Fractional Cascading: II.
Applications

Bernard Chazelle and Leonidas J. Guibas

Abstract. This paper presents several applications of fractional cascading, a new searching technique which has been described in a companion paper. The applications center around a variety of geometric query problems. Examples include intersecting a polygonal path with a line, sorted range search, orthogonal range search, computing locus functions, and others. Some results on the optimality of fractional cascading, and certain extensions of the technique for retrieving additional information are also included.

Keywords. Fractional cascading, iterative search, Multiple look-up, Binary search, B-tree, Iterative search, Multiple look-up, Range query, Densification of data structures

1. Introduction. As we saw in Part I, fractional cascading is an algorithmic technique for searching several sets at once. This generalized form of searching often arises in the solution of query problems. Imagine that you come upon a wump of unknown origin, which you wish to identify. One solution is to look up the word in as many dictionaries as it will take to find it. Fractional cascading gives you a way out of this repetitive search. It offers you the following alternative: look up the word in one dictionary, and from there on jump directly into each of the other dictionaries in constant time. To make this happen, the dictionaries will have to be somehow reorganized, and linked together by some appropriate mechanism. We showed in Part I that all this rearrangement can be done at fairly little cost.

The goal of this second part is to present a number of problems whose solutions can be significantly improved by using fractional cascading. Most of the algorithms presented are short and simple. We believe that fractional cascading is a speed-up mechanism of practical as well as theoretical relevance. One goal of this paper will be to justify the former part of this belief. From now on, we will assume that the reader is familiar with the basic terminology of fractional

1 The first author was supported in part by NSF grants MCS 83-03923 and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786. Part of this work was done while the second author was employed by the Xerox Palo Alto Research Center.

2 Brown University and Ecole Normale Superieure.

3 DEC/ SRC and Stanford University.

4 Contact author's address: Leonidas J. Guibas, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA.

5 Received August 30, 1985; revised February 18, 1986. Communicated by C. K. Wong.
Fractional Cascading. Let $G$ be a catalog graph of size $s$ and locally bounded degree $d$. In $O(s)$ space and time, it is possible to construct a data structure for solving the iterative search problem. The structure allows multiple look-ups along a generalized path of length $p$ to be executed in time $O(p \log d + s \log s)$. If $d$ is a constant, this is optimal. The data structure is dynamic in the following sense. If only insertions are performed, the amortized time for each insertion will be $O(s \log d)$; the same holds for deletions. Arbitrary insertions and deletions can also be done in $O(s \log s)$ amortized time, but the query time becomes $O(p \log d + s \log s)$. How is this paper organized and what will we find in it? The applications we consider in this part all revolve around the notion of a query problem. In each case, one must design a database to answer efficiently certain types of queries relative to some given objects. This will lead us to examine problems of interecting a tree with a fixed polygondal path (Section 5), repeatedly partitioning within a transposed region (Section 4), or a hyperrectangle (Section 3), constructing segment trees (Section 8), and extending query problems (Section 9). The reader puzzled by these rather vague descriptions can skip to the appropriate sections for clarification. On the last application, however, we wish to say a little more at this point. It concerns a fairly general principle which best illustrates the power of fractional cascading. In a standard query problem, call it (T), a query specifies a certain subset of the given objects, and the goal is to compute this subset as fast and economically as possible. Often, however, the objects themselves are pointers to files which, once identified, must then be searched in a later stage. We call the resulting problem an E-extension of (T) (Recursive search extension). What we will show in Section 9 is that with the use of fractional cascading, almost any solution to a query problem can be transformed into a solution to its E-extensions with little or no degradation of performance. This touches on a central aspect of fractional cascading: its use as a post-optimizing device. Most often, fractional cascading is applied to a data structure at the very end stage of its development. What is remarkable is that its applicability depends on structural rather than amicable characteristics of the data structure. To have the basic appearance of a catalog graph is what really matters, and not so much the particular mathematical domain within which the data structure's semantics is defined. This feature grants fractional cascading great versatility.

The notion of iterative search comes in two flavors. It is called explicit if the problem to be solved makes explicit reference to a collection of catalogs. Queries are specified by a subset of this collection along with a search key. In the applications we just mentioned, however, iterative search is implicit. That is to say, the problems do not make mention of it in their statements; they don't even allude to it. It is only in the specific solutions chosen that iterative search shows its face. For practical reasons, implicit iterative search is what justifies the use of fractional cascading. We may still legitimately ask ourselves: how well under-
multiple look-ups, etc. For con-
venience, we assume that $s$ and $\log_2 s$ are integers and $s$ is locally bounded.

The graph of size $s$ is locally bounded if it satisfies the following conditions necessary for fractional cascading (Section 7). A vertex is dynamic in the following sense: if an insertion or deletion occurs in the graph, then the query time becomes

the query time becomes

find it? The applications we consider are of a query problem. We are given a set of $n$ objects, each with a unique identifier, and we are asked to retrieve a subset of the objects based on a given query. The problem is of particular interest when the objects have dynamic properties, such as insertions and deletions.

We examine several types of queries, including

1. **Select**: Given a query $q$, find all objects $o$ such that $f(o) = q$.
2. **Insert**: Given an object $o$ and a query $q$, insert $o$ into the database if $f(o) = q$.
3. **Delete**: Given an object $o$, delete $o$ from the database.
4. **Update**: Given an object $o$ and a new query $q$, update the query of $o$ to $q$.

The query time for each query is $O(\log n + \log s)$, where $s$ is the size of the database and $n$ is the number of objects in the database.

**2. Explicit Iterative Search.** Let $S = \{C_1, \ldots, C_p\}$ be a collection of $p$ catalogs, and let $s = \sum_{i=1}^{p} |C_i|$ be the combined size of the catalogs. An explicit iterative search is the following problem: given a query of the form $(q, H)$, where $q$ is a real number and $H$ is a subset of $\{1, \ldots, p\}$, compute the successor of $q$ in $C_i$ for each $i \in H$ (recall that the successor of $q$ in $C_i$ is the smallest element in $C_i$ larger than or equal to $q$). We solve this problem by setting up the conditions necessary for fractional cascading. Let $G$ be a complete binary tree with $n$ nodes, each associated with a distinct catalog $C_i$. We call $G$ an *emulation graph* of $S$ for convenience, since we refer to the elements of $H$ as nodes of $G$.

The idea is to apply fractional cascading to the emulation graph and answer the query by traversing the minimum spanning tree $T$ of $H$ (Figure 1). Each node of $G$ will have a flag for marking purposes. We compute $T$ by iterating on the following process. Initially, all nodes of $G$ are unmarked. For each node $x$ of $H$, traverse the path from $x$ to the root, marking each node along the way, and stopping as soon as a node already marked is encountered. At the end of this process, the set of marked nodes forms a spanning tree of $H$. It is not necessary minimum since it always contains the root. We must now remove the branch joining the root of $G$ to the lowest common ancestor of the nodes of $H$.

Let $r$ be the root of $G$; if it is not a node of $H$ and has a single marked child $x$, then unmark $x$ and iterate with respect to $x$ itself. Stop. With $T$ in hand, we answer the query by performing multiple look-ups in the catalogs attached to the nodes of $T$.

A simple analysis shows that the time taken by the construction of $T$ as well as the search in each catalog is $O(T + \log n)$. Let $v_1, \ldots, v_m$ be the vertices of $H$ sorted by increasing in-order ranks. Let $l_i$ be the lowest common ancestor of $v_i$ and $v_{i+1}$, and let $h_i$ denote the number of ancestors of $l_i$ in $G$. A rough analysis shows that $|T| \leq \sum_{i=1}^{m} (\log p - h_i)$.

Since fewer than $2^k$ vertices among the $l_i$'s can have fewer than $j$ ancestors, we have

$$\sum_{i=0}^{h_i-1} (\log p - h_i) \leq \sum_{i=0}^{h_i-1} 2^i (\log p - j).$$

**Figure 1.** The emulation graph.
and therefore $|T| = O(m + m \log (p/m))$. We conclude that the running time of the algorithm is $O(|H| \log p/|H| + \log n)$. The next question to decide is whether this result is optimal. After all, fractional cascading allows us to use any graph of bounded degree as a supporting search structure, so one might wonder whether a fanucher catalog graph with, say, cycles to provide shortcuts can yield a better performance. We show that this is not the case.

**Lemma 1.** Among all emulation graphs of $S$ of bounded degree, the complete binary tree on $p$ nodes is asymptotically optimal.

**Proof.** Let $G$ be an emulation graph of degree $\leq d$, and let $H$ be a subset of $m$ vertices carefully chosen so as to make the minimum spanning tree $T$ of $H$ as large as possible. Let the distance between two vertices $u$ and $v$ be defined as the number of edges on the shortest path between $u$ and $v$. Let $l = \lceil \log_2 (p/m) \rceil - 1$. Pick a vertex $v$ in $H$ and mark off all vertices at a distance less than or equal to $l$ from $v$ (this includes $v$). Next, pick a non-marked vertex and iterate on this process until all vertices are marked. Since $G$ has bounded degree $d$, each iteration will mark at most $d^{l+1}$ vertices, therefore at least $m$ vertices will be picked in the process. Let $H = \{v_1, \ldots, v_n\}$ be the chosen vertices and let $T$ be any spanning tree of $H$. For each $v_i$, there must exist at least one vertex $w_i$ in $T$ at a distance $\lceil l/2 \rceil$ from $v_i$. Let $p_i$ be the path in $T$ between $v_i$ and $w_i$. By construction of $H$, the paths $p_1, \ldots, p_n$ are vertex-disjoint, therefore the size of $T$ is at least the added size of the $p_i$, that is, $\Omega(|H| \log p/|H|))$.

