Fast Fractional Cascading and Its Applications *

(CS-TR-4502 & UMIACS-TR-2003-71)

Qingmin Shi and Joseph JaJa
Institute for Advanced Computer Studies,
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742
{*qshi,joseph@umiacs.umd.edu*}

Abstract

Using the notions of Q-heaps and fusion trees developed by Fredman and Willard, we develop a faster version of the fractional cascading technique while maintaining the linear space structure. The new version enables sublogarithmic iterative search in the case when we have a search tree and the degree of each node is bounded by $O(\log^{\epsilon} n)$, for some constant $\epsilon > 0$, where n is the total size of all the lists stored in the tree. The fast fractional cascading technique is used in combination with other techniques to derive sublogarithmic time algorithms for the geometric retrieval problems: orthogonal segment intersection and rectangular point enclosure. The new algorithms use $O(n)$ space and achieve a query time of $O(\log n / \log\log n + f)$, where f is the number of objects satisfying the query. All our algorithms assume the version of the RAM model used by Fredman and Willard.

1 Introduction

Fractional cascading [5] is a powerful technique for dealing with the so called *iterative search* problem [5]. Let G be a graph such that a sorted list is associated with each of its vertices. Given a key q and a subgraph G', the iterative search problem is to search for q in each of the lists associated with the vertices in G'. This problem appears to be crucial in developing efficient algorithms for a wide variety of problems, and in particular for *geometric retrieval* problems [4, 6, 17, 3, 7, 24].

Let the overall size of the lists be n, and let the number of vertices in the subgraph G' be p such that the degree of each vertex is bounded by c. Then the fractional cascading structure enables searching for the key q in these lists in $O(\log n + p \log c)$ time. This result is optimal under the pointer machine model [22]. However, its optimality in more powerful computational models, such as the RAM model and its variations[1, 18, 10], is not clear. In

*Supported in part by the National Science Foundation through the National Partnership for Advanced Computational Infrastructure (NPACI), DoD-MD Procurement under contract MDA90402C0428, and NASA under the ESIP Program NCC5300.

particular, Fredman and Willard showed in [10] that such search operations can be achieved in worst case sublogarithmic time, which is impossible on the pointer machine model. An important fact of the fractional cascading structure is that the complexity of the search operations on the vertices other than the first one is affected significantly by the bound c on the degree of each vertex in G. We show in this paper that, under the RAM model used by Fredman and Willard, if G is a tree whose degree is bounded by a logarithmic function of n, i.e. $c = \log^{\epsilon} n$ for some constant ϵ, it is possible to search each list, except the one associated with the root, in constant time (independent of n) while still maintaining $O(n)$ space for the data structure. We call this new fractional cascading structure *fast fractional cascading*.

We believe that, by applying the fast fractional cascading technique, one should be able to improve the asymptotic upper bounds of a wide range of geometrical retrieval problems. This is due to the fact that binary search trees whose nodes are each equipped with sorted lists are often the crucial components in many data structures for geometrical retrieval problems. Ensuring that a constant amount of time is spent at each node is often critical in designing these structures and can be accomplished using the standard fractional cascading technique. The *fusion tree* technique [10, 11] makes it possible to further reduce the search complexity of some of these structures by increasing the degree of the binary trees to $\log^{\epsilon} n$ so that the height of the tree is reduced to $O(\log n / \log\log n)$. Even though at query time the *branch* operation at each node can be performed in constant time, the fact that the degree of the tree is now dependent on n makes the standard fractional cascading technique ineffective. In fact, a search operation performed on the list associated with a tree node will require $O(\log\log n)$ time, which negates the effect of the "fattened" tree. Only in some special cases, such as when the list of a node is a superset of that of its each children, constant search time can be achieved by equipping with each element of the parent of $\log^{\epsilon} n$ pointers interconnecting the elements in the parent with those of its children, as Willard did in solving the three-dimensional dominance aggregation problem [24]. Even in such cases, the storage cost is increased by a factor of $\log^{\epsilon} n$. On the other hand, our fast fractional cascading technique achieves the constant time search at each node while maintaining the linear space constraint.

We apply our techniques to two well-known problems: *orthogonal segment intersection*, and *rectangular point enclosure*. Each of our algorithms achieves $O(n)$ space and $O(\log n / \log\log n + f)$ query time where f is the number of objects satisfying the query. The best previous results require query time $O(\log n + f)$ when using linear space [4, 17].

We now introduce these two problems formally. To facilitate our explanation, we will denote a horizontal (resp. vertical) segment in a two-dimensional space as $(x_1, x_2; y)$ (resp. $(x; y_1, y_2)$), where (x_1, y) and (x_2, y) (resp. (x, y_1) and (x, y_2)) are its two endpoints and $x_1 \leq x_2$ (resp. $y_1 \leq y_2$)).

- *Orthogonal segment intersection.* Given a set S of n horizontal segments, report the subset Q of segments that intersect a given vertical segment. We say a horizontal segment $(x_1, x_2; y)$ *intersects* a vertical segment $(x; y_1, y_2)$ if and only if $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. We call the segments in Q *proper segments* relative to the given query.

- *Rectangular point enclosure.* Let $(x_1, x_2; y_1, y_2)$ denote a rectangle in a two-dimensional space with edges parallel to the axes, where the intervals $[x_1, x_2]$ and $[y_1, y_2]$ are the projections of this rectangle to the x-axis and y-axis respectively. Given a set of

S of rectangles, report the subset Q of *proper rectangles* such that each rectangle $(x_1, x_2; y_1, y_2)$ in Q contains a query point (x, y), i.e. $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

In this paper, we use the modified RAM model described in [10]. In this model, it is assumed that each word contains w bits, and the size of a data set never exceeds 2^w, i.e. $w \geq \log_2 n$. In addition to arithmetic operations, bitwise logical operations are also assumed to take constant time.

The next section introduces some well-known techniques that will be heavily utilized in the rest of the paper. In Section 3, we present the fast fractional cascading structure, while Sections 4, and 5 respectively give the improved algorithms for orthogonal segment intersection and rectangular enclosure.

2 Preliminaries

Given a set S of multi-dimensional points $(x_1, x_2, \ldots, x_d)$, a point with the largest x_i-coordinate smaller than or equal to a real number α is called the *x_i-predecessor* of α and the one with the smallest x_i-coordinate larger than or equal to α is called the *x_i-successor* of α.

2.1 Cartesian Trees

The notion of a *Cartesian* tree was first introduced by Vuillemin [23] (and rediscovered by Seidel and Aragon [20]). A Cartesian tree is a binary tree defined over a finite set of 2-D points sorted by their x-coordinates, say $(p_1, \ldots, p_n)$. Let p_i be the point with the largest y-coordinate. Then p_i is associated with the root w of C. The two children are respectively the root of the Cartesian trees built on $p_1, \ldots, p_{i-1}$ and $p_{i+1}, \ldots, p_n$. Note that the left (resp. right) child of w does not exist if $i = 1$ (resp. $i = n$). Figure 1 shows an example of the Cartesian tree.

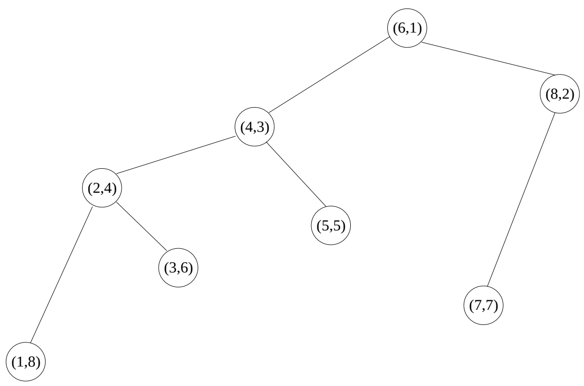


Figure 1: Cartesian tree.

