

FILTERING SEARCH: A NEW APPROACH TO QUERY-ANSWERING*

BERNARD CHAZELLE†

Abstract. We introduce a new technique for solving problems of the following form: preprocess a set of objects so that those satisfying a given property with respect to a query object can be listed very effectively. Well-known problems that fall into this category include *range search*, *point enclosure*, *intersection*, and *near-neighbor* problems. The approach which we take is very general and rests on a new concept called *filtering search*. We show on a number of examples how it can be used to improve the complexity of known algorithms and simplify their implementations as well. In particular, *filtering search* allows us to improve on the worst-case complexity of the best algorithms known so far for solving the problems mentioned above.

Key words. computational geometry, database, data structures, filtering search, retrieval problems

AMS (MOS) subject classifications. CR categories 5.25, 3.74, 5.39

1. Introduction. A considerable amount of attention has been recently devoted to the problem of preprocessing a set S of objects so that all the elements of S that satisfy a given property with respect to a query object can be listed effectively. In such a problem (traditionally known as a *retrieval problem*), we assume that queries are to be made in a repetitive fashion, so preprocessing is likely to be a worthwhile investment. Well-known retrieval problems include *range search*, *point enclosure*, *intersection*, and *near-neighbor* problems. In this paper we introduce a new approach for solving retrieval problems, which we call *filtering search*. This new technique is based on the observation that the complexity of the *search* and the *report* parts of the algorithms *should* be made dependent upon each other—a feature absent from most of the algorithms available in the literature. We show that the notion of filtering search is versatile enough to provide significant improvements to a wide range of problems with seemingly unrelated solutions. More specifically, we present improved algorithms for the following retrieval problems:

1. *Interval Overlap*: Given a set S of n intervals and a query interval q , report the intervals of S that intersect q [8], [17], [30], [31].
2. *Segment Intersection*: Given a set S of n segments in the plane and a query segment q , report the segments of S that intersect q [19], [34].
3. *Point Enclosure*: Given a set S of n d -ranges and a query point q in \mathbb{R}^d , report the d -ranges of S that contain q (a d -range is the Cartesian product of d intervals) [31], [33].
4. *Orthogonal Range Search*: Given a set S of n points in \mathbb{R}^d and a query d -range q , report the points of S that lie within q [1], [2], [3], [5], [7], [9], [21], [22], [23], [26], [29], [31], [35], [36].
5. *k-Nearest-Neighbors*: Given a set S of n points in the Euclidean plane E^2 and a query pair (q, k) , with $q \in E^2$, $k \leq n$, report the k points of S closest to q [11], [20], [25].
6. *Circular Range Search*: Given a set S of n points in the Euclidean plane and a query disk q , report the points of S that lie within q [4], [10], [11], [12], [16], [37].

* Received by the editors September 15, 1983, and in final revised form April 20, 1985. This research was supported in part by the National Science Foundation under grants 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 no. 4786.

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

For all these problems we are able to reduce the worst-case *space* \times *time* complexity of the best algorithms currently known. A summary of our main results appears in a table at the end of this paper, in the form of complexity pairs (*storage*, *query time*).

2. Filtering search. Before introducing the basic idea underlying the concept of filtering search, let us say a few words on the applicability of the approach. In order to make filtering search feasible, it is crucial that the problems specifically require the exhaustive enumeration of the objects satisfying the query. More formally put, the class of problems amenable to filtering search treatment involves a finite set of objects S , a (finite or infinite) query domain Q , and a predicate P defined for each pair in $S \times Q$. The question is then to preprocess S so that the function g defined as follows can be computed efficiently:

$$g: Q \mapsto 2^S; [g(q) = \{v \in S \mid P(v, q) \text{ is true}\}].$$

By computing g , we mean reporting each object in the set $g(q)$ exactly once. We call algorithms for solving such problems *reporting* algorithms. Database management and computational geometry are two areas where retrieval problems frequently arise. Numerous applications can also be found in graphics, circuit design, or statistics, just to pick a few items from the abundant literature on the subject. Note that a related, yet for our purposes here, fundamentally different class of problems, calls for computing a single-valued function of the objects that satisfy the predicate. *Counting* instead of *reporting* the objects is a typical example of this other class.

On a theoretical level it is interesting to note the dual aspect that characterizes the problems under consideration: *preprocessing resources* vs. *query resources*. Although the term “resources” encompasses both space and time, there are many good reasons in practice to concentrate mostly on *space* and *query time*. One reason to worry more about storage than preprocessing time is that the former cost is permanent whereas the latter is temporary. Another is to see the preprocessing time as being amortized over the queries to be made later on. Note that in a parallel environment, *processors* may also be accounted as resources, but for the purpose of this paper, we will assume the standard sequential RAM with infinite arithmetic as our only model of computation.

The worst-case query time of a reporting algorithm is a function of n , the input size, and k , the number of objects to be reported. For all the problems discussed in this paper, this function will be of the form $O(k + f(n))$, where f is a slow-growing function of n . For notational convenience, we will say that a retrieval problem admits of an $(s(n), f(n))$ -algorithm if there exists an $O(s(n))$ space data structure that can be used to answer any query in time $O(k + f(n))$. The hybrid form of the query time expression distinguishes the *search* component of the algorithm (i.e. $f(n)$) from the *report* part (i.e. k). In general it is rarely the case that, chronologically, the first part totally precedes the latter. Rather, the two parts are most often intermixed. Informally speaking, the computation usually resembles a partial traversal of a graph which contains the objects of S in separate nodes. The computation appears as a sequence of two-fold steps: (*search*; *report*).

The purpose of this work is to exploit the hybrid nature of this type of computation and introduce an alternative scheme for designing reporting algorithms. We will show on a number of examples how this new approach can be used to improve the complexity of known algorithms and simplify their implementations. We wish to emphasize the fact that this new approach is absolutely general and is not a priori restricted to any particular class of retrieval problems. The traditional approach taken in designing

reporting algorithms has two basic shortcomings:

1. The search technique is independent of the number of objects to be reported. In particular, there is no effort to balance k and $f(n)$.

2. The search attempts to locate the objects to be reported and *only* those.

The first shortcoming is perhaps most apparent when the entire set S is to be reported, in which case no search at all should be required. In general, it is clear that one could take great advantage of an *oracle* indicating a lower bound on *how many* objects were to be reported. Indeed, this would allow us to restrict the use of a sophisticated structure only for small values of k . Note in particular that if the oracle indicates that $n/k = O(1)$ we may simply check all the objects naively against the query, which takes $O(n) = O(k)$ time, and is asymptotically optimal. We can now introduce the notion of filtering search.

A reporting algorithm is said to use *filtering search* if it attempts to match searching and reporting times, i.e. $f(n)$ and k . This feature has the effect of making the search procedure *adaptively* efficient. With a search increasingly slow for large values of k , it will be often possible to reduce the size of the data structure substantially, following the general precept: *the larger the output, the more naive the search*.

Trying to achieve this goal points to one of the most effective lines of attack for filtering search, also thereby justifying its name: this is the prescription to report more objects than necessary, the excess number being at most proportional to the actual number of objects to be reported. This suggests including a postprocessing phase in order to *filter out* the extraneous objects. Often, of course, no “clean-up” will be apparent.

What we are saying is that $O(k + f(n))$ extra reports are permitted (as long, of course, as any object can be determined to be good or bad in constant time). Often, when k is much smaller than $f(n)$, it will be very handy to remember that $O(f(n))$ extra reports are allowed. This simple-minded observation, which is a particular case of what we just said, will often lead to striking simplifications in the design of reporting algorithms. The main difficulty in trying to implement filtering search is simulating the oracle which indicates a lower bound on k beforehand. Typically, this will be achieved in successive steps, by guessing higher and higher lower bounds, and checking their validity as we go along. The time allowed for checking will of course increase as the lower bounds grow larger.

The idea of filtering search is not limited to algorithms which admit of $O(k + f(n))$ query times. It is general and applies to *any* retrieval problem with nonconstant output size. Put in a different way, the underlying idea is the following: let $h(q)$ be the time “devoted” to searching for the answer $g(q)$ and let $k(q)$ be the output size; the reporting algorithm should attempt to match the functions h and k as closely as possible. Note that this induces a highly nonuniform cost function over the query domain.

To summarize our discussion so far, we have sketched a data structuring tool, *filtering search*; we have stated its underlying philosophy, mentioned one means to realize it, and indicated one useful implementation technique.

1. The *philosophy* is to make the data structure cost-adaptive, i.e. slow and economical whenever it can afford to be so.

2. The *means* is to build an oracle to guess tighter and tighter lower bounds on the size of the output (as we will see, this step can sometimes be avoided).

3. The *implementation technique* is to compute a coarse superset of the output and filter out the undesirable elements. In doing so, one must make sure that the set computed is at most proportional in size to the actual output.

Besides the improved complexity results which we will describe, the novelty of filtering search lies in its abstract setting and its level of generality. In some broad sense, the steps guiding the computation (search) and those providing the output (report) are no longer to be distinguished. Surprisingly, even under this extended interpretation, relatively few previous algorithms seem to use anything reminiscent of filtering search. Noticeable exceptions are the scheme for *circular range search* of Bentley and Maurer [4], the *priority search tree* of McCreight [31], and the algorithm for *fixed-radius neighbor search* of Chazelle [10].

After these generalities, we are now ready to look at filtering search in action. The remainder of this paper will be devoted to the retrieval problems mentioned in the introduction.

