INFORMATION AND CONTROL 64, 77–99 (1985)

How to Search in History*

BERNARD CHAZELLE†

Department of Computer Science,
Brown University, Providence, Rhode Island 02912

This paper considers the problem of granting a dynamic data structure the capability of remembering the situation it held at previous times. We present a new scheme for recording a history of h updates over an ordered set S of n objects, which allows fast neighbor computation at any time in the history. The novelty of the method is to allow the set S to be only partially ordered with respect to queries and the time measure to be multi-dimensional. The generality of the method makes it useful for a number of problems in 3-dimensional geometry. For example, we are able to give fast algorithms for locating a point in a 3-dimensional complex, using linear space, or for finding which of n given points is closest to a query plane. Using a simpler, yet conceptually similar technique, we show that with $O(n^2)$ preprocessing, it is possible to determine in $O(\log^2 n)$ time which of n given points in E^3 is closest to an arbitrary query point. © 1985 Academic Press, Inc.

1. INTRODUCTION

Consider the problem of maintaining a dynamic data structure over time. Typical operations will involve inserting new objects, deleting old objects, and of course, querying the data structure about its current state. If the structure is a dictionary a query is to look up a given item; if it is a priority queue it is to retrive the minimum or maximum element from the current set. In some applications it is sometimes needed to keep track of the configurations the data structure held at previous times. This need might arise in databases, for example, when one wishes to retrieve old information, or in other words, *search in the past*. In other contexts, the notion of *time* is only indirectly relevant. In circuit design rule checking, for instance, sweep-line algorithms are often used to report all pairs of intersecting rectangles (McCreight, 1980). It is convenient (and colorful) to think of, say, the sweeping direction as a time axis. This is all the more relevant that a sweep-line algorithm will indeed induce a one-to-one correspondence

* This paper is a revised and expanded version of a paper presented at the International Conference on "Foundations of Computation Theory" held in Borgholm, Sweden, August 21–27, 1983.

† This research was supported in part by NSF Grant MCS 83-03925 and the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-83-K-0146 and ARPA Order 4786.

0019-9958/85 $3.00
Copyright © 1985 by Academic Press, Inc.
All rights of reproduction in any form reserved.

between sweeping time and position on the sweeping axis. The problem of *searching in the past* corresponds in this case to asking questions about the state of the data structures attached to the sweep line at any particular point in time. Study on *search in the past* was initiated by Dobkin and Munro (1980) and taken up exhaustively by Overmars in his doctoral thesis (Overmars, 1983).

Consider the following problem, typical of searching in the past. Let S be a universe of n objects $v_1,..., v_n$ subject over time to h deletions and insertions in arbitrary order. Assuming that there exists a total order among the objects, Dobkin and Munro have described a method for computing, in time $O(\log n \log h)$, the *rank* of any object at any given time, i.e., the number of objects that preceded it at that time. One remarkable feature of their method is to avoid recording the "state of the universe" at all times, which would take $O(nh)$ space. Instead, they use a clever tree structure to limit the storage requirement to $O(n + h \log n)$. The query time of their algorithm was subsequently reduced to $O(\log(h+n))$ by Overmars (1981). Overmars also observed that if we are only interested in testing membership, i.e., finding if object v_i was present in the data structure at time θ, we can further reduce the storage to $O(n+h)$ by simply recording the history of each object individually. To each v_i we associate a list of non-overlapping intervals corresponding to its life periods.

In this paper we look at the problem of finding which v_i immediately preceded a new object q at time θ. We will see that if either the time measure is 1-dimensional or if there exists a total order among the objects that the query can "use," there are simple solutions to the problem. Unfortunately, neither condition is true in the geometric applications given in this paper, therefore more general alternatives must be sought. We will present a data structure that requires $O(n+h)$ storage and allows the computation of any neighbor, in the sense defined above, in $O(\log n \log h)$ time. Aside from its improved performance, the novelty of the method is to allow the set S to be only partially ordered and the time measure to be multidimensional, if necessary. This generality allows us to use the method for solving a number of problems in 3-dimensional geometry. For example, we are able to give an $O(n)$ space, $O(\log^2 n)$ query time algorithm for locating a point in a 3-dimensional complex with n faces endowed with some ordering property. This can be applied to the complex formed by n hyperplanes, which leads to an $O(n^3)$ space, $O(\log^2 n)$ query time algorithm. This also allows us to determine which of n given points in E^3 is closest to a query plane with the same time and space complexity. Using a slightly different technique, we show that with only $O(n^2)$ preprocessing, we can determine, in $O(\log^2 n)$ time, which of n given points is closest to an arbitrary query point. This result considerably improves the best solution previously known (Yao, in press).

2. The Problem

Before describing our data structure, let us give a more formal presentation of the problem. Let $S = \{v_1, ..., v_n\}$ be a set of n objects, provided with a partial order R. W.l.o.g. we can assume that $v_1 \leqslant \cdots \leqslant v_n$ gives a total order that embeds the partial order R (i.e., the v_i are topologically sorted). Each object is given the possibility of being either *active* or *inactive*, depending on the value of a parameter θ. More formally, we introduce the sets $S(\theta) = \{v_i \mid 1 \leqslant i \leqslant n \text{ and } h_i(\theta) = 1\}$, where $h_i(\theta)$ is a *characteristic* function in $\{0, 1\}$. Note that S induces a total order on $S(\theta)$. The parameter θ takes on any value in a domain Θ, which in most applications will be $\mathfrak{R}^d$ ($\mathfrak{R}$ = set of real numbers). Note that only in the case $d = 1$ it is legitimate to refer to θ as a measure of "time." We will assume that the number of distinct sets $S(\theta)$ is always finite. The collection of all these sets is called a *history* and its cardinality is denoted h. Next, we define a *query* as a pair (q, θ), where q is an object (not necessarily in S) which "extends" the total order in $S(\theta)$, i.e., the outcome of "$q \leqslant v$?" for each object v in $S(\theta)$ does not contradict the total order on $S(\theta)$. This outcome is *meaningless*, however, if v does not belong to $S(\theta)$. We define the *neighbor* of q as the largest object v in $S(\theta)$ such that $v \leqslant q$. By convention, if no such object exists, the neighbor of q is denoted $-\infty$. The first problem of interest in this paper thus is:

Preprocess S so that the neighbor of any query can be computed very effectively.

Such a statement is, of course, too general to lead to practical solutions, so we refine it to deal with some interesting cases. First of all, the case $\Theta = \mathfrak{R}$.

3. The Canal-Tree

3.1. *The Basic Ideas*

When $\Theta = \mathfrak{R}$ (as in Dobkin and Munro, 1980; Overmars, 1981), it is natural to refer to θ as a time measure, since its values can be totally ordered. W.l.o.g. we assume in the following that between two consecutive time intervals, $S(\theta)$ can change in at most one place. This can always be ensured by duplicating time breaks if necessary. Before proceeding with a description of our data structure, let's briefly review the basic features of Dobkin and Munro's method. It essentially consists of looking at the interval spanned by each node of the complete binary tree over $\{1, ..., n\}$, and keeping a chronological list of their cardinality, i.e., the number of active objects they span at any given time. It is then easy to retrieve any of these

cardinalities in time $O(\log h)$, since each list has at most h elements. Furthermore, a simple search in the tree enables us to sum up all the relevant cardinalities and compute the rank of any object. Since there are at most $\lceil \log n \rceil$ such cardinalities, the query time is $O(\log n \log h)$, and since each insertion/deletion need be recorded in at most $\lceil \log n \rceil$ lists, no more than $O(n + h \log n)$ space is thus required. Using a *layered* structure (Willard, in press), Overmars (1981) was able to improve the query time to $O(\log(n + h))$.

