Algorithmica (1986) 1: 133–162

Algorithmica
© 1986 Springer-Verlag New York Inc.

Fractional Cascading: I. A Data Structuring Technique[1]

Bernard Chazelle[2] and Leonidas J. Guibas[3,4]

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 this paper 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. A companion paper gives numerous applications of this technique to geometric problems.

Key Words. Binary search, B-tree, Iterative search, Multiple look-up, Range query, Dynamization of data structures.

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 [CG].

[1] The first author 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.
[2] Brown University and Ecole Normale Supérieure.
[3] DEC/SRC and Stanford University.
[4] Contact author's address: Leonidas J. Guibas, DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA.

Received August 30, 1985; revised February 18, 1986. Communicated by C. K. Wong.

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 [CG] amply demonstrates, iterative search is a fundamental component of many query-answering algorithms. Let us take Problem 3, 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 [C] 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 $\mathfrak{R} \cup \{-\infty, +\infty\}$. The records are stored in the catalog in nondecreasing 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.[5] 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 values 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 a string of 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 $x \in \mathfrak{R}$ 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 $j < i$.

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.

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

DEFINITION 3. A *multiple look-up query* is a pair (x, π), where x is a key value in $\mathfrak{R}$ 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, 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 $O(\sum_{i=1}^{p} \log(|C_{v_i}|))$, 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 such 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 catlog 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 proved 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 only dealt 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 see [T].

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 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

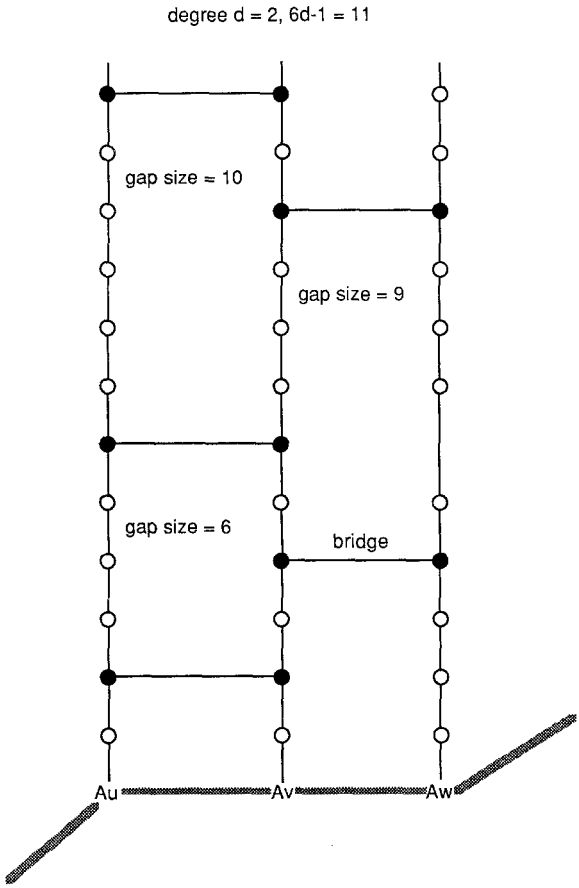

Fig 1. Gaps and bridges.

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 agumented 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 one word long each. 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.[6]

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

[6] 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.

