, F. F. (1983), A 3-space partition and its applications, *in* "Proc. 15th Ann. SIGACT Sympos.," pp. 258–263.