Since searching for neighbors is our main concern, we can avoid computing ranks altogether and, doing so, save a factor of $\log n$ in space. Before proceeding with a description of our method, a few important remarks are in order. As observed by Overmars (1981), we can view the life periods of each object as vertical intervals in the Euclidean plane, whereby the ordered list $v_1, ..., v_n$ is spread along the X axis and the time corresponds to the Y axis. A simple solution consists of storing the v_is in a complete binary tree, in inorder, with a chronological list of life periods attached at each node. This is a viable solution when it is possible to decide whether to branch left or right on the basis of a comparison $q :: v_i$, even when v_i is not present at time θ. Unfortunately in all of our applications, there is only a partial order among S and query objects, therefore any comparison between a query object and an inactive element of S is meaningless. This rules our this solution as well as the following one, valid only when, as before, we do have a total order among S and the query space and furthermore θ are 1-dimensional. Since the problem essentially reduces to finding the vertical segment immediately to the left of a query point, we can introduce horizontal segments to subdivide the plane into regions with common "answers." This is the *adjacency-map* of Lipsky and Preparata (1981), and we refer the reader to this reference for details of the construction. With this structure in hand, it suffices to locate the query in the subdivision, which we can do using any optimal planar point-location algorithm. This leads to an $O(n + h)$ space, $O(\log(n + h))$ query time algorithm, which unfortunately does not fulfill our needs because of our insistence on (1) no assumption of total order between S and the query space, and (2) multi-dimensionality.

To circumvent these difficulties, we introduce a new data structure T, called a *canal-tree*. T is a complete binary tree with n leaves, the ith from the left corresponding to v_i. Since T is essentially a static tree, it should probably be stored in an array so as to avoid the use of pointers. Simplicity will dictate our choice in the matter, however, and we will avoid overburdening our exposition with issues of implementation optimization. Let us first describe a tentative data structure, which we will use as a stepping-stone for constructing the *canal-tree*. Ideally, we would like to keep in each internal node v of T a pointer to a list, $L(v)$, which gives a chronological

account of all the largest active objects in the interval spanned by the left subtree rooted at v. More precisely, let z be the left child of v and let $I(z)$ be the interval $[i, j]$ such that the leaves of the subtree rooted at node z are from left to right $\{v_i, ..., v_j\}$. Let $v_{i_1}, ..., v_{i_k}$ be the list, in chronological order, of the largest objects in $S(\theta)$ during the history, with indices in $I(z)$, and let θ_j be the time corresponding to v_{i_j}'s promotion. $L(v)$ is simply the list of pairs $\{(\theta_1, v_{i_1}), ..., (\theta_k, v_{i_k})\}$. We define the θ-entry of $L(v)$ as the pair (θ_j, v_{i_j}) such that $\theta_j \leqslant \theta < \theta_{j+1}$ (for consistency, we may assume that θ_1 is always 0). It is clear that by traversing T in inorder, each internal node v can be uniquely associated with an interval (v_i, v_{i+1}); we can therefore extend the concept of neighbor to v itself, and define $n(v, \theta)$ as the largest object v_l in $S(\theta)$, with $l \in I(z)$. Answering a query (q, θ) can now be easily described. Starting at the root of T, we find $n(\text{root}, \theta)$ in $O(\log h)$ time, by performing a simple a binary search in $L(\text{root})$. If $q = n(\text{root}, \theta)$, we are clearly finished; otherwise if $q > n(\text{root}, \theta)$, we keep $n(\text{root}, \theta)$ as a potential candidate and we iterate on the right child of the root, and if $q < n(\text{root } \theta)$, we blithely branch to the left. This type of binary search is fairly standard and we may omit the details.

The unfortunate feature of this scheme is to be wasteful in its use of space. Indeed, let $z_0, ..., z_m$ be the internal nodes of T on the leftmost path from the root, and let w_i be the right child of z_i (Fig. 1). Suppose now that, at times $\theta_1, ..., \theta_h$, none of the objects in $I(w_1), ..., I(w_m)$ is ever active but, instead, v_1 is alternatively active and not active. This will cause each of the lists $L(z_1), ..., L(z_m)$ to contain the h-element sequence $\{(\theta_1, v_1), (\theta_2, -\infty), (\theta_3, v_1), (\theta_4, -\infty, ...\}$, which will entail the use of $O(n + h \log n)$ storage.

The *canal-tree* is a simple modification of the tree described above. Here we avoid duplicates by recording events only once: let $z_1, ..., z_p$ be the list of internal nodes, on the path from v_i to the root, that appear *after* v_i in inorder. Informally, $z_1, ..., z_p$ are the nodes encountered after each rightward

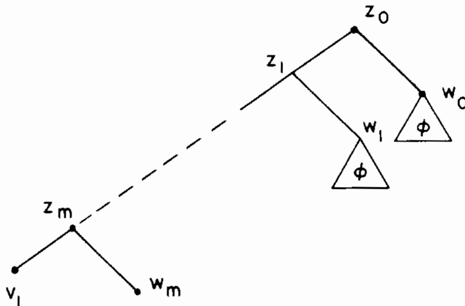

FIGURE 1

move on the way up from v_i to the root. Let w_i denote the right child of z_i. Suppose now that an insertion or a deletion of v_i takes place at time θ, and consider the largest index j such that $I(w_1),\ldots, I(w_{j-1})$ are all free of active objects at θ (Fig. 2). It is clear that whether v_i is inserted or deleted, the only nodes of T which witness a change in neighbor are precisely $z_1,\ldots, z_j$. The main feature of the canal-tree, however, will be to record the event in $L(z_j)$, only. It will result from this restriction that, since only one update is necessary per operation, the total space needed to store the lists $L(v)$ will be $O(n+h)$. We will use the remainder of this section to describe the update operations and show that the canal-tree still allows efficient searching.

Before proceeding, let us give an image to help visualize the workings of the canal-tree and also justify its name. Figure 3 illustrates the canal-like structure of T: let $S(\theta)=\{v_{i_1}, v_{i_2},\ldots, v_{i_k}\}$ be, as usual, the list of objects active at time θ, in left-to-right order. We introduce the *canal* of v_{i_j} as the path from v_{i_j} to a node z_j, defined iteratively as follows: z_k is a *pseudo-root*, situated right on top of the usual root. In the general case, z_{i_j} is the first point of contact with the canal of $v_{i_{j+1}}$. We will refer to v_{i_j} (resp. z_j) as the *source* (resp. *sink*) of the canal, which itself will be denoted $C(v_{i_j})$. Let v (resp. w) be the left (resp. right) child of the root. We observe that when the objects in $I(w)$ are not all inactive, the root is the sink of the canal whose source is the largest active element in $I(v)$. We can now state the basic requirement of the canal-tree: for all j; $1 \leqslant j \leqslant k$, the index i_j should appear in the θ-entry of $L(z_j)$, paired of course with the value of θ when v_{i_j} was last activated. In other words, $L(z_j)$ should contain a pair of the form (θ^*, v_{i_j}); $\theta^* \leqslant \theta$.