An important property of the Cartesian tree is given by the following observation [12]:

Observation 2.1. *Consider a set S of 2-D points and the corresponding (x,y)-Cartesian tree C. Let $x_1 \leq x_2$ be the x-coordinates of two points in S and let α and β be their respective nodes* *in C. Then the point with the largest y-coordinate among those points whose x-coordinates are between x_1 and x_2 is stored in the nearest common ancestor of α and β.*

Using Observation 2.1, combined with the techniques to compute the nearest common ancestors [13] (see also [2]) in constant time, we have shown in [21] that we can handle the so-called 3-sided two dimensional range queries efficiently. Briefly, a point (a, b) satisfies the 3-sided query (x_1, x_2, y), $x_1 \leq x_2$, if $x_1 \leq a \leq x_2$ and $b \geq y$.

Lemma 2.1. *By preprocessing a set of n two-dimensional points to construct a (x,y)-Cartesian tree C, we can handle any three-sided two-dimensional range query given as (x_1, x_2, y), with $x_1 \leq x_2$, in $O(t(n) + f)$ time, where $t(n)$ is the time it takes to find in C the leftmost and right most nodes whose x-coordinates fall within the range $[x_1, x_2]$ and f is the number of points reported.*

Note that C should be transformed into a suitable form to enable the computation of nearest common ancestors in constant time.

2.2 Q-heaps and Fusion Trees

Q-heaps and *fusion trees*, developed by Fredman and Willard [10, 11], achieve sublogarithmic search time on one-dimensional data. While the Q-heap data structure was proposed later than fusion tree, it can be used as a building block for the fusion tree [24]. Using Q-heaps and fusion trees, Willard demonstrated in [24] theoretical improvements for a number of range search problems.

Q-heap [11] supports insert, delete, and search operations in constant time for small subsets of a large data set of size n. Its main properties are given in the following lemma (the version presented here is taken from [24]).

Lemma 2.2. *Suppose S is a subset with cardinality $m < \log^{1/5} n$ lying in a larger database consisting of n elements. Then there exists a Q-heap data structure of size $O(m)$ that enables insertion, deletion, member, and predecessor queries on S to run in constant worst case time, provided access is available to a precomputed table of size $o(n)$.*

Note that the look-up table of size $o(n)$ referred to above is shared by all the Q-heaps built on subsets of this larger database.

The fusion tree built on the Q-heap achieves linear space and sublogarithmic search time. The following lemma is a simplified version of Corollary 3.2 from [24].

Lemma 2.3. *Assume that in a database of n elements, we have available the use of precomputed tables of size $o(n)$. Then it is possible to construct a data structure of size $O(n)$ space, which has a worst-case time $O(\log n / \log\log n)$ for performing member, predecessor and rank operations.*

Notice that the assumptions of the RAM model introduced in Section 1 are critical to achieving the bounds claimed in the above lemmas.

2.3 Adjacency Map and Hive Graph

The notion of the *adjacency map* was first introduced by Lipski and Preparata [15]. Given a set S of n horizontal segments, the *vertical adjacency map* $G(S)$ is constructed by interconnecting the horizontal segments in S using vertical (infinite, semi-infinite, or finite) segments as follows: from each endpoint of the segments in S, draw two rays shooting upward and downward respectively until they meet other segments in S except possibly at an endpoint. This creates a planar subdivision $G(S)$ with $O(n)$ vertices, which are the joints of the horizontal and vertical segments. $G(S)$ can be represented in $O(n)$ space using the adjacency lists associated with the vertices. We call the edges supported by the horizontal segments *horizontal* edges and those supported by the vertical segments *vertical* edges.

Chazelle noticed in [4] that the adjacency map is a useful tool which, when modified appropriately, can be used to handle the orthogonal segment intersection problem efficiently. His modification of the vertical adjacency map is called the *hive graph*. A hive graph $H(S)$ is derived from $G(S)$ by adding only vertical segments to $G(S)$ while maintaining $O(n)$ vertices and $O(n)$ space representation. However, it has the important property that each face may have, in addition to its four (or fewer) corners, at most two extra vertices, one on each horizontal edge. Figure 2 shows a vertical adjacency map and its corresponding hive graph, in which the additional vertical edges are depicted as dashed lines. By assuming that the endpoints of the segments in S all have distinct x- and y-coordinates, as [4] did, one can conclude that each face of $H(S)$ has $O(1)$ vertices on its boundary. Given a query segment $(x; y_1, y_2)$, the segment intersection query can be handled as follows. We first find the face in $H(S)$ that contains the endpoint (x, y_1) in $O(\log n)$ time by using one of the well known planar point location algorithms [8, 9, 14, 16, 19]. Then we traverse a portion of $H(S)$ from bottom up following the direction from (x, y_1) to (x, y_2). Only a constant number of vertices are visited between two consecutive encounters of the horizontal edges that intersect the query segment.

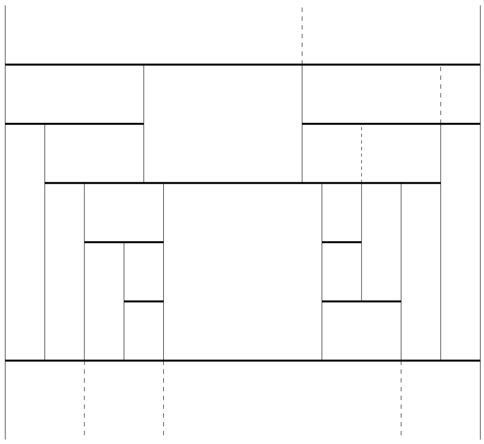


Figure 2: A hive graph

Note that the vertical boundary of a face of the hive graph corresponding to a vertical adjacency map will not necessarily contain a constant number of vertices if the assumption that the endpoints of the segments in S have distinct x coordinates does not hold. Since we will need to deal later with such a case, we get around this problem by associating with each vertex pointers to the upper-right or upper-left corner in the same face as follows. We modify $H(S)$ by associating with each vertex β two additional pointers $p(\beta)$ and $q(\beta)$. Let $\beta = \delta_1, \delta_2, \ldots, \delta_l = \gamma$ be the maximal chain of vertices such that each pair of consecutive vertices δ_i and δ_{i+1} is connected by a vertical edge e_i and δ_{i+1} is above δ_i, for $i = 1, \ldots, l-1$. Note that this chain can be empty ($l = 1$) in which case both $p(\beta)$ and $q(\beta)$ are null. If there exists a vertex in $\{\delta_2, \delta_3, \ldots, \delta_l\}$ which has a horizontal edge connecting it to a vertex to the left of it, then $p(\beta)$ points to the lowest such vertex. Otherwise, $p(\beta)$ is null. Similarly, if there exists a vertex that has a horizontal edge connecting it to a vertex to the right of it, then $q(\beta)$ points to the lowest such vertex. Otherwise, $q(\beta)$ is null. It is easy to see that, using these additional pointers, we can in constant time reach the next proper segment without the distinct x coordinates assumption. Figure 3 shows such a modified hive graph. The additional pointers $p(\beta)$ and $q(\beta)$ are depicted respectively as dashed and dotted arrows. To simplify the drawing, we omit the pointer $\alpha = p(\beta)$ or $\alpha = q(\beta)$ if α is null or (α, β) is an edge in $H(s)$. This figure also illustrates the search path of an exemplary segment intersection query by highlighting the pointers involved.

