12

Fractional Cascading

Bernard Chazelle and Leonidas J. Guibas

June 23, 1986

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center and two other corporate research laboratories are committed to filling that need.

SRC opened its doors in 1984. We are still making plans and building foundations for our long-term mission, which is to design, build, and use new digital systems five to ten years before they become commonplace. We aim to advance both the state of knowledge and the state of the art.

SRC will create and use real systems in order to investigate their properties. Interesting systems are too complex to be evaluated purely in the abstract. Our strategy is to build prototypes, use them as daily tools, and feed the experience back into the design of better tools and the development of more relevant theories. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

During the next several years SRC will explore applications of high-performance personal computing, distributed computing, communications, databases, programming environments, system-building tools, design automation, specification technology, and tightly coupled multiprocessors.

SRC will also do work of a more formal and mathematical flavor; some of us will be constructing theories, developing algorithms, and proving theorems as well as designing systems and writing programs. Some of our work will be in established fields of theoretical computer science, such as the analysis of algorithms. computational geometry, and logics of programming. We also expect to explore new ground motivated by problems that arise in our systems research.

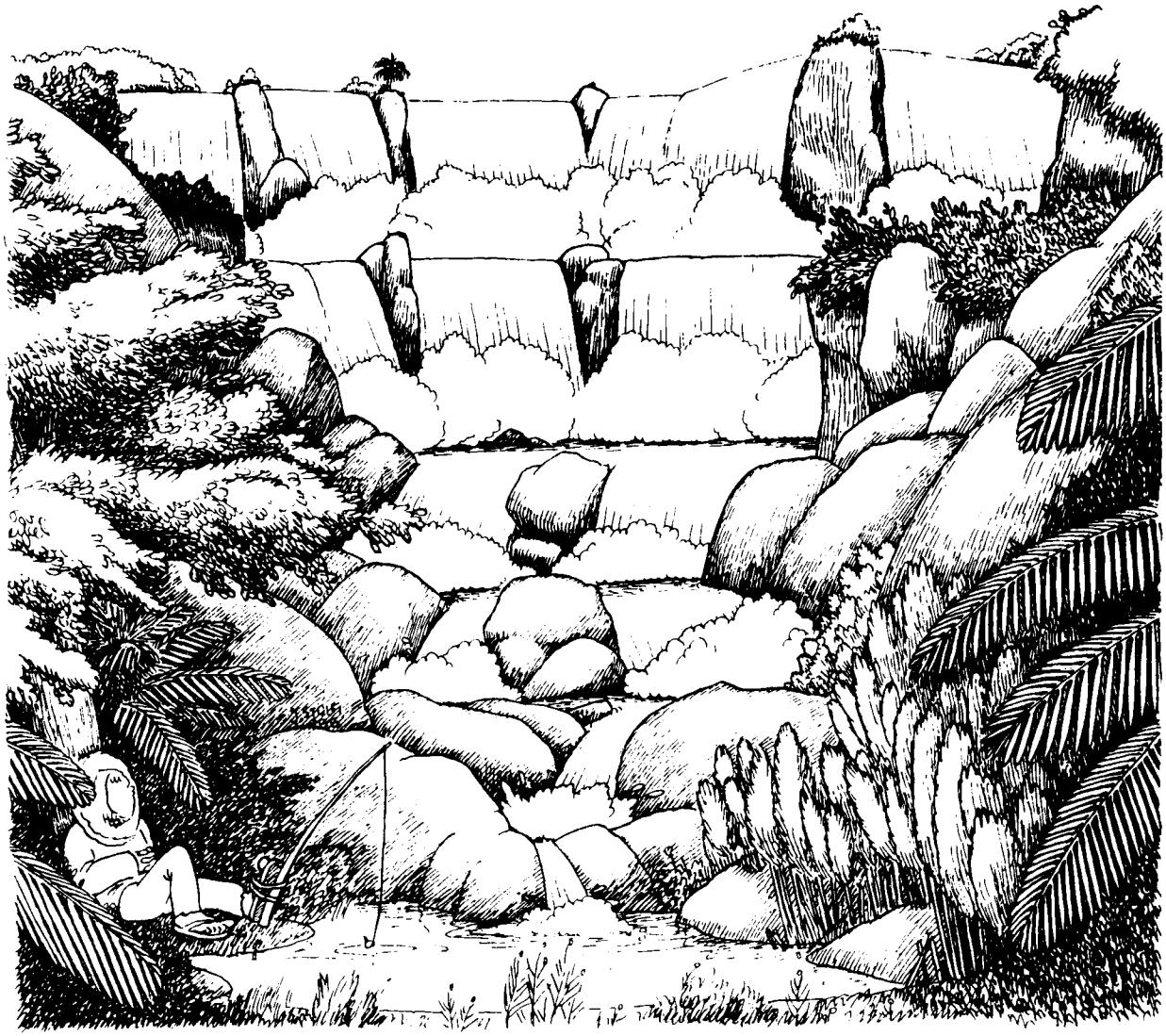
DEC is committed to open research. The improved understanding that comes with widespread exposure is more valuable than any transient competitive advantage. SRC will freely report results in conferences and professional journals. We will actively seek users for our prototype systems among those with whom we have common research interests. We will encourage visits by university researchers and conduct collaborative research.

Robert W. Taylor, Director

Fractional Cascading

Bernard Chazelle and Leonidas J. Guibas

June 23, 1986



Publication history

An earlier version of this report appeared in the Proceedings of 12th ICALP Colloquium, 1985, 90–100, and was published as Lecture Notes in Computer Science, 194, by Springer-Verlag, 1985. This material will also appear in Algorithmica.

Acknowledgements

Bernard Chazelle is currently on leave of absence from Brown University at Ecole Normale Supérieure. He was supported in part by NSF 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. Part of this work was done while the second author was employed by the Xerox Palo Alto Research Center. Contact author's address: Leonidas J. Guibas, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, Ca. 94301.

Copyright and reprint permissions

Authors' abstract

In computational geometry many search problems and range queries can be solved by performing an iterative search for the same key in separate ordered lists. In Part I of this report we show that, if these ordered lists can be put in a one-to-one correspondence with the nodes of a graph of degree d so that the iterative search always proceeds along edges of that graph, then we can do much better than the obvious sequence of binary searches. Without expanding the storage by more than a constant factor, we can build a data-structure, called a *fractional cascading structure*, in which all original searches after the first can be carried out at only $\log d$ extra cost per search. Several results related to the dynamization of this structure are also presented. Part II gives numerous applications of this technique to geometric problems. Examples include intersecting a polygonal path with a line, slanted 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: binary search, B-tree, iterative search, multiple look-up, range query, dynamization of data structures, fractional cascading.

Bernard Chazelle and Leonidas J. Guibas

Capsule review

Suppose we have to search the same key in several sorted lists, each of size n. The obvious approach — perform a binary search in each list — requires $O(\log n)$ operations for each list. Fractional cascading is a method of cross-linking those lists in such a way that the $O(\log n)$ cost of binary search has to be paid only once: to locate the key in one of the lists. The cross-links then allow the key to be located in each additional list with only a constant number of operations.

The first part of the paper describes algorithms for the construction, use, and updating of the fractional cascading structure. The total number of cross-links (and the time required to build them) is proved to be linear in the total size of the lists. The second part shows how fractional cascading can be used to reduce the theoretical complexity of several geometric search problems.

The bias towards geometrically flavored examples here reflects the authors' background and interests, and not any intrinsic limitation of the technique. Fractional cascading is a purely combinatorial data structuring method, and it will certainly be of great value in many other areas.

Jorge Stolfi

Contents

Fractional Cascading: I
A Data Structuring Technique

1. Introduction

This paper introduces a new data structuring technique for improving existing solutions to retrieval problems. For illustrative purposes, let us consider the following three classical problems in computational geometry:

(a) Given a collection of intervals on the line, how many of them intersect an arbitrary query interval?

(b) Given a polygon P, which sides of P intersect an arbitrary query line?

(c) Given a collection of rectangles, which of them contain an arbitrary query point?

What do these problems have in common? Except that they each fall into the broader class of *geometric retrieval problems*, little seems to relate them together in one way or the other. Yet, we can speed up the best algorithms known for solving these problems using a single common technique, which we call *fractional cascading*. This novel technique is general enough to speed up the solutions not only of these three problems but of a host of others; we will give numerous examples in part II of this paper.

In a nutshell, fractional cascading is an efficient strategy for dealing with the following problem, termed *iterative search*: let G be a graph whose vertices are in one-to-one correspondence with a set of sorted lists; given a query consisting of a key q and a subgraph π of G, search for q in each of the lists associated with the vertices of π. This problem has a trivial solution involving repeated binary searches. Fractional cascading establishes that it is possible to do much better: under some weak assumptions, we show that with only linear space it is possible to organize the set of lists so that all the searches can be accomplished in optimal time, at roughly constant cost per search.

As the second part of this paper amply demonstrates, iterative search is a fundamental component of many query-answering algorithms. Let us take Problem (c), for instance: *given a collection of rectangles, which of them contain an arbitrary query point*? The data structure for this problem with the most efficient asymptotic performance [Ch1] is a complete binary tree whose nodes point to auxiliary lists. Answering a query involves tracing a path in the tree, while searching for a given value (one of the coordinates of the query point) in *each* auxiliary list associated with the nodes visited on the path. Here, as well as in many other algorithms for retrieval problems, iterative search is the main computational bottleneck. For this reason, it is desirable to treat the problem in an abstract setting, so the results obtained can be directly applied to as many problems as possible.

Following this approach, we present an optimal solution to iterative search, which we then apply to a number of retrieval problems. By doing so, we are able to improve upon a host of previous complexity results. It is worth noting, and this will become even more apparent when we go into applications of fractional cascading, that this technique can be usefully thought of as a postprocessing step that can be applied to speed up already existing solutions of various problems.

Part I of this paper describes and analyzes fractional cascading in a general setting. We present and discuss the construction of the data structure, its use for query-answering, and the issues involved in making our solution dynamic. In part II we present a number of specific applications of the technique, and examine the complexity of iterative search in the light of fractional cascading. The two parts can be read almost independently of each other. Only Section 2.1 of this part, which introduces the basic concepts and presents the main results, is necessary for reading the second part.

2. The Fractional Cascading Technique

In this section we present a static description of what the fractional cascading structure is and how it can be used to solve the iterated search problem.

2.1. Preliminaries: Setting the Stage; Summary of the Main Result

We consider a fixed graph $G = (V, E)$ of $|V| = n$ vertices and $|E| = m$ edges. The graph G is undirected and connected, and contains no loops or multiple edges. In addition to this classical graph structure, we have associated with each vertex v of G a catalog C_v, and associated with each edge e a range R_e.

A *catalog* is an ordered collection of records, where each record has an associated value in the set $\Re \cup \{-\infty, +\infty\}$. The records are stored in the catalog in non-decreasing order of their value; note that different records may contain the same value. A catalog is never empty: it always contains one record with value $-\infty$ and one record with value $+\infty$. These special records play the role of sentinels so as to simplify the algorithms.[1] A *range* is simply an interval of the form $[x, y]$, $[-\infty, y]$, $[x, +\infty]$, or $[-\infty, +\infty]$. In all cases, it is specified by two endpoints chosen from the linear order. We will refer to our graph G, together with the associated catalogs and ranges, as a *catalog graph*. This is the combinatorial structure to which fractional cascading can be applied.

For notational convenience we make the following assumption: if value K is an endpoint of the range $R_{(u,v)}$ associated with edge (u, v), then K appears as the value of some record in both catalogs C_u and C_v. In fact, if two ranges $R_{(u,v)}$ and $R_{(v,w)}$ have an endpoint in common, its value will appear *twice* in the catalog C_v of their shared vertex v. This requirement does not in any way restrict the generality of our discussion and, since G is connected, it provides a notational advantage. Indeed the space required to store a catalog graph is proportional to the total size of its catalogs. If $s = \sum_{v \in V} |C_v|$, then the $O(m + n)$ storage required to represent the graph structure itself, plus the storage for all the sorted multisets which are the catalogs, plus that for the intervals which are the ranges, adds in total to $O(s)$.

Next, we give three definitions to introduce some basic concepts. We start with a notion related to the degree of the vertices because, as we will see, the performance of our data structure will be very sensitive to high degrees, and more accurately, to high *local degrees*.

Definition 1. A catalog graph is said to have *locally bounded degree d* if for each vertex v and each value $x \in \Re$ the number of edges incident on v whose range includes x is bounded by d.

Note that if G has bounded degree it also has locally bounded degree, but the converse is not true in general. From now on, unless specified otherwise, we will assume that G has locally bounded degree d. The next definition formalizes the intuitive notion of enumerating the vertices of a subgraph in a "connected" way. The one after that makes precise the type of query underlying the notion of iterative search.

Definition 2. A *generalized path* π in G is a sequence of vertices $v_1, v_2, \ldots, v_p$ and corresponding edges $e_2, \ldots, e_p$ such that for each vertex v_i, $i > 1$, the edge e_i connects v_i to a vertex v_j of the path, with $j < i$.

[1] Our assumption that the values are real numbers is only for notational convenience; any linearly ordered set will do.

Since our graph G is connected, it is obvious that there exist permutations of V that are generalized paths of G. In general, any connected subgraph of G gives rise to a generalized path.

Definition 3. A *multiple look-up query* is a pair (x, π), where x is a key value in $\Re$ and π is a generalized path of G. The value x *must* fall within the range of every edge of π. The path π may be specified *on-line*, in other words, one edge at a time.

For a catalog C we will denote by $\sigma(x, C)$ the first record in C whose value is greater than or equal to x; we will call the value of this record the *successor* of x in C. Computing this value is equivalent to locating x in the sorted multiset of values represented by C. The main subject of this work, the *iterative search problem*, can now be formally stated as follows:

> Given a multiple look-up query (x, π), look up x successively in the catalogs C_v associated with each vertex v of π, and in each case report $\sigma(x, C_v)$. If π is given on-line, then the reporting is to be done on-line as well.

The problem which we are confronting is to preprocess a catalog graph G, along with its associated catalogs and ranges, so as to answer any multiple look-up query efficiently. If we do no preprocessing whatsoever, the catalog graph takes up $O(s)$ space, as previously observed. In order to answer a particular query, we look up x in each catalog along π. If this is done by using binary search in each catalog, the total reporting cost will be $O\left(\sum_{i=1}^{p} \log(|C_{v_i}|)\right)$, where the sum is over all vertices of π.

The strategy adopted by fractional cascading is to do only one binary search at the beginning, and then, as each vertex v of π is specified, locate x in C_v with an additional effort that only depends on d (the locally bounded degree). If for simplicity we assume that each catalog has the same size c, and that d is a constant, then fractional cascading reduces the query time from $O(p \log c)$ in the naive method to $O(p + \log c)$. Of course if the catalogs to be queried are unrelated, then knowing the position of x in one catalog might not help to locate it in its neighboring catalogs. So fractional cascading has to build auxiliary structures that correlate these catalogs.

One way to attain query time additive in $\log c$ and p is to merge all the catalogs into a master catalog M, and then for each catalog C to keep a correspondence dictionary between positions in C and positions in M. If we do this, we can look up x in M once and for all when a query is specified, and subsequently, for each vertex of π, simply follow the appropriate correspondence dictionary to locate x in the catalog of that vertex in constant time. Unfortunately the correspondence dictionaries altogether take up space $\Omega(n \sum_{v \in V} |C_v|)$, which is not $O(s)$. For example, in the special case considered above, the storage required grows from optimal $\Theta(nc)$ with the naive method, to $\Theta(n^2 c)$ when the master catalog is used. An important accomplishment of fractional cascading is that it attains the query time claimed while still keeping the overall storage linear.

A side remark is appropriate here: the reason the edges of G have been assigned ranges is to make fractional cascading more general and unifying. If G has bounded degree, however, the notion of ranges becomes irrelevant and the requirement "x must fall within the range of every edge of π" (Definition 3) can be dropped altogether, as each range can be taken to be $[-\infty, +\infty]$. The range enhancement is not gratuitous; it will come in very handy in some of the applications treated later on. Now, before embarking on a fairly long technical development, let us summarize our main result concerning fractional cascading, as will be proven in Sections 3 through 5.

Theorem S. *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 $O(p \log d + \log s)$. If d is a constant, this is optimal.*

So far we have dealt only with static catalogs. In many applications, however, allowing insertions and deletions of records into or from these catalogs is necessary. Thus Section 7 investigates how fractional cascading can be made dynamic. The results we have obtained there are less conclusive:

Theorem D. *The fractional cascading data structure can be made dynamic with the following bounds: If only insertions and look-ups are performed, the amortized time for each insertion can be $O(\log s)$, while the look-up cost remains the same as before. Here we are amortizing over a sequence of $O(s)$ insertions. The same bounds hold for deletions and look-ups only. If intermixed insertions and deletions are desired, then each of them can still be done in $O(\log s)$ amortized time, but the time required for a query increases to $O(p \log d \log \log s + \log s)$.*

For a discussion of amortized computational complexity the reader is referred to a paper by Tarjan [Ta].