3. Interval overlap problems. These problems lead to one of the simplest and most illustrative applications of filtering search. Let $S = \{[a_1, b_1], \dots, [a_n, b_n]\}$; the problem is reporting all the intervals in S that intersect a query interval q . Optimal solutions to this problem can be found in [8] (w.r.t. query time) and [17], [30], [31] (w.r.t. both space and query time). All these methods rely on fairly sophisticated tree structures with substantial implementation overhead. We can circumvent these difficulties by using a drastically different method based on filtering search. This will allow us to achieve both optimal space ($s(n) = n$) and optimal query time ($f(n) = \log_2 n$), as well as much greater simplicity. If we assume that the query endpoints are taken in a range of $O(n)$ elements, our method will give us an $(n, 1)$ -algorithm, which constitutes the only optimal algorithm known for the discrete version of the problem. One drawback of our method, however, is that it does not seem to be easily dynamized.

To begin with, assume that all query and data set endpoints are real numbers. We define a *window-list*, $W(S)$, as an ordered sequence of linear lists (or *windows*), W_1, \dots, W_p , each containing a number of intervals from S (Fig. 1). Let I_j be the interval (in general not in S) spanned by W_j . For simplicity, we will first look at the case where the query interval is reduced to a single point x . Note that this restriction is precisely the one-dimensional point enclosure problem. Let $S(x) = \{[a_i, b_i] \in S \mid a_i \leq x \leq b_i\}$ be the set of intervals of S that contain x . The idea is to ensure that the window enclosing x contains a superset of $S(x)$, but this superset contains at most $\delta - 1$ times too many intervals, where δ is a parameter to be defined later on. We can then simply read out the entire contents of the window, keeping only the desired intervals. More precisely, let $\alpha_1, \dots, \alpha_{2n}$ be the set of endpoints in S in ascending order. Each window W_j has associated with it an *aperture*, i.e. an open interval $I_j = (\alpha_{i_j}, \alpha_{i_{j+1}})$ with $\alpha_{i_j} = \alpha_1$, $\alpha_{i_{j+1}} = \alpha_{2n}$ and $\alpha_{i_{j+1}} \leq \alpha_{i_{j+1}}$. As can be seen, the windows of

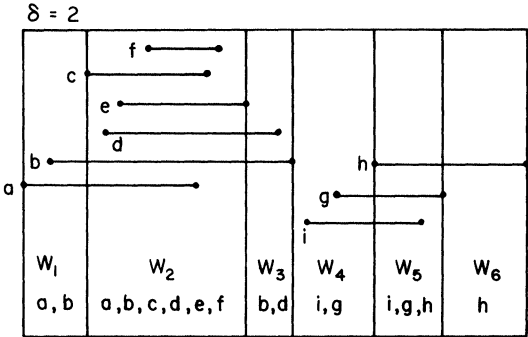


FIG. 1

$W(S)$ induce a partition of $[\alpha_1, \alpha_{2n}]$ into contiguous intervals I_1, \dots, I_p and their endpoints.

Let us assume for the time being that it is possible to define a window-list which satisfies the following density-condition: (δ is a constant > 1)

$$\sum_{1 \leq j \leq p} |W_j| < \frac{2\delta}{\delta - 1} n$$

and

$$\forall x \in I_j, \quad S(x) \subseteq W_j \quad \text{and} \quad 0 < |W_j| \leq \delta \max(1, |S(x)|).$$

It is then easy to solve the point enclosure problem in optimal time and space. To do so, we first determine which aperture I_j contains the query x and then simply check the intervals in the window W_j , reporting those containing x . If x falls exactly on the boundary between two apertures, we check both of their windows. Duplicates are easily detected by keeping a flag bit in the table where S is stored as a list of pairs. Access to windows is provided by a sorted array of cells. Each cell contains two pointers: one to α_{ij} (for binary search) and one to W_j (or a null pointer if $j = p + 1$). Each window W_j is an array of records, each pointing to an entry in S ; the end is marked with a sentinel. The storage needed is $3p + 2\delta/(\delta - 1)n + O(1)$ words of roughly $\log_2 n$ bits each. This is to be added to the $O(n)$ input storage, whose exact size depends on the number representation. Because of the density-condition, the algorithm requires $O(\delta|S(x)| + \log p)$ comparisons per query, which gives an $O(k + \log n)$ query time. Note that the parameter δ allows us to trade off time and space. We next show that window-lists can be easily constructed. We first give an algorithm for setting up the data structure, and then we prove that it satisfies the density-condition.

Procedure Window-Make

```

 $W_1 = \emptyset; j := 1; T := 0; \text{low} := 1; \text{cur} := 0$ 
for  $i := 1$  to  $2n$  do
    if  $\alpha_i$  = left-point then
         $\text{cur} := \text{cur} + 1$ 
         $T := T + 1$ 
        if  $\delta \times \text{low} < T$ 
            then  $\text{Clear}(i, j)$ 
                 $W_j := W_j \cup$  "current interval"
                 $\text{low} := T := \text{cur}$ 
        else  $W_j := W_j \cup$  "current interval"
    else " $\alpha_i$  = right-point"
         $\text{cur} := \text{cur} - 1$ 
         $\text{low} := \min(\text{low}, \text{cur})$ 
        if  $\delta \times \text{low} < T$  then
             $\text{Clear}(i, j)$ 
             $T := \text{cur}; \text{low} := \max(1, T)$ 
    
```

Procedure Clear (i, j)

```

 $j := j + 1$ 
 $W_j := \{\text{intervals } [x, y] \text{ of } W_{j-1} \text{ s.t. } y > \alpha_i\}$ 
    
```

We assume that the α_i are sorted in preprocessing; ties are broken arbitrarily. This requires $O(n \log n)$ operations, which actually dominates the linear running time of the construction phase Window-Make. This part involves scanning each α_i in ascending order and setting up the windows from left to right. We keep inserting new intervals into the current window W_j as long as T , the number of intervals so far inserted in W_j , does not exceed $\delta \times \text{low}$, where "low" is the size of the smallest $S(x)$

found so far in W_j . The variable “cur” is only needed to update “low”. It denotes the number of intervals overlapping the current position. Whenever the condition is no longer satisfied, the algorithm initializes a new window (Clear) with the intervals already overlapping in it.

A few words must be said about the treatment of endpoints with equal value. Since ties are broken arbitrarily, the windows of null aperture which will result from shared endpoints may carry only partial information. The easiest way to handle this problem is to make sure that the pointer from any shared endpoint (except for α_{2n}) leads to the first window to the right with *nonnull* aperture. If the query falls right on the shared endpoint, both its left and right windows with *nonnull* aperture are to be examined; the windows in between are ignored, and for this reason, need not be stored.

By construction, the last two parts of the density-condition, $\forall x \in I_j, S(x) \subseteq W_j$ and $0 < |W_j| \leq \delta \max(1, |S(x)|)$, are always satisfied, so all that remains to show is that the overall storage is linear.

LEMMA 1. $\sum_{1 \leq j \leq p} |W_j| < 2\delta/(\delta-1) n$.

Proof. Any interval $[a_i, b_i]$ can be decomposed into its window-parts, i.e. the parts where it overlaps with the I_j 's. Parts which fully coincide with some I_j are called A-parts; the others are called B-parts. In Fig. 1, for example, segment b has one B-part in W_1 , one A-part in W_2 , and one A-part in W_3 . Let A_j (resp. B_j) be the number of A-parts (resp. B-parts) in the window W_j . Note that for the purposes of this proof we substitute ranks for endpoints. This has the effect that no interval can now share endpoints (although they may in S); this implies in particular that $A_p \leq 1$. Since a new window W_{j+1} is constructed only when $\min_{x \in I_j} (\delta |S(x)|) < |W_j| + 1$, if scanning a start-point, and $\delta (\min_{x \in I_j} |S(x)| - 1) < |W_j|$, if scanning an endpoint, we always have $\delta A_j \leq \min_{x \in I_j} (\delta |S(x)|) < A_j + B_j + \delta$, for $j < p$. From this we derive

$$\sum_{1 \leq j \leq p} (A_j + B_j) < \frac{\delta}{\delta-1} \sum_{1 \leq j < p} B_j + \frac{\delta}{\delta-1} (p-1) + A_p + B_p.$$

Each endpoint a_i, b_i gives rise to at most one B-part (although some might be shared), but exactly $p+1$ endpoints do not give rise to any (falling on the boundary of a window). This implies that $\sum_{1 \leq j \leq p} B_j \leq 2n - p - 1$, and since $A_p \leq 1$ and $\delta > 1$, we have

$$\sum_{1 \leq j \leq p} |W_j| = \sum_{1 \leq j \leq p} (A_j + B_j) < \frac{2\delta}{\delta-1} n - \frac{2\delta + B_p}{\delta-1} + A_p < \frac{2\delta}{\delta-1} n. \quad \square$$

A major asset of window-lists is their simplicity of implementation. Another advantage which, at first sight, may seem paradoxical, is that window-lists are *not* based on a tree-structure. This allows us to solve the discrete version of the point enclosure problem extremely efficiently. This variant of the problem accepts only queries from a set of integers Q of range $O(n)$. We can then dispense with the preliminary binary search and find the corresponding window in constant time. To do so, we simply have to provide a table providing the correspondence point/window, which requires only $O(n)$ additional words of storage. This is a new result in that it departs from the structures in [8], [17], [30], [31], where roughly $\log_2 n$ comparisons are always required, even in the discrete case.

For the *interval overlap problem*, where the query now becomes an interval I , and all the $[a_i, b_i]$ overlapping with I are to be reported, we can still use the window-list by checking all the windows overlapping with I . An argument similar to the proof of Lemma 1 would show that, there again, a superset of $O(k)$ intervals will be reported. We omit the proof.