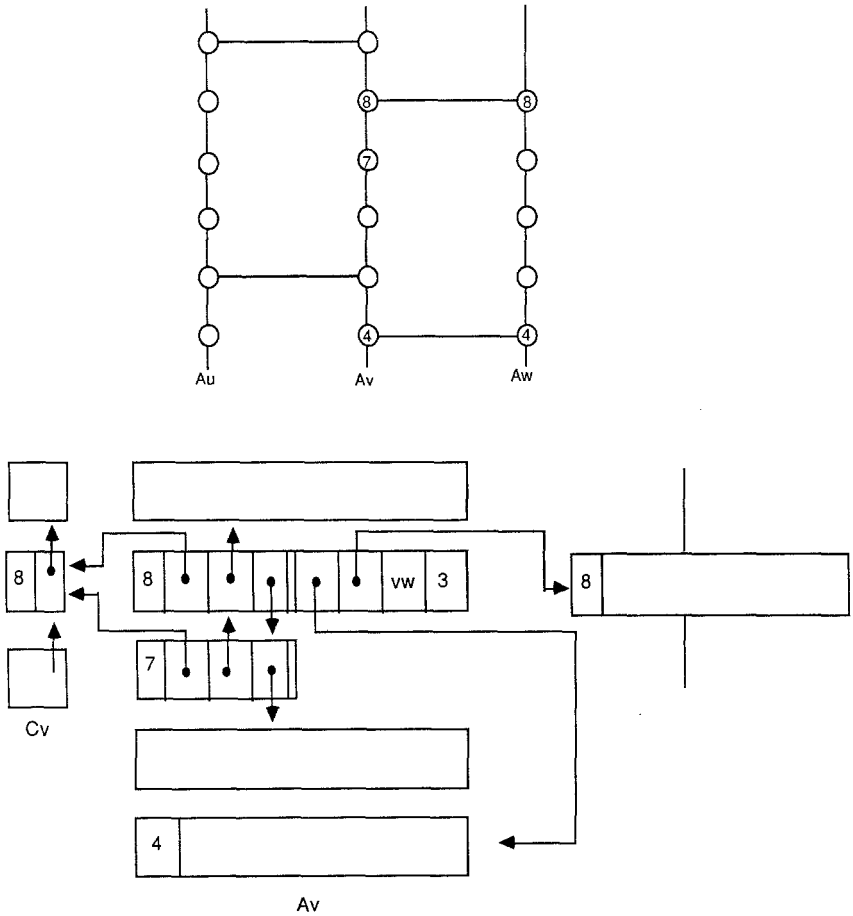

Fig. 2. A close-up of the augmented catalogs.

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. □

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 need not be 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 catlogs, 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 specified 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 details, 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 above conditions 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 nondecreasing 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 $i = \lceil g/3d \rceil$ 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

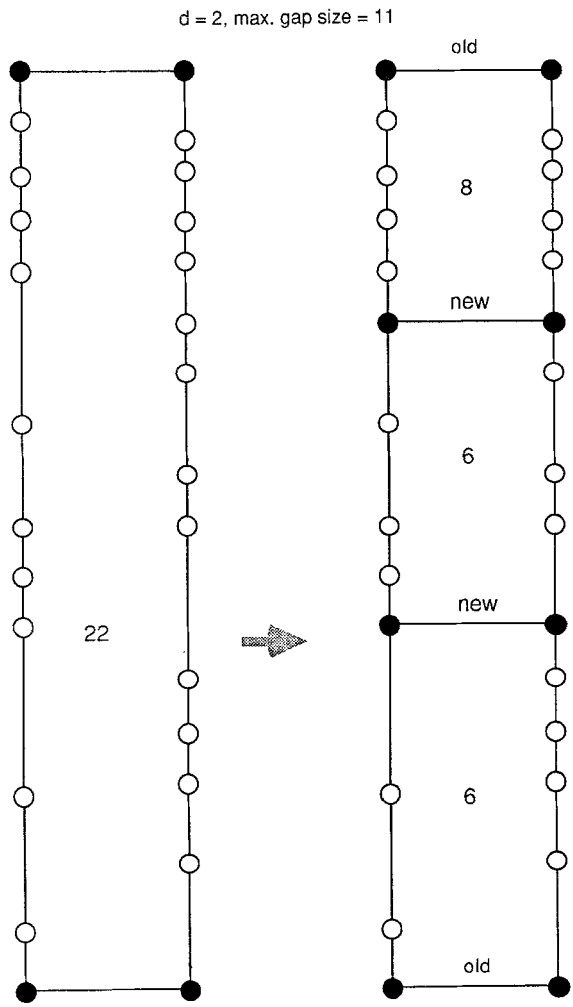

Fig. 3. The gap splitting process.

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, not 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 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 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 [T]. 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 contain 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 now 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.

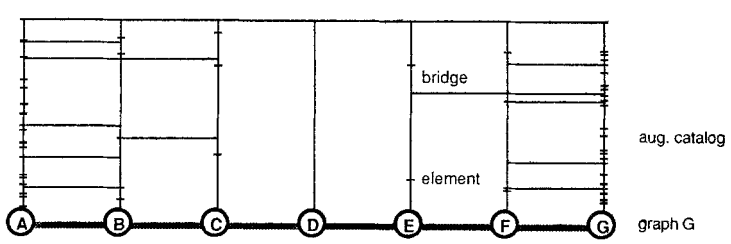

Fig. 4. Example (1) of fractional cascading.

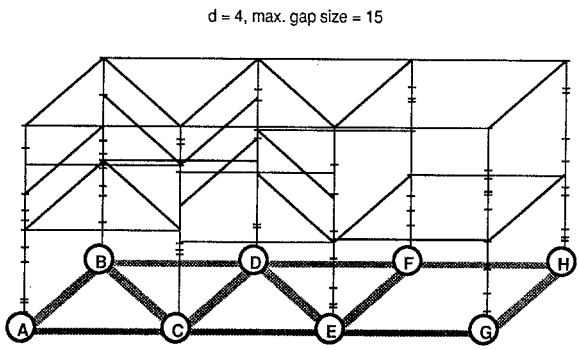

Fig. 5. Example (2) of fractional cascading.