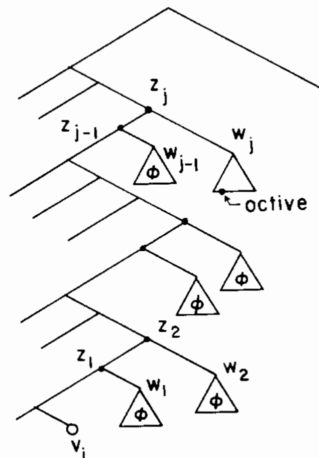

FIGURE 2

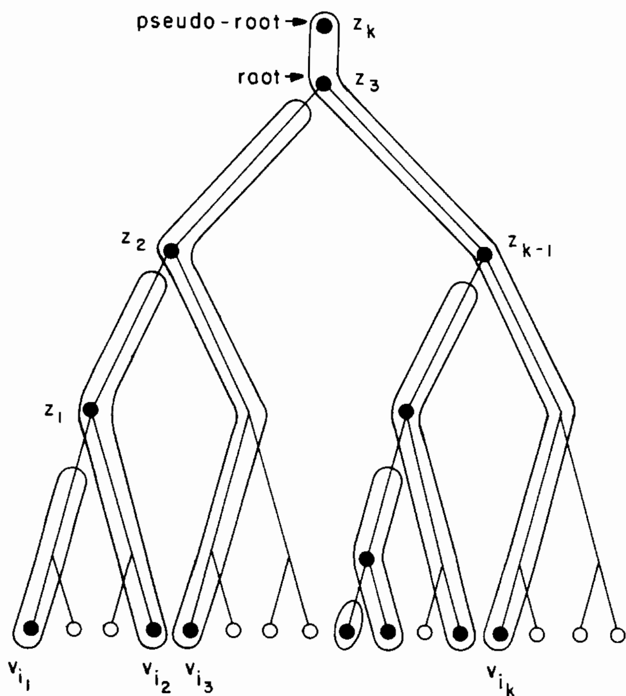

FIGURE 3

With this new, more economical scheme, it is clear that the lists $L(v)$, considered individually, may give us erroneous information about current neighbors. To allow for proper use of these lists, we must include in them the times θ at which the current information ceases to reflect reality. More precisely, each list $L(v)$ will be a sequence of the form

$$\{(\theta_1, v_{i_1}), \theta_1^{(\text{end})}, ..., (\theta_k, v_{i_k}), \theta_k^{(\text{end})}\},$$

whereby $\theta_j^{(\text{end})}$ signifies that v_{i_j} should not be relied upon as an indicator of $n(v, \theta)$ for $\theta; \theta_j^{(\text{end})} < \theta \leqslant \theta_{j+1}$. By "not to be relied upon," we mean that although the information may be occasionally correct, we should never use it, for it may not always be so. Note that we will often have $\theta_j^{(\text{end})} = \theta_{j+1}$, in which case we should simply omit $\theta_j^{(\text{end})}$ from the list, altogether. For convenience, we will refer to $\theta^{(\text{end})}$ as an *info-unavail* flag. To summarize, we state the fundamental property of the canal-tree:

FACT. *At any time $\theta \in \Theta$, the θ-entry of any interval node v of T is of the form (θ^*, v_i) (with $\theta^* \leqslant \theta$) if v is a currently a sink, or is an* info-unavail *flag, otherwise.*

Let [op, i, θ] be a shorthand for "apply operation op (activate or dactivate) to v_i at time θ." We can represent the history of S as a sequence of triplets [op, i, θ], ordered with respect to θ. Setting up T simply involves going through each instruction of the history in turn, updating the lists $L(v)$ accordingly. We will assume that, initially, all the lists $L(v)$ are empty.

3.2. *Setting Up the Canal-Tree*

We are now ready to give a description of the algorithms for activating and deactivating an object, respectively. We proceed in chronological order, one step at a time.

I. Activating an Object

Informally, activating v_i at time θ involves tracing down the path from the root to v_i and determining the last canal visited. A new canal is started at v_i flowing up towards the root, overtaking any canal to its left, but stopping as soon as it runs into a canal coming from the right. Let v_j be the source of this canal, and z be the last node visited on the canal. There are basically two cases to consider (Fig. 4):

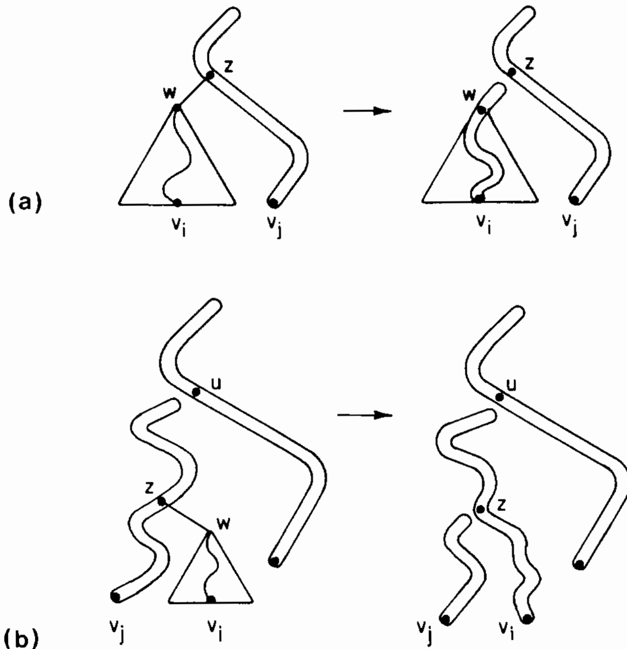

FIGURE 4

1. Suppose that $i<j$ (Fig. 4a). Let w denote the left child of z. Since z is the first canal node on the path from v_i to the root, none of the objects in $I(w)$ is active at time θ, therefore we have $n(z, \theta)=v_i$ and $n(v, \theta)\neq v_i$, for all the ancestors v of z. Furthermore, the global inactivity of $I(w)$ prior to θ shows that all the θ-entries in the subtree rooted at w are *info-unavail* flags, and should remain so. Therefore, the only updating required consists of appending (θ, v_i) to the end of $L(z)$.

2. Suppose that $i>j$ (Fig. 4b). Let w denote the right child of z, and u be the sink of $C(c_j)$. We must introduce a new canal $C(v_i)$ with u for sink, and we must move the sink of $C(v_j)$ down to z. This involves appending (θ, v_i) and (θ, v_j) to the lists $L(u)$ and $L(z)$, respectively. Since no sink is either destroyed or created, no *info-unavail* flag has to be added.