THEOREM 1. *There exist an $(n, \log n)$ -algorithm, based on window-lists, for solving the interval overlap problem, and an $(n, 1)$ -algorithm for solving the discrete version of the problem. Both algorithms are optimal.*

4. Segment intersection problems. *Given a set S of n segments in the Euclidean plane and a query segment q in arbitrary position, report all the segments in S that intersect q .*

For simplicity, we will assume (in § 4.1, § 4.2, § 4.3) that the n segments may intersect only at their endpoints. This is sometimes directly satisfied (think of the edges of a planar subdivision), but at any rate we can always ensure this condition by breaking up intersecting segments into their separate parts. Previous work on this problem includes an $(n \log n, \log n)$ -algorithm for solving the orthogonal version of the problem in two dimensions, i.e. when query and data set segments are mutually orthogonal [34], and an $(n^3, \log n)$ -algorithm for the general problem [19].

We show here how to improve both of these results, using filtering search. We first give improved solutions for the orthogonal problem (§ 4.1) and generalize our results to a wider class of intersection problems (§ 4.2). Finally we turn our attention to the general problem (§ 4.3) and also look at the special case where the query is an infinite line (§ 4.4).

4.1. The hive-graph. We assume here that the set S consists of horizontal segments and the query segment is vertical. We present an optimal $(n, \log n)$ -algorithm for this problem, which represents an improvement of a factor of $\log n$ in the storage requirement of the best method previously known [34]. The algorithm relies on a new data structure, which we call a *hive-graph*.

To build our underlying data structure, we begin by constructing a planar graph G , called the *vertical adjacency map*. This graph, first introduced by Lipski and Preparata [27], is a natural extension of the set S . It is obtained by adding the following lines to the original set of segments: for each endpoint M , draw the longest vertical line passing through M that does not intersect any other segment, except possibly at an endpoint. Note that this “line” is always a segment, a half-line, or an infinite line. It is easy to construct an adjacency-list representation of G in $O(n \log n)$ time by applying a standard sweep-line algorithm along the x -axis; we omit the details. Note that G trivially requires $O(n)$ storage.

Next, we suggest a tentative algorithm for solving our intersection problem: preprocess G so that any point can be efficiently located in its containing region. This planar point location problem can be solved for general planar subdivisions in $O(\log n)$ query time, using $O(n)$ space [15], [18], [24], [28]. We can now locate, say, the lowest endpoint, (x, y_1) , of the query segment q , and proceed to “walk” along the edges of G , following the direction given by q . Without specifying the details, it is easy to see that this method will indeed take us from one endpoint to the other, while passing through all the segments of G to be reported. One fatal drawback is that many edges traversed may not contribute any item to the output. Actually, the query time may be linear in n , even if few intersections are to be reported (Fig. 2).

To remedy this shortcoming we introduce a new structure, the *hive-graph* of G , denoted $H(G)$. $H(G)$ is a refinement of G supplied with a crucial *neighboring* property. Like G , $H(G)$ is a planar subdivision with $O(n)$ vertices, whose bounded faces are rectangles parallel to the axes. It is a “supergraph” of G , in the sense that it can be constructed by adding only vertical edges to G . Furthermore, $H(G)$ has the important property that each of its faces is a rectangle (possibly unbounded) with at most two extra vertices, one on each horizontal edge, in addition to its 4 (or fewer) corners. It

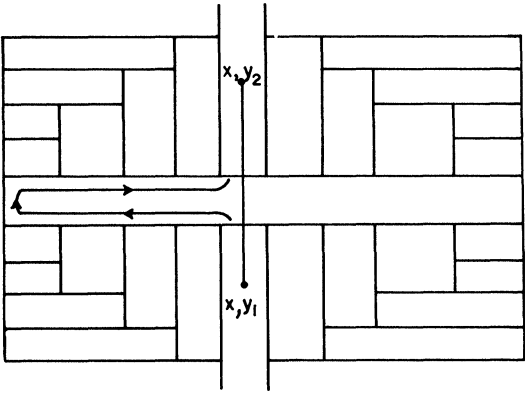


FIG. 2

is easy to see that the existence of $H(G)$ brings about an easy fix to our earlier difficulty. Since each face traversed while following the query's direction contains an edge supported by a segment of S to be reported, the presence of at most 6 edges per face ensures an $O(k)$ traversal time, hence an $O(k + \log n)$ query time.

A nice feature of this approach is that it reports the intersections in sorted order. Its “on-line” nature allows questions of the sort: report the first k intersections with a query half-line. We now come back to our earlier claim.

LEMMA 2. *There exists a hive-graph of G ; it requires $O(n)$ storage, and can be computed in $O(n \log n)$ time.*

Proof. We say that a rectangle in G has an upper (resp. lower) *anomaly* if its upper (resp. lower) side consists of at least three edges. In Fig. 3, for example, the upper anomalies are on s_3, s_7 , and the lower anomalies on s_1, s_2, s_4 . In order to produce $H(G)$, we augment the graph G in two passes, one pass to remove each type of anomaly. If G is given a standard adjacency-list representation and the segments of S are available in decreasing y -order, s_1, \dots, s_m , the graph $H(G)$ can be computed in $O(n)$ time. Note that ensuring these conditions can be done in $O(n \log n)$ steps.

The two passes are very similar, so we may restrict our investigation to the first one. $W \log$, we assume that all y -coordinates in S are distinct. The idea is to sweep

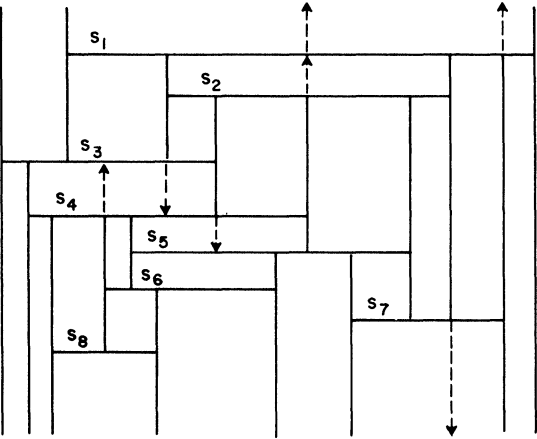


FIG. 3

an infinite horizontal line L from top to bottom, stopping at each segment in order to remove the upper anomalies. Associated with L we keep a data structure $X(L)$, which exactly reflects a cross-section of G as it will appear after the first pass.

$X(L)$ can be most simply (but not economically) implemented as a doubly linked list. Each entry is a pointer to an *old* vertical edge of G or to a *new* vertical edge. We can assume the existence of flags to distinguish between these two types. Note the distinction we make between edges and segments. The notion of edges is related to G , so for example, each segment s_i is in general made of several edges. Similarly, vertical segments adjacent to endpoints of segments of S are made of two edges. Informally, L performs a “combing” operation: it scans across G from top to bottom, dragging along vertical edges to be added into G . The addition of these new segments results in $H(G)$.

Initially L lies totally above G and each entry in $X(L)$ is *old* (in Fig. 3, $X(L)$ starts out with three entries). In the following, we will say that a vertical edge is *above* (resp. *below*) s_i if it is adjacent to it and lies completely above (resp. below) s_i . For $i = 1, \dots, n$, perform the following steps.

1. *Identify relevant entries.* Let a_1 be the leftmost vertical edge above s_i . From a_1 scan along $X(L)$ to retrieve, in order from left to right, all the edges $A = \{a_1, a_2, \dots, a_m\}$ above s_i .

2. *Update adjacency lists.* Insert into G the vertices formed by the intersections of s_i and the new edges of A . Note that if $i = 1$, there are no such edges.

3. *Update $X(L)$.* Let $B = \{b_1, \dots, b_p\}$ be the edges below s_i , in order from left to right (note that a_1 and b_1 are collinear). Let F be a set of edges, initially empty. For each $j = 1, \dots, p - 1$, retrieve the edges of A that lie strictly between b_j and b_{j+1} . By this, we mean the set $\{a_u, a_{u+1}, \dots, a_v\}$ such that the x -coordinate of each a_k ($u \leq k \leq v$) is strictly larger (resp. smaller) than the x -coordinate of b_j (resp. b_{j+1}). Include in F every other edge in this set, i.e. $\{a_{u+1}, a_{u+3}, \dots, a_{u+w}\}$, where w is the largest odd integer not exceeding $v - u$ (whenever these indices are meaningful)—see Fig. 4. Finally, delete from $X(L)$ all entries in A , insert all edges in B and in F , marking the latter *new*.

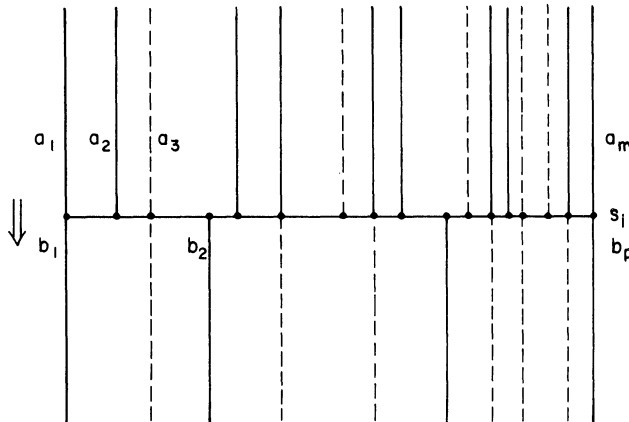


FIG. 4

This part of the algorithm can be easily implemented to run in time proportional to the size of the resulting graph. Access to a_1 in step 1 is provided in constant time by keeping a table associating with each s_i the relevant entry in $X(L)$ (if any). Steps 2 and 3 require a total of $O(m + p)$ operations, which proves our claim.