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 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 vetex 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 4. 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.

Note that this definition is non-deterministic. In all cases, however, T_n has exactly

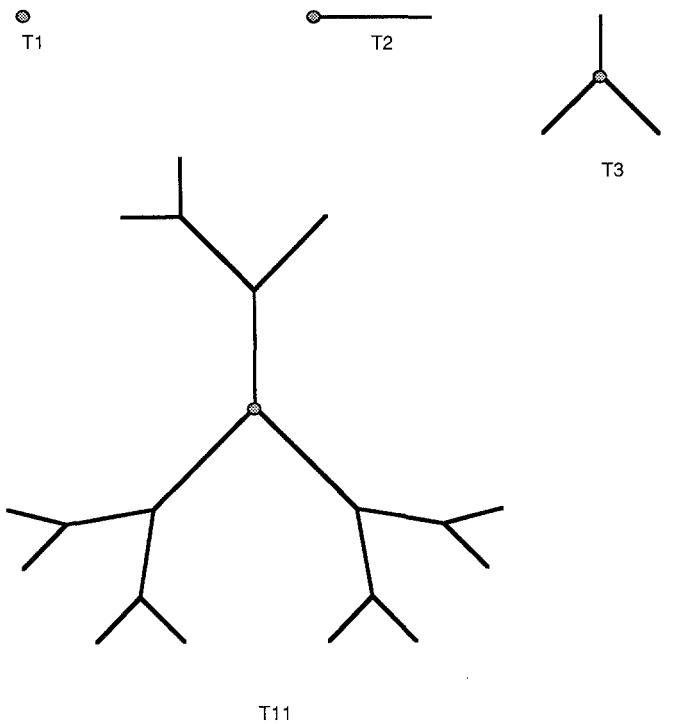

Fig. 6. The star-trees used to resolve high degrees.

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. 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 nondecreasing 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

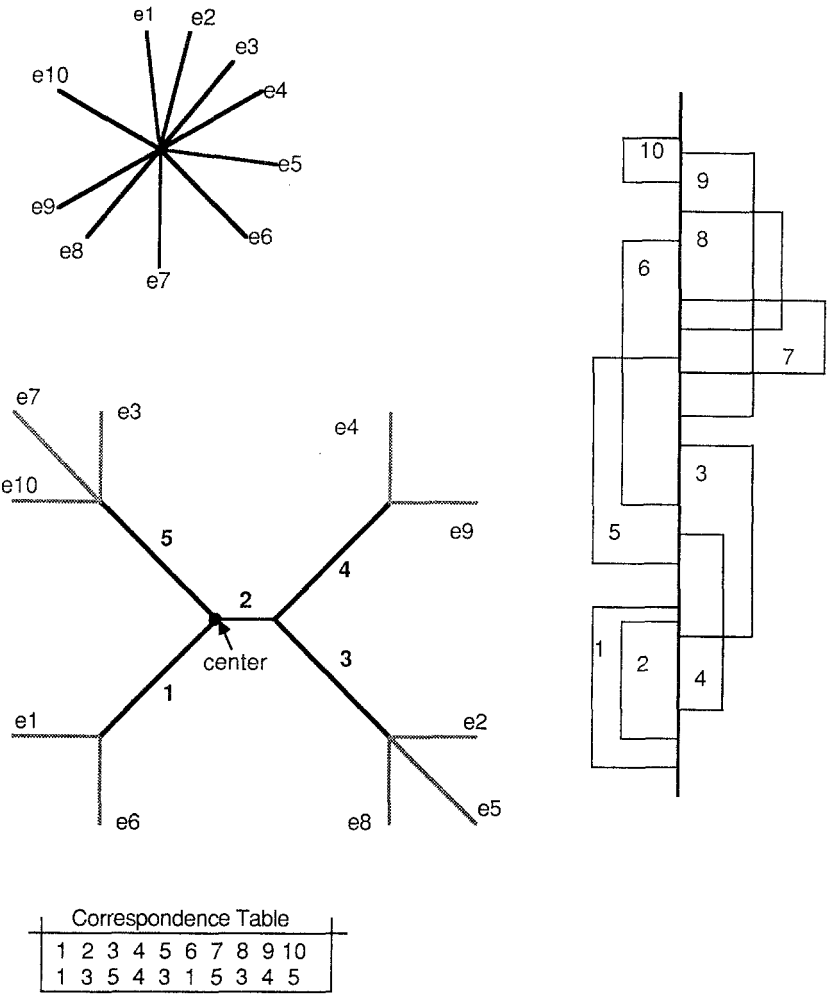

Fig. 7. Using the star-tree T10.