Note that the new canal from v_i can overtake at most one canal, since essentially the canal from v_i ceases to be new as soon as it runs into another canal. In both cases finding the relevant nodes u, w, z mentioned above is straightforward. To do so, we simply have to traverse the tree from the pseudo-root towards v_i, using a standard binary search, and stopping at the first node encountered that is not on a canal. To be able to do this as well as the updating described above, it suffices to keep track, at all times, of the most recent sink visited from which we left a canal. Since this involves only checking whether the current node v is a sink towards whose source we are heading, a simple look at the last item of $L(v)$ suffices, therefore activating v_i requires only $O(\log n)$ operations.

II. Deactivating an Object

Let z be the sink of $C(v_i)$ and let w be the left child of z (Fig. 5). If $I(w)$ is entirely inactive right after θ, the only action to take is clearly to append an *info-unavail* flag to the end of $L(z)$. Otherwise, the only active objects in $I(w)$ can only be of the form v_l for $l<i$ (see iterative definition of the canal-tree). Let j be the largest such index l. Note that the sink u of the canal $C(v_j)$ is the last sink encountered in a downward traversal of $C(v_i)$. Once we have found this node, we only have to append (θ, v_j) to the end of $L(z)$ as well as include an *info-unavail* flag at the end of $L(u)$. To justify the passing of this flag, one should try to imagine the consequences of not doing it if, at the next step, v_j were to be deactivated, i.e., multiple updates would then be necessary.

There again, finding all the appropriate nodes u, w, z can be done easily by walking down the tree, and keeping record of (1) the most recent sink visited from which we left a canal (e.g., z), (2) the most recent sink visited (e.g., u). Since both pieces of information can be updated in constant time at each node v visited, by simply looking at the last item in $L(v)$, the algorithm requires $O(\log n)$ operations. We can finally conclude:

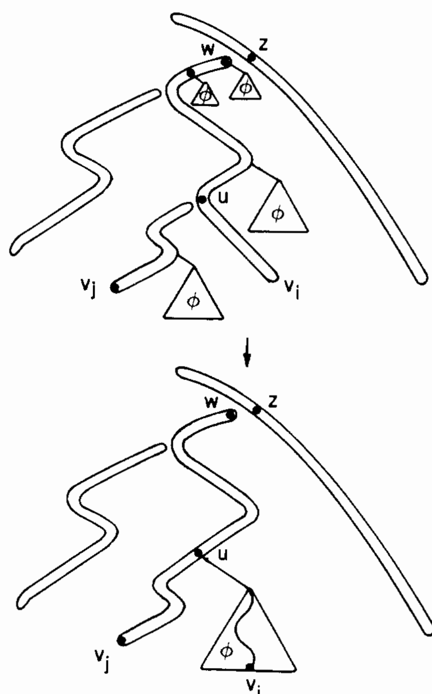

FIGURE 5

LEMMA 1. *The canal-tree can be constructed in* $O(n + h \log n)$ *time and* $O(n + h)$ *space.*

3.3. *Computing Neighbors*

We can now show how to use T for computing the neighbor of a query (q, θ). Starting at the pseudo-root, we retrieve the corresponding θ-entry and terminate if it gives us an *info-unavail* flag. This would, indeed, signify that all the objects were inactive at θ, so $-\infty$ should be our answer. If, instead, the θ-entry is a pair (θ^*, v_i), we start the iterative part of the search. Informally, we follow the current canal always trying to branch left to the first canal $C(v_j)$ with $q \leqslant v_j$ (Fig. 6). More precisely, we need to keep track of two variables: (1) *cur*, the index of the current canal traversed, (2) *last*, the index of the last sink visited to whose canal we did not branch. Initially, $\mathrm{cur} = i$ and $\mathrm{last} = -\infty$. At the generic step, let $C(v_k)$ be the current canal, F be the θ-entry of the current node v, and w be the next nodes after v on $C(v_k)$. If F is an *info-unavail* flag, we simply proceed to the next node towards v_k, and iterate. Assume now that F is of the form

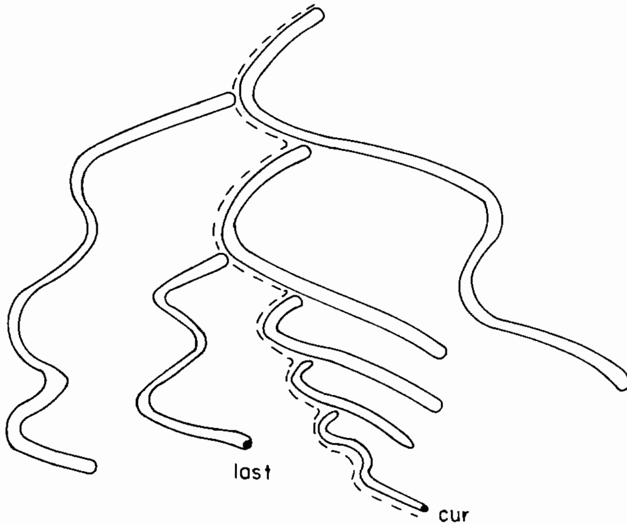

FIGURE 6

(θ^*, v_t), in which case w must be the right child of v. If $q = v_t$, we return v_t and stop. If $q < v_t$, we set cur $= t$ and branch left, otherwise we set last $= t$ and proceed to the right. When we eventually reach a leaf of T, we return the current value of *last*. Since computing each θ-entry takes $O(\log h)$ operations, the entire search requires $O(\log n \log h)$ time. We conclude:

THEOREM 1. *It is possible to record a history of h events involving n objects in $O(n + h)$ space, so that retrieving any neighbor information can be done in $O(\log n \log h)$ time.*

Recently Cole (1983) has shown that it is possible to improve the query time of this method to $O(\log n + \log h)$ method by a clever combination of hive-graphs (Chazelle, 1983) and layers (Willard, in press). This improvement has no incidence on the multi-dimensional use of canal-trees or the 3-dimensional point-location algorithm developed below, however.

3.4. *Generalizing to Multidimensional Parameters*

The key to the method described above was to be able to produce a snapshot of the canal-tree at any time very efficiently. Unfortunately, this is not always easy, in particular when the parameter θ is multi-dimensional. In that case, we may no longer be able to arrange the sinks in linear lists to reflect the chronological sequence of events, since even the very notion of chronology becomes ill-defined. We observe, however, that since the use of

643/64/1/3-7

canal-trees makes the whole problem totally local, i.e., comes down to allowing fast sink retrival at each node, we can still formulate a model in which efficient search in history is always possible by means of canal-trees. Let $\theta \in \Re^d$ $(d > 1)$. Let $T(\theta)$ be a snapshot of the canal-tree T at time θ. We define $J(v)$ to be the set of distinct values of the sinks, for all $\theta \in \Theta$. Assume that there exists a data structure $\mathrm{DS}(J(v))$ that organizes the elements of the set $J(v)$ in such a way that we can compute the sink-value of $T(\theta)$ at v, in time $O(Q(h))$. In this model, it is clearly possible to adapt the algorithm described earlier so as to compute the neighbor of any query (q, θ) in time $O(Q(h) \log n)$. We will show in the next section how this result can be used to derive new algorithms for several geometric problems.