It is not difficult to see that G , in its final state, is free of upper anomalies. For every s_i , in turn, the algorithm considers the upper sides of each rectangle attached to s_i below, and subdivides these rectangles (if necessary) to ensure the presence of at most one extra vertex per upper side. Note that the creation of new edges may result in new anomalies at lower levels. This propagation of anomalies does not make it obvious that the number of edges in G should remain linear in n . This will be shown in the next paragraph.

The second pass consists of pulling L back up, applying the same algorithm with respect to lower sides. It is clear that the new graph produced, $H(G)$, has no anomalies since the second pass cannot add new upper anomalies. To see that the transformation does not add too many edges, we can imagine that we start with $O(n)$ objects which have the power to “regenerate” themselves (which is different from “duplicate”, i.e. the total number of objects alive at any given time is $O(n)$ by definition). Since we extend only every other anomaly, each regeneration implies the “freezing” of at least another object. This limits the maximum number of regenerations to $O(n)$, and each pass can at most double the number of vertical edges. This proves that $|H(G)| = O(n)$, which completes the proof. \square

THEOREM 2. *It is possible to preprocess n horizontal segments in $O(n \log n)$ time and $O(n)$ space, so that computing their intersections with an arbitrary vertical query segment can be done in $O(k + \log n)$ time, where k is the number of intersections to be reported. The algorithm is optimal.*

Note that if the query segment intersects the x -axis we can start the traversal from the intersection point, which will save us the complication of performing planar point location. Indeed, it will suffice to store the rectangles intersecting the x -axis in sorted order to be able to perform the location with a simple binary search. This gives us the following result, which will be instrumental in § 5.

COROLLARY 1. *Given a set S of n horizontal segments in the plane and a vertical query segment q intersecting a fixed horizontal line, it is possible to report all k segments of S that intersect q in $O(k + \log n)$ time, using $O(n)$ space. The algorithm involves a binary search in a sorted list ($O(\log n)$ time), followed by a traversal in a graph ($O(k)$ time).*

We wish to mention another application of hive-graphs, the *iterative search* of a database consisting of a collection of sorted lists S_1, \dots, S_m . The goal is to preprocess the database so that for any triplet (q, i, j) , the test value q can be efficiently looked up in each of the lists S_i, S_{i+1}, \dots, S_j (assuming $i < j$). If n is the size of the database, this can be done in $O((j+i) \log n)$ time by performing a binary search in each of the relevant lists. We can propose a more efficient method, however, by viewing each list S_i as a chain of horizontal segments, with y -coordinates equal to i . In this way, the computation can be identified with the computation of all intersections between the segment $[(q, i), (q, j)]$ and the set of segments. We can apply the hive-graph technique to this problem.¹

COROLLARY 2. *Let C be a collection of sorted lists S_1, \dots, S_m , of total size n , with elements chosen from a totally ordered domain U . There exists an $O(n)$ size data structure so that for any $q \in U$ and i, j ($1 \leq i \leq j \leq m$), the elements of S_i, S_{i+1}, \dots, S_j immediately following q (if any) can be found in $O(j - i + \log n)$ time. The algorithm is optimal.*

4.2. Generalizing the hive-graph. We consider now the case where the segments of S may assume an arbitrary position and the query segment has its supporting line passing through a fixed point O . We can adapt the hive-graph to handle this particular

¹Chazelle and Guibas [13] have recently extended the notion of hive-graph into a general technique for iterative search problems, called *fractional cascading*.

situation, as we proceed to show. Let J be the planar subdivision defined as follows: for each endpoint p in S , draw the longest line collinear with the origin O , passing through p , that does not intersect any other segment except possibly at an endpoint (Fig. 5). We ensure that this “line” actually stops at point O if passing through it, so that it is always a segment or a half-line.

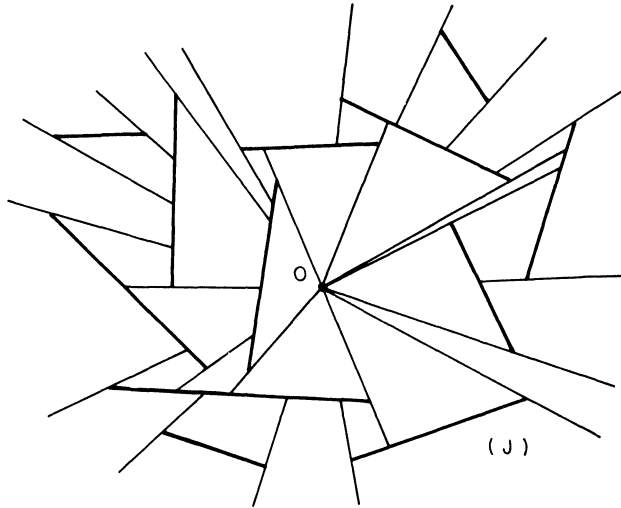


FIG. 5

It is easy to construct the adjacency-list of J in $O(n \log n)$ time, by using a standard sweep-line technique. The sweep-line is a “radar-beam” centered at O . At any instant, the segments intersecting the beam are kept in sorted order in a dynamic balanced search tree, and segments are either inserted or deleted depending on the status (first endpoint or last endpoint) of the vertex currently scanned. We omit the details. In the following, we will say that a segment is *centered* if its supporting line passes through O . As before, the hive-graph H is a planar subdivision with $O(n)$ vertices built on top of J . It has the property that each of its faces is a quadrilateral or a triangle (possibly unbounded) with two centered edges and at most two extra vertices on the noncentered edges (Fig. 6). As before, H can be easily used to solve the intersection problem at hand; we omit the details.

We can construct H efficiently, proceeding as in the last section. All we need is a new partial order among the segments of S . We say that $s_i \preceq s_j$ if there exists a ray (i.e. a half-line) emanating from O that intersects both segments, s_i before s_j . Unfortunately, as shown in Fig. 7, the directed graph induced by this relation may contain cycles. We can easily remedy this shortcoming, however, by breaking up into their two subparts each segment intersecting, say, the vertical ray emanating upwards from O .

LEMMA 3. *The relation \preceq can be embedded in a total order.*

Proof. It suffices to prove that the induced graph does not contain any cycle. Suppose that it does, and let $s_{i_1}, \dots, s_{i_k}, s_{i_1}$ be the shortest cycle in the graph. Call P_{i_j} the smallest wedge centered at O containing s_{i_j} . Since the cycle is the shortest, it is easy to see that the wedges P_{i_1}, \dots, P_{i_k} overlap two by two, but 1) three never overlap at the same point and 2) one never contains another totally. This shows that the sequence of wedges P_{i_1}, \dots, P_{i_k} is monotonically rotating, either clockwise or counter-clockwise. This must stop before crossing the vertical ray, however, since there is no possible relation between two edges on opposite sides of the ray, hence a contradiction. \square

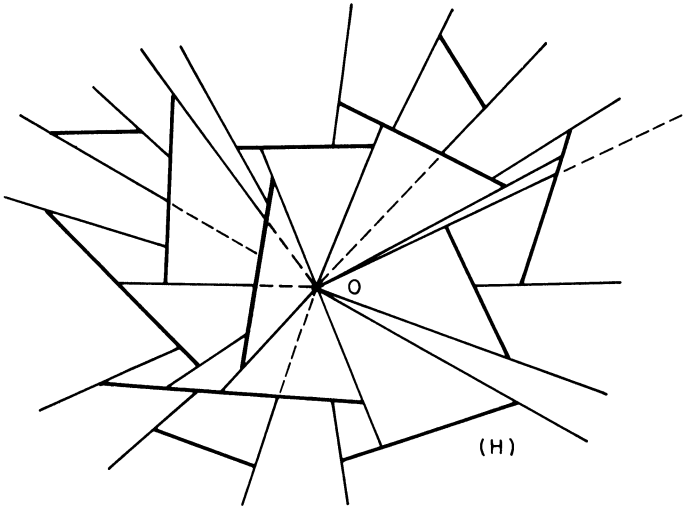


FIG. 6

Lemma 3 allows us to embed the relation \preceq into a total order, called *ray-order*. To do so, we retrieve the subset of the partial order provided by J . This consists of all pairs s_i, s_j such that there exists a ray from O that intersects s_i before s_j , and nothing in between. It is easily shown that this order, denoted \preceq^* , contains $O(n)$ pairs, and its transitive closure coincides with that of \preceq . Therefore we can embed \preceq into a total order by performing a topological sort on \preceq^* , which can be done in $O(n)$ time. Next, we compute H by proceeding as described in § 4.1, the only difference coming from the fact that the order in which to consider the edges is now given by the ray-order. We omit the details and directly conclude:

LEMMA 4. *The graph H exists, requires $O(n)$ storage, and can be computed in $O(n \log n)$ time.*

We observe that setting O at infinity (in the projective plane) gives us an algorithm for the case where the query segment has a fixed slope.

