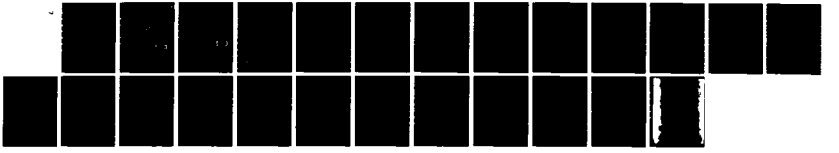
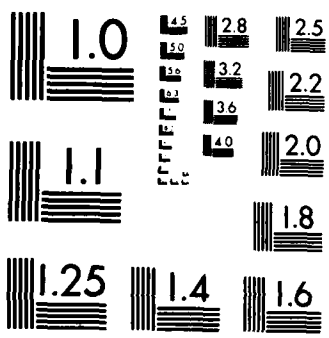
AD-A142 404 NEW ALGORITHMS FOR NEAR NEIGHBOR SEARCHING(U) ILLINOIS 1/1
UNIV AT URBANA APPLIED COMPUTATION THEORY GROUP
B CHAZELLE ET AL. SEP 83 ACT-40 N00014-79-C-0424

UNCLASSIFIED F/G 12/1 NL





(12)

ACT-40 SEPTEMBER 1983

NEW ALGORITHMS FOR NEAR NEIGHBOR SEARCHING

BERNARD CHAZELLE
FRANCO P. PREPARATA

REPORT R-997 UILU-ENG 83-2218

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A142404	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) NEW ALGORITHMS FOR NEAR NEIGHBOR SEARCHING	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER R-997 (ACT-40); UILU-ENG 83-2218
7. AUTHOR(s) Bernard Chazelle and Franco P. Preparata	8. CONTRACT OR GRANT NUMBER(s) N00014-79-C-0424
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois 1101 W. Springfield, Urbana, IL 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program	12. REPORT DATE September, 1983
	13. NUMBER OF PAGES 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Geometric searching, near-neighbo. search, Voronoi diagrams, bounded radius search, filtering search, efficient algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper proposes a new technique for solving near neighbor problems in the plane. We illustrate our method on the following two problems:

1. k-Nearest Neighbor: Given a set S of n points in the plane and a query of the form (q,k), with q a query point and k a positive integer, report the k points of S closest to q.
2. Circular Range Search: Given a set S of n points in the plane and a query of the form (q,d), with q a query point and d a positive real number, report all the points of S that lie inside the circle of radius d, centered at q.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Our main results include $O(n^{1+\epsilon})$ space, $O(k + \log n)$ query time algorithms for solving each of these problems; k denotes the size of the output. We also show that it is possible to solve either problem in $O(k \log^2 n)$ query time, using only $O(n \log n)$ space. These results constitute significant improvements over previous methods, in particular regarding the circular range search problem, which had previously defied efficient solutions.

NEW ALGORITHMS FOR NEAR NEIGHBOR SEARCHING

by

Bernard Chazelle[†]
Dept. of Computer Science
Brown University
Providence, RI

Franco P. Preparata[‡]
Coordinated Science Laboratory
University of Illinois
Urbana, IL

Abstract:

This paper proposes a new technique for solving near neighbor problems in the plane. We illustrate our method on the following two problems:

1. **k-Nearest Neighbor:** Given a set S of n points in the plane and a query of the form (q, k), with q a query point and k a positive integer, report the k points of S closest to q.
2. **Circular Range Search:** Given a set S of n points in the plane and a query of the form (q, d), with q a query point and d a positive real number, report all the points of S that lie inside the circle of radius d, centered at q.

Our main results include $O(n^{1+\epsilon})$ space, $O(k + \log n)$ query time algorithms for solving each of these problems; k denotes the size of the output. We also show that it is possible to solve either problem in $O(k \log^2 n)$ query time, using only $O(n \log n)$ space. These results constitute significant improvements over previous methods, in particular regarding the circular range search problem, which had previously defied efficient solutions.

[†] This research was supported in part by NSF grant MCS 8303925.
[‡] This research was supported in part Joint Services Electronics Program under Contract N00014-79-C-0424.
This document is being simultaneously issued as a Brown Univ. tech. rep. and a report of the Coordinated Science Laboratory, Univ. Illinois.

1. Introduction

The abundant literature on *near-neighbor* problems [1–5,8,11,12] witnesses the central location that the notion of *proximity* occupies in computational geometry. Among the most powerful tools available today for dealing with proximity relationships, the well-known *Voronoi diagram* stands out as one of the most versatile as well. Even its failure to solve farthest point or k-closest point problems can be easily remedied by introducing the notion of *higher order* Voronoi diagram [11]. With this construction in hand, it is possible to solve a number of near-neighbor problems, two of which will be of special interest to us in this paper:

1. **k-Nearest Neighbor:** Given a set S of n points in the plane and a query of the form (q, k), with q a query point and k a positive integer, report the k points of S closest to q.
2. **Circular Range Search:** Given a set S of n points in the plane and a query of the form (q, d), with q a query test point and d a positive real number, report all the points of S that lie inside the circle of radius d, centered at q.