Lemma 1 shows that any choice of $G$ is adequate, but it still falls short of proving the optimality of the technique. Why can't a different method be used that perhaps bears no relation to fractional cascading? What we will show is that no improvement can be expected in a pointer machine model $\mathcal{T}$, if $H$ is given as a set of indices and not as a set of addresses. Why is that so? Let's ask ourselves: how many different collections of dictionaries can be identified by taking $t$ steps on a pointer machine? A single step gives a choice of at most $c$ memory acrosses, for some machine-dependent constant $c$. Therefore "at most $c^t$ collections" is the answer. But there are $|\mathcal{T}|$ possible sets $S$, so $t$ must be at least the order of $\log(|\mathcal{T}|)$. As long as $m = o(p^{1/4})$, we have the elementary asymptotic formula

$$\frac{p}{{m}} \approx \frac{p^m e^{-m(1-1/2m)}}{m!} (1 + o(1)).$$

Using Stirling's approximation

$$m! = m^m e^{-m} \sqrt{2\pi m} (1 + o(1)),$$

we find that $t$ must be at least on the order of

$$\frac{1}{\log(c)} m \log_2 \frac{p}{m} + m \left(1 - \frac{1}{2p^{1/4}} \frac{1}{6p^{1/2}} \right).$$

3. Intersection of a Polygonal Path

The following problem: we are given into a data structure so that, given intersections of $P$ and $L$. The checks each side of $P$ for in $S = \Omega(n)$, where $n$ is the length of a method with a query time of $k$ is the number of intersections. To develop a technique that gives a small constant $t$ the running time, the running time is $O(k)$, and the expression of the query insures the cost of a binary search in the searching strategy. Our solution:

The storage requirement of the polygonal path is simply, that of an interesting phenomenon of a polygon reduces some region $\log n$. Computing the convex hull of the technique we propose in the following observation:

**Lemma 2.** A straight-line $l$ intersects the convex hull $CH(S)$.
that is, \(O(m \log (p/m))\). This shows that, at least for \(m = o(p^{2/3})\), our algorithm is optimal. Keep in mind, however, that this argument assumes that the catalogs are referred to by indices and not by addresses.

**Theorem 1.** Let \(I\) be an explicit iterative search problem involving \(p\) catalogs of combined size \(s\). There exists a data structure for solving \(I\) such that any query can be answered in \(O(m \log (p/m) + \log s)\) time, where \(m\) is the number of catalogs involved in the query. The data structure requires \(O(s)\) space and can be constructed in \(O(s)\) time. Within the context of fractional cascading, this result is optimal.

The naïve method requires \(O(m \log s)\) response time, so the scheme of Theorem 1 is superior whenever the size of the catalogs exceeds their number \((s > p)\), a situation of great likelihood in practice. The solution is optimal when the number of catalogs queried is at least a fixed fraction of the total number of catalogs \((p = O(m))\).

We now turn our attention to implicit iterative search. Ironically, the problems for which fractional cascading seems the best suited do not even suggest the notion of iterative search in their statements. Their solutions, however, are inherently dependent on iterative search. This situation occurs in many query problems, as we will see.

3. **Intersecting a Polygonal Path with a Line.** In this section we investigate the following problem: we are given a polygonal path \(P\) and wish to preprocess it into a data structure so that, given any query line \(l\), we can quickly report all the intersections of \(P\) and \(l\). The obvious method for solving this problem simply checks each side of \(P\) for intersection with \(l\). This method requires storage \(S = O(n)\), where \(n\) is the length of \(P\), and has query time \(C = O(n)\). We desire a method with a query time of the form \(Q = O(f(n) + k)\), where \(f(n) = \omega(n)\) and \(k\) is the number of intersections reported. Using fractional cascading we are able to develop a technique that gives \(Q = O(k + \log s/(k + 1))\). When \(k\) is a small constant the running time is \(O(\log s)\), which is optimal. When \(k = O(\log s)\) the running time is \(O(k)\), and this is also optimal. For intermediate values of \(k\), the expression of the query time suggests that the discovery of each intersection incurs the cost of a binary search. This is actually a fairly accurate reflection of the searching strategy. Our solution represents partial progress towards the desired goal.

The storage requirement of the method is \(O(n \log s)\), but in the case where the polygonal path is simple, it can be reduced to \(O(s)\). This is another instance of an interesting phenomenon in computational geometry, where the simplicity of a polygon reduces some required resource for an algorithm by a factor of \(\log s\). Computing the convex hull is another well-known example.

The technique we propose in this section is based on the recursive application of the following observation:

**Lemma 2.** A straight-line \(l\) intersects a polygonal line path \(P\) if and only if \(l\) intersects the convex hull \(CH(P)\) of \(P\).
PROOF. Obvious.

Let $F(P)$ and $S(P)$ denote respectively the first and second halves of the path $P$, that is, the subpaths of $P$ consisting of the first $\lfloor n/2 \rfloor$ and second $\lfloor n/2 \rfloor$ edges. Then our algorithm is expressed very simply recursively as:

```
Intersect(P, I)
   begin
      if |P| = 1 [single edge] then
         compute $\cap \cap I$ directly
      else if I does not intersect $CH(P)$ then exit
      else
         Intersect(F(P), I)
         Intersect(S(P), I)
   end
```

Since we are allowed to preprocess $P$, it is to our advantage to precompute and store all the convex hulls we may need. We can do this by a recursion similar to that above, where after obtaining $CH(F(P))$ and $CH(S(P))$, we compute $CH(P)$ by any one of a number of linear-time algorithms for computing the convex hull of two convex polygons [PH]. The overall data structure that we thus build is best thought of as a binary tree $T$ whose $n$ leaves are the edges of our path $P$ (which coincide with their own convex hulls and which lie to the right in the same order as in $P$). Interior nodes of the tree correspond to an obvious way to subpaths of $P$ and store the convex hull of their respective subpath (Figure 2).

![Fig. 2. The convex hull decomposition.](image)

The tree $T$ of convex hulls occupies time for computing it is also $O(n)$. The convex hull $T(n)$ satisfies a recurrence of the form

$$T(n) = T(n-1) + O(n)$$

since it costs $O(\log m)$ to test sides and a line, and in $T$ we may store only one intersection, the total

$$O(n \log n)$$

We may now look more closely at the algorithm. We decide whether to move between the convex hull stored in the left or in the right. This is already too expensive into the battle. This is where the trick lies.

The underlying tree $T$ is a perfect binary tree. How are we to view the "two-sides" problem as one of a look-up table by a simple observation. Let $C_0 \ldots C_n$ be the convex hulls of $P$ in clockwise order, and let $O_{C_0} = 1$. Define the slope of $C_0$, $O_{C_0}$ for the sequence $C_0$. It is well known that since $C_i$ is a polygon, the sequence of edges of $C$ is such that the sequence $O_{C_0} \ldots O_{C_n}$ is unique, and is called the slope-sequence.

**Lemma 3.** Let $s$ and $t$ be the slope-sequence of convex $C$. We can store $C$ in $O(n)$ space.

**Proof.** Effect the position $p$ in $C$ where the tangent parallel to $p$ lies between these two tangents.

Thus we view each node $x$ of the slope-sequence of the convex $C$ as having a fractional cascad $x$ still only requiring space of the convex polygon, except for the size of the root, any time we need to decide which subtree to visit, there is, of course, an $O(\log)$ started.
The tree $T$ of convex hulls clearly takes $O(r \log n)$ space to store. The total time for computing it is also $O(n \log n)$ since, by the discussion above, this time satisfies a recurrence of the form

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + O(n).$$

We must now look more closely at the implementation of our intersection algorithm. We decide whether to descend into a subtree by testing for intersections between the convex hull stored in its root and the line $l$. Even if we were to report only one intersection, the total cost of all these tests would be

$$O\left(\sum \log \frac{n}{2}\right) = O(n \log n),$$

since it costs $O(\log m)$ to test for intersection between a convex polygon of $m$ sides and a line, and in $T$ we must trace at least one path down to the intersected edge. This is already too expensive, so some additional weapon must be brought into the battle. This is where fractional cascading comes in.

The underlying tree $T$ is a perfect binary good graph of bounded degree. However, how are we to view the "two-dimensional" (convex polygon, line) intersection problem as one of a look-up in a one-dimensional catalog? The answer is given by a simple observation. Let $c_1, \ldots, c_n$ be the vertices of a convex polygon given in clockwise order, and let $l_x$ be the horizontal ray emanating from $c_0$ towards $x = \infty$. We define the slope of an edge $c_i c_{i+1}$ as the angle $\angle(c_0, c_i c_{i+1}) \in [0, 2\pi)$. It is well known that since $C$ is convex there exists a circular permutation of the slopes of $C$ such that the sequence of slopes is nondecreasing. This sequence is unique, and is called the slope-sequence of $C$.

**Lemma 3.** Let $s$ and $s'$ be the two slopes of $l$ obtained by giving the line its two possible orientations; if we know the positions of $s$ and $s'$ within the slope-sequence of a convex polygon $C$, we can determine whether $C$ and $l$ intersect in constant time.

**Proof.** In effect the position of the slope-sequence tells us the vertices of $C$ where the tangents parallel to $l$ occur. The line $l$ will intersect $C$ if and only if it lies between these two tangents.

Thus we may view each node $x$ of $T$ as containing a catalog consisting of the slope-sequence of the convex polygon associated with $x$. To these catalogs over $T$ we apply fractional cascading. The result is a more elaborate structure, but one still only requiring space $O(n \log n)$. The data structure allows us to implement all the (convex polygon, line) intersection tests required by our algorithm, except for the one at the root, in constant time per test. By the previous lemma, any time we need to decide whether to descend into a subtree, we can look up the slopes of $l$ in that subtree's root catalog and find the answer in constant time.