THEOREM 3. *It is possible to preprocess n segments in $O(n \log n)$ time and $O(n)$ space, so that computing their intersections with a query segment which either has a fixed slope or has its supporting line passing through a fixed point can be done in $O(k + \log n)$ time (k is the number of intersections to be reported). It is assumed that the interior of the segments are pairwise disjoint.*

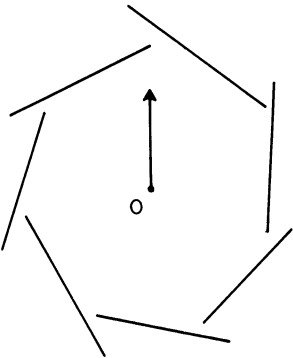


FIG. 7

4.3. The algorithm for the general case. Consider the set of all lines in the Euclidean plane. It is easy to see that the n segments of S induce a partition of this set into connected regions. A mechanical analogy will help to understand this notion. Assume that a line L , placed in arbitrary position, is free to move continuously anywhere so long as it does not cross any endpoint of a segment in S . The range of motion can be seen as a region in a “space of lines”, i.e. a *dual space*. To make this notion more formal, we introduce a well-known geometric transform, T , defined as follows: a point $p: (a, b)$ is mapped to the line $T_p: y = ax + b$ in the dual space, and a line $L: y = kz + d$ is mapped to the point $T_L: (-k, d)$. It is easy to see that a point p lies above (resp. on) a line L if and only if the point T_L lies below (resp. on) the line T_p . Note that the mapping T excludes vertical lines. Redefining a similar mapping in the projective plane allows us to get around this discrepancy. Conceptually simpler, but less elegant, is the solution of deciding on a proper choice of axes so that no segment in S is vertical. This can be easily done in $O(n \log n)$ time.

The transformation T can be applied to segments as well, and we can easily see that a segment s is mapped into a *double wedge*, as illustrated in Fig. 8. Observe that since a double wedge cannot contain vertical rays, there is no ambiguity in the definition of the double wedge once the two lines are known. We can now express the intersection of segments by means of a dual property. Let $W(s)$ and $C(s)$ denote, respectively, the double wedge of segment s and the intersection point thereof (Fig. 8).

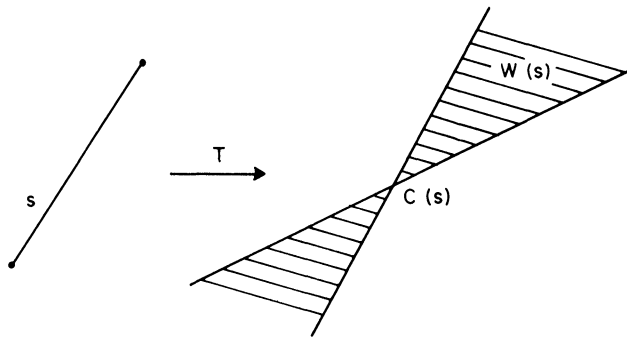


FIG. 8

LEMMA 5. *A segment s intersects a segment t if and only if $C(s)$ lies in $W(t)$ and $C(t)$ lies in $W(s)$.*

Proof. Omitted. \square

This result allows us to formalize our earlier observation. Let G be the subdivision of the dual plane created by the n double wedges $W(s_i)$, with $S = \{s_1, \dots, s_n\}$. The faces of G are precisely the dual versions of the regions of lines mentioned above. More formally, each region r in G is a convex (possibly unbounded) polygon, whose points are images of lines that all cross the same subset of segments, denoted $S(r)$. Since the segments of S are nonintersecting, except possibly at their endpoints, the set $S(r)$ can be totally ordered, and this order is independent of the particular point in r .

Our solution for saving storage over the $(n^3, \log n)$ -algorithm given in [19] relies on the following idea. Recall that q is the query segment. Let $p(q)$ denote the vertical projection of $C(q)$ on the first line encountered below in G . If there is no such line, no segment of S can intersect q (or even the line supporting q) since double wedges cannot totally contain any vertical ray. Since the algorithm is intimately based on the

fast computation of $p(q)$, we will devote the next few lines to it before proceeding with the main part of the algorithm.

From [14], [20], we know how to compute the planar subdivision formed by n lines in $O(n^2)$ time. This clearly allows us to obtain G in $O(n^2)$ time and space. The next step is to organize G into an optimal planar point location structure [15], [18], [24], [28]. With this preprocessing in hand, we are able to locate which region contains a query point in $O(\log n)$ steps. In order to compute $p(q)$ efficiently, we notice that since each face of G is a convex polygon, we can represent it by storing its two chains from the leftmost vertex to the rightmost one. In this way, it is possible to determine $p(q)$ in $O(\log n)$ time by binary search on the x -coordinate of $C(q)$.

We are now ready to describe the second phase of the algorithm. Let H designate the line $T_{p(q)}^{-1}$, and let K denote the line supporting the query segment q (note that $K = T_{C(q)}^{-1}$). The lines H and K are parallel and intersect exactly the same segments of S , say, s_{i_1}, \dots, s_{i_k} , and in the same order. Let $s_{i_a}, s_{i_{a+1}}, \dots, s_{i_{b-1}}, s_{i_b}$ be the list of segments intersecting $q = \alpha\beta$, with $a \leq b$ (Fig. 9). Let p be the point whose transform T_p is the line of G passing through $p(q)$ (note that p is an endpoint of some segment in S). It is easy to see that 1) if p lies between s_{i_a} and s_{i_b} , the segments to be reported are exactly the segments of S that intersect either $p\alpha$ or $p\beta$ (Fig. 9-A). Otherwise, 2) s_{i_a}, \dots, s_{i_b} are the first segments of S intersected by $p\alpha$ or $p\beta$ starting at α or β , respectively (Fig. 9-B). Assume that we have an efficient method for enumerating the segments in S intersected by $p\alpha$ (resp. $p\beta$), in the order they appear from α (resp. β). We can use it to report the intersections of S with $p\alpha$ and $p\beta$ in case 1), as well as the relevant subsequence of intersections of S with $p\alpha$ or $p\beta$ in case 2). Note that in the latter case it takes constant time to decide which of $p\alpha$ or $p\beta$ should be considered. Our problem is now essentially solved since we can precompute a generalized hive-graph centered at each endpoint in S , as described in § 4.2, and apply it to report the intersections of S with $p\alpha$ and $p\beta$.

THEOREM 4. *It is possible to preprocess n segments in $O(n^2 \log n)$ time and $O(n^2)$ space, so that computing their intersections with an arbitrary query segment can be done in $O(k + \log n)$ time, where k is the number of intersections to be reported. It is assumed that the interior of the segments are pairwise disjoint.*

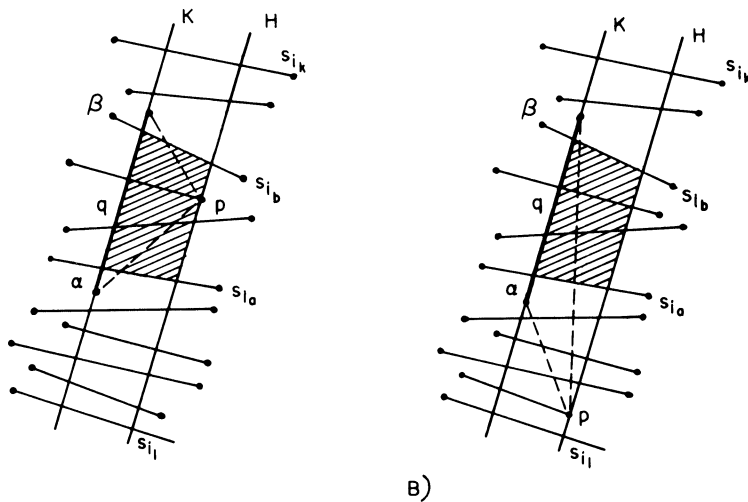


FIG. 9

4.4. The special case of a query line. We will show that if the query q is an infinite line, we can reduce the preprocessing time to $O(n^2)$ and greatly simplify the algorithm as well. At the same time, we can relax the assumption that the segments of S should be disjoint. In other words, the segments in S may now intersect. All we have to do is compute the set R of double wedges that contain the face F where $C(q)(=T_q)$ lies. To do so, we reduce the problem to one-dimensional point enclosure. We need two data structures for each line L in G . Let us orient each of the $\leq 2n$ infinite lines in G , and designate by L_r (resp. L_l) the right (resp. left) side of line L . We construct $D(L_r)$, the data structure associated with L_r by proceeding as follows: let I be the set of intersections between L and each of the n double wedges. These intersections consist of points, segments, pairs of collinear rays, or lines. If L coincides with one of the boundary lines of a double wedge, we do not include the whole line, but only the ray on the boundary of the unique wedge lying in L_r . Finally, we organize I into a window-list, which can be done in $O(n)$ time, given the sorted order provided by G .

The data structure $D(L_l)$ is defined in a similar fashion. It is identical to $D(L_r)$, except for its treatment of wedges sharing a line with L . It is now easy to solve our intersection problem. First, assume that $C(q)$ does not lie on any edge of G . Let e be an edge of F , and let L be its supporting line. *W log*, assume that $C(q)$ lies in L_r . The set R is obtained by probing the window-list $D(L_r)$ with respect to the midpoint of e . If now $C(q)$ lies on e , we probe both $D(L_r)$ and $D(L_l)$, and eliminate duplicate reports with table look-up. Of course, the data structure, as presented, is redundant and can be made more economical. We will not concern ourselves with these implementation issues, however.

THEOREM 5. *It is possible to preprocess n arbitrary segments in $O(n^2)$ time and space, so that computing their intersections with an arbitrary query line can be done in $O(k + \log n)$ time, where k is the number of intersections to be reported.*

5. Point enclosure problems. Given a set S of n d -ranges and a query point q in \mathbb{R}^d , report the d -ranges of S that contain q . Recall that a d -range is the Cartesian product of d intervals. We are able to use filtering search to improve on the most efficient algorithms previously known for the two-dimensional version of the problem: $(n \log^2 n, \log n)$ [33] and $(n, \log^3 n)$ [31]. We present an optimal $(n, \log n)$ -algorithm for this problem; this solution readily generalizes into an $(n \log^{d-2} n, \log^{d-1} n)$ -algorithm in \mathbb{R}^d ($d > 1$). Note that if $d = 1$ we have a special case of the interval overlap problem treated in § 3.