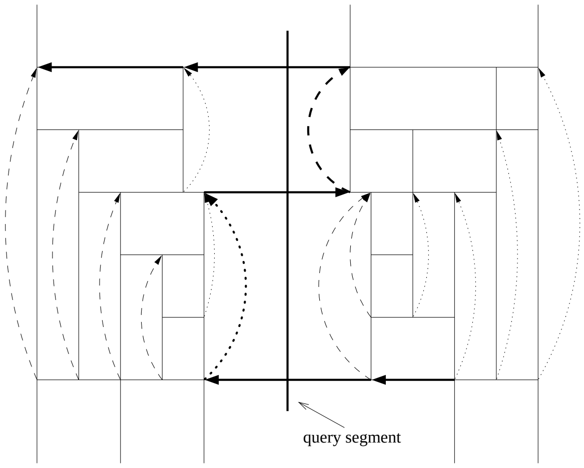


Figure 3: Modified hive graph.

As noted in [4], when the query segment is semi-infinite, that is, consists of a ray $(x; -\infty, y_2)$ shooting downward, there is no need to perform the initial planar point location query. Instead, we can, during preprocessing, sort the x-coordinates of the vertical edges of the faces unbounded from below, and perform as the first step at query time a search on the sorted x-coordinates to locate the face that contains the point $(x, -\infty)$. The following lemma is a restatement of Corollary 1 in [4].

Lemma 2.4. *Given a set S of n horizontal segments in the plane, an $O(n)$ space hive graph can be used to determine all the intersections of the horizontal segments with a semi-infinite vertical query segment $s = (x; -\infty, y_2)$ in $O(t(n) + f)$ time, where f is the number of intersections, and $t(n)$ is the time it takes to search the sorted list of x-coordinates.*

Combining Lemmas 2.3 and 2.4, we have the following Corollary.

Corollary 2.1. *Given a set S of n horizontal segments in the plane and a vertical query segment in the form of $(x; -\infty, y_2)$, it is possible to report all f proper segments of S in $O(\log n / \log\log n + f)$ time using $O(n)$ space.*

Clearly, by rotating the hive graph 90° clockwise (resp. counterclockwise), the same type of techniques will yield a solution for handling any orthogonal segment intersection query that involves a set S of vertical segments and a horizontal query segment of the form $(-\infty, x_2; y)$ (resp. $(x_1, +\infty; y)$). We will denote this rotated hive graph HL(S) (resp. $HR(S)$).

3 Fast Fractional Cascading

Suppose we have a tree $T = (V, E)$ rooted at w such that each node v has a degree bounded by c and contains a catalog $L(v)$ of sorted elements. Let n denote the total number of elements in these catalogs. A key value $k(g)$ from $\Re \cup \{-\infty, +\infty\}$ is associated with each element g in $L(v)$. The elements in $L(v)$ do not need to have distinct key values. We call such a tree a *catalog tree*. Let x be a real number and F be an arbitrary forest with p nodes consisting of subtrees of T determined by some of the children of w. Both x and F can be specified online, i.e., not necessarily at preprocessing time. Let $\sigma_L(x)$ denote the successor of x in a catalog L. The *iterative search problem* is defined as follows [5]: report $\sigma_{L(v)}(x)$ for each v in F. Fractional cascading is a technique for solving the iterative search problem efficiently, and is briefly introduced next.

3.1 Fractional Cascading

The following lemma is a direct derivation from the one given by Chazelle and Guibas for identifying the successor of a value x in each of the catalogs in F [5].

Lemma 3.1. *There exists a linear size fractional cascading data structure that can be used to determine the successors of a given value x in the catalogs associated with F in $O(p \log c + t(n))$ time, where $t(n)$ is the time it takes to identify the successor of x in $L(w)$.*

The main component of a fractional cascading structure is the notion of the *augmented catalogs*. At each node v in T, in addition to the original catalog $L(v)$, we store another augmented catalog $A(v)$, which is a superset of $L(v)$ and contains additional copies of elements from the augmented lists associated with its parent and children. With each element h in $A(v)$, we associate a pointer to its successor $\sigma_{L(v)}(h)$ in $L(v)$. Since $A(v)$ is a superset of $L(v)$, we have $\sigma_{L(v)}(g) = \sigma_{L(v)}(\sigma_{A(v)}(g))$. Note that the elements in an augmented list $A(v)$ form a multiset $S(v)$, that is, a single element can appear multiple times in an augmented list. The elements in an augmented list are chained together to form a doubly linked list.

As illustrated in Figure 4, let u and v be two neighboring nodes in T, u being v's parent. There exists a subset $B(u, v)$ of $A(u) \times A(v)$ such that, for each pair of elements $(g, h) \in B(u, v)$, $k(g) = k(h)$. The pair of elements (g, h) are called a *bridge*. There is a pointer to h associated with the element g, and similarly a pointer to g is associated with h. We will call g a *down-bridge*, and h an *up-bridge*, associated with the edge (u, v). It is important to point out that each element in an augmented list can serve as at most one up-bridge or one down-bridge, but not both. Bridges respect the ordering of equal-valued elements and thus do not "cross". This guarantees that $B(u,v)$ can be ordered and the concept of *gap* presented next is well defined. In this ordered set $B(u,v)$, the bridge (g,h) appears after the (g',h') if and only if g appears after g' in $A(u)$. A *gap* $G_{(u,v)}(g,h)$ of bridge (g,h) is defined as the multiset of elements from both $A(u)$ and $A(v)$ which are strictly between two bridges (g,h) and (g',h'), where (g',h') is the bridge that appears immediately before (g,h) in $B(u,v)$. Accordingly, we define the *up-gap* (resp. *down-gap*) $G_{(u,v)}(g)$ (resp. $G_{(u,v)}(h)$) as the subset of $G_{(u,v)}(g,h)$ containing elements from $A(u)$ (resp. $A(v)$), preserving their orders in the respective augmented catalogs.

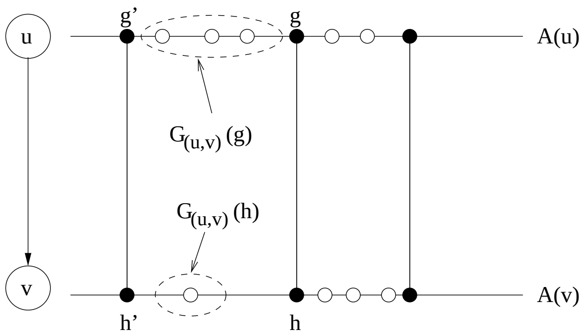


Figure 4: Fractional cascading.

The fractional cascading structure maintains the invariant that the size of any gap cannot exceed $6c-1$. Chazelle and Guibas provided in [5] an algorithm that can in $O(n)$ time construct such a data structure; and they prove that it requires $O(n)$ space.

Given a parent-child pair $(u,v) \in E$, suppose we know the successor $\sigma_{A(u)}(x)$ of a value x in $A(u)$, we follow $A(u)$ along the direction of increasing values to the next down-bridge g connecting u and v (it could be $\sigma_{A(u)}(x)$ itself if it is a down-bridge), cross it to its corresponding up-bridge h, and scan $A(v)$ in the opposite position until the successor of x in $A(v)$ is encountered. Clearly, $\sigma_{A(v)}(x)$ is guaranteed to be found by this process. The constraint on the gap size ensures that the number of comparisons required is $O(c)$.

When c is a constant, the above result is optimal. When c is large, Chazelle and Guibas used the so called *star tree* to achieve $O(\log c)$ search time on each catalog except the one stored at the root.