2.2. The Fractional Cascading Data Structure

There are two key goals that the fractional cascading structure must accomplish: (1) somehow correlate each pair of neighboring catalogs in the catalog graph so a look-up in one of them aids the look-up in the other, and (2) keep the overall storage linear. The former goal suggests augmenting each catalog by introducing additional records borrowed from neighboring catalogs.

2.2.1. Bridges and Gaps

Each *original* catalog C_v will be enlarged with additional records to produce an *augmented* catalog A_v, which too will be a linear list of records whose values form a sorted multiset. Exactly how this is to be done is explained in Section 3. Here we will be content simply to describe the desired state of affairs after this augmentation has occurred. A related idea has been described in [VW]. Augmented catalogs for neighboring nodes in G will contain a number of records with common values. The corresponding pairs of records will be linked together to correlate locations in the two catalogs. More formally, for each node u and edge e connecting u with v in G we will maintain a list of *bridges* from u to v, D_{uv}, which will be an ordered subset of the records in A_v having values common to both A_u and A_v and lying in the range R_e; in particular, the endpoints of R_e are the first and last records in D_{uv}. We will have a symmetric situation with node v, where we maintain, for each bridge in D_{uv}, a *companion* bridge in D_{vu}. We call D_{uv} the *correspondence dictionary* from A_u to A_v. Remember that, in order to allow for the occasional presence of duplicates, we distinguish between a record of a catalog and its value. For example, D_{uv} and D_{vu} have no record in common, although they have the same set of values. A bridge is most usefully considered as a variant record in an augmented catalog pointing to a record with the same value in a neighboring augmented catalog. Bridges respect the ordering of equal-valued records, so they never "cross".

In order to disambiguate communication between catalogs of adjacent vertices, we add the requirement that each bridge should be associated with a *unique* edge of G. This means that

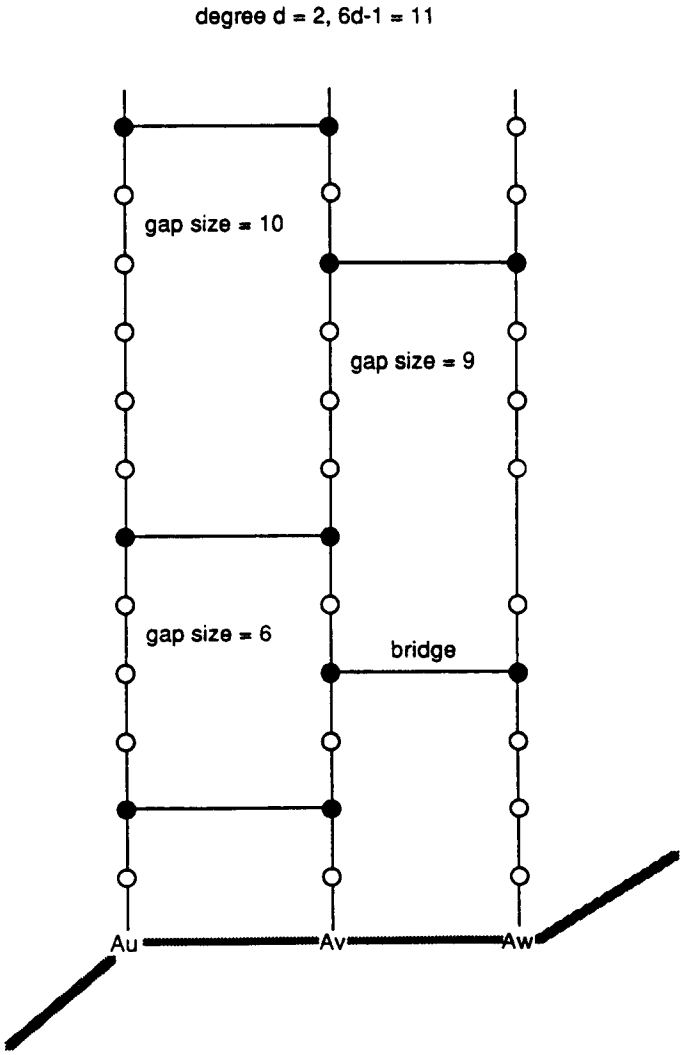


Figure 1. Bridges and gaps

if a given value in A_u is to be used to form a bridge in both D_{uv} and D_{uw} then it must be duplicated and stored in two separate records of A_u.

A pair of consecutive bridges associated with the same edge $e = (u, v)$ defines a *gap*. Let a_u and b_u be two consecutive bridges in D_{uv} and let a_v (resp. b_v) be the companion bridge of a_u (resp. b_u). Assume that b_u occurs after a_u in A_u. We form the *gap* of b_u by including into it each element of A_u positioned strictly between a_u and b_u and each element of A_v positioned strictly between a_v and b_v (a gap does *not* contain the bridges which define it). Note that the gap of b_u is the same as the gap of b_v. The element b_u (or b_v) is called the *upper bridge* of the gap. Except for the bridges formed by the endpoints of the range R_e, all other bridges associated with the edge e are both the upper bridge of some gap and the lower bridge of another. See figure 1. A key property of the structure built by fractional cascading is that gap size is kept small. This guarantees that the bridges correlating two adjacent catalogs are never too far apart. The particular constraint we maintain is:

The gap invariant: No gap can exceed $6d - 1$ in size.

We will see in Section 4 why the magic bound of $6d - 1$ has been chosen. Figure 1 illustrates the gaps and bridges of the augmented catalogs associated with three vertices on a single path. We end this subsection with some general remarks, before proceeding to the detailed description of the data structures needed for fractional cascading

The key to the design of the fractional cascading data structures is maintaining the correspondences between adjacent augmented catalogs, and between augmented catalogs and the associated original catalogs. The former facilitate the iterative search; the latter allow positions in the augmented catalogs to be translated into positions in the original catalogs. About the former correspondence and its implementation via bridges we will have a lot to say shortly in section 3.

Surprisingly, it is the latter correspondence, that between augmented and original catalogs, which becomes the bottleneck in the complexity when we need to deal with dynamic catalogs, where insertions and deletions are allowed. This is so because the records of C_v define an ordered partition of A_v into disjoint sets, each corresponding to a range of values between two successive records of C_v; by convention each such range contains its upper endpoint only. In the dynamization of the fractional cascading structures, we will need to implement insertions and deletions into both augmented and ordinary catalogs. While augmented catalog modifications clearly correspond to insertions/deletions of elements into one of the sets of the ordered partition, original catalog modifications give rise to splits and joins of adjacent sets in the partition. Thus we will need a data structure for handling the operations of *find* (what set contains a given element), *insert*, *delete*, *split*, and *join* in an ordered set partition. Maintaining the ordered set partition is an interesting data structure problem in its own right, which we will examine in Section 7.

For now we are confining our attention to building a static fractional cascading structure, so the correspondence between augmented and original catalogs can be finessed by just keeping, for each augmented catalog element, a separate pointer to indicate its successor in the associated original catalog. Formally, for a record r of an augmented catalog A_v with value x we define its *original successor* $\nu(r)$ to be $\sigma(x, C_v)$.

2.2.2. A Close-Up of the Data Structure

Original and augmented catalogs will be represented by linked-list structures. Each record in C_v consists of two one-word (used here in the generic sense of a unit of storage) fields (*key*,*up-pointer*). The *key* field contains the value of the record, while the *up-pointer* field refers to the record in C_v immediately following the current one in increasing order. The last record in this chain has a key of $+\infty$ and its pointer refers to NIL. The structure for A_v is more complex. It can be described as a doubly-linked list of records containing cross-references to the records in C_v and with additional information stored in nodes that are bridges.

A record in A_v consists of five fields; four of these are each one word long. We assume that a word is large enough to contain a key value, or a pointer to another record, or an integer count. The fifth field is a single bit used internally by the algorithms. More specifically, the fields for a record r are:

(1) *key*: stores the value K of r.

(2) *C-pointer*: holds a pointer to $\nu(r)$, the successor of r in C_v (thus giving us a constant-time implementation of the *find* operation above).

(3) *up-pointer*: points to the next element in A_v (or NIL if last).

(4) *down-pointer*: points to the previous element in A_v (or NIL if first).

(5) *flag-bit*: a bit used during construction or update of the structure.

Bridge records need to store more specialized information, so they have the following additional five fields. These are all one word long.

(6) *prev-bridge-pointer*: if r is a bridge in D_{vw}, then this field points to the previous (lesser value) bridge in D_{vw}. A NIL pointer is used to indicate that this record is the lower endpoint of a range.

(7) *companion-pointer*: points to the companion bridge.

(8) *edge*: if r is a bridge in D_{vw}, then this field stores the label of edge vw.

(9) *count*: This field stores the number of records in A_v that belong to the gap of which r is the upper bridge. Set to 0 for the lowest bridge in a correspondence dictionary. Its sum with the corresponding count field in the companion bridge gives the gap size of this bridge.

(10) *rank*: used internally in the construction phase and during updates.

Figure 2 illustrates the data structure on a small example. Note that, aside from catalog-related information, the structure also contains a full description of the graph G because the range endpoints become bridges providing the node adjacency information. In the next section we describe how to answer an incoming query; we postpone discussion of the construction of the data structure until Section 3.[1]

2.2.3. Answering a Multiple Look-Up Query

How do we proceed to answer a multiple look-up query (x, π)? The idea is to follow the generalized path π via the bridges provided in the data structure. Each time a search is performed in an augmented catalog A_v, the result of the look-up must be carried over to the associated original catalog C_v as well. The following lemmas provide the two basic primitives needed.

Lemma 1. *If we know the position of a value x in the augmented catalog A_v, in other words a record r with the smallest value greater than or equal to x, then we can compute the position (in the same sense) of x in C_v in exactly one step.*

Proof: Use the C-field of the record to retrieve $\nu(r)$. ∎

Lemma 2. *If we know the position of a value x in the augmented catalog A_v, and $e = (v, w)$ is an edge of G such that x is in the range R_e, then we can compute the position of x in A_w in $O(d)$ time.*

Proof: From the position of $r = \sigma(x, A_v)$ in A_v follow up-pointers until a bridge is found that connects to A_w. To do so, simply check the edge-field of every bridge visited. Because of the gap invariant, such a bridge will be found within $6d$ steps. At this point, follow the bridge-pointer and traverse A_w following down-pointers until x has been located. Again because of the gap invariant, both these traversals can be accomplished in at most $6d + 2$ comparisons. ∎

[1] A structure such as the above can easily be built by following the naive approach, mentioned earlier. Construct a master catalog M by merging all catalogs together and repeating each record as many times as the degree of the vertex it came from. Then make M the augmented catalog of each vertex. We can easily choose the bridges so that each gap has size at most $2d$. The interesting task ahead will be how to avoid the blow-up in storage which this simple-minded approach implies.

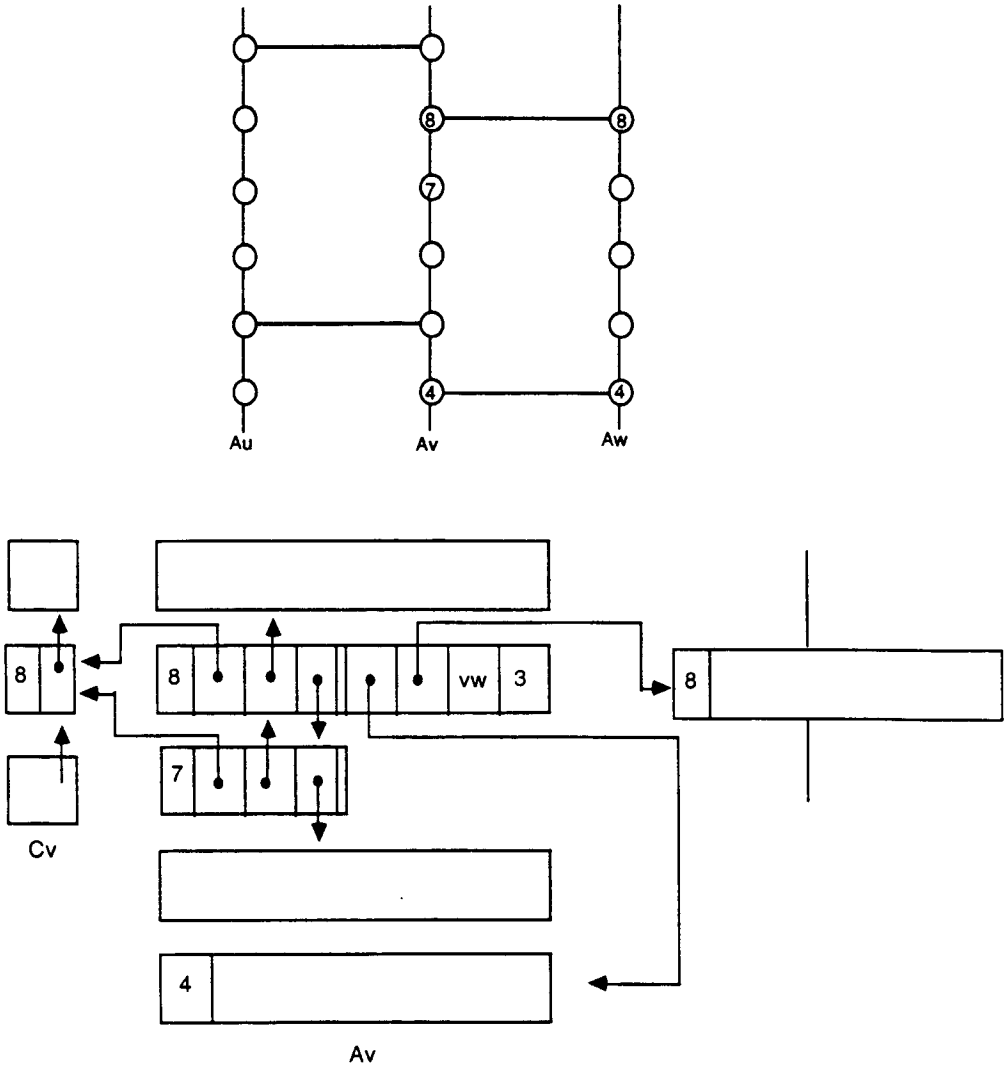


Figure 2. A close-up of the augmented catalogs

Lemmas 1 and 2 show that a multiple look-up query (x, π) can be answered very efficiently, provided that the position of x in A_f is known, where f is the first vertex in π. It is too early now to describe in detail how to compute the position of x in the initial catalog A_f. If we were to store A_v as a one-dimensional array as well, then we can certainly locate x in it in $O(\log s)$ time. However, this solution is rather inconsistent with our previous list-based structures. We will show in Section 6 that this initial search for x can be accomplished in $O(\log s)$ time using a technique which preserves the unity and simplicity of the data structure.

To summarize the situation at this point, we can handle any multiple look-up query satisfactorily, provided that the fractional cascading structures have already been built, and that efficient search is possible for the first augmented catalog to be considered. In the following section we show that the fractional cascading structure can be constructed in time $O(ds)$ and space $O(s)$. One remarkable feature of this data structure is that its size is independent of d. In the ensuing developments, d is considered a parameter and not a constant. It will therefore not disappear in the O-notation.

3. The Construction of the Fractional Cascading Structures

In order to add motivation to our discussion, we will start by describing an approach which, although flawed and ultimately inadequate, introduces the basic idea of fractional cascading in very simple terms. The reader who does not care for motivation at this point may skip the next paragraph.

Since this discussion is only for motivation, let us be concrete and assume that G is regular of degree d and each catalog C_v has size exactly c. Define a k-sample of a catalog C to be a maximal subcatalog of C obtained by taking values k apart; we call k the *sampling order*. Then A_v will be simply C_v, together with a $(2d)$-sample of each neighbor of v one away, a $(2d)^2$-sample of each neighbor two away, and so on. Here we are counting distances according to the underlying graph G. The size of A_v will be bounded by

$$c + d\frac{1}{2d}c + d^2\left(\frac{1}{2d}\right)^2 c + \cdots = 2c,$$

and thus the size of all the augmented catalogs is bounded by twice the size of the original catalogs. Any two adjacent nodes in G differ in their distances to a third node by at most ± 1. Therefore any two samples merged into the adjacent catalogs A_v and A_w may differ by a factor of at most $2d$ in the sampling order. This might make us hope that the gap invariant would also be satisfied. Unfortunately this is not necessarily the case, as the merge of two k-samples can leave gaps of size $2k$. It is this problem that makes the argument above only a heuristic and not a rigorous construction. To overcome this difficulty we must do the sampling in parallel with the construction of the augmented catalogs, as described in the sequel. Specifically, our plan will be to insert one new record at a time, maintaining the gap invariant as we go along. To ensure this, splitting some gaps into smaller gaps will occasionally be necessary. Although the time taken by a specific insertion is fairly unpredictable, the total running time of the algorithm can be made optimal with a careful implementation. Incidentally, the key idea of propagating geometrically decreasing samples of each catalog to nodes further away is responsible for the term "fractional cascading".