We look at the case $d = 2$, first. S consists of n rectangles whose edges are parallel to the axes. Let L be an infinite vertical line with $|S|$ vertical rectangle edges on each side. The line L partitions S into three subsets: S_l (resp. S_r) contains the rectangles completely to the left (resp. right) of L , and S_m contains the rectangles intersecting L . We construct a binary tree T of height $O(\log n)$ by associating with the root the set $R(\text{root}) = S_m$, and defining the left (resp. right) subtree recursively with respect to S_l (resp. S_r).

Let v be an arbitrary node of T . Solving the point enclosure problem with respect to the set $R(v)$ can be reduced to a generalized version of a one-dimensional point enclosure problem. *W log*, assume that the query $q = (x, y)$ is on the left-hand side of the partitioning line $L(v)$ associated with node v . Let h be the ray $[(-\infty, y), (x, y)]$; the rectangles of $R(v)$ to be reported are exactly those whose left vertical edge intersects h . These can be found in optimal time, using the hive-graph of § 4.1. Note that by carrying out the traversal of the graph from $(-\infty, y)$ to (x, y) we avoid the difficulties

of planar point location (Corollary 1). Indeed, locating $(-\infty, y)$ in the hive-graph can be done by a simple binary search in a sorted list of y -coordinates, denoted $Y_l(v)$. Dealing with query points to the right of $L(v)$ leads to another hive-graph and another set of y -coordinates, $Y_r(v)$.

The data structure outlined above leads to a straightforward $(n, \log^2 n)$ -algorithm. Note that answering a query involves tracing a search path in T from the root to one of its leaves. This method can be improved by embedding the concept of hive-graph into a tree structure.

To speed up the searches, we augment $Y_l(v)$ and $Y_r(v)$ with sufficient information to provide constant time access into these lists. The idea is to be able to perform a single binary search in $Y_l(\text{root}) \cup Y_r(\text{root})$, and then gain access to each subsequent list in a constant number of steps. Let $Y(v)$ be the union of $Y_l(v)$ and $Y_r(v)$. By endowing $Y(v)$ with two pointers per entry (one for $Y_l(v)$ and the other for $Y_r(v)$), searching either of these lists can be reduced to searching $Y(v)$. Let w_1 and w_2 be the two children of v , and let W_1 (resp. W_2) be the list obtained by discarding every other element in $Y(w_1)$ (resp. $Y(w_2)$). Augment $Y(v)$ by merging into it both lists W_1 and W_2 . By adding appropriate pointers from $Y(v)$ to $Y(w_1)$ and $Y(w_2)$, it is possible to look up a value y in $Y(w_1)$ or $Y(w_2)$ in constant time, as long as this look-up has already been performed in $Y(v)$.

We carry out this preprocessing for each node v of T , proceeding bottom-up. Note that the cross-section of the new data structure obtained by considering the lists on any given path is similar to a hive-graph defined with only one refining pass (with differences of minor importance). The same counting argument shows that the storage used is still $O(n)$. This leads to an $(n, \log n)$ algorithm for point enclosure in two dimensions. We wish to mention that the idea of using a hive-graph within a tree structure was suggested to us by R. Cole and H. Edelsbrunner, independently, in the context of planar point location. We conclude:

THEOREM 6. *There exists an $(n, \log n)$ -algorithm for solving the point enclosure problem in \mathbb{R}^2 .*

The algorithm can be easily generalized to handle d -ranges. The data structure $T_d(S)$ is defined recursively as follows: project the n d -ranges on one of the axes, referred to as the x -axis, and organize the projections into a segment tree [8]. Recall that this involves constructing a complete binary tree with $2n-1$ leaves, each leaf representing one of the intervals delimited by the endpoints of the projections. With this representation each node v of the segment-tree “spans” the interval $I(v)$, formed by the union of the intervals associated with the leaves of the subtree rooted at v . Let w be an internal node of the tree and let v be one of its children. We define $R(v)$ as the set of d -ranges in S whose projections cover $I(v)$ but not $I(w)$. We observe that the problem of reporting which of the d -ranges of $R(v)$ contain a query point q whose x -coordinate lies in $I(v)$ is really a $(d-1)$ -point enclosure problem, which we can thus solve recursively. For this reason, we will store in v a pointer to the structure $T_{d-1}(V)$, where V is the projection of $R(v)$ on the hyperplane normal to the x -axis. Note that, of course, we should stop the recursion at $d=2$ and then use the data structure of Theorem 6. A query is answered by searching for the x -coordinate in the segment tree, and then applying the algorithm recursively with respect to each structure $T_{d-1}(v)$ encountered. A simple analysis shows that the query time is $O(\log^{d-1} n + \text{output size})$ and the storage required, $O(n \log^{d-2} n)$.

THEOREM 7. *There exists an $(n \log^{d-2} n, \log^{d-1} n)$ -algorithm for solving the point enclosure problem in \mathbb{R}^d ($d > 1$).*

6. Orthogonal range search problems.

6.1. Two-dimensional range search. A considerable amount of work has been recently devoted to this problem [1], [2], [3], [5], [7], [9], [21], [22], [23], [26], [29], [31], [35], [36]. We propose to improve upon the $(n \log^{d-1} n, \log^{d-1} n)$ -algorithm of Willard [35], by cutting down the storage by a factor of $\log \log n$ while retaining the same time complexity. The interest of our method is three-fold: theoretically, it shows that, say, in two dimensions, logarithmic query time can be achieved with $o(n \log n)$ storage, an interesting fact in the context of lower bounds. More practically, the algorithm has the advantage of being conceptually quite simple, as it is made of several independent building blocks. Finally, it has the feature of being “parametrized”, hence offering the possibility of trade-offs between time and space.

When the query rectangle is constrained to have one of its sides lying on one of the axes, say the x -axis, we are faced with the *grounded 2-range search* problem, for which we have an efficient data structure, the *priority search tree* of McCreight [31]. We will see how filtering search can be used to derive other optimal solutions to this problem (§ 6.3, § 6.4). But first of all, let's turn back to the original problem. We begin by describing the algorithm in two dimensions, and then generalize it to arbitrary dimensions.

In preprocessing, we sort the n points of S by x -order. We construct α vertical slabs which partition S into α equal-size subsets (called *blocks*), denoted B_1, \dots, B_α , from left to right. Each block contains $\lceil n/\alpha \rceil$ points, except for possibly B_α (we assume $1 < \alpha \leq n$). The underlying data structure T is an α -ary tree, defined recursively on the following pattern. Each node v of T points to a data structure $D(v)$, constructed with respect to a certain subset $B(v)$ of S . To the root corresponds the set S itself, and to its i th child from the left corresponds the block B_i . The data structure $D(v)$ pointed to by each node v consists of either the list of points in $B(v)$ if it contains fewer than α points, or else the following items: if v_i is the i th child of v from the left,

1. each $B(v_i)$ is organized into two priority search trees, each grounded on a distinct side of the slab;
2. each $B(v_i)$ is sorted by y -coordinates and the points of $B(v_i)$ are linked together in this order. This forms a polygonal line monotone with respect to the y -axis. Consider the projections on the y -axis of the $|B(v)| - \alpha$ segments thus defined, and organize these projections into a window-list (see § 3).

The tree T being defined recursively in this manner, it is then easy to answer a query $R = \{(x_1, y_1), (x_2, y_2)\}$; $x_1 \leq x_2$ and $y_1 \leq y_2$. Searching for the values x_1 and x_2 in T induces a decomposition of the rectangle R into one, two, or three canonical parts; one, if it fits entirely into a slab; two, if it overlaps with two consecutive ones; three, if it overlaps with more than two. In the general case, R can be rewritten as the juxtaposition of R_1, R^*, R_2 . Both R_1 and R_2 partly overlap with a slab, which gives us two instances of the *grounded 2-range search* problem for which we use the priority search trees. The next step involves determining the polygonal lines intersected by a horizontal side of R , say, $s = [(x_1, y_2), (x_2, y_2)]$. Using the window-list, this can be done in time $O(\alpha + \log n)$. Indeed, it suffices to check the window whose aperture covers y_2 , which will give at most $\alpha - 1$ spurious segments. Finally, for each of the segments intersected by s , we scan the polygonal line to which it belongs, downwards, stopping as soon as we encounter a point below the lower side of R .

It is clear that each level of the tree entails $O(n)$ storage; furthermore the partition of S into equal-size blocks leads to a height of at most $\lceil \log n / \log \alpha \rceil$, therefore the total amount of storage required is $O(n(\log n / \log \alpha))$. Decomposing a query rectangle

R into its ≤ 3 canonical parts can be simply done in $O(\log n)$ time by performing an $O(\log \alpha)$ time binary search at each node traversed ($\lceil \log n / \log \alpha \rceil + O(1)$ nodes are visited). Finally, solving the two grounded 2-range search problems and checking all the appropriate windows can be done in time $O(\alpha + \log n + k)$, which leads to an $(n(\log n / \log \alpha), \alpha + \log n)$ algorithm. Note that setting $\alpha = n^\epsilon$, for $\epsilon < 1$ leads to an (n, n^ϵ) -algorithm which matches the performance given in [5]. More interestingly, matching the first two terms in the expression of the query time leads to an $(n(\log n / \log \log n), \log n)$ -algorithm, which outperforms the algorithm given in [35].