3.2 Fast Fractional Cascading

The fractional cascading structure described above is strictly list based, and hence all the related algorithms can run on a pointer machine within the complexity bounds stated. Using the variation of the RAM model introduced in Section 1 and the Q-heap technique of Fredman and Willard [11], summarized in Section 2.2, we can achieve constant search time (independent of c) per node for the class of catalog trees whose degree is bounded by $c = \log^{\epsilon} n$, while simultaneously maintaining a linear size data structure. We call this version *fast fractional cascading*. This result improves over our previous result in [21], which achieves the same search complexity but requires non-linear space. We will first revisit the non-linear space solution and then explain how to reduce the storage cost to linear.

3.2.1 Fast Fractional Cascading with Non-linear Space

We augment the fractional cascading structure described in Section 3.1 by adding two types of components to each augmented catalog $A(v)$. First, we associate c additional pointers $p_1(g), p_2(g), \ldots, p_c(g)$ with each element g in $A(v)$ such that $p_i(g)$ points to the next down-bridge (possibly g itself) connecting v to w_i, where w_i is the ith children of v from the left. Second, we build for each up-gap $G_{(u,v)}(h)$ a Q-heap $Q(h)$, containing elements in $G_{(u,v)}(h)$ with distinct values (choosing the first one if multiple elements have the same value). For large enough n we have $6c - 1 < \log^{1/5} n$; and therefore Lemma 2.2 is applicable. We have added c pointers for each of the elements in the augmented catalogs, whose overall size cannot exceed $O(n)$. In addition, a global look-up table of size $O(n)$ is used to serve all the Q-heaps. And finally, since no two up-gaps in an augmented catalog overlap, since they correspond to the same edge in T (which is not true for a general graph) the Q-heaps cannot consume more than $O(n)$ space.

Now suppose we have found $g = \sigma_{A(u)}(x)$ in $A(u)$. Let v be the ith child of u. By following the pointer $p_i(g)$, we can reach in constant time the next down-bridge in u and then its companion up-bridge h in v. Using $Q(h)$, we can find the successor of x in $G_{(u,v)}(h)$ in constant time.

Lemma 3.2. *Let $c = O(\log^{\epsilon} n)$. The fast fractional cascading structure described above allows the identification of the successors of a given value x in the catalogs associated with F in $O(p + t(n))$ time, where $t(n)$ is time it takes to identify the successor of x in $L(w)$. This structure requires $O(cn)$ space.*

3.2.2 Fast Fractional Cascading with Linear Space

We partition each augmented catalog $A(u)$ into $p = \lceil |A(u)|/c \rceil$ blocks $B_1, B_2, \ldots, B_p$ each, except possibly the last one, containing c elements. For each block B_i starting from the lth element of $A(u)$, we construct a set C_i of $t \le 7c-1$ records as follows. For each down-bridge g that is the dth element in $A(u)$, where $l \le d \le l + 7c - 2$, we include in C_i a record r that contains two entries $r.ptr$ and $r.key$. The entry $r.ptr$ is a pointer to g, and $r.key$ is the key of r whose value is defined as $r.key = j * (7c - 1) + (d - l)$ if g is associated with the edge connecting u and its $(j + 1)$th child (note that $r.key$ can fit in a word). The records in C_i are sorted in increasing order by their key values. Now let g be the successor of a value x in $A(u)$ and suppose we want to find the successor of x in the augmented catalog associated with the $(j + 1)$th child v of u. It is easy to determine in constant time the block B_i to which g belongs and its position f relative to the starting position of B_i ($f = 0$ if g is the first element in B_i). If g is itself a down-bridge associated with (u, v), then we are done. Otherwise, due to the invariant regarding the gap size, the next down-bridge h associated with (u, v) must have a corresponding record in C_i. The following lemma transforms the problem of finding h to a successor search in C_i.

Lemma 3.3. *The record in C_i that corresponds to h is the successor of the value $y = j \cdot (7c - 1) + f$.*

Proof. First we notice the fact that all the keys of the records in C_i are distinct. Let $y' = j \cdot (7c - 1) + f'$ be the key of the record in C_i that corresponds to h. It is obvious that

$y < y'$. Now let $y'' = j'' \cdot (7c+1) + f''$ be the key of a record r in C_i such that $y \le y''$. We only need to show that $y' \le y''$. Since both f'' and f are non-negative integers less than $7c-1$, the fact that $y \le y''$ leads to either $j < j''$, or $j = j''$ and $f \le f''$. If $j < j''$, we immediately have $y' < y''$. On the other hand, if $j = j''$, then the record r also corresponds to a down-bridge associated with the edge (u, v). Since h is the leftmost down-bridge closest to g, we have $f' \le f''$. Thus $y' \le y$. $\square$

The problem of finding the successor of an integer value in a small set C_i can be solved, again using the Q-heap data structure. The following straightforward observations ensure the applicability of Lemma 2.2:

- $|C_i| < \log^{1/5} n$ for n large enough; and
- The total number of distinct keys created for all the augmented catalogs is bounded by $O(n)$.

Finally, it is easy to see that the overall additional space introduced by the new Q-heaps is $O(n)$, and thus we have the following theorem.

Theorem 3.1. *For $c = O(\log^{\epsilon} n)$ for some $\epsilon < \frac{1}{5}$, our fast fractional cascading structure allows the identification of the successors of a given value x in the catalogs associated with F in $O(p + t(n))$ time, where $t(n)$ is time it takes to identify the successor of x in $L(w)$. This structure requires $O(n)$ space.*

4 Orthogonal Segment Intersection

Before tackling the general orthogonal segment intersection problem, we develop a linear size data structure to handle a special case in which the x-coordinates of the endpoints of the segments and the query segment can only take integer values over a small range of values. We will later show how to use the solution of the special case to derive a solution to the general problem.

4.1 Modified Vertical Adjacency Map

Assume that the x-coordinates of the endpoints of each segment $(k_1, k_2; y)$ in the given set R of n horizontal segments can take values from the set of integers $\{1, 2, \ldots, c\}$, where $c = \log^{\epsilon} n$ is an integer, and furthermore, assume that the x-coordinate k of the query segment $r = (k; z_1, z_2)$ is an integer between 1 and c. Let $Y(R) = (y_1(R), y_2(R), \ldots, y_{n'}(R))$ be the list of distinct y-coordinates of the segments in R sorted in increasing order.

Our overall strategy consists of augmenting the vertical adjacency map with auxiliary structures so that we will be able to identify the lowest segment in R intersecting r very quickly, followed by progressively determining the next sequence of lowest segments, each in $O(1)$ time. The details of this strategy are described next.

Our indexing structure $D(R)$ consists of two major components: $H(R)$ and $M(R)$. $H(R)$ is a *directed* vertical adjacency map with auxiliary information attached to it. We define the direction of the horizontal edges to be from right to left and that of the vertical edges to be from bottom up. Note that we do not require that each face of $H(R)$ has a constant number of vertices on its boundary.

Each vertex of $H(R)$ is naturally associated with a pair of x, y-coordinates. We call the vertex with an outgoing horizontal edge a *tail*. We augment $H(R)$ with three types of components as follows:

- For each distinct y-coordinate $y_j(R)$ of $Y(R)$, we create a Q-heap $Q_j(R)$ to index the x-coordinates of the vertices whose y-coordinates are equal to $y_j(R)$.
- For each integer $1 \leq i \leq c$ that serves as the x-coordinate of at least one tail, we create a list $P_i(R)$ of records. Each record g corresponds to a tail α whose x-coordinate is i and contains two elements: $g.key$, which is the y-coordinate of α, and $g.ptr$, which is a pointer to α. This list is sorted in increasing order by the key values.
- With each node β we associate two pointers $p(\beta)$ and $q(\beta)$. Let $y_j(R)$ be the y-coordinate of β. Then $p(\beta)$ points to the Q-heap $Q_j(R)$. If β is not a tail, $q(\beta)$ is null. Otherwise, there is at least one vertex with the same y-coordinate as β that has an outgoing vertical edge and is to the strict left of β. Let γ be the rightmost such vertex, and $e_1, e_2, \ldots, e_l$ be the shortest chain of vertical edges starting from γ such that the head ξ of e_l has an incoming horizontal edge. If such a chain exists, then $q(\beta)$ points to ξ. If not, $q(\beta)$ is null. Note that intuitively $q(\beta)$ is the top left corner (if it exists) of a face containing β.