4. Geometric Applications

We will apply the previous ideas to point-location problems: consider the task of locating a point in a planar subdivision. Although this problem has already been given several optimal solutions (Cole, 1983; Edelsbrunner, Guibas, and Stolfi, in press; Kirkpatrick, 1983; Lipton and Tarjan, 1977), we will for the sake of illustration show how to use canal-trees to produce a very simple near-optimal algorithm:

4.1. *Planar Point-Location*

Let $S = \{s_1, ..., s_n\}$ be the segments of a straight-line subdivision of the plane. We can obtain a total order on S by topologically sorting the relation $\preccurlyeq$, defined as follows: $s_i \preccurlyeq s_j$ whenever there exists a line parallel to the X axis that intersects s_i (resp. s_j) in p (resp. q), with $p \leqslant q$ with respect to the X coordinate. We precompute this partial order by sweeping a horizontal line L downwards, maintaining the current order of the intersections with L in a dynamic balanced search tree. The line L starts at the highest vertex of G and proceeds to visit each vertex of G in turn in descending Y order. If the current vertex is the upper end of a segment, this segment is inserted into the tree, otherwise it is deleted. This method is very standard (Bentley and Ottmann, 1979), so we may omit the details. The next step is to embed the partial order in a total order, which simply requires a topological sort. We are now ready to set up a canal tree T on the following basis: θ is the Y coordinate of the line L, and $S(\theta)$ is the set of segments that intersect L when positioned at $y = \theta$. Since an event in the history corresponds to the promotion or demotion of an edge, we have $h = 2n$. Finally, comparing a query point against an object simply involves computing the relative position of a point with respect to a line. Note that

this scheme will also handle subdivisions made of n curves monotone in the Y direction.

THEOREM 2. *It is possible to use a canal-tree to solve the planar-point location problem in $O(n)$ space and $O(\log^2 n)$ time.*

This method is akin to Lee and Preparata's (1977) algorithm; it shows that the latter is a particular instance of a general searching technique to which canal-trees are especially tailored. The basic difference with their algorithm is that we do away with the actual computation of "geometric chains." Instead, our method relies on the topological (rather than geometrical) nature of the problem, and thus, reduces the geometric part of the algorithm to its simplest expression. This has the effect of granting the algorithm great conceptual simplicity. Also, the basic generality of the method makes it directly applicable to other problems as well. The algorithm can be used, for example, to compute the horizontal neighbors of a query point, given a set of n pairwise disjoint segments. This involves reporting the first segment to the right and to the left of the query point that intersect a horizontal line passing through it.

4.2. *Spatial Point-Location*

Using a more general method, Dobkin and Lipton (1976) have shown how to solve the point-location problem in higher dimensions. Assume that the regions are defined by n arbitrary hyperplanes in d-dimensional Euclidean space, i.e., the regions form a d-dim complex. Dobkin and Lipton's method requires $O(n^{2^d - 1})$ space and $O(2^d \log n)$ time per query. The purpose of this section is to show how point-location problems in general can be viewed as history retrieval problems, for which canal-trees can be used. We will illustrate our point by presenting an improved algorithm for searching a 3-dimensional complex.

Let P be a 3-dimensional complex, regarded for our purposes as a partition of E^3 into polyhedra. We assume that either all the polyhedra are convex (think for example of a Voronoi diagram in E^3) or exactly one of them is non-convex. In the latter case what we have in mind is the convex partition of a convex polyhedron to which we adjoin the outside, unbounded polyhedron. We assume that no face in P can intersect a given line parallel to the X axis in more than one point. This can always be satisfied by slightly rotating the axes, if necessary. Let n denote the number of faces in P. It is easy to show that, up to within a constant factor, n gives the size of any standard representation of P. Let $\{P_1,..., P_p\}$ be the set of polyhedra in P and let V_i, E_i, F_i denote respectively the number of vertices, edges, and faces of P_i $(1 \leqslant i \leqslant p)$. Since any standard representation (e.g.,

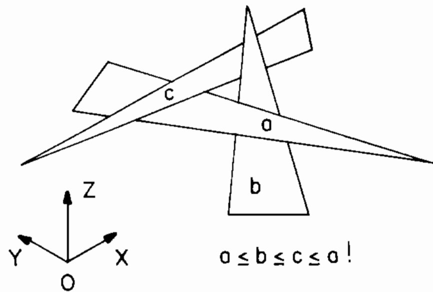

FIGURE 7

adjacency-lists) will be of size $O(\sum_{1 \leqslant i \leqslant p}(V_i + E_i + F_i))$, it suffices to show that for some onstant $c > 0$, we have

$$\sum_{1 \leqslant i \leqslant p} (V_i + E_i + F_i) \leqslant cn. \tag{1}$$

Since each vertex in P_i is adjacent to at least three edges, we have $3V_i \leqslant 2E_i$, therefore by Euler's formula $(F_i - E_i + V_i = 2)$, we derive $E_i \leqslant 3F_i - 6$ and $V_i \leqslant 2F_i - 4$. This shows that $V_i + E_i + F_i < 6F_i$, and since each face appears in at most two distinct polyhedra, we have $\sum_{1 \leqslant i \leqslant p} F_i \leqslant 2n$, therefore $\sum_{1 \leqslant i \leqslant p}(V_i + E_i + F_i) < 12n$, which establishes (1).

Next, we define the *cap* of a convex polyhedron of P as the subset of its faces looking to the right inward, i.e., faces whose inward-directed normal vector has a positive X coordinate. From our assumptions on P, it easily follows that the cap of a polyhedron Q is a connected set of faces whose projection on the YZ plane is a convex partition of a convex polygon (Fig. 8).

We define the following order among caps: $C \preccurlyeq C'$ if there exist two point $(x, y, z) \in C$ and $(x', y, z) \in C'$ with $x < x'$. Whenever it is possible to find a direction for the X axis such that this order is embeddable in a total order, we say that the complex P is *acyclic*. Unfortunately, complexes are often non-acyclic as suggested by Fig. 7. It is however always possible to

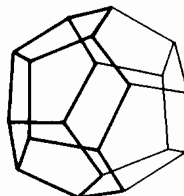

FIGURE 8

refine the complex so as to make it acyclic (Fuchs *et al.*, 1980) so we will assume from now on that P is acyclic. This refinement typically involves splitting faces, therefore usually causes the introduction of new vertices. Let $S = \{C_1, ..., C_m\}$ be the set of caps given in an order that embeds the partial order.

For the sake of clarity, we will first describe a method for locating a point in P that is slightly wasteful of space. With this background it will then be easy to proceed with the description of a linear space algorithm. The underlying search structure T will be a canal-tree defined over the m caps, with the left-to-right order of the leaves corresponding to the order of the caps. We attach an additional search structure to each leaf, so that once the neighboring cap of a query point has been found, we can refine that piece of information and obtain its actual neighboring face. Since the projection of a cap on the YZ plane is a planar graph, we can use any optimal planar point-location algorithm for that purpose (e.g., Cole, 1983; Edelsbrunner *et al.*, in press; Kirkpatrick, 1983; Lipton and Tarjan, 1977). This requires an amount of space linear in the size of caps, so it does not affect the asymptotic space complexity of the whole data structure.