We now explain rigorously how, for every vertex v of G, the augmented catalogs A_v can be built efficiently. We will present the construction of the fractional cascading structures in an incremental fashion. By incremental we mean that we will show how to update these structures when a new record is added to one of the original catalogs. Starting then from a graph G with all catalogs empty, we can arrive at the desired state with repeated insertions.

The overall algorithm consists of two nested loops. For each vertex v of G in turn, we consider the elements of C_v in increasing order and insert them into A_v one at a time. Before inserting an element we make sure that *all* the gap invariants have been restored since the previous insertion. Note that even before any element of C_v has been inserted into A_v, this augmented catalog is already likely to contain elements originating from other catalogs. Therefore we must implement the insertion by merging C_v into A_v. Each insertion of a given element of C_v may cause serious changes in A_v, as well as in other augmented catalogs, necessitated by the restoration of gap invariants. The total cost of these operations, however, will be at most proportional to the final size of all the augmented catalogs.

3.1. Adding a New Record

We will partition the processing required when inserting a new record into three stages. In stage 1 we simply insert the new record r into the appropriate place in its augmented catalog A_v. After such an insertion we must update the count-fields of all gaps containing r and then split excessive gaps into smaller ones. These splits will cause additional insertions in neighboring catalogs, so count-fields must be checked again, and so forth. The counting of gap sizes and the splitting of excessive gaps constitute respectively stages 2 and 3. We may

need to loop around stages 2 and 3 several times, but eventually all gap invariants will be restored and this process will terminate. We now describe these operations in detail.

Stage 1: *Insert new record* — Let p be the next record from C_v to be inserted into A_v. Recall that p may possibly be the endpoint of a range. Let r' be the record from C_v previously inserted into A_v (or the first record of A_v, if none has been). Starting from r', follow the up-pointers of A_v until the correct position of p has been found. At this point, insert a copy r of p into A_v (breaking ties arbitrarily). The initialization of the first five fields is straightforward; the flag bit is set to 0. We also add a pointer to r into a set of newly inserted records, called the *count-queue*. When the previous insertion was fully processed, the count-queue became empty, so now its only element is r.

Stage 2 is invoked next to update the count-fields. In the general situation the count-queue will contain references to several new records created by the gap splitting process of stage 3.

Stage 2: *Update count fields* — Our task is to find all gaps containing each of the records referenced by the count-queue and update their count-fields. A simple solution consists of identifying these gaps, and then traversing each of them in order to evaluate their current size. The difficulty with this method is that gaps can grow to be very large and these repeated traversals can be costly. It is not so obvious how such a bad situation can arise, but appendix A shows that it really does. This forces us to use a cleverer method, which is described below.

We process the pointers in the count-queue twice. In the first traversal we identify the maximal groups of new records belonging to the same augmented catalog such that no two consecutive new records in the group are further than $6d$ apart. These groups are called *clusters*. Note that no gap can contain new elements in a given augmented catalog that come from more than one cluster. In the next traversal we visit each cluster and update the count-fields of the bridge records covered by the cluster. If some gap sizes have overflowed, then these gaps are added to the *wide-gap-queue*, which forms the input to stage 3. In more detail, the traversals work as follows:

First Traversal: For each reference to a new record in the count-queue go to that record and walk $6d-1$ steps down from it in its augmented catalog. In the process mark the $6d$ records thus visited by setting their flag bit to the value 1. In each augmented catalog the maximal runs of records with flag bits set to 1 define the clusters discussed above.

Second Traversal: Now visit every reference in the count-queue once more, this time removing each reference from the queue as it is processed. If a reference points to a record r, in (say) A_v, with its flag bit set to 0 then do nothing: the cluster of A_v in which that record belongs has already been taken care of. Otherwise we must process that cluster. As long as we see records with their flag bit set to 1, we walk down A_v from r to the last such record, or to the bottom of the catalog, whichever comes first. Let p denote the bottom record thus identified. We next walk up from p and in the process update the count fields of all bridges in A_v whose gaps contain new records in the cluster of r. We call this the *ranking process*.

The ranking process proceeds from p up to $6d$ steps past the last record encountered whose flag bit is set to 1. A running count of the records visited during the ascent is maintained, called the *rank*. We start out by giving p rank 1. Whenever we come to a bridge b we take a number of actions. First we store in the rank field of b the current value of this count. By following the prev-bridge-pointer of b to b' and looking at the rank field of that record, we can compute the number j of records from A_v currently belonging to the gap of b. Let i be the current value of the count-field of b, and k the count-field of the companion bridge of b. In general $j > i$ and the gap of b has increased in size from $i+k$ to at least $j+k$ ($j+k$ need not be the true size since the other side of the gap might not have been ranked yet). We now set the count-field of b to j and, if $j+k \geq 6d$

but $i + k < 6d$, then add the gap of b to the wide-gap queue for splitting during stage 3. The conditions above guarantee that a gap is added to the wide-gap queue only the first time a ranking process shows it has overflowed. Our last action in processing the bridge b is to set the rank of b' to 0. When the ranking process has reached its last record, the rank field of the last bridge encountered is also set to 0. In addition, the flag bit of each record visited in the process is set to 0—thus marking the cluster as processed.

At the end of stage 2 the count-queue is empty, all count fields of bridges are correct, and all gaps whose size exceeds $6d - 1$ have been placed in the wide-gap queue.

Stage 3: ***Restore gap invariants*** — If the wide-gap queue is not empty, remove its top element and split the gap of the upper bridge to which it points. To do so, merge all the elements of the gap into a temporary linked list. Let $K_1, \ldots, K_g$ be a labeling of this list in non-decreasing order, and let H_1 be the first group of $3d$ elements, H_2 the second group of $3d$ elements, etc. chosen from this list. Since the gap count g satisfies $g \geq 6d$, we have at least two groups, and more precisely $i = \lceil \frac{g}{3d} \rceil$ of them. If the last group H_i contains fewer than $3d$ elements, then we merge H_i and H_{i-1} together. Let j be the new number of groups ($j = i$ or $j = i - 1$). We separate H_1 from H_2 by making two copies of the largest element in H_1, each to become a bridge in the augmented catalogs associated with the gap. We then iterate on this process for the j groups, which leads to the introduction of $2(j - 1)$ bridges. All gaps produced have size exactly $3d$, except possibly for the last one, which has size $g - 3(j - 1)d \leq 6d - 1$.

Note that each partitioning element already occurs on one side of the gap. If it is not already a bridge to another neighbor on either side, then it need be duplicated only on the missing side. Otherwise it must be duplicated also on the side where it already occurs as a bridge, because of our convention that a record in an augmented catalog can only function as a bridge for a single edge. See figure 3 for an illustration of the splitting process. We omit the details of the initialization of the new records; we just mention that it is imperative to insert references to them into the count-queue.

At the end of stage 3 the wide-gap queue is empty and no gap has size exceeding $6d - 1$, according to the count fields present in the structure. All new records created from the splitting are referenced in the count-queue.

We now recapitulate the basic flow of operations. Stage 1 is called to insert a new key. At this point, stage 2 updates all count fields. Stage 3 is then called to restore the gap invariants. At termination, all gaps will have acceptable size, *if we discount the new elements that stage 3 has created.* To remedy this discrepancy, we call stage 2 again to obtain the list of flawed gaps. Stage 3 is then invoked to fix them, and the process iterates in this way until stage 2 fails to reveal any flawed gaps. It is important to ensure that stage 2 and stage 3 operate completely separately. All count fields must be correct before restoring any gap invariant and all gaps must be valid (up to the discrepancies caused by newcomers) before stage 2 is called again into action.

3.2. Proof of Correctness

Why is this process correct, and why should it always terminate? Let us leave termination aside for the time being; we first prove the two assertions made earlier: (1) after completion of stage 2, all count-fields are correct; (2) after completion of stage 3, no gap contains more than $6d - 1$ elements which were also in existence before.

We prove these assertions by induction. The second one follows directly from the description of the algorithm. Incidentally, note that after stage 3 has started, some splits may occur with a value of the count-field less than the correct one, because of earlier insertions caused by this stage. We now turn to stage 2. By the induction hypothesis (stating the correctness of the previous applications of stages 2 and 3), only the gaps containing the elements in

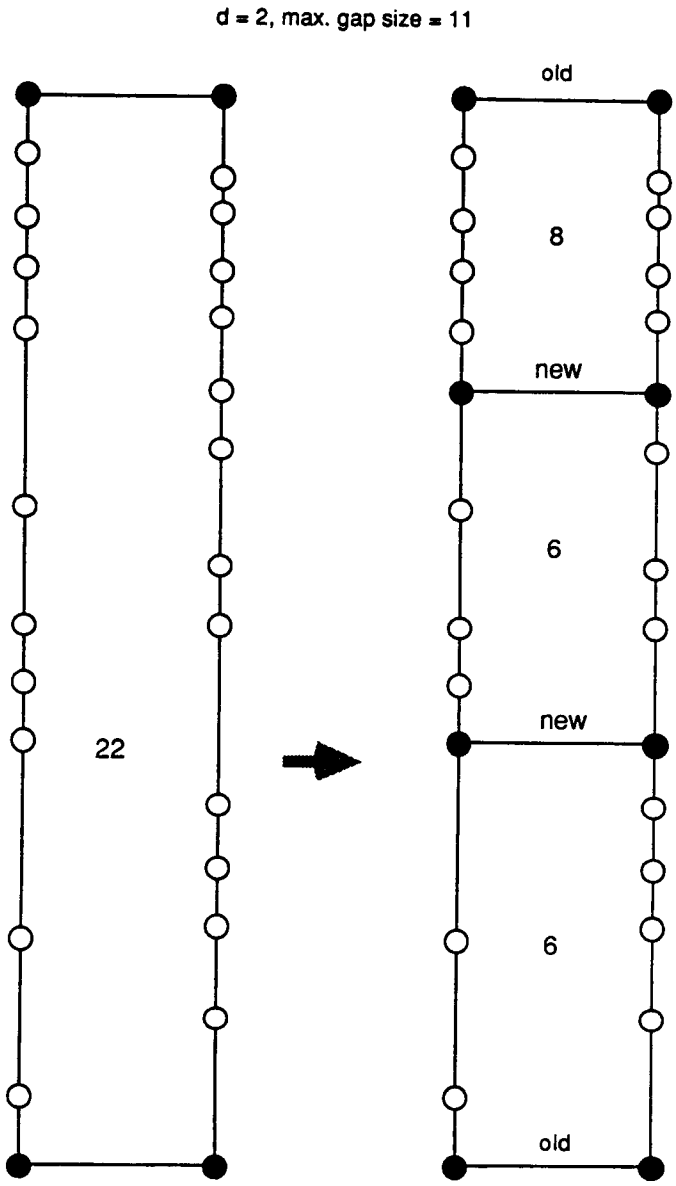


Figure 3. The gap splitting process

the count-queue need have their count-fields updated. We will first show that the updating performed in stage 2 correctly restores the counts of the gaps it touches, and then that all affected gaps are processed.

Let us concentrate our attention on the augmented catalog A_v. We call *new* any element on the count-queue just before stage 2; other elements will be called *old*. The key to the correctness of stage 2 is that no gap can contain more than $6d - 1$ *consecutive* old elements in A_v. Indeed, this would contradict the induction hypothesis that gaps were valid before the

introduction of new elements. Consequently, all new elements in A_v within a given gap must be linked together in stage 2 into one cluster, so the updating cannot miss any of them. The use of ranks is to identify exactly how many new elements lie in a given gap.

To see that the work in stage 2 is sufficient, we will prove that the algorithm does examine any gap which contains a new element. Let γ be a gap with upper bridge $K \in A_v$, and let K_u be the new element positioned highest in A_v such that K_u occurs within γ. K cannot lie more than $6d$ places above K_u in A_v (by the gap invariant), therefore K will be processed in that stage when the cluster of the new element K_u is handled. This completes the proof of correctness of our algorithm.

4. The Complexity of Fractional Cascading

4.1. Time Requirement

We now show that the insertion algorithm not only terminates, but that it does so with delay which is $O(d)$ when amortized over all insertions performed during the construction of the fractional cascading structures. In order to prove this bound we will use certain accounting techniques common in amortized complexity analysis [Ta]. The essence of those techniques is to associate "bank accounts" with parts of the data structure, into which deposits and withdrawals are made at appropriate instants during the execution of the algorithm. It is important to realize that these book-keeping operations are only an artifact of the analysis and not part of the algorithm proper.

Each gap has associated with it a ***piggy-bank*** holding some ***tokens***. A token can pay for a constant amount of computation (recall that $O(d)$ is not interpreted as "constant"). We choose this amount large enough so as to cover the actual cost in our implementation for any of the following: (1) creating a record for a new element or a new bridge and properly linking it into its augmented catalog (stages 1 and 3), (2) processing an element during the traversals designed to update the count fields in stage 2, and (3) processing an element in the merge preceding the gap splitting procedure of stage 3. Besides the piggy-banks, we have a ***cash-bank*** associated with each new element in the count-queue. We will make deposits or withdrawals from these banks in order to cover the ***restoration*** costs of an insertion: these are the costs associated with restoring the gap conditions. We will maintain the following invariant.

> Each gap of size k,$0 \leq k < 6d$, holds in its piggy-bank a number of tokens equal to at least $21 \max(0, k - 3d)$. Each new element contains $6d$ tokens in its cash-bank.

When an element K of C_v is to be inserted into A_v, it is given $27d + 1$ tokens. Twenty-one tokens go into the piggy-bank of each gap containing K. Since there are at most d such gaps, there are at least $6d + 1$ remaining tokens; K keeps $6d$ tokens for its own cash-bank, uses one token for the creation of its new record, and throws away the others. All bank conditions are then satisfied. Except for the double loop which performs the actual updating of the count fields, the time taken by stage 2 is clearly proportional to dN, where N is the number of new elements. As to the double loop, its time of execution is $O(dN + V)$, where V is the number of new elements visited during each count update. But because of the locally bounded degree condition, no new element can be examined more than d times. Therefore the total running time of stage 2 is still $O(dN)$. By our assumption about the token value, all this can be paid for with the $6d$ tokens from the cash-bank of each new element.

Processing each element G in the wide-gap queue during stage 3 takes time proportional to the size of the gap being split. Consider the new gaps produced during the splitting. We can distribute the tokens of the old piggy-bank of G into packets. The highest new gap H

is of size between $3d$ and $6d-1$; it receives a packet containing $21t$ tokens, where t is the excess of the size of H over $3d$. This packet supplies the (new) piggy-bank of H. Each of the other gaps can thus receive a packet containing $21 \times 3d = 63d$ tokens. Since, however, they all have size $3d$, their piggy-banks do not need any tokens at all. Now each new gap, except H, has to pay for the creation of one or two new bridges, as well as the necessary deposits to the piggy-banks of other gaps that the insertion of these bridges necessitates. Obviously at most $2(d-1)$ other gaps are affected. Thus we need the following: 2 tokens to create the two new bridge records; $2 \times 6d = 12d$ tokens to deposit into their cash-banks; and $21 \times 2(d-1) = 42d - 42$ tokens for deposits to other piggy-banks. Since we have a total of $63d$ tokens on hand, we can do all that and still have $9d + 40$ tokens left over.

We must still account for the work of splitting the gap G. If k denotes the size of G, then k tokens suffice to pay for splitting. Suppose that G is broken up into H, the highest gap of size between $3d$ and $6d-1$, and j other gaps, $j \geq 1$, of size exactly $3d$. By the analysis above each of the latter gaps has a surplus of $9d+40$ tokens, for a total of $(9d+40)j$. Since $(9d+40)j \geq 3jd + 6d - 1 \geq k$, we have enough to pay for the splitting out of the pooled surpluses.

In conclusion, the entire insertion process can be paid for with $(27d+1)s$ tokens (recall that s is the total catalog size) and therefore the preprocessing time of the algorithm is $O(ds)$. Next, we turn our attention to the storage utilized by the data structure.

4.2. Space Requirement

A space-token, or token for short, will buy 10 words of memory—that is, storage for one record in A_v. We take space tokens to be divisible units and divide each such token into d equal credits. We maintain the following invariant:

> At the completion of each stage, every gap of size g has at least $2\max(0, g-3d)$ credits in its (space) piggy-bank.