It is clear that $H(R)$ is of size $O(n)$.

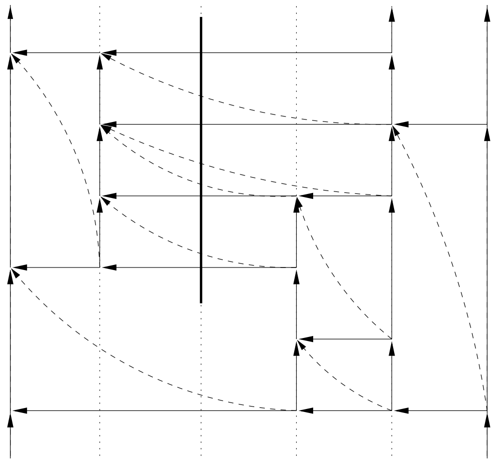
In addition to $H(R)$, we have a bitmap $M(R)$ consisting of a list of bit-vectors. Each vector $V_j(R)$ corresponds to a distinct y-coordinate $y_j(R)$ and contains c bits. The ith bit, starting from the most significant one, is set to one if there is a vertical edge in $H(R)$ passing through the point $(i, y_j(R))$ and zero otherwise. Each vector can easily fit in a single word and thus the storage cost of $M(R)$ is $O(n)$. These vectors are aligned with the lower end of the words and are stored in increasing order by the values of the corresponding y-coordinates.

As an example, Figures 5(a) and 5(b) illustrate the structures $H(R)$ and $M(R)$. In Figure 5(a), the dotted lines depict the c possible x-coordinates, the dashed pointers are the q-pointers that are not null, and the thick line represents the query segment.

Given a vertical segment $r = (k; z_1, z_2)$, we first identify the lowest segment that intersects r and then report each of the remaining proper segments in the direction of increasing y-coordinates.

Locating the lowest segment that intersects r is performed using $M(R)$. Let $y_j(R)$ be the smallest y-coordinate greater than or equal to z_1. If no such y exists, then there is no segment in R which intersects r. Otherwise, we find the largest value $i \leq k$ such that the ith bit in $V_j(R)$ is one. (This number always exists because the vertical edges whose x-coordinates are equal to c form a infinite line and therefore the lowest bits of all the vectors are set to one.) This can be accomplished by first masking out the highest $w - k$ bits of $V_j(R)$, w being the number of bits in a word, and then locating its most significant bit. In [10], Fredman and Willard describe how to compute the most significant bit of a word in constant time.

After identifying i, we use $P_i(R)$ to determine the record g with the smallest key larger than or equal to z_1. We can then immediately obtain the vertex α pointed to by $g.ptr$.



(a)

	6	5	4	3	2	1
$V_6(R)$	1	1	0	0	1	1
$V_5(R)$	1	1	0	0	1	1
$V_4(R)$	1	1	0	1	1	1
$V_3(R)$	1	1	0	1	1	1
$V_2(R)$	1	0	0	1	1	1
$V_1(R)$	1	0	0	1	1	1

(b)

Figure 5: $H(R)$ and $M(R)$.

Lemma 4.1. *Let (k_α, y_α) be the coordinates of α. Then for any segment $(k_1, k_2; y)$ in R such that $k_1 \le k$ and $y_\alpha > y \ge y_1$, we have $k_2 < k$. That is, any horizontal segment between y_α and y which starts to the left of r ends before meeting r.*

Proof. The proof is by contradiction. Suppose $k_2 \ge k$. We then have $k_2 < k_\alpha$, because otherwise the vertical line passing through α would have had at least one vertex α' lying on it with its coordinates $(k_{\alpha'}, y_{\alpha'})$ satisfying $k_{\alpha'} = k_\alpha = i$ and $y_{\alpha'} < y_\alpha$, which contradicts the way we chose α. Now consider the vertical line passing through the endpoint (k_2, y). Either it passes through the point $(k_2, y_j(R))$ or intersects a horizontal segment whose left endpoint is to the left of α and whose y-coordinate is strictly between y and $y_j(R)$. In the first case, we have a contradiction because there would have been a more significant one-bit than i in $V_j(R)$. In the second case, the right endpoint of that horizontal segment has to be to the strict left of α, following the same argument for the segment $(k_1, k_2; y)$. By repeatedly applying this argument, we can show that either there is a one-bit in $V_j(R)$ more significant than the i, or there is a record in $P_i(R)$ whose key is smaller than y_α but larger than y_1, each leading to a contradiction. $\square$

Lemma 4.2. *If $y_\alpha \le y_2$, then the horizontal segment $t = (k_1, k_2; y)$ on which α lies intersects r.*

Proof. The only possible scenario in which t does not intersect r is when $k_1 > k$. If this is the case, then there has to be a vertical segment $(k_1; y_1', y_2')$ consisting of several edges in H and passing through the point (k_1, y). This segment cannot cross the horizontal line corresponding to $V_j(R)$ because otherwise there would have been a more significant one-bit than the ith in $V_j(R)$. Therefore there has to be a horizontal segment $t' = (k_1', k_2'; y_1')$ with $k_2' > k_1 > k$. Lemma 4.1 implies that $k_1' > k$. Repeating this argument will ultimately lead to a contradiction. $\square$

Lemmas 4.1 and 4.2 show that the horizontal segment t on which α lies is the lowest segment that intersects r. Using the Q-heap pointed to by $p(\alpha)$, we can find the vertex β

with the same y-coordinate as α and the smallest x-coordinate greater than or equal to k. Since t intersects r, we are sure that β is also on t. The following lemma explains how to iteratively find the remaining segments that intersect r.

Lemma 4.3. *Let t be a horizontal segment that intersects r and suppose we know the vertex β of $H(R)$ on t with the smallest x-coordinate k_β larger than or equal to k. We can in constant time decide whether there is another segment t' above t that intersects r, and furthermore, if there is one, identify in constant time such a t' having the smallest y-coordinate larger than that of t.*

Proof. We first give the algorithm to compute the vertex β' on t' with the smallest x-coordinate $k_{\beta'}$ larger than or equal to k. Consider the following cases.

Case 1 β has an outgoing vertical edge e and $k = k_\beta$.

Case 1.1 e is an infinite edge, i.e. e is a ray shooting upwards. Then there are no other segments intersecting r.

Case 1.2 The edge e is finite. In this case, the vertex β' is the head of e and t' is the horizontal segment on which β' lies.

Case 2 β does not have an outgoing vertical edge e or $k \neq k_\beta$.

Case 2.1 $q(\beta)$ is null. There are no other segments intersecting r.

Case 2.2 $q(\beta)$ is not null. β' corresponds to the successor of k in the Q-heap pointed to by $p(q(\beta))$ and t' is the horizontal segment on which β' lies.

We now show the correctness of this algorithm. We only discuss Case 2, as the correctness of our algorithm for Case 1 is obvious. First consider the case when $q(\beta)$ is null. Since β has to be a tail, the vertical ray starting from γ (introduced in the definition of $q(\beta)$) shooting upward does not contain a vertex with an incoming horizontal edge. Hence if there were a horizontal segment above t that intersects r, γ would not be the rightmost vertex to the left of β that has an outgoing vertical edge. Hence no segment above t intersects r.