Let L be any line parallel to the X axis; we say that the cap C_i is *right-visible* with respect to a subset of caps W, if there exists a position of L for which the intersection of C_i and L is a point with maximum X coordinate among all intersections between W and L. Let r be the pseudo-root of the underlying tree structure T. We define $J(r)$ as the set of all caps C_i in S that are right-visible with respect to S. $J(\text{root})$ is defined in a similar manner with respect to the set $\{C_1, ..., C_{\lceil m/2 \rceil}\}$. In general, for any internal node v, the set $J(v)$ contains all the caps in $I(z)$ that are right-visible with respect to $I(z)$, where z is the left child of v. Since a query (q, θ) is now a point (x, y, z), with $q = x$ and $\theta = (y, z)$, let us define $L(\theta)$ as the line parallel to the X axis that passes through (x, y, z). The *neighbor* of (q, θ) in $I(z)$ is defined as the cap $C_i \in I(z)$ that intersects $L(\theta)$ at the point with largest X coordinate $\leqslant x$ (or any of them if there are several). If there is no such point, the neighbor is taken to be $-\infty$, as usual. Since we have organized each cap with a planar point location structure, it is easy to retrieve in $O(\log n)$ operations the face of the cap C_i that intersects $L(\theta)$. The next step is to organize $J(v)$ into a data structure, $\text{DS}(J(v))$, which allows us to compute the neighbor of (q, θ) in $I(v)$ very efficiently.

I. A Tentative Solution

For the sake of clarity we will, in a first stage, drop the compaction feature of canal-trees, i.e., the requirement that an object be stored in its corresponding sink and only there. This simplification implies that each node v contains, at all times, the proper information to ensure correct branching. We can regard the problem of branching at node v as a

generalized planar point-location problem. Indeed, let $C_i, \ldots, C_j$ be the caps of $I(z)$ in increasing order (recall that z is the left child of v). Assign a different color to each cap and project their boundaries on the YZ plane. We obtain a set of convex polygons, which we next fill with their respective color, applying the painter's algorithm (i.e., in the order $C_i, \ldots, C_j$) so as to resolve conflicts. This produces a subdivision $K(v)$ of the plane into polygons $t_1, \ldots, t_q$, each part t_l emanating from a cap $C_{f(l)}$. It is important to note that $K(v)$ contains the projection of the boundaries of the caps (cap-projections, for short) and none of the edges within the interior. Finding the rightmost intersection of L with $\{C_i, \ldots, C_j\}$ clearly reduces to locating the region t_l that contains the point (y, z), and reporting the cap $C_{f(l)}$. From that information, we can next turn to the planar point-location structure stored at the leaf $C_{f(l)}$, and retrieve the intersecting face in $O(\log n)$ time.

The choice of caps, rather than faces, as our basic objects is motivated by the following, crucial fact: each edge of $K(v)$ is the projection of a full cap edge. This implies, in particular, that each edge in the caps of $I(z)$ contributes at most one edge in $K(v)$, therefore storing $K(v)$ takes $O(V)$ storage, where V is the number of vertices in all the caps of $I(z)$. To prove the former claim, let us show that each edge u of $K(v)$ is the projection of a whole edge e of some cap C_k and not just some sub-part of it. Let Q be the polyhedron whose cap is C_k, and let f_1 and f_2 be the two faces of Q adjacent to e, with $f_1 \preccurlyeq f_2$. Since some part of C_k in the neighborhood of e is visible, the cap containing f_2 is not in $I(v)$, and for that reason, neither is any face that could prevent a point of e from being visible. This shows that the entire edge e is visible, which proves our claim. We should still be aware that $K(v)$ may not be a collection of disjoint cap projections. It may, indeed, contain projections embedded into one another (Fig. 9).

To summarize, our tentative solution involves associating with v a data structure $\mathrm{DS}(J(v)) = K(v)$, preprocessed for optimal point location. With each face of $K(v)$ we associate a pointer to the corresponding cap so as to be able to retrieve the point $n(v, \theta) \cap L(\theta)$. This scheme will clearly provide an $O(\log^2 n)$ search time while requiring $O(n \log n)$ space.

II. An Improved Solution

Let us now use the compaction feature of canal-trees in order to reduce the space requirement to $O(n)$. Consider any point of a subdivision $K(w)$. This point corresponds uniquely to a point M on some cap C. The basic principle of a canal-tree stipulates that M should be stored at the highest node v from which it is right-visible (a shorthand for saying "right-visible with respect to $I(z)$, where z is the left child of v"). Of course, M will be stored implicitly by keeping at v the set of all points of C sharing the same property. To save space, we will actually only keep the projection of this

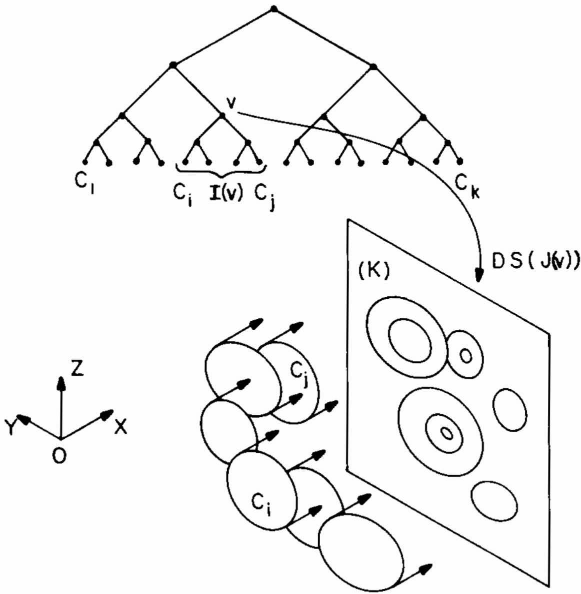

FIGURE 9

set on the YZ plane with a pointer to the cap C. Each point of a cap C has a sink in the canal-tree, i.e., the highest node v from which it is right-visible. This induces a partition of C into parts with common sinks. To visualize this partitioning, just consider each subdivision $K(v)$ as defined in the tentative solution, and decompose the corresponding caps into the parts induced by the subdivision. This defines new caps whose projections will now be pairwise disjoint. This will grant caps the nice property of being either totally visible or totally invisible from the right at any given node of T. The idea is now to store each new cap at the highest node from which it is right-visible. The *holes* thus left in lower levels will correspond to the *info-unavail* elements of the canal-tree.