Every algorithm for a search problem to be solved in a repetitive mode over a fixed database of n items is characterized by three complexity measures —$M(n)$, $Q(n)$, and $P(n)$— which are, respectively, the storage, the response time, and the preprocessing time. For obvious reasons, the time spent organizing the data structure (preprocessing time) is not as important a cost measure as $M(n)$, so we shall temporarily ignore $P(n)$ until the last section of this paper.

Previous work on the *k-nearest neighbor* problem includes a number of algorithms based on the Voronoi diagram, the most efficient of which have the following characteristics [4,6]: $M(n) = O(n^3)$ and $Q(n) = O(k + \log n)$. Other (less efficient) methods were found earlier [5,8,11]. The *circular range search* problem is also amenable to efficient treatment by means of Voronoi diagrams. A straighforward extension of an algorithm given in [1] led to the best method known until now [4]; the method allows us to report the k points within the query circle in time $Q(n) = O(k + \log n \log\log n)$ and requires $M(n) = O(n^3)$ storage. Although we are mainly concerned in this work with algorithms that achieve optimal (or near-optimal) query times, we should still mention the existence of a space-

optimal algorithm, with the following characteristics [12]: $M(n) = O(n)$ and $Q(n) = O(k + n^{0.98})$. For other methods, consult [1–3].

The purpose of this paper is to describe a novel representation scheme for higher Voronoi diagrams that allows us to improve on the best algorithms previously known for the problems mentioned above. The following table summarises our main results; k designates the output size.

problem	space	query time
k-nearest neighbor (fixed k)	$O(k(n-k))$	$O(k + \log n)$
k-nearest neighbor	$O(n^{1+\epsilon})$	$O(k + \log n)$
k-nearest neighbor	$O(n \log n)$	$O(k \log^2 n)$
circular range search	$O(n^{1+\epsilon})$	$O(k + \log n)$
circular range search	$O(n \log n)$	$O(k \log^2 n)$

2. Some Background

Let's briefly review the main steps of the algorithms proposed in [4] for solving the *k-nearest neighbor* and the *circular range search* problems. In the following, ALG_{NN} (resp. ALG_{CR}) will denote the algorithm of [4] for the former (resp. latter) problem. Let $\text{Vor}_k(S)$ denote the order-k Voronoi diagram of a set S of n points in the plane. Recall that $\text{Vor}_k(S)$ is a subdivision of the plane into convex regions, all of whose points have the same k nearest neighbors. More precisely,

$$\text{Vor}_k(S) = \bigcup_{T \subseteq S;\, |T| = k} V^*(T),$$

where $V^*(T)$ is the locus of points that are closer to any point in T than any point in $S - T$. The complexity of higher-order Voronoi diagrams has been thoroughly analyzed by Lee [8], who showed that the size of $\text{Vor}_k(S)$ (e.g. the number of edges) is always $O(k(n-k))$, a fact which we express by the relation

$$|\text{Vor}_k(S)| = O(k(n-k)). \tag{1}$$

Both ALG_{NN} and ALG_{CR} involve computing the set of diagrams $\{\mathrm{Vor}_{2^i}(S) \mid 0 \leq i \leq \lfloor \log_2 n \rfloor\}$. In order to efficiently retrieve the neighbors associated with each Voronoi polygon, we augment the graph representation of each diagram $\mathrm{Vor}_k(S)$ with its *neighbor-lists*, i.e., the set of k neighbors corresponding to each face. From Lee's findings, it then follows that $O(n^3)$ is an upper bound on the storage required by both ALG_{NN} and ALG_{CR}.

Both algorithms were presented in [4] in order to illustrate the concept of *filtering search*. This notion prescribes to trade traditional 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 the case of, say, ALG_{NN}, this idea materializes as follows: first, compute the integer j such that $2^{j-1} < k \leq 2^j$; then determine the face f of $\mathrm{Vor}_{2^j}(S)$ that contains the point q; finally, retrieve from the list of neighbors associated with f, the k points closest to q. Using an optimal planar point location algorithm [7,10] to locate q and a linear-selection algorithm to retrieve the k neighbors, ALG_{NN} can be easily shown to have the following performance: $M(n) = O(n^3)$ and $Q(n) = O(k + \log n)$.

ALG_{CR} proceeds along similar lines. The main idea was originally proposed by Bentley and Maurer in [1]. It involves locating q in $\mathrm{Vor}_{2^i}(S)$ for $i = 0, 1, 2, \ldots$, and examining the corresponding neighbor-list, proceeding in this manner until we first encounter a point of S 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. A simple analysis shows that ALG_{CR}'s performance is given by $M(n) = O(n^3)$ and $Q(n) = O(k + \log n \log\log n)$.

We will show how to apply the notion of *filtering search* to make the representation of neighbor-lists more economical, and thus improve on the above results. The basic idea is to partition $\mathrm{Vor}_k(S)$ into sets of adjacent faces to which are attached *augmented* neighbor-lists. This allows us to save a factor of n in the space requirements of ALG_{NN} and ALG_{CR}. Further application of *filtering search* will lead to still greater improvements.