We now consider Case 2.2. In this case, γ and the chain starting from it always exist. Let $e_1, e_2, \ldots, e_l$ be the chain of vertical edges used to define $q(\beta)$, and $\gamma = \delta_1, \delta_2, \ldots, \delta_{l+1} = \xi$ be the sequence of vertices such that for each $1 \leq j \leq l$, $e_j = (\delta_j, \delta_{j+1})$, and (k', y_γ) and (k', y_ξ) be the respective coordinates of γ and ξ. We claim that: (i) no horizontal segment whose y-coordinate are strictly between those of t and t' intersects r; (ii) the horizontal segment t' on which ξ lies does intersect r; and (iii) the successor β' of k in the Q-heap pointed to by $p(q(\beta))$ always exists.

To see why the first claim is true, suppose there is a horizontal segment $(k_1', k_2'; y')$ intersecting r that satisfies $y_\gamma < y < y_\xi$. Then it has to be true that $k' < k_1' \leq k$. Since we are discussing Case 2, there has to be another horizontal segment $(k_1'', k_2''; y'')$ such that $k' < k_1'' < k$ and $y_\gamma < y'' < y_\xi$. Following similar arguments as in the proof of Lemma 4.1, we can show that either there exist a vertex on t between β and γ with an outgoing vertical edge, or there exists a vertex ξ' with an incoming horizontal edge such that its coordinate $(k_{\xi'}, y_{\xi'})$ satisfies $k_{\xi'} = k'$ and $y_\gamma < y_{\xi'} < y_\xi$. Either case leads to a contradiction.

To show that t' indeed intersects r, we notice that the right endpoint of the horizontal segment of which the horizontal incoming edge of ξ is a part cannot be to the (strict) left of s, because otherwise either there would be a chain of vertical edges closer to β than the one we have, or there would be a horizontal segment lying vertically between t and t' that intersects r, each leading to a contradiction. This also justifies the last claim (iii), and the proof of the lemma is complete. □

Lemma 4.4. *Given a set R of n horizontal segments in the plane, whose endpoints can only have $c = \log^{\epsilon} n$ possible x-coordinates $\{1, 2, \ldots, c\}$, it is possible to report using $O(n)$ space all f proper segments of R which satisfy a query $r = (k; y_1, y_2)$, where $k = 1, 2, \ldots, c$, in $O(t(n) + f)$ time, where $t(n)$ is the time it takes to compute the successor of y_1 in $Z(R)$ and $P_i(R)$ for some $i = 1, 2, \ldots, c$.*

Note that we can apply the fusion tree technique to index the distinct y-coordinates using linear space so that $t(n) = O(\log n / \log\log n)$. In the next section, we will show how to use the algorithm of Lemma 4.4 to solve the general orthogonal segment intersection problem. By applying the fast fractional cascading technique, The time $t(n)$ in Lemma 4.4 can be reduced to $O(1)$ except for the initial search, in which $t(n) = O(\log n / \log\log n)$.

4.2 Handling the General Orthogonal Segment Intersection Problem

In this section we consider the general orthogonal segment intersection problem involving a set S of n horizontal segments. To simplify the presentation, we assume that the endpoints of the segments in S have distinct x-coordinates. The primary data structure is a tree T of degree $c = \log^{\epsilon} n$, built on the endpoints of the n segments sorted in increasing order of the x-coordinates. Each leaf node v is associated with c endpoints. Let x_{l} and x_{r} be respectively the x-coordinates of the leftmost endpoints associated with v and the leaf node to its immediate right ($x_{\mathrm{r}} = +\infty$ if v is the rightmost leaf node); then the *x-range* of v is defined as $[x_{\mathrm{l}}, x_{\mathrm{r}})$. For an internal node u with c children $v_0, v_1, \ldots, v_{c-1}$, whose corresponding x-ranges are $[x_0, x_1), [x_1, x_2), \ldots, [x_{c-1}, x_c)$, its x-range is $[x_0, x_c)$. The set of $c-1$ infinite horizontal lines $b_1(u), b_2(u), \ldots, b_{c-1}(u)$, whose x-coordinates are $x_1, x_2, \ldots, x_{c-1}$ respectively, are called the *boundaries* of u. When the context is clear, we will use $b_i(u)$ to represent its corresponding x-coordinate as well.

The segments in S are distributed among the nodes of T as follows. A horizontal segment is associated with an internal node u if it intersects one of the boundaries of u but none of the boundaries of u's ancestors. A segment is associated with a leaf node v if its endpoints both lie within the x-range of v.

The set $S(v)$ of segments associated with an internal node v is organized into several secondary data structures as described below and illustrated in Figure 6.

- The $c-1$ boundaries of each node v are indexed by a Q-heap so that given an arbitrary value x the left most boundary $b_i(v)$ that satisfies $x \leq b_i(v)$ can be identified in constant time.

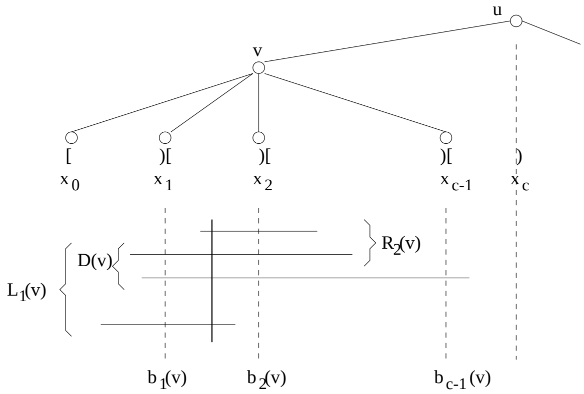


Figure 6: Data structures for the segments associated with node v.

- With each boundary $b_i(v)$ with $1 \leq i \leq c-1$, we associate two Cartesian trees $L_i(v)$ and $R_i(v)$. The Cartesian tree $L_i(v)$ contains the endpoints of those segments $(x_1, x_2; y)$ in $S(v)$ which satisfy $b_{i-1}(v) < x_1 \leq b_i(v)$ ($b_0(v) = -\infty$) and $x_2 \geq b_i(v)$, and is used to the answer the three-sided range query of the form: $(x_1 \leq a, b \leq y \leq d)$; and $R_i(v)$ contains the endpoints of those segments that satisfy $b_i(v) \leq x_2 < b_{i+1}(v)$ ($b_{i+1}(v) = +\infty$ for $i = c-1$) and $x_1 \leq b_i(v)$, and is used to answer the three-sided range query of the form: $(x_2 \geq a, b \leq y \leq d)$. Each Cartesian tree thus created has its nodes doubly linked in the order of increasing y-coordinates.

- Let $S'(v)$ be a subset of $S(v)$ containing segments that each intersects at least two boundaries of v. We organize these segments using the data structure $D(v)$ discussed in Section 4.1. We will later explain how to transform the problem corresponding to $S'(v)$ to the one discussed in Section 4.1.

The number of horizontal segments associated with a leaf node is at most $c/2$ since there are only c different endpoints associated with a leaf node, which are simply stored in a list.

We analyze the storage cost of the structures involved in our overall data structure. Obviously, each segment in S is associated with exactly one node v of T. For any segment associated with an internal node v, it appears in at most three secondary structures, once in $L_i(v)$ associated with the left most boundary $b_i(v)$ it intersects, once in $R_j(v)$ associated with the rightmost boundary $b_j(v)$ it intersects, and possibly once in $D(v)$. Any segment associated with a leaf node is stored exactly once. Note that all these data structures are linear-space. Hence the total amount of space used by these structures is $O(n)$..