6.2. Orthogonal range search in \mathbb{R}^d . We can use known techniques to extend our method to higher dimensions [2]. The basic idea is to reduce a problem in \mathbb{R}^d to a similar problem in \mathbb{R}^{d-1} . The underlying data structure, $T_d(S)$, is a complete binary tree with n leaves. Let x be one of the axes; the i th leaf from left to right corresponds to the i th point of S in x -order. In this manner, each node v “spans” the set $S(v)$, formed by the points associated with leaves of the subtree rooted at v . As usual, we keep in v a pointer to a data structure $D(v)$, defined as follows: $D(v) = T_{d-1}(S^*(v))$, where $S^*(v)$ is the projection of the set $S(v)$ on the hyperplane ($x=0$). Of course, $T_2(S^*(v))$ is defined by using the method described above. To answer a query, we start by searching for the two x -values of the query d -range in the tree $T_d(S)$. This induces a canonical decomposition of the d -range into at most $2\lfloor \log_2 n \rfloor$ parts, each of them dealt with by projection on the hyperplane ($x=0$) and reduction to an orthogonal range search problem in \mathbb{R}^{d-1} .

Each increment in dimensionality adds a factor of $\log n$ in the time and space complexity of the algorithm, therefore since for $d=2$, we have an $(n(\log n / \log \log n), \log n)$ -algorithm, we conclude with

THEOREM 8. *There exists an $(n(\log^{d-1} n / \log \log n), \log^{d-1} n)$ -algorithm for solving the orthogonal range search problem in \mathbb{R}^d ($d > 1$).*

6.3. Grounded 2-range search. Before closing this section, we wish to show how filtering search leads to efficient algorithms for the *grounded 2-range search* problem. The problem is to preprocess a set S of n points $\{(x_1, y_1), \dots, (x_n, y_n)\}$, so that for any query triple (a, b, c) with $a \leq b$ and $0 \leq c$, it is possible to report all the points of $\{(x_i, y_i) | a \leq x_i \leq b \text{ and } y_i \leq c\}$ effectively.

Recall that this problem can be solved optimally using a priority search tree [31]. This structure is essentially a balanced binary tree with one point stored at each node. The tree is a heap with respect to the y -order, i.e. each node stores a point with smaller y -coordinate than all of its descendants. As is the case with most balanced tree-structures, although insertions and deletions can be performed in logarithmic time, overhead in implementation and running times occurs because rotations (or equivalent operations) are often necessary.

Filtering search can be used to alleviate the average cost of these operations. After sorting the n points in x -order, partition S into p blocks of roughly $\log_2 n$ points each, denoted B_1, \dots, B_p , from left to right. Each block B_j is arranged as a linked-list sorted in ascending y -order. With each B_j we associate the vertical slab $V_j = \{(x, y) | b_j \leq x \leq b_{j+1}\}$, where the b_j 's correspond to vertical lines separating the blocks. Let (α_j, β_j) be the point of B_j with smallest y -coordinate. We construct a priority search tree T with respect to the p points $(\alpha_1, \beta_1), \dots, (\alpha_p, \beta_p)$. Answering a query (a, b, c) involves:

1. finding the two slabs V_l, V_r ($l \leq r$) containing $(a, 0)$ and $(b, 0)$, respectively;
2. if $l+1 < r$, answering the query (b_{l+1}, b_r, c) , using T ;

3. for each point found in step 2, scanning the corresponding block upwards as long as the y -coordinate does not exceed c , and reporting all the points visited, including the starting ones;

4. checking all the points in B_l and B_r , and reporting those satisfying the query.

Note that, since the blocks B_i correspond to fixed slabs, their cardinality may vary from $\log n$ to 0 in the course of dynamic operations. We assume here that the candidate points are known in advance, so that slabs can be defined beforehand. This is the case if we use this data structure for solving, say, the all-rectangle-intersection problem (see [30] for the correspondence between the two problems). This new algorithm clearly has the same time and space complexity as the previous one, but it should require much fewer dynamic operations on the priority search tree, on the average. The idea is that to insert/delete a point, it suffices to locate its enclosing slab, and then find where it fits in the corresponding block. Since a block has at most $\log n$ points, this can be done by sequential search. Note that access to the priority search tree is necessary only when the point happens to be the lowest in its block, an event which we should expect to be reasonably rare.

6.4. Other optimal schemes. In light of our previous discussion on the hive-graph, it is fairly straightforward to devise another optimal algorithm for the grounded 2-range search problems. Simply extend a vertical ray upwards starting at each point of S . The problem is now reduced to intersecting a query segment, $[(a, c), (b, c)]$ with a set of nonintersecting rays; this can be solved optimally with the use of a hive-graph (§ 4).

Incidentally, it is interesting to see how, in the static case, filtering search allows us to reduce the space requirement of one of the earliest data structures for the ECDF (empirical cumulative distribution function) [7] and the grounded 2-range search problems, taking it from $O(n \log n)$ to $O(n)$. Coming back to the set of points $(\alpha_1, \beta_1), \dots, (\alpha_p, \beta_p)$, we construct a complete binary tree T , as follows: the root of the tree is associated with the y -sorted list of all the p points. Its left (resp. right) child is associated with the y -sorted list $\{(\alpha_1, \beta_1), \dots, (\alpha_{\lfloor p/2 \rfloor}, \beta_{\lfloor p/2 \rfloor})\}$ (resp. $\{(\alpha_{\lfloor p/2 \rfloor + 1}, \beta_{\lfloor p/2 \rfloor + 1}), \dots, (\alpha_p, \beta_p)\}$). The other lists are defined recursively in the same way, and hence so is the tree T . It is clear that searching in the tree for a and b induces a canonical partition of the segment $[a, b]$ into at most $2 \lfloor \log_2 p \rfloor$ parts, and for each of them all the points lower than c can be obtained in time proportional to the output size. This allows us to use the procedure outlined above, replacing step 2 by a call to the new structure. The latter takes $O(p \log p) = O(n)$ space, which gives us yet another optimal $(n, \log n)$ -algorithm for the grounded 2-range search problem.

7. Near-neighbor problems. We will give a final illustration of the effectiveness of filtering search by considering the k -nearest-neighbors and the circular range search problems. Voronoi diagrams are—at least in theory—prime candidates for solving neighbor problems. The Voronoi diagram of order k is especially tailored for the former problem, since it is defined as a partition of the plane into regions all of whose points have the same k nearest neighbors. More precisely, $\text{Vor}_k(S)$ is a set of regions R_i , each associated with a subset S_i of k points in S which are exactly the k closest neighbors of any point in R_i . It is easy to prove that $\text{Vor}_k(S)$ forms a straight-line subdivision of the plane, and D. T. Lee has shown how to compute this diagram effectively [25]. Unfortunately, his method requires the computation of all Voronoi diagrams of order $\leq k$, which entails a fairly expensive $O(k^2 n \log n)$ running time, as well as $O(k^2(n - k))$ storage. However, the rewards are worth considering, since $\text{Vor}_k(S)$ can be preprocessed to be searched in logarithmic time, using efficient planar point location methods [15], [18], [24], [28]. This means that, given any query point,

we can find the region R_i where it lies in $O(\log n)$ time, which immediately gives access to the list of its k neighbors. Note that the $O(k^2(n-k))$ storage mentioned above accounts for the list of neighbors in each region; the size of $\text{Vor}_k(S)$ alone is $O(k(n-k))$. A simple analysis shows that this technique will produce an $(n^4, \log n)$ -algorithm for the k -nearest-neighbors problem. Recall that this problem involves preprocessing a set S of n points in E^2 , so that for any query (q, k) ($q \in E^2, k \leq n$) the k points of S closest to q can be retrieved efficiently [11], [20], [25].

Using the basic principle of filtering search: “*be naive when k is large*”, we can save a factor of n space over this algorithm. The method consists of computing $\{\text{Vor}_{2^i}(S) | 0 \leq i \leq \lceil \log_2 n \rceil\}$ (let $\text{Vor}_j(S) = \text{Vor}_n(S)$, if $j \geq n$). This requires $S(n)$ storage, with

$$S(n) = O\left(\sum_{0 \leq i \leq \lceil \log n \rceil} 2^{2i}(n - 2^i)\right) = O(n^3).$$

All we then have to do is search the diagram $\text{Vor}_{2^j}(S)$, where $2^{j-1} < k \leq 2^j$, and apply a linear-time selection algorithm to retrieve from the corresponding set of points the k closest to the query point. Since 2^j is at most twice k , answering a query takes $O(k + \log n)$ time. We conclude to the existence of an $(n^3, \log n)$ -algorithm for solving the k -nearest-neighbors problem. This result matches the performance of the (very different) algorithm given in [20], which is based on a compact representation of Voronoi diagrams.

We must mention that a similar technique has been used by Bentley and Maurer [4] to solve the *circular range search* problem. This problem is to preprocess a set S of n points in E^2 , so that for any query disk q , the points of S that lie within q can be reported effectively. The approach taken in [4] involves searching $\text{Vor}_{2^i}(S)$ for $i = 0, 1, 2, \dots$, until we first encounter a point in S further than r away from the query point. At worst we will have checked twice as many points as needed, therefore the query time will be $O(k + \log k \log n)$, which as noticed in [4] is also $O(k + \log n \log \log n)$. The space complexity of the algorithm, stated to be $O(n^4)$ in [4], can be shown to be $O(n^3)$ using Lee's results on higher-order Voronoi diagrams [25]. For this problem, see also the elegant reduction to a three-dimensional problem used in the sublinear solution of [37].