There is, of course, an $O(\log n)$ cost at the root of $T$ to get the whole process started.
As a net result, the cost of our intersection algorithm is now reduced to \(O(n \log n + \text{size of subset of } T \text{ actually visited})\), since, once we pass the root, we spend only constant effort per node visited. Our claimed query time bound of \(O((n+1) \log n/(k+1))\) now follows from the next lemma.

**Lemma 4.** Let \( T \) be a perfectly balanced tree on \( n \) leaves and consider any subtree \( S \) of \( T \) with \( k \) leaves chosen among the leaves of \( T \). Then,

\[
|S| \leq k \log n - k \log k + 2k - 1.
\]

**Proof.** In \( S \) there are \( k \) leaves and \( k - 1 \) branching nodes (outdegree 2). The size of \( S \) is maximized when all the branching nodes occur as high in \( T \) as possible. Then the number of remaining non-branching nodes in \( S \) is at most \( k \log n \) - \( k \log k \).

We have finally shown,

**Theorem 2.** Given a polygonal path \( P \) of length \( n \), it is possible in time \( O(n \log n) \) to build a data structure of size \( O(n \log n) \), so that given any line \( l \), if \( l \) intersects \( P \) in \( k \) edges, then these edges can be found and reported in time \( O((k+1) \log n/(k+1)) \).

We next show how the storage used can be reduced to \( O(n) \) when \( P \) is known to be simple (i.e., non-self-intersecting). The key lemma is

**Lemma 5.** If \( P \) is simple, then \( CH(F(P)) \) and \( CH(S(P)) \) have at most two common tangents (Figure 3).

**Proof.** Consider \( CH(F(P)) \); the interior of this polygon is partitioned by the simple path \( F \) into a number of simply connected regions \( CH(F(P)) = \{L_i\}, R_i \). The regions \( R_i \) are in one-to-one correspondence with the edges of \( CH(P) \) that are not edges of \( P \), except for possibly the interior of \( P \), if \( P \) is closed. To see this, note that for any point in \( CH(F(P)) \) (except for points inside \( P \), if \( P \) is closed) there is a path to infinity that avoids \( P \). Thus, regions containing such points must have on their boundary edges of \( CH(P) \) that are not part of \( P \). Furthermore, a particular region \( R \) can never have more than one such edge on its boundary because \( F \) is connected.

Let us now examine the remaining boundary edges of this region \( R \). Naturally, they are all edges of \( P \). Since they form a connected set, they must form a subpath of \( P \). The order of the edges along the subpath corresponds to the order of the same edges around \( R \), with one exception. That arises when the initial or final vertex of \( P \) is interior to \( P \), in which case (i) an initial segment of \( P \) respectively may occur on the boundary of \( R \). (ii) twice.

Now let \( e \) be the midpoint of \( P \) that is the vertex separating \( P \) from \( S(P) \). If an edge \( e \) of \( CH(P) \) is a common tangent of \( CH(F(P)) \) and \( CH(S(P)) \), then \( e \) cannot be an edge of \( P \). The boundary of the region \( R \) of \( CH(P) \) bounded by
is now reduced to $O(n \log n)$, once we pass the root, we assumed query time bound of $O(n)$.

Hence and consider any subtree there.

$\text{Fig. 3. Storing common tangents.}$

is a subtree of $P$, in a subpath of $P$ joining $F(P)$ to $S(P)$. Therefore $x$ is on the boundary of $R$. Since $x$ can be on the boundary of at most two regions, there can be at most two common tangents. Note that the regions on either side of $x$ can be the same region $R$. In that case the edge $x$ of $CH(F(P))$ associated with $R$ is an edge of either $CH(F(P))$ or $CH(S(P))$, since $x$ is encountered twice when walking along the boundary of $R$. This implies that there are no common tangents of $CH(F(P))$ and $CH(S(P))$. One is fully enclosed in the other. Note also that one of the common tangents can be degenerate. In case $x$ is on $CH(P)$.}

Since $CH(P)$ can be obtained from $CH(F(P))$ and $CH(S(P))$ by drawing the two common tangents if any exist and then throwing away the interior segments of the convex hulls of the parts, it follows that the total number of distinct edges used by all the convex hulls of $T$ is at most $n + 2(n - 1) + 3a - 2$. Therefore an algorithm with $O(n)$ storage may be feasible. Of course, a particular edge $e$ may appear in many convex hulls. If we are to store it only once, where should we store it? The answer is: "at the highest node of $T$ whose associated hull contains $e". It is easy to check that this node is well-defined. A similar trick has been used by Lef and Preparata for edges that appear on many separators in their classic point location paper (LP). Thus, at each node of $T$, only a certain subset of the edges of its convex hull is stored, namely those that do not appear in hulls higher up in the tree. This particular choice has a fortunate consequence.

**Lemma 6.** If we store each edge in the highest node in $T$ in whose convex hull it appears, then all the edges stored at a particular node form a contiguous interval of the edges of its convex hull.

**Proof.** The edges stored with a node $v$ of $T$ are exactly those which are not also edges of the parent of $v$ is $t$. By the previous lemma, $v$ and its brother have common hulls with at most two common tangents. The assertion follows.
Thus we can view the stored edges at each node as a catalog of slopes, and apply fractional cascading: the lemmas above imply that if the slope of a line \( l \) we are looking up falls outside of the stored catalog of a node \( v \), then the answer we want is the same as what we get for the parent of \( v \). Again, we can in constant time per node locate the two tangents of the convex hull associated with the node and parallel to \( l \) (root excepted). So we have shown:

**Theorem 3.** Given a simple polygon path \( P \) of length \( n \), it is possible in time \( O(n \log n) \) to build a data structure of size \( O(n) \), so that given any line \( l \) that intersects \( P \) in \( k \) edges, then these edges can be found and reported in time \( O((k+1)\log n/(k+1)) \).

---

4. Slanted Range Search. Let \( R \) be the Euclidean plane endowed with a Cartesian system of axes \((O_x, O_y)\). We will use the term **aligned rectangle** to refer to the Cartesian product \([a, b] \times [c, d]\), for some positive reals \(a, b, c, d\). The **aligned range search problem** involves preprocessing a set \( V \) of \( n \) points so that for any aligned rectangle \( R \), the set \( V \cap R \) can be computed efficiently. McCreight \([16]\) has described a data structure, called a **priority search tree**, which allows us to solve this problem in optimal space and time. The data structure requires \( O(n) \) space and offers \( O(k \log n) \) response time, where \( k = |V \cap R| \) is the size of the output. Can the priority search tree be extended to solve a more general class of range search problems? For example, consider adding one degree of freedom to the previous problem. We define an **aligned trapezoid** as a trapezoid with corners \((a, 0), (b, 0)\) and \((a, c), (b, d)\), with \( a < b, c > 0, \) and \( d > 0 \). In the **aligned range search problem**, the set to be computed is of the form \( V \cap R \), where \( R \) is an aligned trapezoid. Figure 4 illustrates the difference between the two problems. Note that slanted range search is strictly more general than aligned range search. Informally, the "root" of the range is now of arbitrary slope. For this reason the priority search tree is inadequate. Instead, we turn to a slightly more complicated data structure, which we develop in two stages. First, we outline a data structure of linear size. Its response time number of points to be reported application of fractional cascading:

A special case of slanted set \( CGL \) - given a query line \( l \) which is optimal in both space convex layer, a structure for convex hull of \( V \). This preprocessor, each of which lends slanted range search, we add a.

To begin with, observe that \( \ast \) points in \( V \) have distinct \( x \)-coordinates and \( y \)-coordinates. In this way, we each time a point is reported, a point falling outside of the subtable of negative \( y \)-coordinates \( \Delta \) of lower hull of the point set \( \Gamma \) vertices of the convex hull of \( a \) the point with minimum \( \Gamma \) (the sequence of points \( a_1, \ldots, a_n \).

We are now ready to describe by associating the list \( D(s) = \{ \) the points of \( V \) not in \( V \} \). The right children of node \( a \). The depth of the sequence of points \( V \) structure \( D(a) \) is defined as \( \psi \) (Figure 5). The recursion stop trivially bounded above by \( n \).

![Aligned range search](https://example.com/aligned-range-search.png)

![Slanted range search](https://example.com/slanted-range-search.png)

**Fig. 4. Two cases of range searching.**
node as a catalog of slopes, and
implies that a slope of a line
catalog of a node n, then the
parent of n. Again, we can
of the convex hull associated
we have shown.
length n, it is possible in time
v, so that given any line L \not \parallel
be found and reported in time
plane endowed with a
term \textit{aligned rectangle} to refer
positive reals a,b,c. The \textit{aligned}
set V of \( n \) points so that for any
rectangle efficiently. McCreight [M2]
search tree, which allows us to
the data structure requires \( O(n) \)
\( k = |V \cap R| \) is the size of the
solve a more general class of
adding one degree of freedom to
and a trapezoid with corners
and \( d > 0 \). In the \textit{slanted range}
the set \( v \) form \( V \cap R \), where \( R \) is as
not between the two problems.
less than aligned range search.
slope. For this reason the
a slightly more complicated
we outline a data structure

A special case of slanted search has been solved by Chazelle, Guibas and Lee [CGL]: given a query line \( L \), report all points of \( V \) on one side of \( L \). The algorithm, which is optimal in both space and time, is intimately based on the notion of \textit{convex layers}, a structure obtained by repeatedly computing and removing the convex hull of \( V \). This preprocessing partitions the point set into a hierarchy of subsets, each of which lends itself to efficient searching. For the purpose of slanted range search, we add a recursive component to the construction of layers.