To handle an initial insertion (stage 1), we grant each new key three tokens. One of them covers the storage for the key itself. The other two tokens are exchanged for $2d$ credits: two of the credits are then deposited in the space piggy-bank of each containing gap; the remaining credits are thrown away. Note that this transaction preserves the piggy-bank invariant. To handle the gap splitting of stage 3, we use the packet argument of the previous section. This shows that each bridge of a newly created pair receives $6d/2 = 3d$ credits to use, after preserving all bank conditions. Two of them are deposited into the piggy-bank associated with each of the gaps containing the endpoints of the bridge. This still leaves at least $3d - 2(d-1) = d+2$ credits per bridge, which is more than one token, so the bridge can then pay for its own record. As a net result, only $3s$ tokens must be used to account for all the space used, so this space is $O(s)$. More precisely, only 30 words of memory are necessary per catalog element (on the average).

Theorem 1. *(Preliminary result) — Let G be a catalog graph of size s and locally bounded degree d. In $O(s)$ space and $O(ds)$ 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 $O(dp + \log s)$. If d is a constant, this is optimal.*

We conclude by remarking that in our $6d-1$ bound for the gap size invariant, the constant 6 can be reduced to $4+\varepsilon$, for any $\varepsilon > 0$. As it turns out, when ε goes to zero, the implied constants in our time and space analysis (in other words, the number of time or space tokens needed per insertion) go to infinity. Although the analysis breaks down for gap sizes less than

or equal to $4d$, the algorithms we have presented continue to work correctly. Figures 4 and 5 show two examples of fractional cascading structures on two simple graphs, where we in fact used $4d - 1$ as the maximum allowed gap size.

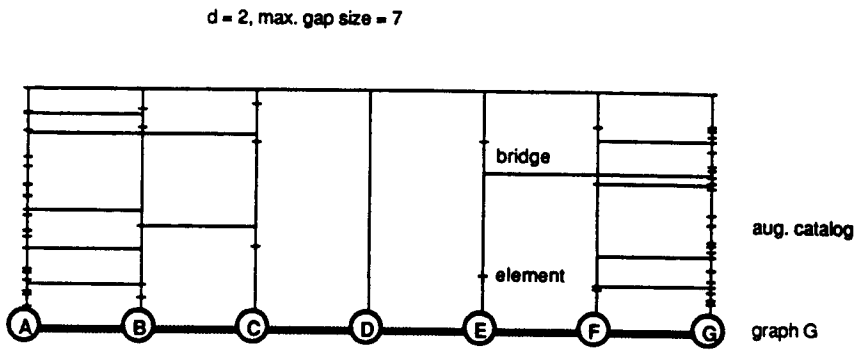


Figure 4. Example (1) of fractional cascading structure

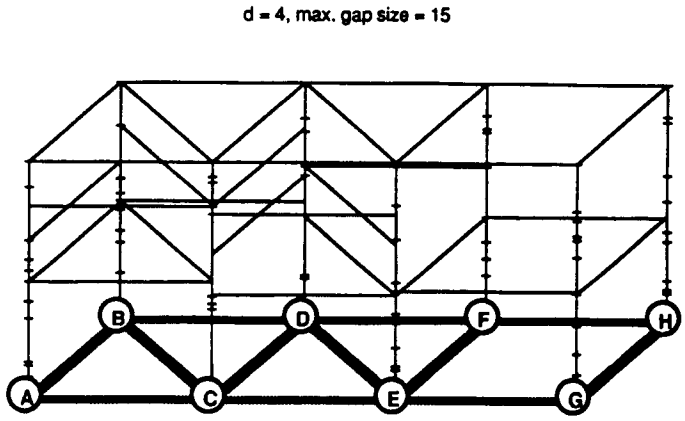


Figure 5. Example (2) of fractional cascading structure

Although our current result will be improved shortly, it is interesting in its own right because it does not attempt to modify the combinatorial nature of the graph. We will see in the next section that by rewriting the graph G in a canonical manner so that it has bounded degree, the preprocessing time can be reduced to $O(s)$, while the query time goes down to $O(p \log d + \log s)$ from $O(pd + \log s)$. In practice, on the other hand, d is most likely to be a small constant, so these asymptotic considerations are immaterial.

5. An Improved Implementation of Fractional Cascading

We have seen that the complexity of the query-answering process is proportional to the degree d. This is unavoidable given the approach taken here: the gap size must be proportional to the

degree if the overall storage is to remain linear. Through the medium of bridges, the query-answering process simulates a traversal of a graph of degree d represented by traditional adjacency lists. This means that in the worst case, to go from node v to its neighbor w, we may have to look at *all* d neighbors of v. To avoid this delay, we choose to resolve high degrees in the graph G by rewriting it in a canonical fashion. This will lead to a graph G^* of bounded degree which emulates G and allows us to go from a vertex v of G to a particular neighbor w in $O(\log d)$ time. Briefly, G^* is constructed by adding a small balanced tree at each node, called a *star-tree*. We solve the iterative search problem on G by applying fractional cascading to G^*, as described in the previous section.

Definition 3. A star-tree T_n is an oriented tree with n leaves (vertices of degree 1), endowed with a distinguished vertex called its *center*, and obtained inductively as follows.

(1) The tree T_1 is a single vertex which, of course, is also its center. The tree T_2 has two vertices connected by an edge; one of them is arbitrarily chosen to be the center.

(2) For $i > 2$, a T_i can be obtained from a T_{i-1} as follows: choose a leaf w of T_{i-1} which has minimum distance to the center of that tree. To form T_i attach two new edges to w and leave the center the same—see figure 6.

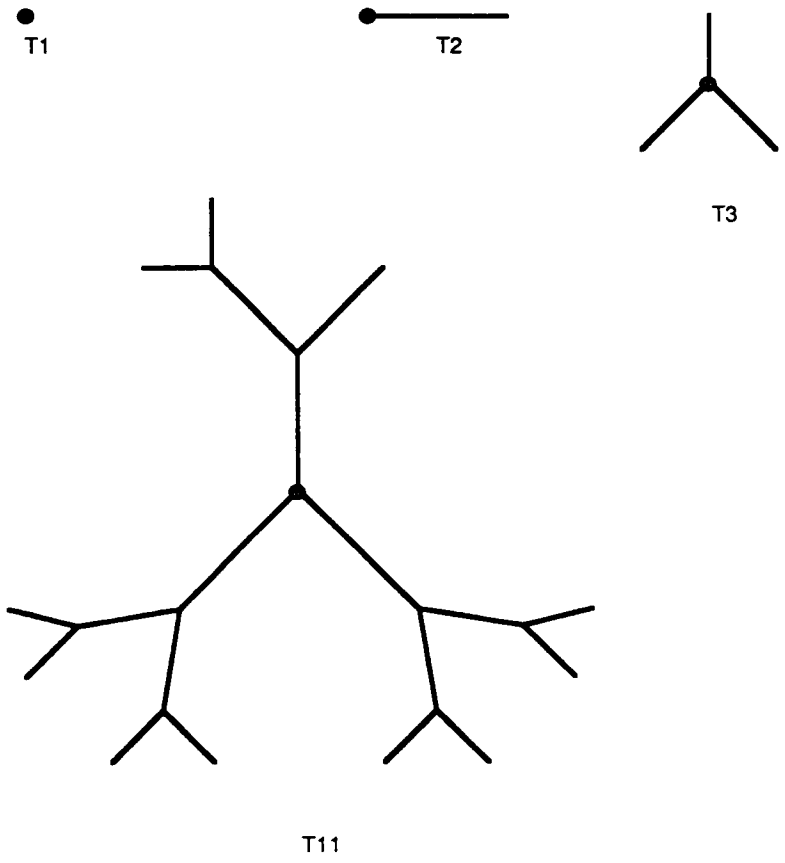


Figure 6. The star-trees used to resolve high degrees

Note that this definition is non-deterministic. In all cases, however, T_n has exactly n vertices of degree one, all interior vertices have degree three, and no vertex is at a distance greater

than $\lceil \lg n \rceil$ from the center.

Now let $e_1, \ldots, e_k$ be the edges of G adjacent to a vertex v of V. If $t_1, \ldots, t_d$ are the leaves of some T_d, we will attach each e_i to some t_j so that the leaves of T_d have a local degree of at most 2 in G^*. Computing the assignment is straightforward. At the outset, the index of each t_j is inserted into a *leaf-queue*. We also extract from C_v the $2k$ endpoints of $R_{e_1}, \ldots, R_{e_k}$ in sorted order (this does not require sorting, since these endpoints form a subset of C_v, which is itself assumed to be given in non-decreasing order). We perform the assignment by going through the endpoints in order, as follows. If the endpoint is a lower endpoint of R_{e_i}, remove any index j from the leaf-queue and assign edge e_i to leaf t_j. If the endpoint is an upper endpoint of R_{e_i}, re-insert back into the leaf queue the index l of the leaf t_l to which e_i had been previously assigned. Because of the locally bounded degree condition, the queue will always contain at least one label when one is needed. This whole process can be carried out in $O(k+d)$ time—see figure 7.

The graph G^* is obtained from G by replacing each node of G with a copy of T_d. (Actually, if a particular node of G has local degree $f < d$, a tree T_f could be used instead to save space). Each edge $e = (u, v)$ of G becomes an edge in G^* connecting the two leaves of the star-trees corresponding to u and v to which e has been assigned by the previous algorithm. In the star-tree T used to replace node u we assign empty catalogs to all nodes, except for the center to which we assign C_u. Also, each edge of T is given a range $[-\infty, +\infty]$. It is now easy to check that all the nodes of G^* have local degree bounded by 3. The graph G^* thus constructed has a number of edges proportional to m, the number of edges in G, and can clearly be built in time $O(s)$.

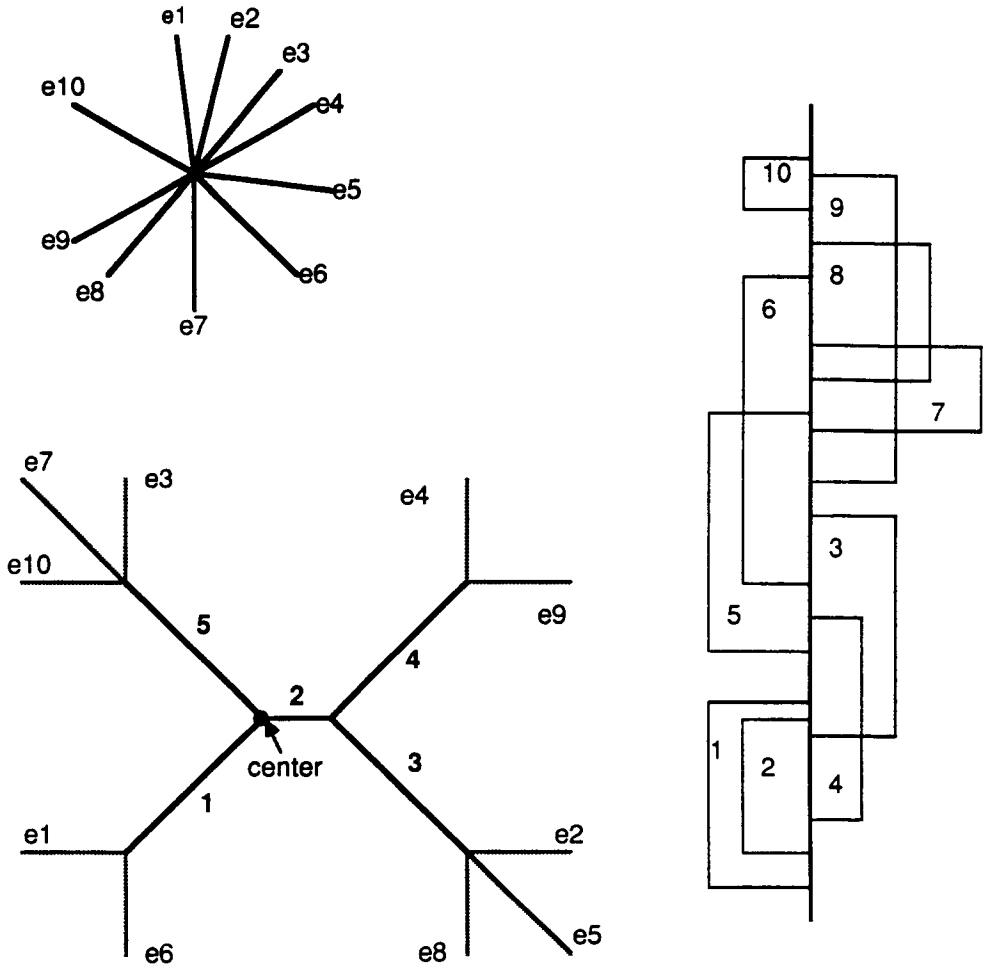
Searching for neighbors in G^* is trivial. Each tree T_d (or T_f) used in G^* has its edges labeled in a depth-first traversal. This allows us to go from one leaf to another in $O(\log d)$ time, provided that we know the labels of the starting and ending edges. All we have to do then is provide a correspondence table to translate the name of an edge in G into the local label of its new adjacent edges (see figure 7). Each edge of G will appear in at most two correspondence tables.

The emulation catalog graph is now ready for use. Note incidentally that the path of a star-tree between two of its leaves may avoid the center. Since the center *must* be visited in order to retrieve the desired information, we will fork the traversal into two paths: one going towards the center, the other pursuing its route towards the exit leaf. The emulation path is obviously still a generalized path. We conclude with an improved version of Theorem 1.

Theorem 2. *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 $O(p \log d + \log s)$. If d is a constant, this is optimal.*

6. The Notion of Gateways

We address here one of the points left open in previous sections: the location of a query value in the first catalog of the generalized path. The solution proposed earlier consisted of keeping a copy of each augmented catalog in a table, with the idea of performing a binary search in one of them in order to get a multiple look-up started. This is unsatisfying for at least two reasons. For one thing, the solution is inherently static and will support modifications only with great difficulty. Also, it breaks the unity of fractional cascading by stepping out of the list-based world in which we have (implicitly) pledged to remain.



Correspondence Table										
1	2	3	4	5	6	7	8	9	10	
1	3	5	4	3	1	5	3	4	5	

Figure 7. Using the star-tree T_{10}

The answer to these objections will be found in the notion of *gateways*. To each vertex v of G, attach an extra edge connecting v to a new vertex $g(v)$, called the *gateway* of v. The vertex $g(v)$ will have an augmented catalog attached to it but no catalog per se. The edge $(v, g(v))$ is called a *transit* edge; its range is $[-\infty, +\infty]$ — note that these definitions are made with respect to G and not its emulation graph G^*: the transition to G^* must come, as prescribed above, in a postprocessing phase and will ignore differences between edges and transit edges, etc. The augmented catalog of $g(v)$ is required to have exactly three elements. When the preprocessing takes place, if $A_{g(v)}$ should end up with less than three elements,

it is not created. On the other hand, if it ends up with more than three elements, another gateway is attached to it. This process might go on for a while, creating a chain of new vertices emanating from each vertex of G—see figure 8. Only the last vertex in the chain is called a gateway; all the others are called transit vertices.

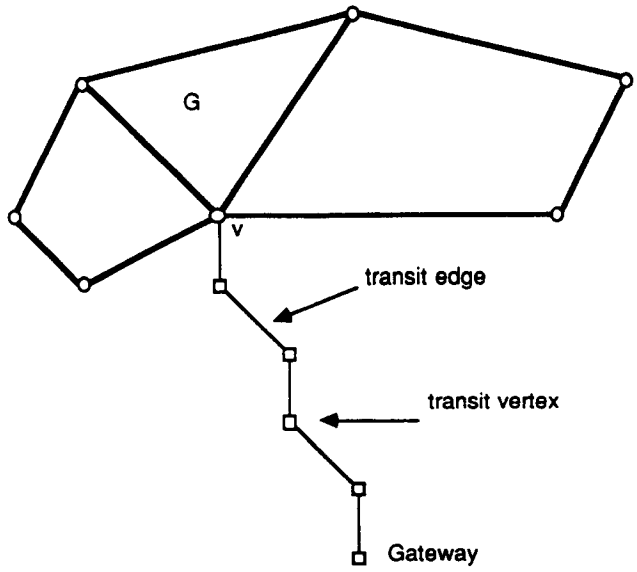


Figure 8. A gateway

It is clear that every time a new gateway is created, there are enough tokens around to pay for this creation. This is all the more obvious as the degree of a transit vertex is two. It is not hard to see that the length of a gateway will be roughly proportional to the logarithm of the size of the augmented catalog at its attachment to G. To answer a query we perform the initial search in A_v by starting at the gateway of v and proceeding to v. This will take $O(\log s)$ time.