We next outline our search algorithm and then fill in the details as we go along. Let $s = (a; b, d)$ be a vertical segment. To avoid the tedious but not difficult task of treating special cases, we make the assumption that the endpoints of s is different from any of the endpoints of the segments in S. To compute the set of proper segments in Q, we recursively search the tree T, starting from the root. Let v be the node we are currently visiting. We search v as follows.

1 If v is a leaf node, check each segment associated with v and report those that intersect s, after which the algorithm terminates.

2 If x lies outside the x-range of v, then no segment in S intersect s and the algorithm terminates. (This can only happen at the root, when s is to the left of all the segments in S.)

3 Otherwise do the following:

3.1 Find the pair of consecutive boundaries $b_i(v)$ and $b_{i+1}(v)$ of v such that $b_i(v) < a < b_{i+1}(v)$. (The boundary $b_i(v)$ does not exit if $x < b_1(v)$; and $b_{i+1}(v)$ does not exist if $a > b_{c-1}(v)$.)

3.2 If $b_i(v)$ exist, use $R_i(v)$ to report segments $(x_1, x_2; y)$ that satisfy $x_2 \geq a$ and $b \leq y \leq d$.

3.3 If $b_{i+1}(v)$ exist, use $L_{i+1}(v)$ to report those segments $(x_1, x_2; y)$ that satisfy $x_1 \leq a$ and $b \leq y \leq d$.

3.4 If both $b_i(v)$ and $b_{i+1}(v)$ exist, use $D(v)$ to report those proper segments with no endpoints in the interval $(b_i(v), b_{i+1}(v))$.

3.5 Recursively visit the $(i+1)$th child of v (the first child being the leftmost).

The correctness of the algorithm is obvious, provided that Step 3.4 can be performed correctly, a fact we will show shortly. First we note that Step 3.1 can be done in constant time using the Q-heap. Furthermore the access of the Cartesian trees in Steps 3.2 and 3.3 can be done in time proportional to the number of segments reported if the successor of b and the predecessor of d in the list of nodes for each Cartesian tree can be identified in constant time. We will show later that we can indeed achieve this goal by applying the fast fractional cascading structure. Finally, it is clear that only one node is visited at each level of T, which consists of $O(\log n / \log\log n)$ levels.

Now we focus on Step 3.4. The difficulty is to keep the size of $D(v)$ linear and at the same time be able to execute this step in time proportional to the number of segments reported. Let n' be the size of $S'(v)$. One obvious choice is to keep as $D(v)$ as $O(c^2)$ lists of segments. Each list corresponds to a pair of boundaries and consists of segments sorted by their y-coordinates that cross both boundaries. The storage cost is obviously $O(n')$. However, we will have to visit each list to report the proper segments, since there is no obvious way to decide beforehand which lists contain at least one proper segment (as Willard cleverly did in the design of the fusion priority tree [24]). At least $\Omega(\log^{2\epsilon} n)$ time seems to be required as a result. On the other hand, we can associate with each pair of consecutive boundaries the sorted list of segments that crosses both of them. This approach satisfies the requirement on the query complexity but increases the storage cost by a factor of $\log^{\epsilon} n$.

We now present our solution to handle these segments. We first transform the x-coordinates x_1 and x_2 of the endpoints of each segment s into two integers k_1 and k_2 between 1 and $c-1$. More specifically, k_1 and k_2 are the indices of the leftmost and rightmost boundaries of v crossed by s. By doing this, we transform $S'(v)$ into another set $W(v)$, in which the segments have their y-coordinates unchanged but their x-coordinates replaced by the indices of the boundaries. At query time, we also transform the query segment $s = (x; y_1, y_2)$ into another segment r by replacing its x-coordinate with the index k of the boundary to its immediate right. It is straightforward to see that a segment in $S'(v)$ is proper if and only if its corresponding segment $(k_1, k_2; y)$ in $W(v)$ satisfies $k_1 < k \leq k_2$ and $y_1 \leq y \leq y_2$. (In the case where $k_1 = k$, the original segment corresponding to $(k_1, k_2; y)$ is already found using $L_{k_1}(v)$ and thus need not be reported here.) We now have exactly the problem we tackled in Section 4.1. Hence by Lemma 4.4, we can find the f' proper segments in $W(v)$ in $O(f')$ time, provided that we can in constant time identify the successor of b in the various sorted lists of y-coordinates associated with $H(v)$.

To complete the description of our algorithm, we show how to apply the fast fractional cascading structure to search the sorted lists at different levels of the tree. The sorted lists stored at each node v consist of the $2(c-1)$ lists $R_i(v)$ and $L_i(v)$ for $i = 1, \ldots, c-1$, the list of vectors in $M(v)$, and up to $c-1$ lists of $P_i(v)$. Note that during the query time, we only need to search $O(1)$ such lists at each level. Using the fusion tree, we can search the relevant list at the root of T in $O(\log n / \log \log n)$ time.

To see how the various lists are linked through fast fractional cascading, We can imagine a virtual forest F consisting of c "virtual" trees $T_1, T_2, \ldots, T_c$ of degree $3c-2$, such that the lists stored at the roots are respectively $L_1(v), L_2(v), \ldots, L_{c-1}(v), R_{c-1}(v)$, where v is the root of our search tree. The children of the root containing $L_i(u)$ contain the lists in the ith children of u from the left; and the children of $R_{c-1}(u)$ are the lists in the cth children. Figure 7 illustrates the concept of the virtual forest. It is straightforward to see that a node in F is searched only if its parent is searched. Since $c = \log^{\epsilon} n$ with $\epsilon < 1/5$, $3c - 2 < \log^{1/5} n$ for large enough n. Therefore we can apply the fast fractional cascading technique to interconnect the lists according to the topology of the virtual forest so that we can search in constant time each list after the initial search at the root of F without increasing the space requirements.

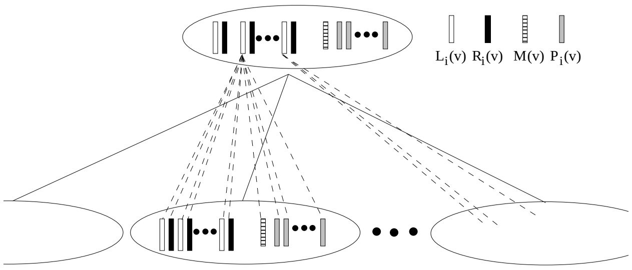


Figure 7: The virtual forest.

In summary, handling a query consists of processing the nodes on a path from the root to a leaf node. Processing the root w of T takes $O(\log n / \log \log n + f(w))$ time. The time spent at processing any other internal node u is $O(f(u))$. To search the leaf node w, we simply check each segment stored there. Since there are at most $O(c)$ such segments and $c = \log^{\epsilon} n < \log n / \log \log n$ for large enough n, the overall query time is $O(\log n / \log \log n)$ and therefore we have the following theorem.

Theorem 4.1. *There exists a linear-space algorithm to handle the orthogonal segment intersection problem in $O(\log n / \log \log n + f)$ query time, where f is the number of segments reported.*

5 Rectangle Point Enclosure

To simplify our presentation, we assume that the corners of the rectangles in S all have distinct x- and y-coordinates. As in the case of the segment intersection problem, the primary data structure consists of a tree T of degree $c = \log^{\epsilon} n$. Let v be the root of T and $b_1(v), b_2(v), ..., b_{c-1}(v)$ be a set of infinite vertical lines, called the *boundaries* of v, which partition the set of $2n$ vertical edges of the rectangles in S into c subsets of equal size, thereby creating c stripes $P_1(v), P_2(v), \ldots, P_c(v)$. We define the c subtrees rooted at the children of v recursively, each with respect to the vertical edges that fall into the same stripe. If the number of vertical edges is less than $\log^{\epsilon} n$ in a stripe, the child node corresponding to this stripe becomes a leaf node. Clearly the height of this tree is $O(\log n/\log\log n)$. A Q-heap $Q(v)$ holding the boundaries of v is built for each node v, which will enable a constant time identification of the stripe of v the query point belongs to. In addition, for each node v, except for the root, we define its *x-range* as the the stripe $P_i(u)$ of its parent u, assuming v is the ith child of u from the left.