To begin with, observe that without loss of generality we can assume that all points in \( V \) have distinct \( x \)-coordinates. If this is not the case, we sort each group of points with the same \( x \)-coordinates in a linked list sorted by increasing \( y \)-coordinates. In this way, we may ignore every point that is not first in its list. Each time a point is reported, the corresponding list is scanned until we run into a point falling outside of the range. Of course, we can assume that all points with negative \( y \)-coordinates have been removed. Next, we introduce the notion of \textit{lower half} \( L(V) \) of the point set \( V \), denoted \( L(V) \). If \( a_1 \), \( a_2 \), \( a_3 \), \ldots, \( a_k \) are the
the vertices of the convex hull of \( V \), given in counterclockwise order with \( a_1 \) (resp. \( a_k \)) the point with minimum (resp. maximum) \( x \)-coordinate, \( L(V) \) is defined as the sequence of points \( a_1 \), \ldots, \( a_k \). If \( V \) consists of a single point, then \( L(V) = V \).

We are now ready to describe the data structure. It is constructed recursively by associating the list \( D(v) = L(V) \) with the \textit{root} of a binary tree \( G \). Let \( W \) be the points of \( V \) not in \( L(V) \), and let \( l \) and \( r \) denote respectively the left and right children of node \( n \). The data structure \( D(n) \), associated with \( n \), is defined as the sequence of points \( l(W) \), where \( W \) is the leftmost half of \( W \). A data structure \( D(n) \) is defined similarly with respect to the rightmost half of \( W \) (Figure 5). The recursion stops as soon as \( W \) is empty, so \( G \) is finite; in size is

Fig. 5. A tree of convex hulls.

Each data structure $D(v)$ is now refined as follows: let $D(v) = ((x_1, y_1), \ldots, (x_n, y_n))$ be the lower hull at node $v$, with $x_1 < x_2 < \cdots < x_n$. The two pieces of information of interest at node $v$ are:

1. $\text{Abs}(v) = (x_1, \ldots, x_n)$, the sorted list of $x$-coordinates in $D(v)$,
2. $\text{Slope}(v) = ((y_2, -y_1)/(x_2 - x_1), \ldots, (y_n - y_{n-1})/(x_n - x_{n-1}))$, the sorted list of edge-slopes in $D(v)$.

For explanatory purposes, we describe the query-processing answer in two-stages.

A preliminary phase marks selected vertices of $G$ using two colors, blue and red. The red vertices are then used as starting points for the second stage of the algorithm, where the remaining candidate vertices are examined. We successively describe the algorithm, prove its correctness, and examine its complexity. As a convenient piece of terminology, we introduce the notion of an $L$-peak. Let $L$ be the line passing through the two points $(a, c)$ and $(b, d)$, and let $L$ be the half-plane below $L$. We define the $L$-peak of $D(v)$ as the point of $L \cap D(v)$ whose orthogonal distance to $L$ is maximum (break ties arbitrarily). The $L$-peak of $D(v)$ is $0$ if $L \cap D(v) = \emptyset$.

Stage 1. The algorithm is recursive and starts at the root $v$ of $G$. In the following, $D(v)$ is regarded as the polygonal line with vertices $(x_1, y_1), \ldots, (x_n, y_n)$. The query taxa $(x, y) \in R$ is $(a, 0), (b, 0), (a, v), (b, v)$ falls in one of three positions with respect to $D(v)$:

1. $D(v)$ intersects the vertical segment $r_v = [(a, y), (b, y)]$ at some edge $[(x_1, y_1), (x_{v+1}, y_{v+1})]$: as long as the point $(x_v, y_v)$ is defined and lies in $R$, report it and increment $v$ by one. If $D(v)$ does not intersect $r_v$ but intersects $r_v$ at some edge $[(x_v, y_v), (x_{v+1}, y_{v+1})]$, then perform a similar sequence of operations. As long as the point $(x_v, y_v)$ is defined and lies in $R$, report it and decrement $v$ by one. As a final step, mark $v$ blue. If $v$ is a leaf of $G$ then return, else recur on its children (Figure 6, Case 1).

2. $D(v)$ is completely to the left or to the right of $R$, i.e., $x_v < a$ or $x_v > b$: return (Figure 6, Case 2).
(3) None of the above: mark \( v \) red and return (Figure 6, Case 3).

**Stage 2.** As long as there are some unhandled red vertices left in \( G \), pick any one of them, say \( v \), mark it “handled” and compute \( (x_v, y_v) \), the \( L \)-peak of \( D(v) \). Next, perform the following case analysis:

1. The \( L \)-peak of \( D(v) \) lies in \( R \), report it, and initialize \( j \) to \( i + 1 \). As long as the point \( (x_v, y_v) \) is defined and lies in \( r \), repeat it and increment \( j \) by one.
2. Next, re-initialize \( j \) to \( i - 1 \), as long as the point \( (x_v, y_v) \) is defined and lies in \( R \), report it and decrement \( j \) by one.
3. The \( L \)-peak of \( D(v) \) does not lie in \( R \), mark red the children of \( v \) (if any).

The description of the algorithm will be complete after a few words on the implementation of its basic primitives. The case-analysis of Stage 1 is performed by binary search in \( \text{Slope}(v) \) with respect to \( a \) and \( b \). In Stage 2, the \( L \)-peak of \( D(v) \) is computed by performing a binary search in \( \text{Slope}(v) \) with respect to the slope of \( L \).

![Fig. 6. The various cases.](image-url)
The correctness of the algorithm is established with the following observations.
First of all, it is clear that each point computed by the algorithm lies in \( R \) and is
reported only once. Secondly, for each node \( v \) examined, the points of
\( D(v) \cap V \) are all reported. It then suffices to show that each lower hull contributing
a point in \( R \) is indeed examined. Let \( U \) denote the set of vertices that must be
examined by a correct algorithm, i.e., \( U = \{ v | D(v) \text{ contributes at least one point}
\) to \( V \cap R \). Set

\[
U_1 = \{ v \in U | D(v) \cap r_v = \emptyset \text{ and } D(v) \cap r_v \neq \emptyset \}
\]

and

\[
U_2 = \{ v \in G | D(v) \cap r_v = \emptyset \text{ or } D(v) \cap r_v \neq \emptyset \}.
\]

The sets \( U_1 \) and \( U_2 \) contain respectively red and blue vertices. Clearly \((U \setminus U_1) \subseteq U_2\),
and this inclusion may often be strict. We omit the proof that:

1. the path from any vertex of \( U_1 \) to the root of \( G \) is a sequence of vertices in
   \( U_1 \), followed by a sequence of vertices in \( U_2 \) (the latter sequence possibly
   empty);
2. the path from any vertex of \( U_2 \) to the root of \( G \) consists exclusively of vertices in
   \( U_2 \).

These two remarks show that the algorithm visits each vertex of \( U_1 \) and \( U_2 \),
and is therefore correct. Note that the computation of \( U_2 \) may fail to contribute
any point to the output, although it provides an important guiding mechanism,
quite similar to the scheme followed by the priority search tree. In particular,
if a red node \( v \) has no intersections with the trapezoid, then we never descend
in \( G \) below \( v \). This allows us to bound the number of such "fruitless" visits by
\( 2(U_1 \cup U_2) \). The recursive definition of nested lower hulls ensures that \( U_2 \)
consists of at most two paths, each of length \( O(\log n) \). Since each visit of a vertex
in \( U_1 \) provides at least one output point, we easily bound the running time of
the algorithm by \( O(\log^2 n + \log n) \), where \( k \) is the output size. The storage
required by the algorithm is clearly \( O(n) \). The preprocessing time can be kept
down to \( O(n \log n) \), provided that the points of \( V \) are sorted by \( x \)-coordinates
at the outset of the computation. Repeated Graham scans will provide each lower
hull in linear time.

But now we have the stage set for fractional cascading. Visiting vertex \( v \) of \( G \)
involves a binary search in either \( \text{Abs}(v) \) or \( \text{Slope}(v) \). The keys to be searched
are \( a \), \( k \), or the slope of \( L \). The graph \( G \) is of bounded degree and its traversal
always involves a subgraph whose vertices are examined in a connected sequence.
We immediately conclude.

Theorem 4. Given a slanted range search problem on \( n \) points, there exists a data
structure of size \( O(n) \) that allows us to answer any query in \( O(k + \log n) \) time,
where \( k \) is the size of the output. The data structure can be constructed in \( O(n \log n) \)
time and is optimal.

5. Orthogonal Range Search, with \( n \) points in \( R \), \( A \) is the set of \( n \) points
in \( R \) a set specified by two points \( a \) and \( b \). Let \( R \) be a set of \( n \) points in
\( \mathbb{R}^d \) and \( \{ x \} \in V \) be a set of \( n \) points in \( \mathbb{R}^d \) and \( \{ y \} \in \mathbb{R}^d \).

This problem can be solved with the priority search tree of Moeller
[1978]. However, we must consider vertical rays emanating from
unbounded segments of the form \( y = \{ x \} \). To compute \( V \), we search the
ray \( R \) and the ray \( H = \{ x \} \). The h-bisection graph structure described
here is the only known structure of size \( O(n) \) that can compute all desired
intersections of the original set of rays. Figure

The catalog.
5. Orthogonal Range Search. Let $\mathbb{R}^d$ be the real $d$-dimensional Euclidean space endowed with a Cartesian system of reference $(Ox_1, \ldots, Ox_d)$. A $d$-range $R$ is a set specified by two points $(a_1, \ldots, a_d)$ and $(b_1, \ldots, b_d)$, with $a_i < b_i$: we have

$$R = \{ (x_1, \ldots, x_d) | a_i < x_i < b_i \}.$$

Let $V$ be a set of $n$ points in $\mathbb{R}^d$. The orthogonal range search problem can be stated as follows: given a query $d$-range, report all points of $V \cap R$. We direct our interest here to data structures that require only $O(n \log^d n)$ space, for some constant $d$, and provide a response time of $O(\log n + \text{output size})$. As of yet, such a data structure has been found only for the case $d = 2$ [Ch.1, GBT, W1]. We show that one also exists for the case $d = 3$. The algorithm relies on successive reductions to easier problems. We will proceed from the bottom, treating the easy cases first. We put together a kit of building blocks which we use in the end to produce the desired result.