As it turns out, a gateway chain is quite similar to a B-tree [K]. In a way it corresponds to a B-tree where all nodes at a given level have been combined into a supernode. The bridges between adjacent levels play the role of the inter-level links in the tree. The whole fractional cascading structure can be viewed as a generalization of B-trees. The upper and lower bounds that must be maintained on the gap size correspond naturally to the upper and lower bounds on the node size of a B-tree. The main difference, and one of the most intriguing aspects of fractional cascading as well, is that in the latter the gap splittings (or mergings) can cycle back to a node previously visited, and so go on for an unpredictable length of time.

7. Dynamic Fractional Cascading

We now examine how the fractional cascading structures can be made dynamic. When building the static structure as described in Section 3, we took advantage of inserting the keys present in each original catalog in increasing order. This sorting allowed us to use a simple linear scan to locate the position of each new key in the augmented catalog, and at the same time to set the value of the C-pointer of each augmented catalog element passed over. Both

the location problem, and the augmented-to-original correspondence problem are much more difficult in the dynamic case.

7.1. Insertions or Deletions Only

Let us first tackle insertions only. Suppose a new key K is to be inserted in C_v. We must (1) compute the position of K in A_v, (2) update whatever representation we are using to relate positions in A_v to positions in C_v, and (3) restore any gap invariants that have been violated by the new insertion. The similarity between gateways and B-trees makes dynamization a straightforward operation, at least as regards (1). Unfortunately, the static structure is grossly inadequate when it comes to problem (2). Too many C-pointers may need to be changed following a single insertion to allow any hope for a logarithmic update cost. In fact, what we must solve is an instance of the ordered set partition problem where we allow the operations *find*, *insert*, and *split*, as described in Section 2.2.1. The *find* operation replaces the C-pointers of the static structure, the *split* operation corresponds to the insertion of a new key in an original catalog, and the *insert* operation is used for the secondary insertions occasionally necessary for the restoration of the gap invariants. A recent paper by Imai and Asano [IA] has shown how to solve this particular case of the ordered set partition problem in constant amortized time per *insertion* or *split*, and constant actual time per *find*. The only assumption their argument requires is that we start with an empty structure. Finally problem (3) can be handled — for a change — exactly as in the static case. We must, however, use the Imai-Asano structure for the secondary insertions associated with stage 3 (the gap splitting). In conclusion:

Theorem 3. *If we allow only insertions, then fractional cascading can be made dynamic while preserving all the previous bounds for space, preprocessing, and query time. The cost of an insertion is $O(\log s)$ when amortized over a sequence of s insertions into an initially empty structure.*

It is possible to handle deletions (and only deletions) in a way analogous to that for insertions. The correspondence between augmented catalogs and original catalogs now requires a solution to the ordered set partition problem where the allowed operations are *merge*, *delete*, *insert*, and *find*. Although not explicitly stated as a result in their paper, Imai and Asano show in fact how to adapt the Gabow and Tarjan [GT] method to handle the first three of the operations above in constant amortized time and *find* in constant actual time. With deletions a new issue arises. It seems hopeless to try to eliminate all copies of an element being deleted from the structure at once. On the other hand, leaving these propagated copies lying around raises the possibility that the structure may no longer remain of linear size in the number of elements present in the original catalogs. But, as observed by Fries, Mehlhorn, and Näher, we can allay this fear if we simply impose a lower bound on the gap size as well [FMN].

Lemma 3. *If the minimum gap size is kept to γd for some $\gamma > 2$, then the size of the fractional cascading structure will be $O(\gamma s/(\gamma - 2))$.*

Proof:

Let (v, w) be an edge of G. We use lower case a's and c's to denote the size of the corresponding augmented and original catalogs, $b_{(v,w)}$ to designate the number of bridges between A_v and A_w, and $g_{(v,w)}$ to designate the number of elements in $A_v \cup A_w$ whose value falls in the range of (v, w). We then have $g_{(v,w)} \geq (b_{(v,w)} - 1)\gamma d + 2b_{(v,w)}$, and therefore $b_{(v,w)} \leq (g_{(v,w)} + \gamma d)/(\gamma d + 2)$.

Since we have made the convention that the original catalogs contain the endpoints of the ranges of their adjacent edges, we obtain $a_v \leq c_v + \sum_{(v,w)\in E}(b_{(v,w)} - 2)$. It follows that

$$\begin{aligned}\sum_{v\in V} a_v &\leq \sum_{v\in V} c_v + \sum_{v\in V}\sum_{(v,w)\in E}(b_{(v,w)} - 2)\\ &= s + 2\sum_{(v,w)\in E}(b_{(v,w)} - 2)\\ &\leq s + 2\sum_{(v,w)\in E}\left[\frac{g_{(v,w)} + \gamma d}{\gamma d + 2} - 2\right]\\ &\leq s + \frac{2d}{\gamma d + 2}\sum_{v\in V} a_v.\end{aligned}$$

From this the desired result follows. ∎

Maintaining both a lower and an upper bound on the gap size gets to be quite intricate. The accounting has to be modified to leave tokens in the piggy-banks for underflowing as well as overflowing gaps, following the method described by Fries, Mehlhorn, and Näher [FMN]. We will not give the details here, but simply state the end result.

Theorem 4. *If we allow only deletions, then fractional cascading can be made dynamic while preserving all the previous bounds for space, preprocessing, and query time. The cost of a deletion is $O(\log s)$ when amortized over a sequence of s deletions leading to an empty structure.*

Next, we will attack the general problem of handling *both* insertions and deletions at the same time. Instead of placing a lower bound on the gap size, we let the data structure degenerate gradually and rebuild it every now and then. The idea is just to mark the deleted elements, but not expend the effort to remove them right away from the structure. The obvious problem with this scheme is that since the data structure never decreases in size, it may become intolerably large compared to the number of *live* elements it contains after many deletions. To deal with this difficulty, we could stop the computation when the ratio of live elements to the total of those present drops below some threshold and re-insert every element still alive from scratch. But we now face the problem that although this scheme might have a good amortized performance, the occasional interruptions might be simply too long to be acceptable. Think for example of an on-line system where requests have to be handled immediately. The next section will bring an answer to this dilemma.

7.2. A General Scheme for Efficient Deletions

Consider a database reacting to three types of requests: insertions, deletions, and queries. Each insertion can be performed in ν amortized time, and each deletion can be recorded in δ actual time, where $\nu, \delta = O(1)$. The notion of recording a deletion as opposed to performing it is the following: an element can be marked off in δ time so that queries may go on and provide correct answers. Recording a deletion, however, does not free any storage, so it is not a viable alternative in the long run. To prove the following result, we use a dynamization technique originating in a paper by Bentley and Saxe [BSa], and one by Overmars [O].

Lemma 4. *Consider a data structure in which we can only insert new elements and answer queries. Let $M(s)$ be the storage used to store s elements, assumed polynomial in s, and let ν indicate the amortized time for an insertion, assumed to be constant. If the time δ*

to mark off an element to be deleted is also constant (thus ensuring that queries can still be answered correctly), it is then possible to implement each deletion in constant actual (non-amortized) time. The storage used is $O(M(s))$, and the time for inserting a new element or answering a query is the same as before, up to within a constant factor.

Proof: The idea, as mentioned earlier, is to "mark" the deleted elements as dead, then periodically garbage collect them to prevent the dead elements from swamping the live ones. Consider the situation at a generic time t. We always keep two identical copies of the data structure, so called the *query* copy and the *survival* copy. All requests are handled simultaneously (i.e., in a time-sharing fashion) in the two structures, except for queries, which are handled exclusively in the query structure. Recall that deletions are handled by simply recording the event, which will take a total of 2δ time. By keeping counters, we check that the live elements always outnumber the dead ones. As soon as this is not the case, we fork two concurrent processes, as described below; in the following, we let A denote the value of the two counters when they meet.

(1) Process 1 continues to handle all three types of requests in the query structure as though nothing was happening.

(2) Process 2 consists of two subprocesses, which can *pipe* information between them. Subprocess 2.1 will keep a transcript of all incoming requests during the entire lifetime of process 2. This includes all insertions and deletions but not queries. Subprocess 2.2 will go through three consecutive stages. In the first one, the subprocess re-inserts every element that is *alive* in the survival structure into a new survival structure. When this is done, the subprocess enters its second stage, where it makes a copy of the new survival structure; from then on the subprocess will work in double, performing the same operations in both copies of the new survival structure. We will not mention this duplication of effort in the following. In the third stage, the subprocess goes through the transcript maintained by subprocess 2.1 and starts responding to each request in chronological order. As soon as process 1 has spent $\delta A/2$ cycles in deletions and insertions (not counting queries), we complete the current request and immediately terminate all processes and subprocesses. The query structure is thrown away, and the two copies maintained by subprocess 2.2 become the query and survival structures. We will make sure that at this point the number of dead elements cannot exceed the number of live ones, so we are back to the initial conditions.

The idea is to have process 2 operate faster than process 1. First of all observe that after a while process 2 mimics process 1, although the duplicating task and the bookkeeping of subprocess 2.1 make this work about three times as hard. At any rate, giving a little more than three cycles to process 2 for every cycle of process 1 should be enough for process 2 eventually to catch up with process 1. However, this catching up should not be delayed too long or we may end up in a forbidden situation where more elements are dead than alive. Recall that from the moment processes 1 and 2 are triggered, no incoming deletion is effectively taken into account until the next process fork. [1]

We will show that setting the speed of process 2 to be $3 + \lceil 8\frac{\nu}{\delta} \rceil$ times the speed of process 1 satisfies all our conditions. Let I and D be respectively the number of insertions and

[1] Note also that the maintenance of multiple copies of the same structure makes the assumption that elements can be marked "dead" in constant time a bit tricky to implement. We cannot refer to an element to be deleted from both the copy and surival structures by a single pointer. We must instead access the element by naming an insertion or query operation record that referenced that element earlier. This "name" could, for example, be the serial number of the operation, which would be the same in the two structures.

deletions handled by process 1. We have

$$\nu I + \delta D \leq \delta \frac{A}{2}. \tag{1}$$

We must show that during these $\delta A/2$ cycles of process 1, process 2 has had time to go through its third stage and handle all I insertions and D deletions. The time necessary for these operations is

(1) *subprocess* 2.1: $I + D$ transcript operations which can be generously accomplished in $\nu I + \delta D$ time; these constants are chosen for convenience.

(2) *subprocess* 2.2 (stage 1): going through every element of the data structure cannot take more time than it would to rebuild it from scratch, so $2\nu A$ is an upper bound on the scanning time. Re-inserting the A elements alive will take νA time.

(3) *subprocess* 2.2 (stage 2): copying the data structure takes less time than rebuilding it, that is, at most νA cycles.

(4) *subprocess* 2.2 (stage 3): implementing the $I + D$ requests twice takes $2(\nu I + \delta D)$ time.

The total running time is dominated by $3(\nu I + \delta D) + 4\nu A$, which by (1) is less than $(\frac{3}{2}\delta + 4\nu)A$. This corresponds to a number of cycles in process 1 at most equal to

$$\frac{(\frac{3}{2}\delta + 4\nu)A}{3 + 8\frac{\nu}{\delta}} = \frac{\delta}{2}A.$$

Therefore, process 2 and its subprocesses will be complete when process 1 is. Note that during that time no more than $A/2$ requests for deletions can be accepted since process 1 lasts only $\delta A/2$ cycles. It follows that during the time the processes are active there are at least $A/2$ live elements and at most $A/2$ dead ones. Therefore the initial invariant is preserved: the dead count never exceeds the live count. Since the function $M(s)$ is polynomial in s, the space will be at all times proportional to what it could be at best, that is, $M(A/2)$. The proof is therefore complete. ∎

Lemma 4 provides a method for the general dynamization of fractional cascading. Whereas insertions are handled as usual, we use a lazy deletion mechanism to remove elements. This means ignoring deletions from augmented catalogs altogether, but reacting to deletion requests by just removing the appropriate elements from their catalogs. As in lemma 4 we will maintain a count of the elements alive and a count of those removed since the last cleanup operation. All the pieces of this process have been described above, except for the correspondence between original and augmented catalogs. Fries, Mehlhorn, and Näher [FMN] have shown how to modify the van Emde Boas priority queue [BKZ] so as to reduce the storage to linear and allow all five operations needed by the ordered set partition problem to be performed in time $O(\log\log s)$, where s is the total number of elements *present in the structure*. This implies that a fully dynamic version of fractional cascading is possible, but at the expense of increasing the cost per look-up to $\log\log s$ from constant:

Theorem 5. *If we allow both insertions and deletions, then a fractional cascading structure can be built whose size is $O(s)$ and where a multiple look-up along a generalized path of length p costs $O(p \log d \log\log s + \log s)$ in time. Both deletions and insertions can be handled in amortized time $O(\log s)$.*

8. General Remarks

In part II of this paper we give a large number of applications of fractional cascading to query problems. In fact, our discovery of this technique is due to noticing that tricks bearing a certain similarity had been used in a number of published algorithms [Ch1,Co,EGS] to deal with the problem of iterative search. Examples are the *hive-graph* of Chazelle [Ch1] and the chain refinement scheme of Edelsbrunner et al. [EGS]. These connections are developed more fully in part II.

The most unsatisfactory aspect of our treatment of fractional cascading is the handling of the dynamic situation. Is our method optimal? Whether it is or not, can it be simplified to the point of being useful in practice? Even in the insertion-only or deletion-only cases, our techniques are more of theoretical than practical interest, because of the large constants involved. We also feel that we do not fully understand the influence of high degree vertices in G on the method (see also [CG] for some additional comments on this). Can fractional cascading be applied to graphs, such as planar graphs, of bounded average degree — or does the presence of a small but non-constant number of high degree vertices really destroy the sampling/propagation?

We conclude by remarking that the philosophy of fractional cascading can be extended to other iterative search problems, beyond that of searching in a linearly ordered catalog. The three main requirements seem to be (1) that two search structures $\mathcal{A}$ and $\mathcal{B}$ can be merged into a joint structure efficiently (spec. in linear space), (2) that once the position of a "key" is known in the merged structure, its position in the component structures should be computable efficiently (spec. in constant time), and (3) that an appropriate notion of "sample" exist such that location of a key in the sample allows efficient (spec. constant time) location in the original. We hope that this paradigm will yield useful results in other areas as well.

Acknowledgments: We wish to thank Cynthia Hibbard, Ian Munro, Jorge Stolfi, and Robert Tarjan for many useful comments on the manuscript. We are also grateful to Marc Brown for programming a preliminary version of fractional cascading in the static case. The idea for the construction in the following appendix is due to Jorge Stolfi.

Appendix A. How gaps can get big.

It is possible to construct a catalog graph with any given degree d, $d \geq 3$, where insertion of a single record in one catalog ultimately propagates to many insertions into the same gap of another catalog. In the example we give below, we achieve secondary insertions into the same gap whose total number is $\Omega(s^\rho)$, where s is the size of our catalog graph and ρ is roughly $\log 2/\log(6d)$. We do not know if this is best possible. This example shows the need for the careful gap size counting we had to do in Section 3.1.

Our catalog graph will consist of two parts: a *multiplier* and a *concentrator*. The concentrator is just a linear chain of $k+1$ nodes, k to be determined later on. Across the last edge e_k of this chain there is only one gap (two bridges). This gap overlaps $6d-1$ gaps of the previous edge e_{k-1}; see figure 9 for an illustration. Now each of these gaps overlaps $6d-1$ gaps of the previous edge e_{k-2}, and so on. Therefore across the first edge e_1 there will be $(6d)^k+1$ bridges. The catalog of the first node contains enough additional records to bring all gaps across the first edge e_1 to saturation. The total size of this structure is $\Theta\big((6d)^k\big)$.