We next specify the sequence of operations in greater detail. Let r be an internal node of T and let v be its left child. Let S_v be the *silhouette* at v, i.e., the shadow created by $I(v)$ on the plane $X = -\infty$ with a source of light placed at $X = +\infty$. S_v can be obtained by coloring every face of $K(r)$ that is the projection of points lying on a cap of $I(v)$; S_v is in this case the set of edge-disjoint polygons whose boundaries lie between colored and uncolored faces. We say that a point is *inside* S_v if it lies inside any of these boundaries. Let v_1 (resp. v_2) be the left (resp. right) child of v. Following the main idea of the canal-tree, we wish to ensure that at node v only the faces of $K(v)$ inside S_{v_2} should be represented, since all other points have

sinks higher than v. Let $S_{1,2}$ denote the set of points both inside some polygon of S_{v_1} and some polygon of S_{v_2}. We will *add* to $K(v)$ every edge in S_{v_2} that lies inside S_{v_1}, i.e., every edge of $S_{1,2}$ that is not already in $K(v)$. The key observation is that as far as $K(v)$ is concerned, any query handled at v that falls outside $S_{1,2}$ is handled further up in the tree, i.e., at an ancestor of v. Indeed, since each point in $K(v)$ outside $S_{1,2}$ corresponds to a point on a cap that is right-visible from r, its corresponding sink is an ancestor of v. As a result, we may simply delete each edge in $K(v)$ that lies outside $S_{1,2}$. This sequence of additions and then deletions leads to a new subdivision, denoted $K^*(v)$. Of course, each face of $K^*(v)$ outside $S_{1,2}$ will be marked *info-unavail*, and $K^*(v)$ will, as usual, be preprocessed for optimal planar point-location. We carry this construction of $K^*(v)$ for each node $v \in T$, including the root.

Let us show that the space used is now $O(n)$. An important feature of $K^*(v)$ is that each of its edges is the projection of a *whole* side of a cap-boundary (i.e., a full cap-projection edge). This property is true because it also holds for $K(v)$ and $K(r)$. This allows us to associate each edge of $K^*(v)$ with a cap edge. Note in particular that no edge of $S_{1,2}$ need be split in order to be added to $K(v)$. We distinguish between old and new edges, i.e., edges of $K(v) \cap K^*(v)$ and those of $K^*(v) - K(v)$. Each old edge appears only once, namely at the common sink of their points. Each new edge appears also only once (as a new edge), but for a different reason: a new edge in $K^*(v)$ is a silhouette edge, i.e., the projection of an edge of a cap C_2 of $I(v_2)$ onto the cap projection of a uniquely defined cap C_1 of $I(v_1)$; this new edge will be introduced at v, and more generally at the lowest common ancestor of the two leaves corresponding to C_1 and C_2. It follows that only $O(n)$ space is needed.

The neighbor search proceeds as specified in Section 3.3. This involves performing a planar point-location at each node visited during the search, branching left if we land in a face marked *info-unavail*, otherwise branching according to the newly computed result. We leave out the details which were given in Section 3.3. Once again we associate with each face of the new subdivisions a pointer to the unique cap they correspond to. We have avoided dealing with the time complexity of the preprocessing altogether, for it is extremely dependent on the form of the input. We conclude:

THEOREM 3. *It is possible to preprocess an acyclic n-face complex so that locating any point can be done in $O(\log^2 n)$ time, using $O(n)$ storage.*

Let us make a few comments about this result. The reason why we chose caps and not faces as our basic objects comes from the fact that projecting sets of right-visible faces on the YZ plane can often entail a quadratic blow-up. Think of two sets of parallel strips, one vertical and the other

horizontal. Another difficulty comes from the fact that there are in general many total orders to embed a given partial order and that consequently little can be assumed on the relative position of consecutive faces besides the known order between comparable ones.

As an application of Theorem 3, consider the 3-dimensional complex formed by n hyperplanes. It is easy to show that this complex is always acyclic. To see this, consider the directed graph G induced by $\preccurlyeq$. It is clearly acyclic since no path from any cap C of a polyhedron Q can lead to a cap outside the unbounded convex polyhedron R, where R is defined as the intersection of the half spaces containing Q and bounded by the faces of C. We may then embed the relation $\preccurlyeq$ in a total order, and to do so, we proceed as follows: for each polyhedron Q set an arc from its cap C to the cap containing each of Q's faces that are not in C. We avoid computing G explicitly but, instead, simply number each cap by performing a topological sort on H.

THEOREM 4. *Given a subdivision of E^3 by n hyperplanes, it is possible to determine which region contains a query point in $O(\log^2 n)$ time, using $O(n^3)$ storage.*

This result has a number of immediate corollaries. One of them is, of course, the existence of an $O(n^3)$ space algorithm for determining, in $O(\log^2 n)$ time, whether a given test point lies on any of n arbitrary planes in 3-dimensional space. A simple duality argument shows that the same result applies to n arbitrary points to be tested for containment in a query plane. Using the fact that our algorithm returns "neighbors" and not only region names, we can also prove that

THEOREM 5. *It is possible to store n arbitrary points in E^3, using $O(n^3)$ storage, so that the closest point to a query plane can be determined in $O(\log^2 n)$ time.*

Proof. Let us map any point $p\colon (x, y, z)$ of E^3 to the plane $f(p)\colon Z = xX + yY + z$. Since all the points of the plane $P\colon \alpha X + \beta Y + \gamma Z + \varepsilon = 0$ will then be mapped to planes which all pass through the point $(\alpha/\gamma, \beta/\gamma, -\varepsilon/\gamma)$, it is consistent, conversely, to map P to the point $f(P)$: $(\alpha/\gamma, \beta/\gamma, -\varepsilon/\gamma)$. In this way, the vertical distance between points and planes in invariant under the mapping. More precisely, let q be the projection, parallel to the Z axis, of the point p (x, y, z) on the plane P; similarly, let t be the projection of the point $f(P)$ on the plane $f(p)$. We easily check that $p_z - q_z = t_z - f(P)_z = z + (\alpha x + \beta y + \varepsilon)/\gamma$. Since the orthogonal distance from p to P is proportional to the quantity $p_z - q_z$, it suffices to organize the dual set as in Theorem 4 to be able to report the closest point to a query plane in $O(\log^2 n)$ time and $O(n^3)$ space. ∎

4.3. *Computing Nearest-Neighbors in* E^3

Some of the ideas developed above can also be used to improve on the best algorithms known for computing the closest neighbor of a query point in three dimensions (Dobkin and Lipton, 1976; Yao, in press). The problem is that of preprocessing a set S of n points $\{p_1,..., p_n\}$ in E^3, so that for any test point q, an index m such that $[\forall i\,(1 \leqslant i \leqslant n) \mid d(q, p_m) \leqslant d(q, p_i)]$ can be determined very effectively. Let $P(n)$, $S(n)$, and $Q(n)$ be respectively the preprocessing time, storage, and query time of a solution to this problem. The solution given in Dobkin and Lipton (1976) for the 2-dimensional version of this problem can be generalized to E^3. It consists of solving a point-location problem in the complex created by the $n(n-1)/2$ bisecting planes. This can be done in $Q(n) = O(\log n)$ time, but the price to pay is a tremendously large $S(n) = O(n^{14})$ storage requirement. This can be improved by using the point-location algorithm described above, which leads to $S(n) = O(n^6)$ and $Q(n) = O(\log^2 n)$. This does not, however, constitute an improvement over the method proposed by Yao (in press), whose complexity in $P(n) = S(n) = O(n^5 \log n)$ and $Q(n) = O(\log^2 n)$.

We show here how the basic idea of nested binary search used throughout in this paper can be used to extend Shamos's scheme for 2*d*-closest-point problems (Shamos, 1975), and produce a substantial savings in storage. We next describe a nearest-neighbor algorithm, quite simple conceptually, with the following features: $S(n) = P(n) = O(n^2)$ and $Q(n) = O(\log^2 n)$.