Subproblem P1. Let $V$ be a set of $n$ points in $\mathbb{R}^2$ and let $(Ox, Oy)$ be a Cartesian system of reference. Consider the problem of computing the set $V(a,b) = \{(x,y) \in V | a \leq x \leq b \}$, given any query point $(a,b)$.

This problem can be solved by a number of known data structures, including the priority search tree of McCreight [M2]. To prepare the ground for fractional cascading, however, we must choose a different approach. Consider the set of $n$ vertical rays emanating upward from the points of $V$. This set consists of the unbounded segments of the form $[x, y)\times \{x \}$, obtained for each point $(x,y)$ of $V$. To compute $V(a,b)$, it suffices to identify all intersections between these rays and the ray $H = [(-\infty, b)\times (a,b)]$. We accomplish this task by using the hive-graph structure described by Chazelle [Ch.1]. This data structure allows us to compute all desired intersections in optimal time and space. Briefly, the hive-graph is a subdivision of the plane built by adding horizontal segments to the original set of rays. Figure 7 illustrates this construction.

![The Hive-Graph](image-url)
correspond to added edges. Without going into the details of the structure, we must mention an essential feature of the query-processing scheme. To find the intersections between the rays and the segments, the binary-graph, will test ask us to compute the successor of b in some given catalog. The result of the search will then trigger the report of each intersection at unit cost per report. The data structure requires O(n) space and can be constructed in O(n log n) time.

**Subproblem P2.** Next, we turn a restricted case of three-dimensional range search, one where the query R is in the form (a, b) x [0, b] x [0, h] (Figure 8(c)). We say that subproblem P2 is based on the two halfspaces x ≤ 0 and y ≥ 0.

Let V be the set of points \((a_1, y_1, z_1), \ldots, (a_n, y_n, z_n)\), given by their coordinates in a Cartesian system of reference, \((x, y, z)\). We use Bentley’s notion of range tree [B] to reduce this problem to O(log n) instances of subproblem P1. It is O(n log n) time, relabel the points of V so that \(x_1 < x_2 < \ldots < x_n\), and let up a complete binary tree \(T\) whose leaves correspond respectively to \((x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\) in left-to-right order. Each leaf of \(T\) has a key, which we define as the \(x\)-coordinate of its associated point. We organize \(T\) as a search tree, so that any successor of an arbitrary value among \(x_1, \ldots, x_n\) can be computed in \(O(\log n)\) time. For each vertex \(v\) of \(T\) let \(U(v)\) be the subset \(V\) induced by the leaves descending from \(v\). Let \(P(v)\) be the projection of \(U(v)\) on the plane \(x = 0\). For each set \(P(v)\) we construct the data structure described in the solution of subproblem P1. Following the paradigm of the range query, we can decompose \(R\) into a logarithmic number of canonical pieces. To do so, we search for \(a_1\) and \(b\) in \(T\). Let \(v_a\) be the leaf whose key is the successor of \(a_1\). Symmetrically, consider the leaf whose key is the successor of \(n_1\), and let \(v_b\) be its predecessor. For simplicity, we assume that all these nodes are well defined. Special cases can easily be integrated into a unified framework, but to preserve the continuity of the exposition, we will not attempt to do so. Let \(w_1\) and \(w_2\) be respectively the left and right children of the lowest common ancestor of \(v_a\) and \(v_b\). We define \(W\) as the set of nodes of \(T\) that are either right children of nodes from \(v_1\) to \(w_1\) or left children of nodes from \(v_2\) to \(w_2\). Our original problem can be solved by solving it with respect to the point \(p\) associated with the nodes of \(W\). The benefits of this multiplication of work is that each subproblem is of lesser dimensionality. So, the original query can be answered by applying the solution of P1 to each of the sets \(P(v) \cap W\). Note that the two-dimensional query for P1 is specified by the point \((b_1, b_2)\) in the yz-plane. Straightforward analysis shows that the time to process the data structure is \(O(n \log^2 n)\), the space used is \(O(n \log n)\), and the response time is \(O(k + \log^2 n)\), where \(k\) is the size of the output.

**Subproblem P3.** Next, we generalize subproblem P2 by considering queries of the form \((a, b), (a_1, b_1) \times [0, b] \times [0, h]\) (Figure 8(b)). Subproblem P3 is said to be based on the halfspace \(x > 0\).

The same complete binary tree \(T\) defined in the previous paragraph is used here, but in a somewhat different way. Let \(v_1\) (resp. \(v_2\)) be the left (resp. right) child of the internal node \(v\) in \(T\), and let \(d(v)\) be any number at least as large as
The details of the structure, querying answering process. To find the
root $x$, the hive-graph will first ask us
in catalog: The result of the search
is at unit count per report. The data
is stored in $O(n \log n)$ time.

The case of three-dimensional range
search $[a_x, b_x] \times [a_y, b_y] \times [0, b_z]$ (Figure 8)
the two halfspaces $z = 0$ and $y = k$.

The set of three-dimensional range
search $[a_x, b_x] \times [a_y, b_y] \times [0, b_z]$ (Figure 8)
the two halfspaces $z = 0$ and $y = k$.

Fig. 8. Reducing the dimensionality of the query.
any x-coordinate in $U(x_1)$ and at least as small as any in $U(x_2)$. Associate with $n$ the data structure of $P_2$ defined with respect to $U(x_1)$ (resp. $U(x_2)$) and based on $(z \geq 0, x \leq d(v) \text{ or } (z \geq 0, x \leq d(v)))$. The data structure can be constructed in $O(n \log^k n)$ time and requires $O(n \log^k n)$ space. How do we answer a query? Starting at the root of $\mathcal{I}$, we compare $d(root)$ against $a_1$ and $a_2$. (1) If $d(root) \leq a_2$, then we can apply the solution of $P_2$ using the two data structures associated with $n$. The computation will thus be complete. (2) If $d(root) < a_1$, then we iterate on this process by branching to the left child of the root. (3) If $d(root) > a_2$, then we iterate on this process by branching to the right child of the root. The running time of this algorithm will be $O(\log n + t(n))$, where $t(n)$ is the time to solve two instances of subproblem $P_2$. This brings the time complexity to $O(\log n + \text{output size})$.

Subproblem $P_4$. We are ready to return to the original problem: $R$ is now specified by two arbitrary points in $R^3$.

We modify the solution of $P_3$ in the obvious manner. Each node of $\mathcal{I}$ becomes associated with two data structures for solving not $P_2$ but, of course, $P_1$. A similar analysis shows that the storage and preprocessing time leap up to $O(n \log^k n)$ and $O(n \log^k n)$, respectively. The response time remains $O(\log^k n + \text{output size})$.

Let's examine the data structure in its full expansion. What we have is essentially an interconnection of hive-graphs. If we ignore the hives of graphs for a moment, but just concern ourselves with their associated catalogs, we obtain a graph (actually a tree) of degree at most five; a typical node is adjacent to one parent, two children, and two roots of auxiliary structures. This gives us a perfect example of implicit iterative search. Or does it really? To be consistent, we must provide catalogs to all the nodes, and not just a happy few. We can do so by supplementing empty catalogs with a single key ($x \in \mathbb{R}$). We are now in a position to apply fractional cascading to this resulting catalog graph. An immediate savings of a factor $\log n$ in query time will follow.

Theorem 5. There exists a data structure for three-dimensional orthogonal range search that allows us to answer any query in optimal $O(k + \log n)$ time, where $k$ is the size of the output. The data structure requires $O(n \log^k n)$ space and can be constructed in $O(n \log^k n)$ time.

6. Orthogonal Range Search in the Past. This section does not make use of fractional cascading per se but of its geometric counterpart, the hives of graphs [Ch 1], already mentioned in Section 5. As we will see in Section 10, however, a hive-graph is a special case of fractional cascading, so the relevance of this material makes its inclusion compelling. Consider the problem of querying a database about its present state as well as about configurations it held at previous times. This leads, traditionally known as searching in the past, has already been well-researched.

[DMO, Ch 2, Co]. The question is addressed in this paper. Briefly, it presents a data structure for orthogonally range queries of theoretical interest. Consider a point $v$ in a point set $V$ representing an employee as sex, age, or salary. Over time, their records update (not the first in a pair $(R, t)$, with the meaning: $R$ at time $t$). The time range is represented as a set $W$ of $n$ points $[a_i, h_i]$ indicating its lifetime. Our solution to this problem assumes that $d = 2$ and distinguishes Cartesian products $[x_i, y_i] \times [y_i]$, performs two-dimensional range searches, where $k$ is the size of the output. The solution of subproblem $P_4$ associated with the set $U(v) \subset W$ does not differ from wth each node sorted by increasing $y$-coordinate. The tests in the tree and retrieve the query range $R$. For each such $y$-coordinates fall between $y_1$ and $y_2$. All this is very well known, so the observation is that since within each $R$ we can see the $x$-dimension: The lifetime of a point $p = (x_1, x_2, y_1, y_2, x_3, y_3)$ is limited to the segments of the $x$-coordinates in $[x_1, x_2, x_3]$. To do so, we use $4$ intersections in time $O(\log n)$. One takes $O(p \log p)$ time and $O(p \log p)$ for the overall data structure amounting to the query time at $O(\log n + k)$. Where $k$ is the number of point dimensions is straightforward, a divide-and-conquer [8].

Theorem 6. It is possible to precompute the $d$-dimensional orthogonal range queries in $O(k + \log n)$ time. The preprocessing step takes $O(n \log^k n)$ time. The output size is $O(n \log n)$ and the time per query is $O(\log n) + k$.
as any in U(τ). Associate with a U(τ) and based on the data structure can be constructed. How do we answer a query against a, and a.