Consider what happens when we simultaneously insert one new record in each of the gaps of the first catalog. By simultaneously we mean during the same invocation of stage 3 as described in Section 3.1. All gaps of e_1 will split, causing $6d$ insertions into each gap of the second catalog. This will cause each gap over e_2 to overflow and reach size $12d-1$. In the

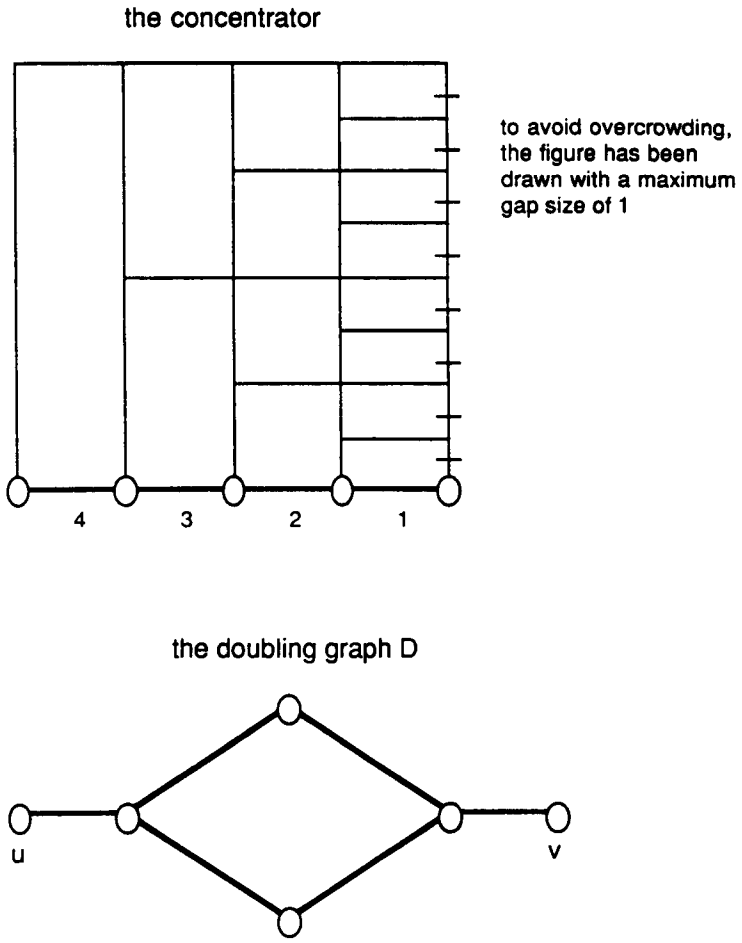


Figure 9. The concentrator and the multiplier

next iteration through stage 3 these gaps will split and will push *two* new records to be inserted into each gap of e_3 (because $12d - 1 = 3d + 3d + 6d - 1$). The gaps of e_3 now will reach a size of $16d - 1$ and will split the next time around, yielding *four* new insertions into each gap of e_4 (since $18d - 1 = 3d + 3d + 3d + 3d + 6d - 1$). This pattern continues, with the number of insertions into each gap doubling at each iteration. In the last splitting, 2^k secondary insertions will occur simultaneously in the one gap over e_k.

The job of the multiplier is to produce in the same stage all insertions needed to start the above process in the concentrator. It uses a doubling graph $D^{(1)}$ having an entry node u and an exit node v, and in addition four other nodes arranged as in figure 9. The catalog A_u has only two records, both bridges delimiting the same gap. The catalog A_v has three records, again all bridges delimiting two gaps. The catalogs of the other nodes are easily arranged so that an insertion into the gap of A_u causes an insertion into each of the gaps of A_v three stages later. We need a total of about $12d + O(1)$ records for this.

If we stack up m copies of these augmented catalogs on top of each other we obtain $D^{(m)}$, a graph where an insertion into each of the m gaps of A_u will cause an insertion into each of the $2m$ gaps of A_v. The multiplier is constructed by concatenating $D^{(1)}, D^{(2)}, \ldots, D^{(n)}$, where

n is chosen so that $2^n = (6d)^k + 1$. This compound graph produces the grouped insertions needed to feed the concentrator. The total size of our catalog graph is $O\big((6d)^k\big)$ and the number of insertions into a sigle gap it produces is 2^k. This proves the bound mentioned at the beginning if we fix k so that $s = \Theta\big((6d)^k\big)$ and thus completes our construction.

Fractional Cascading: II
Applications

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 word 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 then 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 part will be to justify the first part of this belief. From now on, we will assume that the reader is familiar with the basic terminology of fractional cascading, such as *iterative search*, *catalogs*, *multiple look-ups*, etc. For convenience, let's recall the main findings of part I.

> **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 $O(p \log d + \log s)$. 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 $O(\log s)$; the same holds for deletions. Arbitrary insertions and deletions can also be done in $O(\log s)$ amortized time, but the query time becomes $O(p \log d \log \log s + \log s)$.

How is this part 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 intersecting a line with a fixed polygonal path (Section 3), reporting points lying inside a trapezoidal region (Section 4) or a hyperrectangle (Section 5), performing range search in the past (Section 6), computing locus-functions (Section 7), compressing 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. Section 9 concerns a fairly general principle which best illustrates the power of fractional cascading.

In a standard query problem, call it Π, 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[D. We call the resulting problem an *IS-extension* of Π (iterative 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 IS-extensions with little or no degradation of performance. This touches on a central aspect of fractional cascading: its use as a postprocessing 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 *syntactical* rather than *semantic* 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 understood is explicit iterative search? We study this problem in the next section. In particular, we examine the sensitivity of fractional cascading to the presence of high degree vertices in the catalog graph.

2. Explicit Iterative Search

Let $S = \{C_1, \ldots, C_p\}$ be a collection of p catalogs, and let $s = \sum_{1 \leq i \leq p} |C_i|$ be the combined size of the catalogs. ***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 $C_i \cup \{+\infty\}$ 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 on p nodes, each associated with a distinct catalog: G is called an *emulation graph* of S. For convenience, 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 v of H, traverse the path from v 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 necessarily 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 v be the root of G; if v is not a node of H and has a single marked child w, then unmark v and iterate with respect to w, else stop. With T in hand, we can answer the query by performing multiple look-ups in the catalogs attached to the nodes of T.

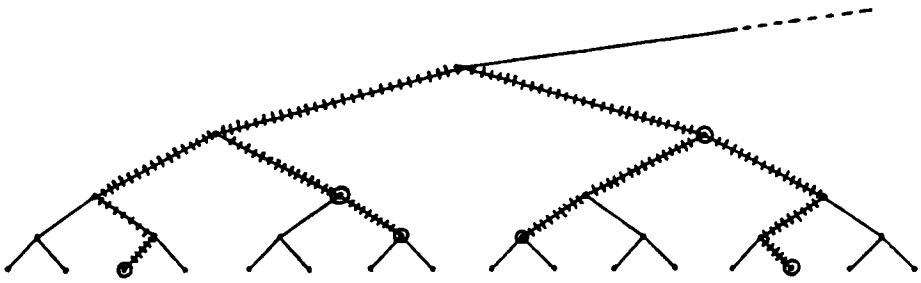


Figure 1. The emulation graph

A simple analysis shows that the time taken by the construction of T as well as the search in each catalog is $O(|T| + \log s)$. Let $v_1, \ldots, v_m$ be the vertices of H sorted by increasing inorder ranks. Let l_i be the lowest common ancestor of v_i and v_{i+1} $(1 \leq i < m)$, and let h_i denote the

number of ancestors of l_i in G. A rough analysis shows that $|T| \leq 2\sum_{1\leq i\leq m-1}(\log p - h_i)$. Since fewer than 2^j vertices among the l_i's can have fewer than j ancestors, we have

$$\sum_{1\leq i\leq m-1}(\log p - h_i) \leq \sum_{1\leq j\leq \lfloor \log m\rfloor} 2^j(\log p - j),$$

and therefore $|T| = O(m + m\log \frac{p}{m})$. We conclude that the running time of the algorithm is $O\big(|H|\log \frac{p}{|H|} + \log s\big)$.

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 fancier 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 v and w be defined as the number of edges on the shortest path between v and w. Let $l = \lfloor \log_d \frac{p}{m}\rfloor - 1$. Pick a vertex v in G 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 $H = \{v_1, \ldots, v_m\}$ 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 $\lfloor l/2\rfloor$ from v_i. Let p_i be the path in T between v_i and w_i. By construction of H, the paths $p_1, \ldots, p_m$ are vertex-disjoint, therefore the size of T is at least the added size of the p_i, that is, $\Omega\big(|H|\log \frac{p}{|H|}\big)$. ∎

Lemma 1 shows that our 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 with fractional cascading? What we will show is that no improvement can be expected in a pointer machine model [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 accesses, for some machine-dependent constant c. Therefore "at most c^t collections" is the answer. But there are $\binom{p}{m}$ possible sets H, so t must be at least on the order of $\log_c \binom{p}{m}$. As long as $m = o(p^{3/4})$, we have the elementary asymptotic formula

$$\binom{p}{m} = \frac{p^m e^{-\frac{m^2}{2p} - \frac{m^3}{6p^2}}}{m!}(1 + o(1)).$$

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

$$\frac{1}{\log c} m\log \frac{p}{m} + m\big(1 - \frac{1}{2p^{1/4}} - \frac{1}{6p^{1/2}}\big),$$

that is, $\Omega\left(m \log \frac{p}{m}\right)$. This shows that, at least for $m = o(p^{3/4})$, 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 Π be an explicit iterative search problem involving p catalogs of combined size s. There exists a data structure for solving Π such that any query can be answered in $O(m \log \frac{p}{m} + \log s)$ 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 naive 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 ($p = \Omega(m)$).