Let $V(S)$ denote the Voronoi diagram of S. In (Seidel, 1981) Seidel describes an optimal method for computing convex hulls in E^{2k}. Using a duality argument it is possible to adapt Seidel's algorithm for 4*d*-convex hulls so as to compute $V(S)$ in $O(n^2)$ time. In the following we will denote by $f_S(i, j)$ the face of $V(S)$ supported by the bisector between p_i and p_j. Suppose now that the points $p_1,..., p_n$ appear X-sorted in this order. As usual, our underlying search structure T is a complete binary tree over the n objects, here in left-to-right order, $p_1,..., p_n$. Let v be an internal node of T and let w_1 (resp. w_2) be the left (resp. right) child of v. We define $J(v)$ to be the set of faces in the Voronoi diagram of $I(v)$, whose corresponding points lie in $I(w_1)$ and $I(w_2)$. More precisely, we have

$$J(v) = \{f_{I(v)}(i, j) \mid i \in I(w_1) \text{ and } j \in I(w_2)\}.$$

Recall that $I(v)$ is the set of points p_i that appear at the leaves of the subtree rooted at v. We next prove the following fact.

LEMMA 2. *Any line L parallel to the X axis intersects one and only one face of $J(v)$.*

Proof. Let L intersect the face $f_{I(v)}(i, j)$; $i \in I(w_1)$, $j \in I(w_2)$. Since the vector normal to $f_{I(v)}(i, j)$, $p_i p_j$, has positive X coordinate, the nearest neighbor of a point traveling along L in ascending X order will be successively p_i then p_j, when crossing the face $f_{I(v)}(i, j)$. This shows that L can intersect at most one face in $J(v)$. On the other hand, for any point position of L, the traveling point will start from $x = -\infty$ with its nearest neighbor in $I(w_1)$, eventually to end up having its nearest neighbor in $I(w_2)$. This shows that L always intersects at least one face of $J(v)$, which completes the proof. ∎

It follows from Lemma 2 that the projection of $J(v)$ on the YZ plane is a planar graph and that the projections of no two faces can intersect strictly. We can thus preprocess this graph for efficient searching. This will allow us to determine, in $O(\log n)$ time, which face of $J(v)$ intersects the line L passing through the query point, from which we can decide where to branch in the tree T. We will thus keep in v a pointer to a data structure $\mathrm{DS}(J(v))$, which will be essentially a planar point location structure (Edelsbrunner *et al.*, in press; Kirkpatrick, 1983). Since Edelsbrunner *et al.*'s method (in press) as well as Kirkpatrick's (1983) require only linear preprocessing time, given the clockwise order of the edges around each vertex, and since we can compute $J(v)$ from $V(I(v))$ by a simple depth-first search, both $P(n)$ and $Q(n)$ satisfy the relation: $R(1) = 1$ and $R(n) = 2R(n/2) + O(n^2)$, whence $R(n) = O(n^2)$. Note that, in general, keeping the adjacencies vertex/edge sorted is not a problem, since the degree of a vertex in a 3-dimensional Voronoi diagram is 4 , barring singularities (i.e., more than four points on a common sphere).

THEOREM 6. *It is possible to preprocess n points in $O(n^2)$ time and space, so that any near-neighbor query can be answered in $O(\log^2 n)$ time.*

5. Conclusions and Further Research

We have presented a general scheme for solving a class of problems related to the history of a dynamic data structure. In the various geometric applications given in this paper, we have used one coordinate to set the order between objects, and the two others to define the history. It would be very interesting to study the possibility of dynamizing this scheme, i.e., allowing updates in the overall geometric structure, especially in light of recent advances in the area of dynamization (Bently and Saxe, 1980; Overmars, 1983; van Leeuwen and Wood, 1980).

One may wonder whether it is possible to generalize the 3-dimensional point-location algorithm given here to arbitrary dimensions. The major dif-

ficulty seems to come from the fact that it is not clear at all that by reducing the problem to lower dimensions one is always guaranteed a total ordering among projections. This problem does not arise in three dimensions, for the edges of a planar graph can always be ordered, but what can be said in E^d? Also in the algorithm presented here, we had to use caps instead of faces as our basic objects so as to avoid a quadratic blow-up in the process of reducing dimension. Generalizing this remedy to higher dimensions may be a difficult problem of topology, and one should consult (Zaslavsky, 1975) for egetting a sense of the intricacies of hyperplane arrangements in E^d.

References

Bentley, J. L., and Ottmann, T. (1979), Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.* **C-28**, 643–647.

Bentley, J. L., and Saxe, J. B. (1980), Decomposable searching problems. I. Static-to-dynamic transformation, *J. Algorithms* **1** 301–348.

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

Cole, R. (1983), "Searching and Storing Similar Lists," Tech. Rep. No. 88, New York University.

Dobkin, D. P., and Lipton, R. J. (1976), Multidimensional searching problems, *SIAM J. Comput.* **5**, 181–186.

Dobkin, D. P., and Munro, J. I. (1980), Efficient uses of the past, *in* "Proc. 21st Annu. Found. of Comput. Sci. Sympos.," pp. 200–206.

Edelsbrunner, H. Guibas, L., and Stolfi, J. (in press), Optimal point location in a monotone subdivision, *SIAM J. Comput.*

Fuchs, H., Kedem, Z. M., and Naylor, B. (1980), On visible surface generation by a priori tree structures, *Comput. Graphics* **14** 124–133.

Kirkpatrick, D. G. (1983), Optimal search in planar subdivisions, *SIAM J. Comput.* **12**, No. 1, 28–35.

Lee, D. T., and Preparata, F. P. (1977), Location of a point in a planar subdivision and its applications, *SIAM J. Comput.* **6**, 594–606.

Lipsky, W., and Preparata, F. P. (1981), Segments, rectangles, contours, *J. Algorithms* **2**, 63–76.

Lipton, R. J., and Tarjan, R. E. (1977), Applications of a planar separator theorem, *in* "Proc. 18th Annu. Found. of Comput. Sci. Sympos., pp. 162–170.

McCreight, E. M. (1980), "Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles," Tech. Report. Xerox PARC, CSL-80-9.

Overmars, M. H. (1981), "Searching in the Past I," Report RUU-CS-81-7, University of Utrecht, The Netherlands.

Overmars, M. H. (1983), "The Design of Dynamic Data Structures," Ph. D. thesis, University of Utrecht, The Netherlands.

Seidel, R. (1981), "A Convex Hull Algorithm Optimal for Point Sets in Even Dimensions," Master's thesis, Techn. Report 81–14, Univ. British Columbia, Vancouver, Canada.

Shamos, M. I. (1975), Geometric complexity, *in* "Proc. 7th ACM SIGACT Sympos., pp. 224–233.

VAN LEEUWEN, J., AND WOOD, D. (1980), Dynamization of decomposable searching problems, *Inform. Process. Lett.* **10**, 51–56.

WILLARD, D. E. (in press), New data structures for orthogonal queries, *SIAM J. Comput.*

YAO, A. C. (in press), "On the Preprocessing Cost in Multidimensional Search, Tech. Rep., IBM San Jose Research Center.

ZASLAVSKY, T. (1975), Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes, Mem. *Amer. Math. Soc.* **1**, Issue 1, No. 154.