Solution of P2, using the two data structures will thus be complete.

Assess by branching to the left child of a (as far as the solution of P2 can be assessed by branching to the right child of a) t(n) + t(n), where t(n) is the time brings the time complexity to be the original problem: R is now 4.

Each node is adjacent to one parent. This gives us a perfect example of the hive-graphs for a moment. After time O(n log n) remains O(log^2 n + output size).

The problem becomes the Cartesian product [x_1, x_2] x [y_1, y_2]. Using Bentley's range tree [8], we can perform two-dimensional range searching in O(n log^2 n) space and O(k log^2 n) time, where k is the size of the output. Our structure is similar to the one defined in the solution of subproblem P2 (Section 5). As usual, each vertex v of S is associated with a list C(v) of points in U(v) sorted by increasing y-coordinate. To answer a query, we perform two binary searches in the tree and retrieve the nodes of the canonical decomposition of the query range R. For each such node t, we compute the points of C(v) whose y-coordinates fall between y_1 and y_2.

As this is well known, so where is the novelty of our structure? The key observation is that since within each list examined only y-coordinates are relevant, we can free the x dimension and use it to represent the lifetime of each point. The lifetime of a point p = (a_n, p) will be represented now by a horizontal segment [(a, p), (b, p)], instead of searching the list C(v), we must now report all the segments of [(a, p), (b, p), p] ∈ C(v)] that intersect the vertical segment [(i, y), (i, y)]. To do so, we use a hive-graph. This allows us to find all desired intersections in time O(log n + k). Constructing a hive-graph for n segments takes O(p log n) time and O(p) space [Ch 1], so preprocessing time and storage for the overall data structure amount respectively to O(n log^2 n) and O(n log n).

The query time is O(log n + k) per node, which gives a total of O(log^2 n + k), where k is the number of points to be reported. Generalization to higher dimensions is straightforward, using Bentley's technique for multidimensional divide-and-conquer [8].

Theorem 1. It is possible to perform range searching in the past over a set of n d-dimensional points in O(k log^d n) time and O(n log^d+1 n) space, where k is the size of the output. The preprocessing time is O(n log^d n).

7. Computing Locust Functions. Let V be a set of m 2-ranges in the Euclidean plane R^2, which we assume endowed with a Cartesian system of reference
A 2-range is the Cartesian product of two closed intervals (recall the definition of a d-range in Section 5). We wish to compute functions of the form

$$f: p \in \mathbb{R}^d \to f(p) \in \{0, \ldots, n\},$$

where \( f(p) \) might be defined as the number of 2-ranges containing \( p \) or as the index of the largest (smallest) 2-range containing \( p \); the notion of large or small refers to the area, perimeter, width/height ratio, or any other suitable function of 2-ranges. We characterize this class of functions as follows: a function \( G: \mathbb{R}^d \to (0, \ldots, n) \) is called decomposable if for any partition of a subset \( X \subseteq \mathbb{R}^d \) and \( Z \), \( G(X) \) can be computed from \( G(Y) \) and \( G(Z) \) in constant time \([\text{BS}a].\)

We restrict our attention to those so-called locus-functions. Let \( V(p) \) be the set of 2-ranges containing \( p \). If \( f \) is a locus-function if there exists a decomposable function \( G \) such that \( f(p) = G(V(p)) \), for any \( p \in \mathbb{R}^d \).

Note that the problem of computing \( V(p) \), given any query point \( p \), has been solved optimally in \([\text{Co}1]\). The fact that \( f \) is single-valued makes the problem of computing locus-functions more difficult. For this reason, we resort to a slightly redundant data structure, inspired by Bentley and Wood's segment tree \([\text{BW}].\)

We assume that the reader is familiar with this notion. Let \((x_1, \ldots, x_d)\) be the \( x \)-coordinates of the 2-ranges of \( V \), sorted in nondecreasing order. We construct a \((2^d - 1)-leaves\) complete binary tree \( G \), placing the \( i \)-th leaf of \( G \) from the left in correspondence with the interval \([x_i, x_{i+1})\). Each vertex \( v \) of \( G \) has a span \( V(v) \), defined as the union of all intervals associated with leaves descending from \( v \). \( G \) induces a canonical decomposition of each 2-range of \( V \) into \( O(\log n) \) canonical parts. With each node \( v \) distinct from the root, we associate the subset \( \mathcal{R}(v) \subseteq V \) made of 2-ranges whose projections on the \( x \)-axis contains the span of \( v \) but not the span of \( v \)'s parent. Vertex \( v \) is assigned a catalog \( C(v) \) containing the \( y \)-coordinates, is sorted order, all of the 2-ranges in \( \mathcal{R}(v) \). Note that each 2-range in \( \mathcal{R}(v) \) contributes two entries to the catalog.

Let \( \rho = (\rho_x, \rho_y) \) and let \( \rho_s \) denote the restriction of \( \rho \) to the subset of 2-ranges in \( \mathcal{R}(v) \). Within the vertical slab \((x, y) \mid x \in \mathcal{R}(v)) \), \( \rho_s \) can be computed in \( O(\log n) \) time by performing a binary search in \( C(v) \) for the key \( \rho_x \). To do so, it suffices to store the proper answer in each entry of \( C(v) \) in preprocessing. We can now respond to any query as follows: in \( O(\log n) \) time, compute the set \( \rho_s \) by performing a binary search in \( G \) for the key \( \rho_y \). The value of \( f(p) \) is obtained by combining together the partial answers \((f(p) : p \in \pi(v))\), at a total cost of \( O(\log n) \) operations. We omit the analysis of the preprocessing time because of its dependence on the particular function \( f \) we are dealing with. If \( f(p) \) denotes the number of 2-ranges that contain \( p \), then it is trivial to guarantee an \( O(\log n) \) preprocessing time by scanning each \( C(v) \) linearly and updating partial counts on the fly. If \( f \) is more exotic, this on-line method might not work, however.

Once again, using Bentley's technique for multidimensional divide-and-conquer \([\text{B}].\), we easily generalize this scheme to higher dimensions. All definitions, necessary facts, and algorithms are extended in a straightforward manner to the

computations of locus-functions and a factor of \( \log \) in storage.

With the algorithms now designed and applied fractionally \([\text{ratio}].\).

For the sake of generality, we can easily modify the structure \( G \) consists of \( d \)-

dimensional \( \Sigma \)-structured trees to obtain other \( \Sigma \)-structured trees.

Theorem 3. Given a set of \( \Sigma \)-function \( f \) is \( O(\log\log n) \).

8. Space-Compression Scheme

Range trees \([\text{B}].\) are suboptimal in terms of their redundancy and time as response time. Fractional removal of degradation. We illustrate this by an example in two dimensions, which can be reduced by a factor \( \log \).

We borrow notation from Section 2. If the order \( 2 \)-range of \( G \) is sorted in integer. Construct a \((2^d - 1)-leaves\) of \( G \) from the left in corresponding order, and \( \rho \) is defined as before. Each leaf descends from the root, \( v \), whose projections on the \( x \)-axis contain the index \( k \) such that \( \rho \in \mathcal{R}(v) \). Unfortunately, storing all \( \mathcal{R}(v) \) is in order. Let \( k_1, \ldots, k_t \) be any 2-range of \( V \) that appears in the indices \( k \) such that \( \rho \in \mathcal{R}(v) \). We have either \( \rho = \rho_s \) or \( \rho \) is only at the highest node u. This reason, we can spend freely in the other trees. We construct the set

(1) If \( i = 1 \), include \( k \) in \( \mathcal{R}(v) \).

(2) If \( i = n \), include \( k \) in \( \mathcal{R}(v) \).

(3) If \( i < n \), include \( k \) in \( \mathcal{R}(v) \).
of 2-ranges containing \( p \) or as the
partitioning \( p \); the notion of large or small
ratio, or any other suitable function
acquisitions as follows: a function \( G: \mathbb{Z} \rightarrow \mathbb{Z} \)
partition of a subset \( X \subseteq Y \) with \( G \) and
functions. Let \( V(p) \) be the set
if there exists a decomposable
by \( p \in \mathbb{Z}^2 \).
Given any query point \( p \) has been
a single-valued makes the problem.
For this reason, we router to a slightly
by Wood's segment tree [4W].
our notion. Let \( x_1, \ldots, x_n \) be the
of \( G \). Let \( x \in \mathbb{Z} \),
with lower descending from \( v \). G
of \( V \) and \( O(n) \) canonical
we associate the subset \( \mathcal{R}(v) \subseteq V \). The
axis contain the span of \( v \) but not
catalog \( C(v) \) containing the
true in \( \mathcal{R}(v) \). Note that each 2-range
function of \( f \) to the subset of 2-ranges
\( f \). \( f_1(p) \) can be computed in
in \( (\mathcal{C}(v)) \) for the key \( p \). To do so,
any of \( \mathcal{C}(v) \) in preprocess.
\( O(n \log n) \) time, compute the set
of \( (x_1, \ldots, x_n) \) are the \( x \)-coordinates
2-ranges excluding \( p \), then it
have \( f \) is more exotic, this on-line
theoretical divide-and-
theoretical dimensionality of the
function \( f \). The
the partial answers \( \mathcal{L}(p) \).
It is. We omit the analysis of
the particular function \( f \) we
2-ranges that contain \( p \), then it
by scanning each \( C(v) \).
\( f \) is more exotic, this on-line

8. A Space-compression Scheme. Data structures such as segment-trees [4W] and
range trees [5B] are suboptimal, space-wise. It is possible to eliminate some
of their redundancy and thus save storage, but this entails some degradation
in response time. Fractional cascading can be used, however, to slow down the rate
of degradation. We illustrate this point by returning to the problem of computing
locus-functions in two dimensions (see Section 7). We will show that the storage
 can be reduced by a factor \( \log n \), while increasing the query time by a factor
\( \log n \). We confess that this result is rather academic interest, and we would
not have included it, had it not illustrated the versatility of fractional cascading
in such a simple way, as we will see now.