3. A New Representation of Higher Order Voronoi Diagrams

The purpose of this section is to show how neighbor-lists can be stored implicitly without increasing the asymptotic complexity of $\mathrm{Vor}_k(S)$, i.e., using only $O(k(n-k))$ storage. We define $\mathrm{Del}_k(S)$ as the 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 in $\mathrm{Del}_k(S)$ correspond to adjacent faces in $\mathrm{Vor}_k(S)$ (faces are adjacent if they share an edge and not just a vertex). Note 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 triangulation. At any rate, the dual graph is always 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 $w_{i-1}, v_{i+1}, w_{i+1}$.

It is easy to see that this transformation of T_k can at most double the original number of vertices: indeed, let P denote the number of vertices of $\mathrm{Del}_k(S)$ and let ν_i be the number of vertices of degree i in T_k. If δ is the maximum degree in T_k, we have $\sum_{1 \le i \le \delta} \nu_i = P$ and $\sum_{1 \le i \le \delta} i\nu_i = 2(P-1)$ (since T_k is a tree). Let $|T^*|$ denote the number of vertices of T^*. Since each vertex of T_k of degree $i \ge 3$ gives way to $i-2$ vertices in T^*, the size of T^* is given by

$$|T^*| = \nu_1 + \nu_2 + \sum_{3 \le i \le \delta} \nu_i(i-2) = \sum_{1 \le i \le \delta} i\nu_i - \nu_2 - 2 \sum_{3 \le i \le \delta} \nu_i < 2P - 2. \qquad (2)$$

The following is common knowledge; thus given without proof.

Lemma: 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 at most $2m/3$ vertices each.

The decomposition process embodied by the lemma can be applied recursively on the tree T^*, until we achieve a decomposition of the original tree into connected subtrees $T_1^*, \ldots, T_q^*$, whose numbers of vertices are all comprised between k and $3k$. This is always possible since in each

splitting step each component has at least one third as many vertices as its parent. Note that a given face of $\mathrm{Del}_k(S)$ may be represented as a vertex in several of the subtrees $T_1^*, \ldots, T_q^*$. We will, however, allow only one instance to be accessible in the search process.

We now argue that neighbor-lists within any subtree T_i^* cannot differ by a large amount. Indeed, 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. This allows us to set up an implicit representation of neighbor-lists within each subtree T_i^*. The simplest solution would consist of merging all the neighbor-lists into a superset $S_i = \bigcup_{u \in T_i^*} \sigma(u)$ for $i = 1, \ldots, q$. 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, $|T_i^*| \geq k$, we have $q \leq |T^*|/k$, from which we readily derive $\sum_{1 \leq i \leq q} |S_i| \leq q \max |S_i| < \frac{|T^*|}{k} \cdot 4k = 4|T^*|$; hence from (2)

$$\sum_{1 \leq i \leq q} |S_i| < 8P. \tag{3}$$

We complete the preprocessing of the planar graph $\mathrm{Vor}_k(S)$ by organizing it for efficient planar point location [7,10], and associating with each face f the index of the subtree T_i^* containing the dual vertex of f. As mentioned earlier, this index may not be unique; if there are several candidates we simply choose any one of them. We are now in a position to determine the k nearest neighbors of a query point q by locating the face of $\mathrm{Vor}_k(S)$ that contains q. Since the corresponding set S_i contains at most $4k$ elements, we can apply a linear-selection algorithm to retrieve the k nearest neighbors of q in $O(k)$ time.

It is possible to circumvent the difficulties inherent to linear-selection methods by slightly refining the representation of S_i. Choose any vertex v^* in T_i^* ($i = 1, \ldots, q$) and call it the root. The set $\sigma(v^*)$ is represented in full by means of a linked list. Since $\sigma(u)$ and $\sigma(v)$ differ in at most one element when u and v are adjacent vertices in T_i^*, we can compute the lists $\sigma(v)$ incrementally. Indeed, if $\sigma(u)$ is available, replacement of one item in a known position of $\sigma(u)$ enables us to compute $\sigma(v)$; this is done by simply specifying the position in $\sigma(u)$ and the item to be inserted as replacement.

Thus, assuming that we have a linked list representation of T_i^*, suppose that we wish to compute $\sigma(v)$, where v is an arbitrary vertex of T_i^*. We identify the unique path in T_i^* from v to v^*, and traverse it backward, at each step updating the current point list. This list will be precisely $\sigma(v)$ at the termination of the traversal. Once again the report time will be $O(k)$, since $|T_i^*| \leq 3k$. This scheme necessitates the storing of $\sigma(v^*)$ and a fixed amount c of data per edge in T_i^*. Thus the added storage C_i associated with T_i^* is

$$C_i = |\sigma(v^*)| + c(|T_i^*| - 1) < k + c|T_i^*|.$$

Therefore the total storage can be bounded from above as follows

$$\sum_{1 \leq i \leq q} C_i < qk + c \sum_{1 \leq i \leq q} |T_i^*| \leq \frac{|T^*|}{k} \cdot k + c|T^*| = (c+1)|T^*| < 2(c+1)P,$$

where use has been made of $q \leq |T^*|/k$ and of Relation (2).