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 trival. 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.

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.

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 proceed into to v. This will take $O(\log s)$ time.

As it turns out, a gateway chain is quite similar to a B-tree [BM]. 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

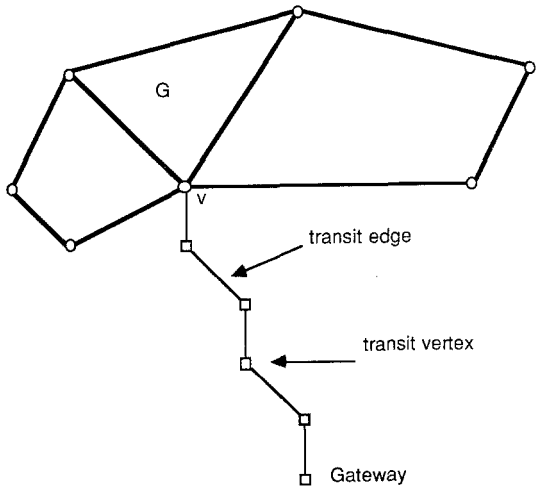

Fig. 8. A gateway.

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 require 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 one. 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

$$\sum_{v \in V} a_v \leq \sum_{v \in V} c_v + \sum_{v \in V} \sum_{(v,w) \in E} (b_{(v,w)} - 2)$$

$$= s + 2 \sum_{(v,w) \in E} (b_{(v,w)} - 2)$$

$$\leq s + 2 \sum_{(v,w) \in E} \left[\frac{g_{(v,w)} + \gamma d}{\gamma d + 2} - 2 \right]$$

$$\leq s + \frac{2d}{\gamma d + 2} \sum_{v \in V} a_v.$$

From this 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 in 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 ν, $\delta = O(1)$. 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 v 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 make 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.[7]

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

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

[7] 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 survival 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.

Therefore, process 2 and its subprocess 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 [CG] 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 [C, Co, EGS] to deal with the problem of iterative search. Examples are the *hive-graph* of [C] and the chain refinement scheme of [EGS]. These connections are developed more fully in [CG].

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 $\Theta((6d)^k)$.

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 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

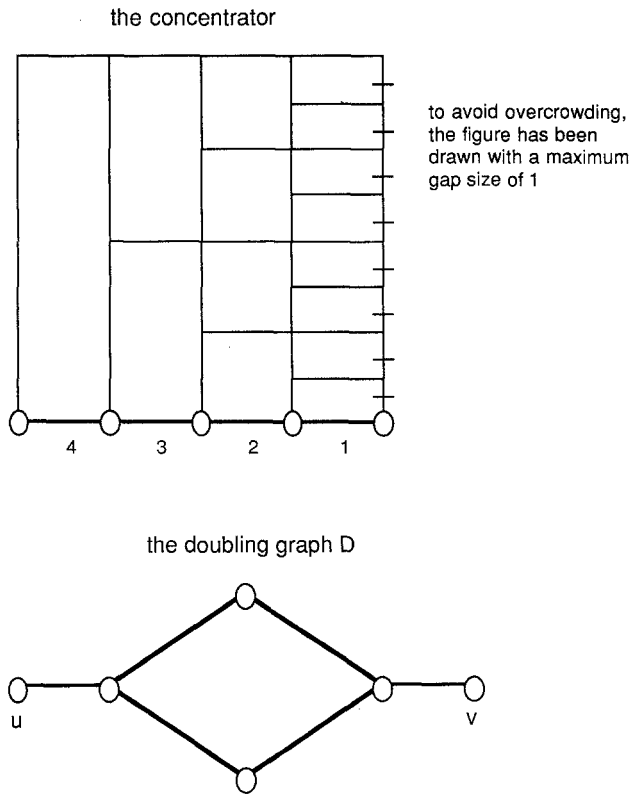

Fig. 9. The concentrator and the multiplier.

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 $O((6d)^k)$ and the number of insertions into a single gap it produces is 2^k. This proves the bound mentioned at the beginning if we fix k so that $s = \Theta((6d)^k)$ and thus completes our construction.

References

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

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

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

[CG] B. Chazelle and L. J. Guibas. *Fractional cascading II: applications.* To appear in Algorithmica (1986).

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

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

[EGS] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. *Optimal point location in a monotone subdivision.* To appear in SIAM J. Comput. Also DEC/SRC Research Report No. 2, 1984.

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

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

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

[T] R. E. Tarjan. *Amortized computational complexity.* SIAM J. Alg. Disc. Meth., **6** (2) (April 1985), 306-318.

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