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 with 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 $Q = O(n)$. We desire a method with a query time of the form $Q = O(f(n) + k)$, where $f(n) = o(n)$ and k is the number of intersections reported. Using fractional cascading we are able to develop a technique that gives $Q = O\left((k+1) \log \frac{n}{k+1}\right)$. When k is a small constant, the running time is $O(\log n)$, which is optimal. When $k = \Omega(n)$, 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 $O(n \log n)$, but in the case where the polygonal path is *simple*, it can be reduced to $O(n)$. 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 $\log n$. 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 $\lceil n/2 \rceil$ edges. Then our algorithm is expressed very simply recursively as:

```
Intersect(P, ℓ)
  begin
  if |P| = 1 { single edge } then
        compute P ∩ ℓ directly
  else if ℓ does not intersect CH(P) then exit
  else
    begin
        Intersect(F(P), ℓ)
        Intersect(S(P), ℓ)
    end
  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 from left to right occur in the same order as in P). Interior nodes of the tree correspond in an obvious way to subpaths of P and store the convex hull of their respective subpaths (figure 2).

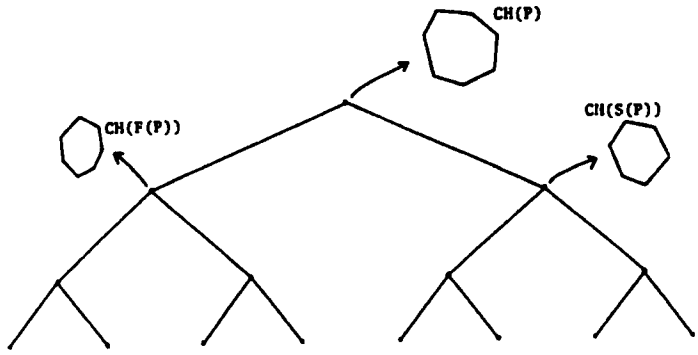


Figure 2. The convex hull decomposition

The tree T of convex hulls clearly takes $O(n \log n)$ 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

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

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

$$\Omega(\sum_i \log \frac{n}{2^i}) = \Omega(\log^2 n),$$

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 weaponry must be brought into the battle. This is where fractional cascading comes in.

The underlying tree T is a perfectly 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 $c_1, \ldots, c_m$ be the vertices of a convex polygon given in clockwise order, and let $c_i x$ be the horizontal ray emanating from c_i towards $x = +\infty$. We define the slope of an edge $c_i c_{i+1}$ as the angle $\angle(c_i x, c_i c_{i+1}) \in [0, 2\pi)$. It is well-known that since C is convex there exists a circular permutation of the edges of C such that the sequence of slopes is non-decreasing. 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 positions in the slope-sequence tell 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 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 just 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 $O(\log n +$ size of subtree of T actually visited) since, once we pass the root, we spend only constant effort per node visited. Our claimed query time bound of $O((k+1)\log \frac{n}{k+1})$ now follows from the following 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,*

$$|S| \leq k\lceil \log n \rceil - k\lfloor \log k \rfloor + 2k - 1.$$

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 $k(\lceil \log n \rceil - \lfloor \log k \rfloor)$. ∎

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 $O\big((k+1)\log \frac{n}{k+1}\big)$.*

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 $CH(P)$; the interior of this polygon is partitioned by the simple path P into a number of simply connected regions: $CH(P) \setminus P = \cup_i R_i$. 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 $CH(P) \setminus P$ (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 P is connected.

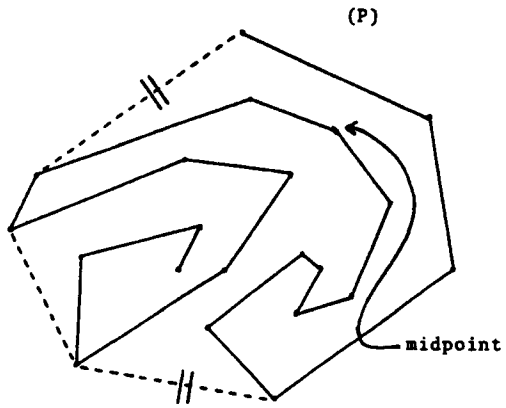


Figure 3. Sharing common tangents

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 R. In these cases an initial or final segment of P may occur on the boundary of R twice.

Now let x be the midpoint of P that is the vertex separating $F(P)$ 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 e, with e removed, is 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 e of $CH(P)$ 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 $n + 2(n-1) = 3n - 2$. 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 Lee 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 this edge appears, then all the edges stored at a particular node form a contiguous interval of the cycle of edges forming the convex hull of the node.*

Proof: The edges stored with a node v of T are exactly those which are not also edges of the parent of v in 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 lemma above implies 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, if ℓ intersects P in k edges, then these edges can be found and reported in time $O\big((k+1)\log \frac{n}{k+1}\big)$.*

4. Slanted Range Search

Let E^2 be the Euclidean plane endowed with a Cartesian system of axes (Ox, Oy). We will use the term *aligned rectangle* to refer to the Cartesian product $[a,b]\times[0,c]$, for some positive reals a, b, c. 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 [M2] 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 *slanted 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 "roof" 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 is $O(\log^2 n + k \log n)$, where k is, as usual, the number of points to be reported. Then we show how to improve this solution by application of fractional cascading.

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 *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

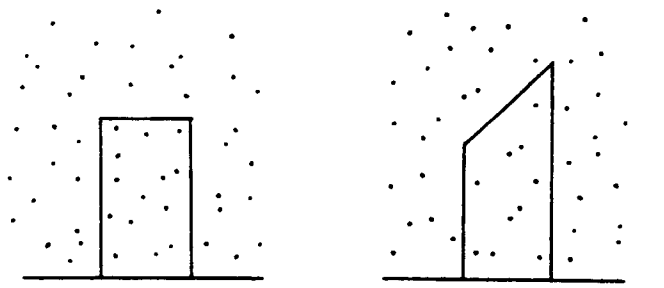


Figure 4. Two cases of range searching

distinct x-coordinates. If this is not the case, we store 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 *lower hull* of the point set V, denoted $L(V)$. If $a_1, \ldots, a_i, a_{i+1}, \ldots, a_j$ are the vertices of the convex hull of V, given in counterclockwise order with a_1 (resp. a_i) the point with minimum (resp. maximum) x-coordinate, $L(V)$ is defined as the sequence of points $a_1, \ldots, a_i$. 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 root v of a binary tree G. Let W be the points of V not in $L(V)$, and let $v.l$ and $v.r$ denote respectively the left and right children of node v. The data structure $D(v.l)$, associated with $v.l$, is defined as the sequence of points $L(W')$, where W' is the leftmost half of W. A data structure $D(v.r)$ 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: its size is trivially bounded above by n. The construction procedure is executed by calling $BUILD(V,\text{root})$.

Build(C, v)

begin
if $C = \phi$ **then** stop
$D(v) \leftarrow L(C)$
$W \leftarrow C \setminus L(C)$
Let α be the $\lceil \frac{|W|}{2} \rceil$th largest x-coordinate in W.
Build$\big(W \cap \{x \leq \alpha\}, l(v)\big)$
Build$\big(W \cap \{x > \alpha\}, r(v)\big)$
end

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

(1) Abs$(v) = \{x_1, \ldots, x_m\}$, the sorted list of x-coordinates in $D(v)$;

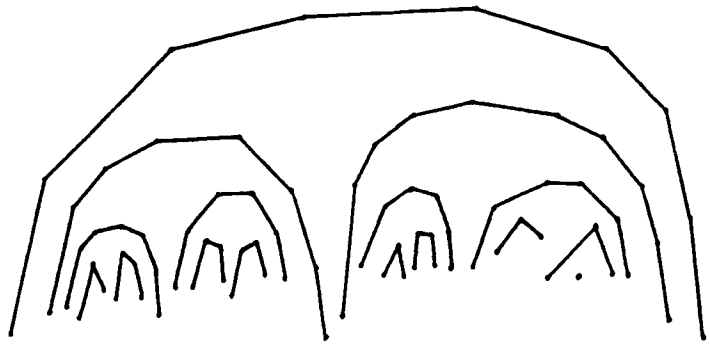


Figure 5. A tree of convex hulls

(2) Slope$(v) = \{\frac{y_2-y_1}{x_2-x_1}, \ldots, \frac{y_m-y_{m-1}}{x_m-x_{m-1}}\}$, the sorted list of edge-slopes in $D(v)$.

For explanatory purposes, we describe the query-answering process in two stages. A phase preliminary to the query-answering process 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 $L^- \cap D(v) = \phi$.

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 $(x_1, y_1), \ldots, (x_m, y_m)$. The query trapezoid $R = \{(a,0), (b,0), (a,c), (b,d)\}$ falls in one of three positions with respect to $D(v)$:

(1) $D(v)$ intersects the vertical segment $r_a = \{(a, y) | 0 \leq y \leq c\}$ at some edge $[(x_{i-1}, y_{i-1}), (x_i, y_i)]$: as long as the point (x_i, y_i) is defined and lies in R, report it and increment i by one. If $D(v)$ does not intersect r_a but intersects the segment $r_b = \{(b, y) | 0 \leq y \leq d\}$ at some edge $[(x_j, y_j), (x_{j+1}, y_{j+1})]$, then perform a similar sequence of operations. As long as the point (x_j, y_j) is defined and lies in R, report it and decrement j 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, that is, $x_m < a$ or $x_1 > b$: 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_i, y_i), 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_j, y_j) is defined and lies in R, report it and increment j by one. Next, re-initialize j to $i - 1$; as long as the point (x_j, y_j) is defined and lies in R, report it and decrement j by one.

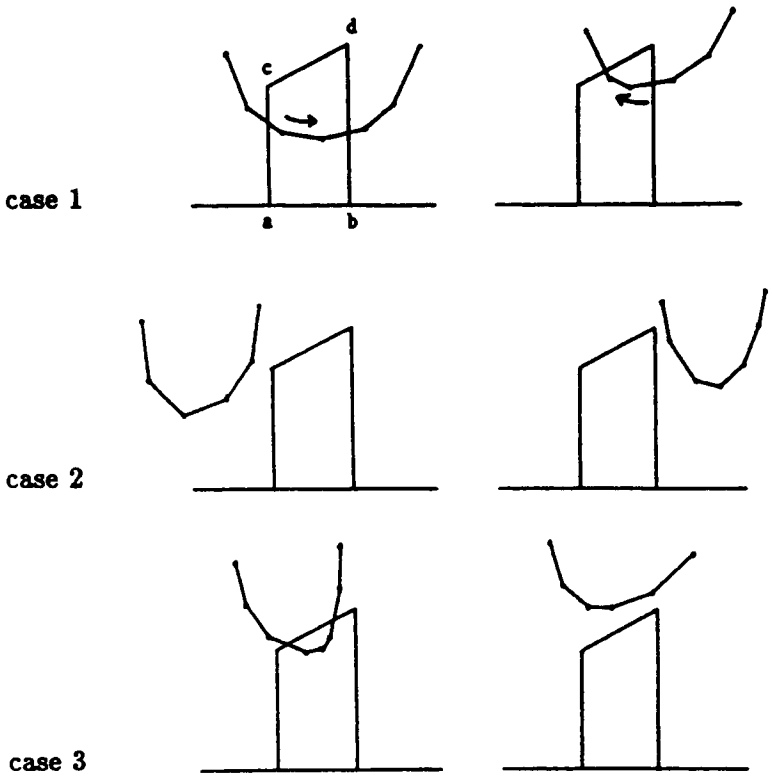


Figure 6. The various cases

(2) 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 Abs(v) with respect to a and b. In *Stage 2*, the L-peak of $D(v)$ is computed by performing a binary search in Slope(v) with respect to the slope of L.

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, all the points of $D(v) \cap V$ are 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., $U = \{v | D(v)$ contributes at least one point to $V \cap R\}$. Set

$$U_1 = \{v \in U | D(v) \cap r_a = \phi \text{ and } D(v) \cap r_b = \phi\}$$

and

$$U_2 = \{v \in G | D(v) \cap r_a \neq \phi \text{ or } D(v) \cap r_b \neq \phi\}.$$

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 $2(|U_1|+|U_2|)$. 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 $O(\log^2 n + k \log n)$, 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 we now have the stage set for fractional cascading. Visiting vertex v of G involves a binary search in either Abs(v) or Slope(v). The keys to be searched are a, b, 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

Let $\Re^d$ 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 $a_i \leq b_i$: we have

$$R = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d].$$

Let V be a set of n points in $\Re^d$. 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 $O(n \log^c n)$ space, for some constant c, and provide a response time of $O(\log n$+output size). As yet, such a data structure has been found only for the case $d = 2$ [Ch1,GBT,W]. 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. The desired result is approached through a series of subproblems in which each new subproblem builds on the previous one.

Subproblem P1: Let V be a set of n points in $\Re^2$ and let (Ox, Oy) be a Cartesian system of reference. Consider the problem of computing the set $V(a,b) = \{(x,y) \in V | x \leq a$ and $y \leq b\}$, 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 $[(x,y),(x,+\infty)]$, 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 $H = [(-\infty,b),(a,b)]$. We accomplish this task by using the *hive-graph* structure described by Chazelle [Ch1]. 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. Dashed lines correspond to added edges. Without going into the details of the structure, we must mention an essential feature of the query-answering process. To find the intersections between the rays and the segment s, the hive-graph will first 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.

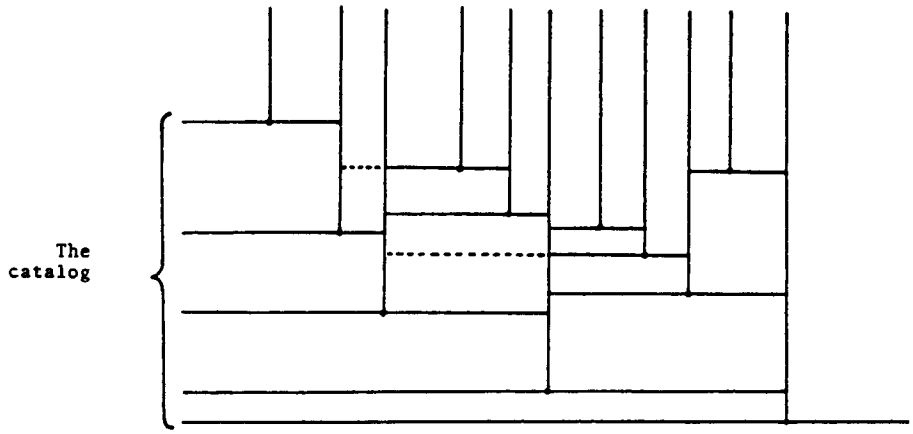


Figure 7. The hive-graph

Subproblem P2: Next, we turn to a restricted case of three-dimensional range search, one where the query R is of the form $[a_1, b_1] \times [0, b_2] \times [0, b_3]$ (figure 8.1). We say that subproblem *P2* is *based* on the two halfspaces $z \geq 0$ and $y \geq 0$.

Let V be a set of points $\{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}$, given by their coordinates in a Cartesian system of reference, (Ox, Oy, Oz). We use Bentley's notion of *range tree* [B] to reduce this problem to $O(\log n)$ instances of subproblem *P1*. In $O(n \log n)$ time, relabel the points of V so that $x_1 \leq x_2 \leq \ldots \leq x_n$, and set up a complete binary tree $\mathcal{T}$ whose n 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 $\mathcal{T}$ has a *key*, which we define as the x-coordinate of its associate point. We organize $\mathcal{T}$ 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 $\mathcal{T}$, let $U(v)$ be the subset of 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 tree, we can decompose R into a logarithmic number of canonical pieces. To do so, we search for a_1 and b_1 in $\mathcal{T}$. Let v_b be the leaf whose key is the successor of b_1. Symmetrically, consider the leaf whose key is the successor of a_1, and let v_a be its predecessor. For simplicity, we assume that all these nodes are well-defined (special cases can easily be integrated in a unified framework, but to preserve the continuity of the exposition, we will not attempt to do so). Let w_a and w_b 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 $\mathcal{T}$ that are either right children of nodes from v_a to w_a, or left children of nodes from v_b to w_b. Our original problem can be solved by solving it with respect to the point sets associated with the nodes of W. The benefit 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 $\{P(v) | v \in W\}$. Note that the two-dimensional query for *P1* is specified by the point (b_2, b_3) in the yz-plane. Straightforward analysis shows that the time to preprocess 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.

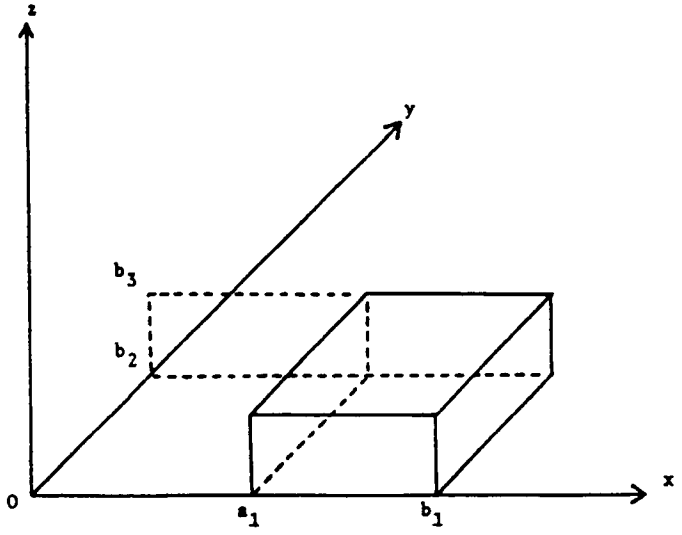


Figure 8.1

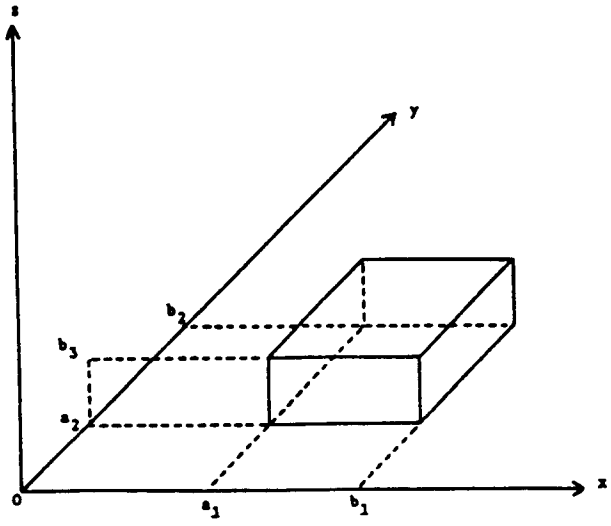


Figure 8.2

Figure 8. Reducing the dimensionality of the query

Subproblem P3: Next, we generalize subproblem *P2* by considering queries of the form $[a_1, b_1] \times [a_2, b_2] \times [0, b_3]$ (figure 8.2). *P3* is said to be *based* on the halfspace $z \geq 0$.

The same complete binary tree $\mathcal{T}$ 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 $v \in \mathcal{T}$, and let $d(v)$ be any number at least as large as any x-coordinate in $U(v_1)$ and at least as small as any in $U(v_2)$. Associate with v the data structure of *P2* defined with respect to $U(v_1)$ (resp. $U(v_2)$) and based on $(z \geq 0, x \leq d(v))$ (resp. $(z \geq 0, x \geq d(v))$). The data structure can be constructed in $O(n \log^3 n)$ time and requires $O(n \log^2 n)$ space. How do we answer a query? Starting at the root of $\mathcal{T}$, we compare $d(\text{root})$ against a_1 and a_2.

(1) If $a_1 \leq d(\text{root}) \leq a_2$, then we can apply the solution of *P2*, using the two data structures associated with v. The computation will thus be complete.

(2) If $d(\text{root}) < a_1$, then we iterate on this process by branching to the left child of the root.

(3) If $d(\text{root}) > a_2$, 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 *P2*. This brings the time complexity to $O(\log^2 n +$ output size).

Subproblem P4: We are ready to return to the original problem: R is now specified by two arbitrary points in $\Re^3$.

We modify the solution of *P3* in the obvious manner. Each node of $\mathcal{T}$ becomes associated with two data structures for solving not *P2* but, of course, *P3*. A similar analysis shows that the storage and preprocessing time leap up to $O(n \log^3 n)$ and $O(n \log^4 n)$, respectively. The response time remains $O(\log^2 n +$ 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 hive-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 $(+\infty)$. 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 $O(n \log^3 n)$ space and can be constructed in $O(n \log^4 n)$ 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 hive-graph [Ch1], 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 task, traditionally known as searching in the *past*, has already been well-researched [DM,O,Ch2,Co]. The question which we ask in this section, however, has not been addressed before. Briefly, it concerns the problem

of recording the previous states of a data structure for orthogonal range search. The question is not only of theoretical interest. Consider the case of a personnel database, where each point of V represents an employee's record. Coordinates indicate attributes such as sex, age, or salary. Over time, employees might be hired, fired, or simply have their records updated (not the first attribute, we hope). A query will then become a pair (R,t), with the meaning: report all points of V that were inside the d-range R at time t. The time range stretches from $-\infty$ to the present. The input is represented as a set V of n points in $\Re^d$; each point p is assigned an interval $[a_p, b_p]$ indicating its lifetime.

Our solution to this problem will be defined in two stages. For the time being, assume that $d = 2$ and disregard the notion of time. The query range R is the Cartesian product $[x_1, x_2] \times [y_1, y_2]$. Using Bentley's range tree [B], we can perform two-dimensional range searching in $O(n \log n)$ space and $O(k + \log^2 n)$ time, where k is the size of the output. The structure is similar to the one defined in the solution of subproblem *P2* (Section 5). As usual, each vertex v of $\mathcal{T}$ is associated with the set $U(v) \subseteq V$ formed by the leaves descending from v. The difference is that with each node v we associate a list $C(v)$ of the points in $U(v)$ sorted by increasing y-coordinates. 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 v, we compute the points of $C(v)$ whose y-coordinates fall between y_1 and y_2.

All this is very 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 $p = (p_x, p_y)$ will be represented now by a horizontal segment $[(a_p, p_y), (b_p, p_y)]$. Instead of searching the list $C(v)$, we must now report all the segments of $\{[(a_p, p_y), (b_p, p_y)] | p \in C(v)\}$ that intersect the vertical segment $[(t, y_1), (t, y_2)]$. To do so, we use a hive-graph. This allows us to find all desired t_v intersections in time $O(\log n + t_v)$. Constructing a hive-graph for p segments takes $O(p \log p)$ time and $O(p)$ space [Ch1], 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 $O(\log n + t_v)$ 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 [B].