Whatever the strategy chosen, Relation 3 and the inequality above show that the total amount of storage needed is $O(P)$. Since, as in any planar graph, the number of faces in $\mathrm{Vor}_k(S)$, P, is dominated by the number of edges, Relation 1 shows that the overall storage used by the algorithm is $O(k(n-k))$.

Theorem 1. When the value of k is fixed (i.e., k is not part of the query), it is possible to solve the *k-nearest neighbor* problem in $O(k + \log n)$ time, using $O(k(n-k))$ space.

Throughout this paper we will use the notation $\mathrm{Vor}_k^*(S)$ to designate the data structure used in Theorem 1, i.e., the order-k Voronoi diagram of S, augmented with the implicit representation of neighbor-lists and preprocessed for efficient planar point location.

4. The k-Nearest Neighbor Problem

Algorithm ALG_{NN} can be used exactly as described in Section 2; the only difference coming from the new representation of higher-order Voronoi diagrams now used. Theorem 1 shows that the

overall storage needed is

$$O\Big(\sum_{0\le i\le\lfloor\log_2 n\rfloor} 2^i(n-2^i)\Big) = O(n^2).$$

The query time is clearly $O(2^j + \log n)$, with $2^{j-1} < k \le 2^j$, hence $O(k + \log n)$. This shows that it is possible to solve the *k-nearest neighbor* problem in $O(k + \log n)$ time, using $O(n^2)$ space. As will be established in the following, this improvement can be taken much further with a more subtle use of *filtering search*.

4.1. Outline of the approach

We begin with an informal description of the basic idea of the approach. In the data structure $\mathrm{Vor}_j^*(S)$ we call the parameter j the *scope* of the search. In the previously outlined ALG_{NN} the storage cost is mainly due to the necessity of having a Voronoi diagram with the adequate search scope over the *entire* set S for any query: this is because we stipulated to carry out the filtering search as a single stage process. If instead we explore the idea of a multistage search, at each stage we could use the information so far acquired to devise the best strategy for the next stage. Specifically, suppose that in the course of the process we have partitioned the original set S into several nontrivial subsets, and —as a policy— we explore the "most promising" subset with an increasing search scope. Such a scheme would have the property that, while the search scope increases, the size of the searched set decreases, which bears the promise of reduced storage requirement.

More formally, the main (*primary*) search structure is appropriately described as a *rooted tree* T. A search, prompted by the query (q, k), is to be viewed as the visit of a subtree T_q of T, where the term "subtree" refers here to any connected subgraph of T that contains the root. Associated with each node v of T there is a set $S(v) \subseteq S$, and a (*secondary*) search data structure $\mathrm{Vor}^*_{k(v)}(S(v))$, where $k(v)$ is the search scope at node v. We also define: $\Gamma(v)$ is the *set of offspring* nodes of v; $h(v)$, the *depth* of v, is the number of ancestors of v in T; *level* j of T is the set of all nodes v with $h(v) = j$.

If $\Gamma(v) = \{w_1, \ldots, w_c\}$ $(c \ge 2)$, then we stipulate that $\{S(w_1), \ldots, S(w_c)\}$ is a non-trivial partition of $S(v)$. Thus is we set $S(\mathrm{root}) = S$, it is immediate to recognize that the set of nodes on

a given level of T define a partition of S, and that the level-$(j+1)$ partition is a proper refinement of the level-j partition ($j \geq 0$).

We now restrict the structure of T by imposing the following regularity constraints:

1. All internal nodes of T have the same degree $c \geq 1$;
2. The subsets of $S(v)$, assigned to the offsprings of v, are (nearly) equally sized (this is trivial when c is equal to 1 and is readily obtained by requiring $n = c^m$ otherwise);
3. If w is the parent of v, then $k(v) = f \times k(v)$, where $f > 1$.

Note that we may choose $k(\text{root}(T)) = \lfloor \log_2 n \rfloor$, since this will equalize search time and report time for root(T). Moreover, $\text{Vor}^*_{k(v)}(S(v))$ is replaced by $S(v)$ itself if $|S(v)| \leq k(v)$; thus, if v is a leaf of T and w is its parent, we wish to have $|S(v)| \leq k(v)$ and $|S(w)| > k(w)$. This characterizes the leaves, and the depth h of T is the smallest integer satisfying the following inequality:

$$|S(v)| = n/c^h \leq \lfloor \log_2 n \rfloor \cdot f^h = k(v)$$

or equivalently $(fc)^h \geq n/\lfloor \log_2 n \rfloor$. We can thus express h as follows:

$$h = \left\lceil \frac{\log n - \log\lfloor \log_2 n \rfloor}{\log(fc)} \right\rceil. \tag{4}$$

The storage requirement of T is readily evaluated. The data structure $\text{Vor}^*_{k(v)}(S(v))$ is stored in $O(k(v) \cdot |S(v)|)$ space (see Section 3); since $k(v)$ is constant for all nodes v at the same depth, the storage requirement for all nodes at level $h(v)$ of T is $O(k(v)\sum|S(v)|) = O(nk(v))$. Recalling that $k(v) = \lfloor \log_2 n \rfloor \cdot f^{h(v)}$, we derive that the total storage requirement $M(n)$ of T is

$$M(n) = O(n \log n \sum_{1 \leq i \leq h} f^i) = O(nf^h \log n). \tag{5}$$

Substituting (4) into (5), we obtain

$$M(n) = O\left(n^{1+\frac{\log f}{\log fc}} \cdot (\log_2 n)^{1-\frac{\log f}{\log fc}}\right). \tag{6}$$

Therefore, given any $\epsilon > 0$, if we choose f and c so that $\frac{\log f}{\log(fc)} = \epsilon$, we can achieve storage $M(n) = O(n^{1+\epsilon})$.