We borrow notation from Section 7. Let again \( (x_1, \ldots, x_n) \) be the \( x \)-coordinates
of the 2-ranges of \( v \), sorted in non-decreasing order, and let \( \alpha \) be a positive
integer. Construct a \((2n - 1)\)-leaf complete \( \alpha \)-ary tree \( G \) by placing the \( i \)th leaf
of \( G \) from the left in correspondence with the interval \( [x_i, x_{i+1}] \). The span
of \( v \) is defined as before: \( (\mathcal{R}(v)) \) is the union of all intervals associated
with leaves descending from \( v \). As usual, \( \mathcal{R}(v) \subseteq \mathcal{R}(v) \) designates the set of 2-ranges
whose projections on the \( y \)-axis contain the span of \( v \) but not the span of \( v \)'s
parent. Unfortunately, storing all these sets is too expensive, so a redesignation
of \( \mathcal{R}(v) \) is in order. Let \( s_1, \ldots, s_n \) be the children of \( v \) from left to right and let \( \mathcal{R}(v) \)
be any 2-range of \( Y \) that appears in at least one \( \mathcal{R}(s_j) \) \((j = 1, \ldots, n) \). Note
that the indices \( i \) such that \( R \in \mathcal{R}(s_j) \) if any form a consecutive interval \( (i, j) \). In
general, we will have either \( i = 0 \) or \( j = \alpha \). The inequalities \( 1 < i < j < \alpha \) can take
place only at the highest node used in the canonical decomposition of \( R \). For
this reason, we can spend freely in the latter case, but we must show restraint
in the others. We construct the sets \( \mathcal{R}(s_1), \mathcal{R}(s_2), \ldots, \mathcal{R}(s_{\alpha}) \) as follows:

1) If \( i = 1 \), include \( R \) in \( \mathcal{R}(s_1) \).
2) If \( j = \alpha \), include \( R \) in \( \mathcal{R}(s_{\alpha}) \).
3) If \( 1 < i < j < \alpha \), include \( R \) in \( \mathcal{R}(s_i), \ldots, \mathcal{R}(s_{j-1}) \).

computation of locus-functions on \( d \)-ranges. Each increase of one in dimension
adds a factor of \( \log n \) in storage and search time.
With the algorithm now described, we identify its iterative search component
and apply fractional cascading to improve its performance by a logarithmic factor.
For the sake of generality, we consider the case where \( V \) consists of \( d \)-ranges.
The structure \( G \) consists of \( d - 1 \) levels of nested binary trees. Each vertex
is adjacent at most four other vertices (one parent, two children, one root of a
structure of lesser dimension). The trees at the lowest level do not have pointers
to other tree structures but, instead, have a catalog associated with each of their
vertices. For consistency, vertices with no catalogs are assigned dummy catalogs
\( (v) \). Each traversal of \( G \) clearly satisfies the connectivity requirement of frac-
tional cascading, we conclude.

Theorem 7. Given a set of \( n \) \( d \)-ranges in \( \mathbb{R}^d \), it is possible to compute any
locus-function in \( O(n \log d \cdot n) \) time, using a data structure of the form
\( \mathcal{R}(n \log d \cdot n) \).
It is easy to understand the why and wherefore of this construction. Given the interval-like occurrences of \( R \) among brother vertices, the collection of sets \( \mathcal{R}(v_0), \mathcal{R}(v_1), \mathcal{R}(v_n) \) provides an implicit representation of the collection of sets \( \mathcal{R}(v) \). With each set \( \mathcal{R}(v) \), we associate the catalog \( C(v) \) defined in Section 7. Note that except for one level each 2-range \( R \) can appear at most twice at each level of \( G \). This contributes \( O(c \log n/\log a) \) to the storage. The exception corresponds to the highest-level occurrences of \( R \), which come in batches of at most \( a \). Consequently, the data structure requires \( O(n \log n/\log a + \text{one}) \) space.

To answer a query \( p = (p_0, p_n) \), we first collect all vertices whose spans contain \( p \). For each such vertex, we consider the children of its parent in left-to-right order, \( v_1, \ldots, v_n \). Let \( n \) be the vertex in question. For obvious reasons, \( f_0(p) \) can be computed by searching for \( p \) in the catalogs \( C(v_0), C(v_1), \ldots, C(v_n) \), and \( g_0(p), g_1(p), \ldots, g_n(p) \), and if \( 1 \leq i < a \) also \( C_i(v) \). This scheme yields an overall \( O(a \log n/\log a) \) response time.

A standard binary representation of \( G \) allows us to apply fractional cascading (see Knuth [1], for example). Let \( v_1, \ldots, v_n \) be the children of \( v \) from left to right. We remove all pointers from \( v \) to \( v_1, \ldots, v_n \), and replace them by pointers from \( v_i \) to \( v_{i+1} \), for \( i = 1, 2, \ldots, n-1 \) (Figure 9). To each node \( v \), we attach a little chain of three consecutive nodes, assigned to the catalogs \( C_i(v) \), \( C_i(v) \), and \( C_i(v) \), whenever these are well-defined. The data structure forms a catalog graph of bounded degree. Application of fractional cascading immediately takes the running time down to \( O(n \log n/\log a + \log n) \). Setting \( a = \lfloor \log n \rfloor \), we obtain the following result.

![Fig. 9. Putting each node in normal form.](image)

**Theorem 8.** Given a set \( \mathcal{R} \) of axes, we can compute a locus function in \( O(n \log n/\log a) \) time.

**9. Iterative Search Extensions**

with query problems which an literature but natural generalization range search (Section 5) can be involves retrieving the names \( c \). What is often desired, but additional information about some files (or catalogs) are extra work cannot be nicely problem. The only recourse is to select the range search. It of the algorithm by a logarithmic

We will show that with a little in order to retrieve the complete apply only to range search but of our approach is its generality problems and show that by a date the additional requests we must formalize this notion

Consider the following class (finite) query domain, and Preprocess the set \( V \) so that it efficiently:

\[ g: q \in Q \rightarrow g(q) \in V \]

In the orthogonal range search and \( g(q) \) is the set \( V \cap q \). For problem \( \text{II}^* \), each element \( v \) in a totally ordered set \( X \). A is to compute the successor of

**Definition.** Problem \( \text{II}^* \) is

The term "IS-extension" is a special feature shared by many given graph structures. The most generally a graph of bounded manner when answering a query algorithms generally via fragment algorithms to which these are a generic reference algorithm. Problem \( \text{II}^* \), we say that \( \mathcal{R} \) is

1. The underlying data struc
9. Iterative Search Extensions of Query Problems. In practice one is often faced with query problems which are not quite the standard problems studied in the literature but natural generalizations thereof. A typical occurrence of orthogonal range search (Section 5) can be found in a personnel division's database. A query involves retrieving the names of all employees whose attributes fall in a certain range. What is often desired, however, is not so much the names of the employees but additional information about them. To satisfy this request will involve looking up some files (or catalogs) associated with each employee. Unfortunately, this extra work cannot be nicely integrated within a more general range search problem. The only recourse is then to search separately the files of each employee selected by the range search. In the best case, this may multiply the running time of the algorithm by a logarithmic factor.

We will show that with a little care asymptotically no extra work need be done in order to retrieve the complementary information desired. This result does not apply only to range search but to a host of other query problems. One advantage of our approach is its generality. We investigate a number of algorithms for query problems and show that by a generic modification each can be made to accommodate the additional requests mentioned above. Before proceeding further, we must formalize this notion of additional request.

Consider the following cases of problems: let $V$ be a data set, $Q$ a (finite or infinite) query domain, and $P$ a predicate defined for each pair in $V \times Q$. We preprocess the set $V$ so that the function $g$ defined as follows can be computed efficiently:

$$g : q \in Q \rightarrow g(q) \in 2^V$$

$$g(q) = \{v \in V \mid P(v, q) \text{ is true} \}$$

In the orthogonal range search problem, $V$ is a set of points in $\mathbb{R}^d$, $g$ is a $d$-range and $g(q)$ is the set $V \cap q$. For any query problem $P$ we define an iterative search problem $P^* = \{ x \in V \mid \exists \text{ a distinct catalog } C(x) \text{ defined over a totally ordered set } X. \text{ A query for } P^* \text{ is a pair } (q, x) \in Q \times X; \text{ the problem is to compute the successor of } x \text{ in each catalog of } (C(x) \cap g(q)) \}$. DEFINITION. Problem $P^*$ is called the IS-extension of problem $P$.

The term "IS-extension" is a short-hand for iterative search extension. One nice feature shared by many algorithms for query problems is that they operate on graph structures. The memory is often organized as a tree, a dag, or more generally a graph of bounded degree, which the algorithm traverses in a connected manner when answering a query. This feature allows us to transform these algorithms generically via fractional cascading. We next characterize the class of algorithms to which these transformations apply. This leads to the definition of a retrieval reference algorithm or RRA for short. Let $af$ be an algorithm for problem $P$; we say that $af$ is an RRA if and only if:

1. The underlying data structure of $af$ is a graph $G$ of bounded degree. Each
vertex of $G$ is associated with at most one element of $u$, but elements of $V$ may appear in several vertices.

(2) The output of $af$, i.e., $(v \in V) P(u, q)$ is true, is a subset of the data stored at the vertices visited during the computation.

(3) The computation is modelled by a sequence of stages, each of which corresponds to one or several actual steps of the algorithm. To each stage $i$ corresponds a vertex $v(i); G$; for each $v(i)$ (except for at most a constant number of them) there exists an edge of the form $(v(i), v(i))$ with $r < i$.