If the radius of the query disk is fixed, we have the so-called *fixed-radius neighbor problem*. Chazelle and Edelsbrunner [12] have used filtering search to obtain an optimal $(n, \log n)$ -algorithm for this problem. As regards the general problems of computing k -nearest-neighbors and performing circular range search, Chazelle, Cole, Preparata and Yap [11] have recently succeeded in drastically lowering the space requirements for both problems. The method involves a more complex and intensive use of filtering search.

THEOREM 9. [11] *There exists an $(n(\log n \log \log n)^2, \log n)$ -algorithm for solving the k -nearest-neighbors problem as well as the circular range search problem. With probabilistic preprocessing, it is possible to obtain an $(n \log^2 n, \log n)$ -algorithm for both problems.*

8. Conclusions and further research. We have presented several applications of a new, general technique for dealing with retrieval problems that require exhaustive enumeration. This approach, called *filtering search*, represents a significant departure from previous methodology, as it is primarily based on the interaction between *search* and *report* complexities. Of course, the idea of balancing mixed complexity factors is not new *per se*. Our basic aim in this paper, however, has been to introduce the idea of balancing all the various complexity costs of a problem as a *systematic* tool for improving on known *reporting* algorithms. To implement this idea, we have introduced

the general scheme of *scooping* a superset of candidates and then *filtering out* the spurious items. To demonstrate the usefulness of filtering search, we have shown its aptitude at reducing the complexity of several algorithms. The following chart summarizes our main results; the left column indicates the best results previously known, and the right the new complexity achieved in this paper.

problem	previous complexity	filtering search
discrete interval overlap	$(n, \log n)$ [17, 30, 31]	$(n, 1)$
segment intersection	$(n^3, \log n)$ [19]	$(n^2, \log n)$
segment intersection ²	$(n \log n, \log n)$ [34]	$(n, \log n)$
point enclosure in $\mathbb{R}^d (d > 1)$	$(n \log^d n, \log^{d-1} n)$ [33]	$(n \log^{d-2} n, \log^{d-1} n)$
orthogonal range query in $\mathbb{R}^d (d > 1)$	$(n \log^{d-1} n, \log^{d-1} n)$ [35]	$\left(n \frac{\log^{d-1} n}{\log \log n}, \log^{d-1} n \right)$
k -nearest-neighbors	$(n^3, \log n)$ [20]	$(n(\log n \log \log n)^2, \log n)$ [11]
circular range query	$(n^3, \log n \log \log n)$ [4]	$(n(\log n \log \log n)^2, \log n)$ [11]

In addition to these applications, we believe that filtering search is general enough to serve as a stepping-stone for a host of other retrieval problems as well. Investigating and classifying problems that lend themselves to filtering search might lead to interesting new results. One subject which we have barely touched upon in this paper, and which definitely deserves attention for its practical relevance, concerns the dynamic treatment of retrieval problems. There have been a number of interesting advances in the area of dynamization lately [6], [32], [34], and investigating how these new techniques can take advantage of filtering search appears very useful. Also, the study of upper or lower bounds for orthogonal range search in two dimensions is important, yet still quite open. What are the conditions on storage under which logarithmic query time (or a polynomial thereof) is possible? We have shown in this paper how to achieve $o(n \log n)$. Can $O(n)$ be realized?

Aside from filtering search, this paper has introduced, in the concept of the *hive-graph*, a novel technique for batching together binary searches by propagating fractional samples of the data to neighboring structures. We refer the interested reader to [13] for further developments and applications of this technique.

Acknowledgments. I wish to thank Jon Bentley, Herbert Edelsbrunner, Janet Incerpi, and the anonymous referees for many valuable comments.

REFERENCES

- [1] Z. AVID AND E. SHAMIR, *A direct solution to range search and related problems for product regions*, Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, 1981, pp. 123–216.
- [2] J. L. BENTLEY, *Multidimensional divide-and-conquer*, Comm. ACM, 23 (1980), pp. 214–229.
- [3] J. L. BENTLEY AND J. H. FRIEDMAN, *Data structures for range searching*, Comput. Surveys, 11 (1979) 4, pp. 397–409.
- [4] J. L. BENTLEY AND H. A. MAURER, *A note on Euclidean near neighbor searching in the plane*, Inform. Proc. Lett., 8 (1979), pp. 133–136.
- [5] ———, *Efficient worst-case data structures for range searching*, Acta Inform., 13 (1980), pp. 155–168.

² With supporting line of query segment passing through a fixed point, or with fixed-slope query segment.

- [6] J. L. BENTLEY AND J. B. SAXE, *Decomposable searching problems. I. Static-to-dynamic transformation*, J. Algorithms, 1 (1980), pp. 301–358.
- [7] J. L. BENTLEY AND M. I. SHAMOS, *A problem in multivariate statistics: Algorithm, data structure and applications*, Proc. 15th Allerton Conference on Communication Control and Computing, 1977, pp. 193–201.
- [8] J. L. BENTLEY AND D. WOOD, *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., C-29 (1980), pp. 571–577.
- [9] A. BOLOUR, *Optimal retrieval algorithms for small region queries*, this Journal, 10 (1981), pp. 721–741.
- [10] B. CHAZELLE, *An improved algorithm for the fixed-radius neighbor problem*, IPL 16(4) (1983), pp. 193–198.
- [11] B. CHAZELLE, R. COLE, F. P. PREPARATA AND C. K. YAP, *New upper bounds for neighbor searching*, Tech. Rep. CS-84-11, Brown Univ., Providence, RI, 1984.
- [12] B. CHAZELLE AND H. EDELSBRUNNER, *Optimal solutions for a class of point retrieval problems*, J. Symb. Comput., 1 (1985), to appear, also in Proc. 12th ICALP, 1985.
- [13] B. CHAZELLE AND L. J. GUIBAS, *Fractional cascading: a data structuring technique with geometric applications*, Proc. 12th ICALP, 1985.
- [14] B. CHAZELLE, L. J. GUIBAS AND D. T. LEE, *The power of geometric duality*, Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Nov. 1983, pp. 217–225.
- [15] R. COLE, *Searching and storing similar lists*, Tech. Rep. No. 88, Computer Science Dept., New York Univ., New York, Oct. 1983.
- [16] R. COLE AND C. K. YAP, *Geometric retrieval problems*, Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Nov. 1983, pp. 112–121.
- [17] H. EDELSBRUNNER, *A time- and space- optimal solution for the planar all-intersecting-rectangles problem*, Tech. Rep. 50, IIG Technische Univ. Graz, April 1980.
- [18] H. EDELSBRUNNER, L. GUIBAS AND J. STOLFI, *Optimal point location in a monotone subdivision*, this Journal, 15 (1986), pp. 317–340.
- [19] H. EDELSBRUNNER, D. G. KIRKPATRICK AND H. A. MAURER, *Polygonal intersection searching*, IPL, 14 (1982), pp. 74–79.
- [20] H. EDELSBRUNNER, J. O'ROURKE AND R. SEIDEL, *Constructing arrangements of lines and hyperplanes with applications*, Proc. 24th Annual IEEE Symposium on Foundation of Computer Science, Nov. 1983, pp. 83–91.
- [21] R. A. FINKEL AND J. L. BENTLEY, *Quad-trees: a data structure for retrieval on composite keys*, Acta Informat. (1974), pp. 1–9.
- [22] M. L. FREDMAN, *A lower bound on the complexity of orthogonal range queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 696–705.
- [23] ———, *Lower bounds on the complexity of some optimal data structures*, this Journal, 10 (1981), pp. 1–10.
- [24] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, this Journal, 12 (1983), pp. 28–35.
- [25] D. Y. LEE, *On k -nearest neighbor Voronoi diagrams in the plane*, IEEE Trans. Comp., C-31 (1982), pp. 478–487.
- [26] D. T. LEE AND C. K. WONG, *Quintary trees: A file structure for multidimensional data base systems*, ACM Trans. Database Syst., 1 (1980), pp. 339–353.
- [27] W. LIPSKI AND F. P. PREPARATA, *Segments, rectangles, contours*, J. Algorithms, 2 (1981), pp. 63–76.
- [28] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, this Journal, 9 (1980), pp. 615–627.
- [29] G. LUEKER, *A data structure for orthogonal range queries*, Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 28–34.
- [30] E. M. MCCREIGHT, *Efficient algorithms for enumerating intersecting intervals and rectangles*, Tech. Rep. Xerox PARC, CSL-80-9, 1980.
- [31] ———, *Priority search trees*, Tech. Rep. Xerox PARC, CSL-81-5, 1981; this Journal 14 (1985), pp. 257–276.
- [32] M. H. OVERMARS, *The design of dynamic data structures*, Ph.D. thesis, Univ. Utrecht, 1983.
- [33] V. K. VAISHNAVI, *Computing point enclosures*, IEEE Trans. Comput., C-31 (1982), pp. 22–29.
- [34] V. K. VAISHNAVI AND D. WOOD, *Rectilinear line segment intersection, layered segment trees and dynamization*, J. Algorithms, 3 (1982), pp. 160–176.
- [35] D. E. WILLARD, *New data structures for orthogonal range queries*, this Journal, 14 (1985), pp. 232–253.
- [36] A. C. YAO, *Space-time trade-off for answering range queries*, Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 128–136.
- [37] F. F. YAO, *A 3-space partition and its applications*, Proc. 15th Annual ACM Symposium on Theory of Computing, April 1983, pp. 258–263.