4.2. Answering a Query

The subtree T_q of T which describes the search prompted by a given query (q, k) is *grown* one node at a time, starting at the root r. The primitive operation used by the search process is the *visit of a node* v, and consists of the following actions:

Step 1: Locate q in $\mathrm{Vor}^{*}_{k(v)}(S(v))$ and report the set $N_q(v)$ of the $k(v)$ closest neighbors to q in $S(v)$ ($N_q(v)$ is the set retrieved at v.)

Step 2: Determine the member of $N_q(v)$, called $far(v)$, which is farthest from q.

If T^* denotes the subtree of T visited so far by the search process, then

$$\bigcup_{v \in \text{leaves of } T^*} N_q(v)$$

denotes the *total retrieved set*.

The process makes use of a priority queue R that contains a subset of the nodes of T. The ordering in R is based on the value of the distance of *far* from q, and the *top* of R (the node to be extracted) is the node in R for which this distance is minimum. Queue R is managed as follows: initially, the root r of T is visited and inserted into the queue. At the generic step, the node v at the top of R is extracted and, if it is not a leaf of T, all of its offsprings are visited and inserted into the queue. By this process, it is immediately recognized that the nodes in R are leaves of T^*, all nodes of which have been visited by the search. Note, however, that not all leaves of T^* are necessarily in R; indeed, a leaf of T^* that is also a leaf of T, upon leaving R, does not generate any new leaves to be re-inserted. Each visited node is marked and the process terminates when the set of marked nodes is the node set of T_q (i.e., when $T^* = T_q$). We shall now develop the criterion for termination, which is based on the claim that when the total retrieved set is large enough it is guaranteed to contain the desired k nearest neighbors of q.

The C be the disc centered at q that passes through the k-th nearest neighbor of q. Without loss of generality, we may assume that only one point lies on the boundary of C, so as to ensure that the notion of "k nearest neighbors" is well defined. A node v is said to be *saturated* if $far(v)$ lies

inside C, and *unsaturated* otherwise. For each saturated node v, New(v) denotes the set of points that are discovered *for the first time* while visiting v: New(v) represents the contribution of v to the k nearest neighbors of q, and |New(v)| is, in bookkeping terms, the net revenue for a cost of $k(v)$ retrieved points. We claim that $|\text{New}(v)| \geq (1 - 1/f)k(v)$. Indeed, any point $p \in N_q(v)$ previously discovered must have been encountered for the first time while visiting an ancestor β of v and must also belong to $N_q(\text{father}(v))$. Thus the set of newly discovered points is

$$\text{New}(v) = N_q(v) - S(v) \bigcap N_q(\text{father}(v)),$$

and its cardinality is $|N_q(v)| - |S(v) \bigcap N_q(\text{father}(v))|$. Since $N_q(v) = k(v)$ and $|S(v) \bigcap N_q(\text{father}(v))| \leq |N_q(\text{father}(v))| = k(v)/f$, at least $k(v) - k(v)/f$ points are discovered for the first time at v, as claimed. Thus

$$|N_q(v)| \leq \frac{f}{f-1}|\text{New}(v)|. \tag{7}$$

We now observe that an unsaturated node u will not be extracted from the queue R as long as R contains at least one saturated node v; indeed, the distance from q to *far*(u) is dominated by the distance from q to *far*(v), so that the presence of v prevents u from appearing at the top of R. This implies that the visit of T_q is completed when the last saturated node v^* leaves R. Indeed, after v^* has been extracted, R contains only unsaturated nodes; for any such node u, $N_q(u)$ contains at least one point of S outside C, which implies that all the k nearest neighbors of q have already been retrieved (i.e., they belong to the set $\bigcup_{v \in \text{leaves}(T^*)} N_q(v)$), so T_q has been visited entirely.

Let us now return to consider an unsaturated node u, offspring of a saturated node w. After the extraction from R of the (saturated) w, the visits of its offspring (all inserted into R) have been done at a cost of $c|N_q(u)| = cf|N_q(w)|$ retrieved points. We stipulate to charge the cost of the visit of each unsaturated offspring u to its saturated parent w; in the worst case, all offsprings are unsaturated, so that w gets charged the additional cost $c \cdot |N_q(u)| = cf|N_q(w)| \leq \frac{cf^2}{f-1}|\text{New}(w)|$.

Consider now the event consisting of the extraction of the *last* saturated node from R. At this point, $\sum_{\text{saturated}_v} |\text{New}(v)| \leq k$, since all the k nearest neighbors of q belong to the retrieved set.