(4) The mapping between $v(0), v(1), \ldots$ and the steps of the algorithm is trivial. Transforming the algorithm so that it outputs the name of the current vertex $v(i)$ at each step can always be done without slowing down the algorithm by more than a constant factor.

Note that these requirements do not in any way define a model of computation. These are only necessary and sufficient requirements for an algorithm to be an RRA. We will find that although a number of algorithms for query problems can be immediately seen as RRA's, many others have to undergo minor transformations in order to be readily recognized as such. Here are some examples of query problems which admit of RRA's. This list is given for illustrative purposes and is not meant to be comprehensive.

(a) Interval Overlap. Given a set $V$ of intervals and a query interval $q$, report the intervals of $V$ that intersect $q$ [Ch 1, E, M 1, M 2].

(b) Segment Intersection. Given a set $S$ of segments in the plane and query segment $q$, report the segments of $S$ that intersect $q$ [Ch 1, D, EKM].

(c) Point Enclosure. Given a set $V$ of $d$-ranges and a query point $q$ in $R^d$, report the $d$-ranges of $V$ that contain $q$ [Ch 1, E].

(d) Orthogonal Range Search. Given a set $R$ of points in $R^d$ and a query $d$-range $q$, report the points of $R$ that lie inside $q$ [B, Ch 3, GBT, M 2, W].

(e) Rectangle Search. Given a set $V$ of $d$-ranges and a query $d$-range $q$, report the $d$-ranges of $V$ that intersect $q$ [Ch 3, GBT].

(f) Triangle Retrieval. Given a set $V$ of points in $E^3$ (resp. $E^3$) and a query triangle (resp. tetrahedron) $q$, report the points of $V$ that lie within $q$ [CY, EH, EW, Y].

(g) Circular Range Query. Given a set $V$ of points in $E^2$ and a query circle $q$, report the points of $V$ that lie within $q$ [CCP].

(h) $k$-Nearest-Neighbor. Given a set $V$ of points in $E^d$ and a query of the form $(q, k)$, $q \in E^d$, $k$ integral $\geq 0$, report the $k$ points of $V$ closest to $q$ [CCP].

Reference retrieval algorithms are best understood in the broader context of the pointer machine model [T]. This model includes most algorithms free of address calculations: this rules out, for example, hashing, radix sort, and operations on dense matrices. In the pointer machine model, the memory is represented by a directed graph with one vertex per piece of data and one edge per pointer. The computation involves visiting vertices of the graph in such a way that going from one vertex to another requires the presence of a directed edge from the origin to the destination. New pointers are provided by requesting new memory cells from a free list; they cannot be created by arithmetical operations.

Since both interval overlap and time, so can their IS-extents of the other problems mentioned in the references for their IS-extents at least linearly, the storage usage is the size of the input. The name $\alpha$ and looking up the search key $\alpha$ uses only $O(n + f(p))$ space $b$.

10. Other Applications. To illu-

Sometimes, solutions to query form binary search in linear and it is easily fixed by substituting $\alpha$-marks, checking each of the $\alpha$-marks leads to the straightforward code.

Lemmas 7. All solution to the $\alpha$-most efficient known to date a

The main result of this section always be generally transformed.

To alleviate the notation, we may have each of the same size $m$.

Theorem 9. Let $f\alpha$ be a query be an RRA for solving $\alpha$. Assume $k$ response time, where $k$ is the $\alpha$-obtained by associating a one exists a data structure for solving $f, p)$ $+$ $k$ response time.

Proof. Let $G$ be the graph of $G$ corresponds at most one $\alpha$ to $\alpha$ if the vertex does not $\alpha$ we can apply fractional cascading query, look up the search key $\alpha$ subsequent step $\alpha > 0$ retrieve with $\alpha(i)$.
Fusional Cascading: II. Applications

Sometimes, solutions to query problems do require address calculations to perform binary searches in linear arrays. This is not a major handicap, however, since it is easily fixed by substituting balanced search trees for arrays. With these remarks, checking each of the references accompanying the problems listed above leads to the straightforward conclusion:

**Lemma 7.** All solutions to the eight problems referenced above (which include the most efficient known to date) are of the type RRA.

The main result of this section states that any RRA for a query problem II can be generally transformed into an algorithm for solving its IS-extension. To alleviate the notation, we make the simplifying assumption that the catalogs are each of the same size $m$.

**Theorem 9.** Let II be a query problem defined over a set $V$ of size $p$, and let $d$ be an RRA for solving II. Assume that $d$ requires $O(f(p))$ space and has $O(g(p) + k)$ response time, where $k$ is the size of the output. Let II* be the IS-extension of II obtained by associating a catalog of size $m$ with each element in $V$. Then there exists a data structure for solving II*, which requires $O(mf(p))$ space and $O(mg(p) + k)$ response time.

**Proof.** Let $G$ be the graph used in modelling $d$ as an RRA. To each vertex of $G$ corresponds at most one element of $V$, hence one catalog (possibly reduced to $\emptyset$ if the vertex does not store any element). Since $T$ has bounded degree, we can apply fractional cascading to its associated set of catalogs. To answer a query, look up the search key $x$ in the catalog associated with $\xi(x)$; at any subsequent step $t > 0$ retrieve the relevant successor in the catalog associated with $\xi(x)$.

Since both interval overlap and point enclosure can be solved in optimal space and time, so can their IS-extensions [Ch 1]. If $m = O(n)$, the algorithms for each of the other problems mentioned above have the same complexity as the algorithms for their IS-extensions. In general, note that since the function $f$ grows at least linearly, the storage used for solving II* is also $O(f(n))$, where $n = pm$ is the size of the input. The naive algorithm for solving II* consists of applying $d$ and looking up the search key $x$ in each of the $k$ catalogs found. This scheme uses only $O(mf(p))$ space but may need as much as $O(g(p) + k \log m)$ time.

10. Other Applications. To illustrate the wide applicability of fractional cascading, we wish to report briefly on other related work. The idea of propagating fractional samples has already been used in a number of different specific contexts [Ch 1, Co, EGSJ]. Interestingly, in all three cases, fractional cascading provides an unifying framework in which to understand these results. Let's take the case of the Klee-graph, for example. We briefly recall this technique (see [Ch 1] for details). Given a set of horizontal segments, construct a planar subdivision by adding, for each endpoint $p$, the longest vertical segment passing through $p$ that
does not properly intersect any horizontal segment. This is our base subdivision (Figure 10). We refine it by adding new vertical segments, so that every face ends up with at most a constant number of vertices. As we can see, it is not immediate that such a property can be ensured without adding a quadratic number of segments. The novelty of [Ch 1] was to show that by propagating only every other vertical segment, the size of the subdivision remains linear.

How can we interpret this result in terms of fractional cascading? Every horizontal segment corresponds to a node of the catalog graph; catalogs are made of the $x$-coordinates of the vertices on each segment; edges connect nodes whose corresponding segments are visible from each other (where segment $a$ is visible from segment $b$ if there exists a vertical segment that connects $a$ and $b$, and does not intersect any other segment). In Figure 10, for example, node $v_1$ is adjacent to $v_2$, $v_3$, and $v_4$. Its catalog is the list of $x$-coordinates $(g, a, b, f)$.

Other results that can be interpreted in terms of fractional cascading or that make explicit use of it include algorithms for:

(a) **Planar point location.** Locate a point in a planar subdivision [Co, EGS].
(b) **Point enclosure.** Find $d$-ranges containing a query point [Ch 1].
(c) **Homothetic range search.** Report the points falling in a query 2-range of fixed aspect-ratio [CE].
(d) **3d-Domination search.** Range search in $\mathbb{R}^3$ for queries of the form $[0, a] \times [0, b] \times [0, c]$ [CE].
(e) **Intersection search.** Find the intersection of a polygon with a query segment [CG].

11. **Concluding Remarks.** The contribution of this paper has been to show the versatility of a new data structuring technique, called fractional cascading. The technique seems simple and general enough to have many practical applications. Besides those studied in this paper, one should mention the relevance of fractional cascading to external searching in general. Since it works on a pointer machine, fractional cascading can handle situations where the collection of catalogs is very large, but each of them can be stored on one or a small number of pages. It would be interesting to determine if such a scheme can outperform hashing techniques in practice.

One of the most interesting open problems is to determine whether fractional cascading extends to higher dimensions. Imagine that a catalog is a planar subdivision, and the "successor" of a query point is the name of the face that contains it. Can iterative search be speeded up? As usual, we may try to merge all the subdivisions into one master subdivision. The catch is that merging together two subdivisions of respective size $l$ and $m$ may result in a subdivision of size $O(lm)$. This contrasts with the nice property of linear lists: merging two of them only adds their sizes. Why is this extension so important, anyway? Various data structures for near-neighbor problems involve a hierarchy of Voronoi diagrams. A query involves selecting a few of them and performing repeated point locations. Results similar to the ones we have obtained with fractional cascading would bring about dramatic improvements to the best solutions known to date.
This segment. This is our base subdivision of segments, so that every face ends in a segment. As we can see, it is not immediate but adding a quadratic number of vertices that by propagating only every edge remains linear.

If there is 0 of fractional cascading? Every query on the catalog graph; catalogs are made of segments, edges connect nodes whose coordinates (g, a, b, f). If the point a is visible to another (segment a is visible to segment b), for example, node u is adjacent to v. The problem of fractional cascading or that of a planar subdivision [Co, EGS].

Fig. 10. The base subdivision.
Acknowledgments. We wish to thank Bob Tarjan for his many helpful comments and suggestions. The proof of Lemma 1, in particular, is due to him. We also thank Cynthia Hishard for her many suggestions that improved the exposition.

References