Theorem 6. *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 Locus-Functions

Let V be a set of n 2-ranges in the Euclidean plane $\Re^2$, which we assume endowed with a Cartesian system of reference (Ox, Oy). 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

$$f : p \in \Re^2 \mapsto f(p) \in \{0, \ldots, n\},$$

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 $G : 2^V \mapsto \{0, \ldots, n\}$ is called *decomposable* if for any partition of a subset $X \subseteq V$ into Y and Z, $G(X)$ can be computed from $G(Y)$ and $G(Z)$ in constant time [BSa]. We restrict our attention to these so-called *locus-functions*. Let

$V(p)$ be the set of 2-ranges containing p; f is a locus-function if there exists a decomposable function G such that $f(p) = G(V(p))$, for any $p \in \Re^2$.

Note that the problem of computing $V(p)$, given any query point p, has been solved optimally in [Ch1]. 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 [BW]. We assume that the reader is familiar with this notion. Let $\{x_1, \ldots, x_{2n}\}$ be the x-coordinates of the 2-ranges of V, sorted in non-decreasing order. We construct a $(2n-1)$-leaf complete binary tree G, placing the ith leaf of G from the left in correspondence with the interval $[x_i, x_{i+1}]$. Each vertex v of G has a *span*, $I(v)$, 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 contain the span of v but not the span of v's parent. Vertex v is assigned a catalog $C(v)$ containing the y-coordinates, in sorted order, of all the 2-ranges in $\mathcal{R}(v)$. Note that each 2-range in $\mathcal{R}(v)$ contributes two entries to the catalog.

Let $p = (p_x, p_y)$ and let $f_v(p)$ denote the restriction of f to the subset of 2-ranges in $\mathcal{R}(v)$. Within the vertical slab $\{(x,y)|x \in I(v)\}$, $f_v(p)$ can be computed in $O(\log n)$ time by performing a binary search in $C(v)$ for the key p_y. 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 $\pi(p) = \{v \in G|p \in I(v)\}$ by performing a binary search in G for the key p_x. The value of $f(p)$ is obtained by combining together the partial answers $\{f_v(p)|v \in \pi(p)\}$, at a total cost of $O(\log^2 n)$ 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 $O(n \log n)$ 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 [B], we easily generalize this scheme to higher dimensions. All definitions, necessary facts, and algorithms are extended in a straightforward manner to the 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 n d-ranges. The structure G consists of $d-1$ levels of nested binary trees. Each vertex is adjacent to 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 $\{+\infty\}$. Each traversal of G clearly satisfies the connectivity requirement of fractional cascading; we conclude.

Theorem 7. *Given a set of n d-ranges in $\Re^d$, it is possible to compute any locus-function in $O(\log^{d-1} n)$ time, using a data structure of size $O(n \log^{d-1} n)$.*

8. A Space-Compression Scheme

Data structures such as segment-trees [BW] and range trees [B] 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 $\log\log n$, while increasing the query time by a factor $\log^{\epsilon} n$. We confess that this result is of 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 $\{x_1, \ldots, x_{2n}\}$ again be the x-coordinates of the 2-ranges of V, sorted in non-decreasing order, and let α be a positive integer. Construct a $(2n-1)$-leaf complete α-ary tree G by placing the ith leaf of G from the left in correspondence with the interval $[x_i, x_{i+1}]$. The *span* of vertex v is defined as before: $I(v)$ is the union of all intervals associated with leaves descending from v. As usual, $\mathcal{R}(v) \subseteq V$ designates the set of 2-ranges whose projections on the x-axis contain the span of v but not the span of v's parent. Unfortunately, storing all these sets is too expensive, so a redefinition of $\mathcal{R}(v)$ is in order. Let $v_1, \ldots, v_\alpha$ be the children of v from left to right and let R be any 2-range of V that appears in at least one $\mathcal{R}(v_k)$ $(k = 1, \ldots, \alpha)$. Note that the indices k such that $R \in \mathcal{R}(v_k)$ (if any) form a consecutive interval $[i, j]$. In general, we will have either $i = 1$ or $j = \alpha$. The inequalities $1 < i \leq j < \alpha$ 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 $\mathcal{R}_l(v_k), \mathcal{R}_r(v_k), \mathcal{R}_t(v_k)$ as follows:

(1) If $i = 1$, include R in $\mathcal{R}_l(v_j)$.

(2) If $j = \alpha$, include R in $\mathcal{R}_r(v_i)$.

(3) If $1 < i \leq j < \alpha$, include R in $\mathcal{R}_t(v_i), \ldots, \mathcal{R}_t(v_j)$.

It is easy to understand the whys and wherefores of this construction. Given the interval-like occurrences of R among brother vertices, the collection of sets $\mathcal{R}_l(v_k), \mathcal{R}_r(v_k), \mathcal{R}_t(v_k)$ provides an implicit representation of the collection of sets $\mathcal{R}(v_k)$. With each set $\mathcal{R}_*(v_k)$, we associate the catalog $C_*(v_k)$ 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 $O(n\frac{\log n}{\log \alpha})$ to the storage. The exception corresponds to the highest-level occurrences of R, which come in batches of at most α. Consequently, the data structure requires $O(n\frac{\log n}{\log \alpha} + \alpha n)$ space.

To answer a query $p = (p_x, p_y)$, 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, $v_1, \ldots, v_\alpha$. Let v_i be the vertex in question. For obvious reasons, $f_{v_i}(p)$ can be computed by searching for p_y in the catalogs $C_r(v_1), C_r(v_2), \ldots, C_r(v_i)$, and $C_l(v_i), C_l(v_{i+1}), \ldots, C_l(v_\alpha)$, and if $1 < i < \alpha$, also $C_t(v_i)$. This scheme yields an overall $O(\alpha\frac{\log^2 n}{\log \alpha})$ response time.

A standard binary representation of G allows us to apply fractional cascading (see Knuth [K], for example). Let $v_1, \ldots, v_\alpha$ be the children of v from left to right. We remove all pointers from v to $v_2, \ldots, v_k$, and replace them by pointers from v_i to v_{i+1}, for $i = 1, 2, \ldots, \alpha - 1$ (figure 9). To each node v, we now attach a little chain of three consecutive nodes, assigned to the catalogs $C_t(v)$, $C_r(v)$, and $C_l(v)$, 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 $O(\alpha\frac{\log n}{\log \alpha} + \log n)$. Setting $\alpha = \lfloor (\log n)^{\epsilon} \rfloor$, we obtain the following result.

Theorem 8. *Given a set of n 2-ranges in $\Re^2$ and any positive real ϵ, it is possible to compute a locus-function in $O(\log^{1+\epsilon} n)$ time, using $O(n\frac{\log n}{\log\log n})$ space.*

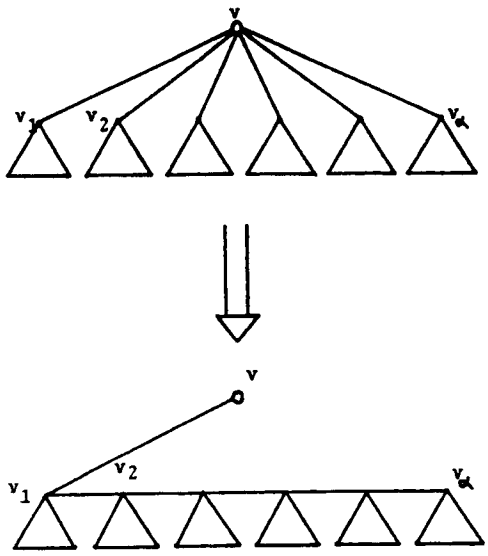


Figure 9. Putting each node in normal form

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 any further, we must formalize this notion of *additional request.*

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

$$g : q \in Q \mapsto g(q) \in 2^V; \quad g(q) = \{v \in V | P(v, q) \text{ is true } \}.$$

In the orthogonal range search problem, V is a set of points in $\Re^d$, q is a *d-range* and $g(q)$ is the set $V \cap q$. For any query problem Π we define an iterative search problem Π^*: each element $v \in V$ is associated with a distinct catalog $C(v)$ defined over a totally ordered set X. A query for Π^* is a pair (q, x) in $Q \times X$; the problem is to compute the successor of x in each catalog of $\{C(v) | v \in g(q)\}$.

Definition . Problem Π^* is called the *IS-extension* of problem Π.

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 $\mathcal{A}$ be an algorithm for problem Π; we say that $\mathcal{A}$ is an RRA if and only if:

(1) The underlying data structure of $\mathcal{A}$ is a graph G of bounded degree. Each vertex of G is associated with at most one element of V, but elements of V may appear in several vertices.

(2) The output of $\mathcal{A}$, i.e., $\{v \in V | P(v,q)$ 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 t corresponds a vertex $v(t) \in G$; for each $v(t)$ (except for at most a constant number of them) there exists an edge of the form $(v(t'), v(t))$ with $t' < t$.

(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 $v(t)$ 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 [Ch1,E,M1,M2].

(b) **Segment Intersection**: Given a set V of segments in the plane and a query segment q, report the segments of V that intersect q [Ch1,DE,EKM].

(c) **Point Enclosure**: Given a set V of d-ranges and a query point q in $\Re^d$, report the d-ranges of V that contain q [Ch1,E].

(d) **Orthogonal Range Search**: Given a set V of points in $\Re^d$ and a query d-range q, report the points of V that lie inside q [B,Ch3,GBT,M2,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 [Ch3,GBT].

(f) **Triangle Retrieval**: Given a set V of points in E^2 (resp. E^3) and a query triangle (resp. tetrahedron) q, report the points of V that lie within x [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^2 and a query of the form (q,k); $q \in E^2$, k integral ≥ 0, report the k points of V closest to q [CCP].

Retrieval reference 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 arithmetic operations. Sometimes, solutions to query problems do require address calculations to perform binary search 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 Π can always be generically 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 Π be a query problem defined over a set V of size p, and let $\mathcal{A}$ be an RRA for solving Π. Assume that $\mathcal{A}$ requires $O(f(p))$ space and has $O(g(p)+k)$ response time, where k is the size of the output. Let Π^* be the IS-extension of Π obtained by associating a catalog of size m with each element in V. Then there exists a data structure for solving Π^*, which requires $O(mf(p))$ space and $O(\log m + g(p) + k)$ response time.*

Proof: Let G be the graph used in modelling $\mathcal{A}$ as an *RRA*. To each vertex of G corresponds at most one element of V, hence one catalog (possibly reduced to $+\infty$ 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 $v(0)$; at any subsequent step $t > 0$ retrieve the relevant successor in the catalog associated with $v(t)$. ∎

Since both interval overlap and point enclosure can be solved in optimal space and time, so can their IS-extensions [Ch1]. 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 Π^* is also $O(f(n))$, where $n = pm$ is the size of the input. The naive algorithm for solving Π^* consists of applying $\mathcal{A}$ and looking up the search key x in each of the k catalogs found. This scheme uses only $O(n + f(p))$ 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 [Ch1,Co,EGS]. Interestingly, in all three cases, fractional cascading provides a unifying framework in which to understand these results. Let's take the case of the *hive-graph*, for example. We briefly recall this technique (see [Ch1] 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 [Ch1] was to show that by propagating only *every other* vertical segment, the size of the planar subdivision remains linear.

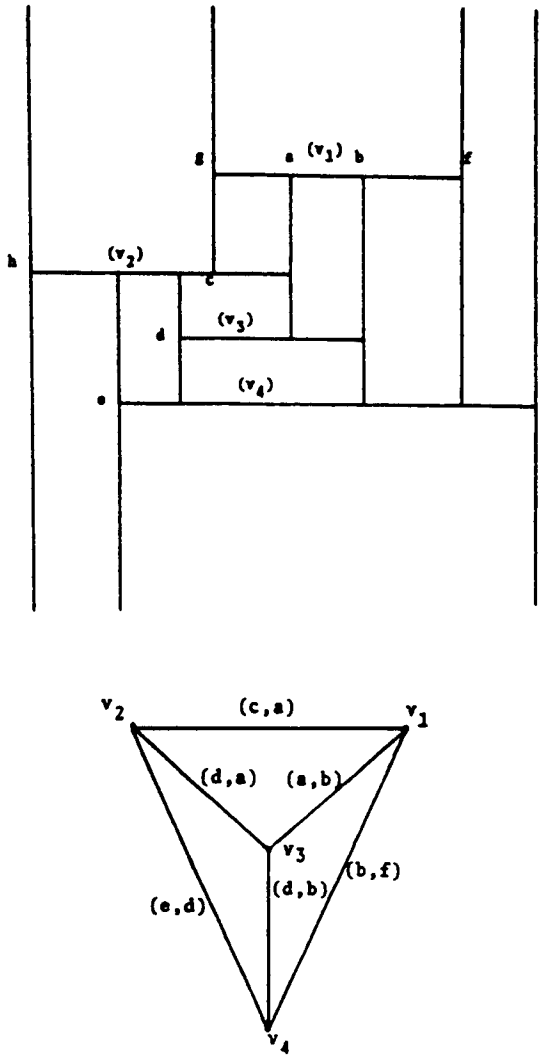


Figure 10. The base subdivision

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 [Ch1].

(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 $\Re^3$ 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 where 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 $\Theta(lm)$. 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.

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 Hibbard for her many suggestions that improved the exposition.

References

[B] Bentley, J.L. *Multidimensional divide-and-conquer*, Comm. ACM, 23, 4 (1980), 214–229.

[BKZ] van Emde Boas, P., Kaas, B., and Zijlstra, E. *Design and implementation of an efficient priority queue*, Math. Syst. Theory **10**, 1977, pp. 99–127.

[BSa] Bentley, J.L., Saxe, J.B. *Decomposable searching problems I: static to dynamic transformations*, J. of Algorithms 1 (1980), 301–358.

[BS] Bentley, J.L., Shamos, M.I. *A problem in multivariate statistics: Algorithms, data structures and applications*, Proc. 15th Allerton Conf. Comm., Contr., and Comp. (1977), 193–201.

[BW] Bentley, J.L., Wood, D. *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., Vol. C–29 (1980), 571–577.

[Ch1] Chazelle, B. *Filtering search: A new approach to query-answering*, Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 122–132. To appear in SIAM J. on Computing, 1986.

[Ch2] Chazelle, B. *How to search in history*, Information and Control, 1985.

[Ch3] Chazelle, B. *A functional approach to data structures and its use in multidimensional searching*, Brown Univ. Tech. Rept, CS–85–16, Sept. 1985 (preliminary version in 26th FOCS, 1985).

[CCP] Chazelle, B., Cole, R., Preparata, F.P., Yap, C.K. *New upper bounds for neighbor searching*, Tech. Rept. CS–84–11 (1984), Brown Univ.

[CE] Chazelle, B., Edelsbrunner, H. *Linear space data structures for a class of range search*, to appear in Proc. 2nd ACM Symposium on Computational geometry, 1986.

[CG] Chazelle, B., Guibas, L.J. *Visibility and intersection problems in plane geometry*, Proc. 1st ACM Symposium on Computational Geometry, Baltimore, MD, pp. 135–146, June 1985.

[CGL] Chazelle, B., Guibas, L.J., Lee, D.T. *The power of geometric duality*, BIT, 25 (1), 1985. Also, in Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 217–225.

[Co] Cole, R. *Searching and storing similar lists*, Tech. Report No. 88, Courant Inst., New York Univ. (Oct. 1983). To appear in J. Algorithms.

[CY] Cole, R., Yap, C.K. *Geometric retrieval problems*, Proc. 24th Ann. Symp. Found. Comp. Sci. (1983), 112–121.

[DE] Dobkin, D.P., Edelsbrunner, H. *Space searching for intersection objects*, Proc. 25th Ann. Symp. Found. Comp. Sci. (1984).

[DM] Dobkin, D.P., Munro, J.I. *Efficient uses of the past*, Proc. 21st Ann. Symp. Found. Comp. Sci. (1980), 200–206.

[E] Edelsbrunner, H. *Intersection problems in computational geometry*, Ph.D. Thesis, Tech. Report, Rep. 93, IIG, Univ. Graz, Austria (1982).

[EGS] Edelsbrunner, H., Guibas, L.J., Stolfi, J. *Optimal point location in a monotone subdivision*, to appear in SIAM J. Comp. Also DEC/SRC research report no. 2, 1984.

[EH] Edelsbrunner, H., Huber, F. *Dissecting sets of points in two and three dimensions*, forthcoming technical report, IIG, Univ. Graz, Austria, 1984.

[EKM] Edelsbrunner, H., Kirkpatrick, D.G. Maurer, H.A. *Polygonal intersection search*, Inform. Process. Lett. 14 (1982), 74–79.

[EW] Edelsbrunner, H., Welzl, E. *Halfplanar range search in linear space and $O(n^{0.695})$ query time*, Tech. Report, F–111, IIG, Univ. Graz, Austria (1983).

[FMN] Fries, O., Mehlhorn, K., and Näher, St. *Dynamization of geometric data structures*, Proc. 1st ACM Computational Geometry Symposium, 1985, pp. 168–176.

[GBT] Gabow, H.N., Bentley, J.L., Tarjan, R.E. *Scaling and related techniques for geometry problems*, Proc. 16th Ann. SIGACT Symp. (1984), 135–143.

[GT] Gabow, H. N., and Tarjan, R. E. *A linear-time algorithm for a special case of disjoint set union*, Proc. of 24-th FOCS Symposium, 1983, pp. 246–251.

[IA] Imai, H. and Asano, T. *Dynamic segment intersection search with applications*, Proc. of 25-th FOCS Symposium, 1984, pp. 393–402.

[K] Knuth, D.E. *The art of computer programmming, sorting and searching*, Vol. 3, Addison-Wesley, 1973.

[LP] Lee, D.T., Preparata, F.P. *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., Vol. 6, No. 3, pp. 594–606, Sept. 1977.

[M1] McCreight, E.M. *Efficient algorithms for enumerating intersecting intervals and rectangles*, Tech. Rep., Xerox PARC, CSL–80–9 (June 1980).

[M2] McCreight, E.M. *Priority search trees*, Tech. Rep., Xerox PARC, CSL–81–5 (1981).

[O] Overmars, M.H. *The design of dynamic data structures*, PhD Thesis, University of Utrecht, The Netherlands, 1983.

[PH] Preparata, F.P., Hong, S.J. *Convex hulls of finite sets of points in two and three dimensions*, Comm. ACM, vol 20, (1977), 87–93.

[Ta] Tarjan, R.E. *Amortized computational complexity*, SIAM J. on Comp., to appear.

[T] Tarjan, R.E. *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), 110–127.

[VW] Vaishani, V.K., and Wood, D. *Rectilinear line segment intersection, layered segment trees, and dynamization*, J. Algorithms, vol. 3, 1982, pp. 160–176.

[W] Willard, D.E. *New data structures for orthogonal queries*, to appear in SIAM J. Comput.

[Y] Yao, F.F. *A 3–space partition and its applications*, Proc. 15th Annual SIGACT Symp. (1983), 258–263.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Report #1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Report #2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Report #3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Report #4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Report #5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Report #6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Report #7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Report #8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Report #9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Report #10, May 1st, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Report #11, May 5, 1986.

by Bernard Chazelle and Leonidas J. Guibas

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301