This event can be detected by controlling a cumulative sum

$$A \triangleq \sum_{\text{visited}} k(v).$$

Indeed, let V denote the set of nodes visited; this set is partitioned into V_1, the subset of saturated nodes, and V_2, the subset of unsaturated nodes.

$$A = \sum_{v \in V} k(v) = \sum_{v \in V} |N_q(v)| = \sum_{v \in V_1} |N_q(v)| + \sum_{u \in V_2} |N_q(u)|.$$

For a saturated v, we have $|N_q(v)| \leq \frac{f}{f-1}|\text{New}(v)|$ (Relation 7), so that;

$$\sum_{v \in V_1} |N_q(v)| \leq \frac{f}{f-1} \sum_{v \in V_1} |\text{New}(v)|.$$

On the other hand, the cost of retrieving $N_q(u)$ for each unsaturated node u is charged to their saturated parents, as discussed earlier. So, let now V_3 denote the subset of V_1 with unsaturated offspring. We have

$$\sum_{u \in V_2} |N_q(u)| \leq \frac{cf^2}{f-1} \sum_{v \in V_3} |\text{New}(v)| \leq \frac{cf^2}{f-1} \sum_{v \in V_1} |\text{New}(v)|,$$

and, in conclusion:

$$A \leq \frac{(1+cf)f}{f-1} \sum_{v \in V_1} |\text{New}(v)| \leq \frac{(1+cf)f}{f-1} k.$$

Since $\frac{k(1+cf)f}{f-1}$ is the maximum value that A can assume as the last saturated node departs from R, the condition

$$A > \frac{(1+cf)f}{f-1} k \triangleq c_1 k$$

can be used to detect the termination of the scooping phase of the filtering search.

We can now describe the search algorithm. To provide for efficient updating, the priority queue R will be a dynamic heap (e.g. 2-3 tree). This will allow us to retrieve the top of the queue, as well as perform insertions and deletions, all in logarithmic time. For convenience, we should keep in the queue pairs of the form $(v, far(v))$. We have:

Initial Step:

If $k \leq \log_2 n$, then visit the root r and halt; else let $A := 0$, $T^* := \emptyset$, and insert $(r, far(r))$ into R.

General Step: Let v be the node at the top of R and let $A = \sum_{v \in T^*} k(v)$.

1. Assume that $A \leq c_1 k$. If v is a leaf of T, delete v from R and iterate. Otherwise, delete v from R, and visit each child z of v, and insert $(z, far(z))$ into both R and T^*. Update the value of A and iterate. Updating A involves adding $ck(z)$ to its current value (where again z is the generic child of v).

2. If $A > c_1 k$, apply a linear-selection algorithm to the set $\bigcup_{w \in \text{leaves}(T^*)} N_q(w)$, and determine the k points closest to q. These points constitute the k nearest neighbors of q in S.

With the previous analysis at hand, evaluating the query time of the algorithm is quite straightforward. Processing node v requires time $O(k(v) + \log n)$, which is also $O(k(v))$, since $k(v) = \Omega(\log n)$. The query time will thus be $O(A)$, so we can conclude

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

Thus, we have derived the main result of this section.

Theorem 2. It is possible to solve the *k-nearest neighbor* problem in $O(k + \log n)$ time, using $O(n^{1+\epsilon})$ space; ϵ is an arbitrarily small real number > 0.

We close this section with two remarks. The first is that the constant time that multiplies k in the expression $Q(n) = O(k + \log n)$ is proportional to $c_1 = (1 + cf)f/(f - 1)$. Since $\epsilon = \log f / \log(fc)$, for fixed f, we easily derive that $c_1 = O(f^{1/\epsilon})$. Finally, we note that if we let $c = 1$, we obtain a scheme where the primary structure T is a chain (and, consequently, $S(v) = S$ for each $v \in T$). In this case the global storage —see (6)— becomes $O(n^2)$ and the method behaves like the one described in Section 3 (save for the replacement of bisection search with a sequential search).

4.3. A More Space-efficient Solution

It is possible to lower the space requirements of the previous algorithm at the price of some increase in the query time. We will present an $O(n \log n)$ data structure that allows a query to be answered in time $O(k \log^2 n)$. This method can be of great interest when the application specifically requires that the k neighbors be sorted by distance to q. As we will see, another asset of this scheme is its utter simplicity. 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 in ascending x-order 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 $S(v)$; this can be done in $O(n \log n)$ time by using the divide-and-conquer algorithm of [11]. Each Voronoi diagram will be preprocessed for efficient planar point location, using Kirkpatrick's algorithm [7]. Aside from its conceptual simplicity, this point location method has the advantage over [10] of requiring only linear preprocessing, provided that the edges of the graph are already ordered around each of their adjacent vertices. This is precisely the case with the Voronoi diagram construction of [11], therefore the entire preprocessing will take $O(n \log n)$ time.

We answer a query (q, k) by first computing the nearest neighbor of q in S, using the structure $\mathrm{Vor}_1(S(r))$, where r is as usual the root of the tree, and $S(r) = S$. Next we visit the two offsprings of r and proceed as in the method described in the preceding section; in the present case the priority queue R 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 $\mathrm{Vor}_1(S(z))$ 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 $O(k \log \frac{n}{k})$ —see [9]— and each visit has a cost of $O(\log n)$, we conclude

Theorem 3. It is possible to solve the *k-nearest neighbor* problem in $O(k \log^2 n)$ time, using $O(n \log n)$ space.

5. The Circular Range Search Problem

Before resorting to the fairly heavy machinery of Section 4 in order to produce a near-optimal algorithm for the circular range search problem, we wish to show how a simple application of Theorem 1 leads to a significant improvement over the algorithm ALG_{CR} of Section 2.

5.1. A Preliminary Algorithm

We will describe an algorithm with performance: $Q(n) = O(k + \log n)$ and $M(n) = O(n^2)$ (with k being the output size). Recall that the basic idea behind ALG_{CR} is to retrieve the neighbor-lists of the regions of $\mathrm{Vor}_j(S)$ containing the query point q, for $j = 2^i$; $i = 0, 1, \ldots$. This process will stop at the first encounter with a point further than d from q. At this stage, the current neighbor-list will be a superset of the desired set, with at most $2k$ points, therefore a simple scan through it will complete the work. Since a total of $O(\log k)$ neighbor-lists will thus be examined, the query time of the algorithm is $O(k + \log k \log n)$, which can be easily shown to be $O(k + \log n \log\log n)$. If we substitute the data structure of Theorem 1, $\mathrm{Vor}_j^*(S)$, for the combination $\{\mathrm{Vor}_j(S),$ neighbor-lists$\}$, we lower the storage requirements to

$$M(n) = O\Big(\sum_{0 \le i \le \lfloor \log_2 n \rfloor} 2^i n\Big) = O(n^2).$$

We can improve the query time by slightly reorganising the computation. Let $r = \lceil \log_2 \log_2 n \rceil$. The first step consists of retrieving the sought neighbor-list in $\mathrm{Vor}_{2^r}^*(S)$. If it contains any point further than d from q, we complete the computation by filtering out the extraneous items, at a total cost of $O(2^r + \log n) = O(\log n)$ operations. If on the other hand all the points in the list lie within a distance d of q, we must pursue the search with larger scope. We return to the previous mode of operation, retrieving the sought neighbor-lists in $\mathrm{Vor}_{2^{r+1}}^*(S), \mathrm{Vor}_{2^{r+2}}^*(S), \ldots$. Let

$\mathrm{Vor}^*_{2^E}(S)$ be the last Voronoi diagram investigated. A total of $E - r + 1$ neighbor-lists will have been examined, therefore the algorithm requires $O((E - r + 1)\log n + \sum_{r \le i \le E} 2^i)$ time, hence $Q(n) = O((E - r + 1)\log n + 2^E)$. Since $\log_2 n \le 2^r$, we have

$$(E - r + 1)\log_2 n + 2^E \le (E - r + 2^{E-r} + 1)2^r < (2^{E-r+1} + 1)2^r \le 2^{E+2},$$

therefore $Q(n) = O(2^{E+2})$. The examination of $\mathrm{Vor}^*_{2^{E-1}}(S)$ produces only points within d of q, therefore $k \ge 2^{E-1}$. This implies that $Q(n) = O(k)$, and completes the proof that in all cases $Q(n) = O(k + \log n)$.

5.2. A Near-Optimal Algorithm

The preprocessing is absolutely identical to the one described in Section 4.1. We construct the tree T with the data structure $\mathrm{Vor}^*_{k(v)}(S(v))$ attached to each of its nodes v. Answering a query can be now described recursively quite simply: starting at v = root, retrieve the $k(v)$ nearest neighbors of the query point q. If any of these neighbors is found not to lie within a distance d of q, the subset of neighbors that do lie within d of q can be reported immediately, and the entire subtree rooted at v need not be further examined. If on the other hand all the neighbors lie within d of q, we must pursue the exploration of T, and to do so we distinguish between two cases: if v is a leaf of T, we report all the neighbors just found, otherwise no reporting takes place; instead, we iterate on the same process with respect to each of the c children of v.

The algorithm is trivially correct, so we only need to investigate its running time $Q(n)$. As before, the set of nodes examined in T forms a subtree T_q of T. From Theorem 1, we derive that $Q(n) = O(\sum_{v \in T_q}(k(v) + \log(n)))$, and since $k(v) \ge \lfloor \log_2 n \rfloor$,

$$Q(n) = O(\sum_{v \in T_q} k(v)).$$

Let v be any internal node of T_q. It follows directly from the algorithm that the $k(v)$ neighbors examined at node v all lie within d of q. As usual, let $N_q(v)$ denote this set of points. We can repeat a previous argument to show that the $\frac{f-1}{f}k(v)$ points of $N_q(v)$ are discovered at node v for the first

time (note that these are the $\frac{f-1}{f}k(v)$ points furthest away from q in $N_q(v)$). Since the total time required to visit all the children of v is $O(ck(v))$, it can be accounted for by the newly discovered neighbors. This scheme involves charging the cost incurred by each node to its parent, except for the root of T_q that will also bear its own cost, i.e., $O(\log n)$ time. We thus derive the relation

$$Q(n) = O(ck + \log n). \tag{15}$$

Relations (6) and (15) allow us to conclude

Theorem 4. It is possible to solve the *circular range search* problem in $O(k + \log n)$ time, using $O(n^{1+\epsilon})$ space; k denotes the size of the output and ϵ an arbitrarily small real number > 0.

5.3. A More Space-efficient Algorithm

Applying the very same technique developed in Section 4.3, yet discarding the priority queue —for which we have no use here— we derive the following result.

Theorem 5. It is possible to solve the *circular range search* problem in $O(k \log^2 n)$ time, using $O(n \log n)$ space; k denotes the size of the output.

6. Some Preprocessing Time Considerations

What is the time required to organize the various data structures introduced in this paper? In the case of Theorem 1, we can use Lee's $O(k^2 n \log n)$ time algorithm to construct the order-k Voronoi diagram of S [8]. Since $\mathrm{Vor}_k(S)$ has $O(k(n-k))$ vertices, the decomposition of its dual into subtrees $T_1^*, \ldots, T_q^*$ can be carried out in $O(k(n-k)\log n)$ time (see Lemma in Section 3). The remaining preprocessing can be easily shown to require $O(k \times |\mathrm{Vor}_k(S)|)$ operations, therefore the overall computation requires $O(k^2 n \log n)$ time.

The algorithms of Theorems 2 and 4 involve the same type of preprocessing. Since several Voronoi diagrams are needed, we can use the ingenious representation of the set $\{\mathrm{Vor}_1(S), \ldots, \mathrm{Vor}_q(S)\}$ recently discovered by Edelsbrunner et al [6]. This scheme allows us to construct and represent any diagram $\mathrm{Vor}_k(S)$, along with all its neighbor-lists, in time $O(k^2(n-k))$, after $O(n^3)$ preprocessing. This shows that the time required to organize the data structure for the algorithm of Theorem 2 is $O(n^3)$ for each level, i.e., $O(n^3 \log n)$ for the entire structure. It is easy to see that the same result holds true for the algorithm of Theorem 4.

Finally, concerning Theorems 3 and 5, recall that the corresponding data structures have already been shown to require $O(n \log n)$ time for their construction.

7. Conclusions

The contribution of this work has been to propose an economical method for representing higher-order Voronoi diagrams and demonstrate its power by describing improved algorithms for a number of near-neighbor problems, in particular, for the circular range search problem, which had previously defied efficient solutions. Several open questions deserve investigation. Order-k Voronoi diagrams are powerful tools, yet prohibitively expensive for large values of k. We partly overcame this shortcoming by using *filtering search*. This rescinded the need for very high-order Voronoi diagrams, yet failed to reduce the space requirement to $O(n \times \mathrm{POLYLOG}(n))$. Whether this bound can be achieved, as is the case for the *range query* problem [4], is an interesting open question.

Another area worthwhile of study concerns the existence of efficient near-neighbor algorithms that use only linear space. As mentioned earlier, F. Yao provided a partial answer to this question by providing a linear space algorithm for the *circular range search* problem with sublinear query time [12]. Can a similar algorithm be devised for solving the *k-nearest neighbor* problem?

REFERENCES

[1] Bentley, J.L., Maurer, H.A. *A note on Euclidean near neighbor searching in the plane*, Inf. Proc. Lett., 8 (1979), pp. 133–136.

[2] Bentley, J.L., Stanat, D.F. and Williams Jr., E.H. *The complexity of finding fixed-radius near neighbours*, Inf. Proc. Lett., 6 (1977), pp. 209–212.

[3] Chazelle, B. *An improved algorithm for the fixed-radius neighbor problem*, IPL, 16(4), pp. 193–198, May 1983.

[4] Chazelle, B. *Filtering search: A new approach to query-answering*, Proc. of 24th FOCS Symp., 1983.

[5] Dehne, F. *An optimal algorithm to construct all Voronoi diagrams for k nearest neighbor searching in the Euclidean plane*, Proc. 20th Allerton Conf. on Comm., Contr., and Comp., 1982.

[6] Edelsbrunner, H., O'Rourke, J., Seidel, R. *Constructing arrangements of lines and hyperplanes with applications*, Proc. of 24th FOCS Symp., 1983.

[7] Kirkpatrick, D.G. *Optimal search in planar subdivisions*, SIAM J. on Comp., Vol. 12, No. 1, Feb. 1983.

[8] Lee, D.T. *On k-nearest neighbor Voronoi diagrams in the plane*, IEEE Trans. Comp., Vol C–31, No. 6, June 1982, pp. 478–487.

[9] Lipski, W.Jr., Preparata, F.P. *Finding the contour of a union of iso-oriented rectangles*, J. Algorithms, 1(3), pp. 235–246, 1980.

[10] Lipton, R.J., Tarjan, R.E. *Applications of a planar separator theorem*, SIAM J. on Comp., Vol. 9, No. 3, Aug. 1980.

[11] Shamos, M.I., Hoey, D.J. *Closest point problems*, 16th Annual FOCS Symp., pp. 63–65, 1976.

[12] Yao, F.F. *A 3-space partition and its applications*, Proc. 15th Annual SIGACT Symp., pp. 258–263, April 1983.