We associate with each internal node v the rectangles that intersect at least one of its boundaries but none of the boundaries of its ancestors. Each leaf node contains the set of rectangles both of whose vertical edges lie within its x-range. Hence at most $O(\log^{\epsilon} n) = O(\log n/\log\log n)$ rectangles are associated with a leaf node, and no preprocessing will be required for these rectangles.

Now consider the set $S(v)$ of rectangles associated with an internal node v. As in [4], we build two hive-graphs $HL_i(v)$ and $HR_i(v)$, as defined in Section 2.3, for each boundary $b_i(v)$, $i = 1, \ldots, c-1$. The hive-graph $HL_i(v)$ is built on left vertical edges lying inside stripe P_i and is used to answer the semi-infinite segment intersection queries of the form: $(x_1 \leq x, y_1 \leq y \leq y_2)$; and $HR_i(v)$ is built on the right vertical edges lying inside stripe P_{i+1} and is used to answer the semi-infinite segment intersection queries of the form: $(x_2 \geq x, y_1 \leq y \leq y_2)$. In addition, let $S'(v)$ be a subset of $S(v)$ such that each rectangle in $S'(v)$ crosses at least two boundaries. We transform the coordinates of $S'(v)$ from the space $\Re \times \Re$ to $W(v)$ in the space $[1, 2, \ldots, c-1] \times \Re$ as follows. We transform each rectangle $[x_1, x_2; y_1, y_2]$ in $S'(v)$ into the rectangle $[k_1, k_2; y_1, y_2]$ in $W(v)$, where $b_{k_1}(v)$ and $b_{k_2}(v)$ are the leftmost and rightmost boundaries it crosses.

We now turn our attention to the query algorithm and postpone the description of the data structure for $W(v)$ until the end of this section. Using the Q-heaps stored at the internal nodes, we can in $O(\log n/\log\log n)$ determine the path from the root to the leaf node whose x-range contains the query point $p = (x, y)$. (We assume for simplicity that the x- and y-coordinates of p are different from that of the corners of the rectangles in S.) It is clear that only the rectangles associated with the nodes on this path can possibly contain the query point.

Consider a node v on this path. If v is a leaf node, we simply examine each rectangles associated with it, a process that takes $O(\log n/\log\log n + f(v))$ time, where $f(v)$ denotes the number of rectangles reported at node v.

If v is an internal node, we first decide which stripe of v the query point p belongs to, a task that can be done in $O(1)$ time, say it belongs to $P_i(v)$. The rectangles stored at node v and which contain p can be classified into three groups: (i) the set $L(v)$ that contains the rectangles whose left vertical edges lie inside $P_i(v)$; (ii) the set $R(v)$ that contains the rectangles whose right vertical edges lie inside $P_i(v)$; and (iii) the set $F(v)$ consisting of those rectangles whose horizontal edges cross $P_i(v)$ entirely. If $i = 1$, $R(v)$ and $F(v)$ do not exist. Similarly, if $i = c$, $L(v)$ and $F(v)$ do not exist.

By Lemma 2.4, the rectangles that belong to the first two groups can be identified in $O(1)$ time per rectangle reported if we apply the fast fractional cascading technique on the lists of the sorted y-coordinates of the vertices of the corresponding hive-graph. For example, we know that each rectangle $(x_1, x_2; y_2, y_2)$ associated with the hive-graph $HL_i(v)$ satisfies $x_2 \geq x$. Therefore, to find rectangles in $L(v)$, we only need to check the criteria: $x_1 \leq x$ and $y_1 \leq y \leq y$. Also note that a proper rectangle can be reported at most once in this process.

The remaining task is to determine the rectangles in group $F(v)$, which requires an additional data structure. We start from the set $W(v)$ consisting of rectangles of the form $(k_1, k_2; y_1, y_2)$, where k_1 and k_2 are integers between 1 and $c-1$. For each pair of different integers $i < j$ between 1 and $c-1$, we construct a cartesian tree $C_{i,j}(v)$ consisting of rectangles $(i, j; y_1, y_2)$ in $W(v)$ to answer the two-sided range queries in the form $y_1 \leq y \leq y_2$. Note that the total space is still linear and the use of the fast fractional cascading technique will enable us to access the appropriate nodes in time proportional to the number of rectangles reported.

However, we still need to resolve the problem of identifying which of these Cartesian trees should be accessed when handling a query. We cannot afford to access such a tree unless we are guaranteed to find at least one proper rectangle. To address this problem, we do the following.

We construct a look-up table $M(v)$ with n' rows, each corresponding to a distinct y-coordinate of the horizontal edges of the rectangles in $W(v)$ and occupying one word (of $\log n$ bits). The rows are sorted by increasing order of the y-coordinates. Let $y_1(v) < y_2(v) < \cdots < y_{n'}(v)$ be the set of distinct y-coordinates of the horizontal edges. Let $V_j(v) = (b_{c^3}, b_{c^3-1}, b_1)$ be a sequence of c^3 bits, where b_i is the ith bit from the lower end of the word representing the jth row of $M(v)$ (note that $c^3 < \log n$). The word $V_j(v)$ is evenly divided into c sections, each corresponding to a stripe of v (actually we only use $c-2$ of them which correspond to $P_2(v), \ldots, P_{c-1}(v)$). Let $(b_{l+c^2} b_{l+c^2-1} \cdots b_{l+1})$ be one of them that corresponds to $P_i(v)$, $i = 2, \ldots, c-1$. For each pair of integers $k_1 < i \leq k_2$ between 1 and $c-1$, we set the bit $b_{l+k_1 \cdot c+k_2}$ to one if there is a rectangle $(k_1, k_2; y_1, y_2)$ in R such that $y_1 < y_j(v) \leq y_2$. All the other bits are set to zero.

To find the proper rectangles in $S'(v)$, we first transform in $O(1)$ time using $Q(v)$ the query point (x, y) to the point (k, y) in the same space as $W(v)$. Let $b_k(v)$ be the leftmost boundary of v whose x-coordinate is greater than or equal to x. It is clear that a rectangle $(x_1, x_2; y_1, y_2)$ in $S'(v)$ contains (x, y) if and only if its corresponding rectangle $(k_1, k_2; y_1, y_2)$ in $W(v)$ contains (k, y). Let $y_j(v) = \min\{y_l(v) | 1 \leq l \leq n', y_l(v) \geq y\}$. We have the following lemma.

Lemma 5.1. *Let $(b_{l+c^2} b_{l+c^2-1} \cdots b_{l+1})$ be the section of $V_j(v)$ which corresponds to $P_k(v)$. Then for each pair of integers $1 \leq k_1 < k_2 \leq c-1$ such that $k_1 < k \leq k_2$, $b_{l+k_1 \cdot c+k_2} = 1$ if and only if there exists a rectangle $(k_1, k_2; y_1, y_2) \in W(v)$ which contains (k, y).*

Proof. By the definition of $V_j(v)$, $b_{l+k_1 \cdot c+k_2} = 1$ if and only if there exists a rectangle $(k_1, k_2; y_1, y_2)$, such that $y_1 < y_j(v) \leq y_2$. If this rectangle indeed exists, we have $k_1 \leq k \leq k_2$ and $y_2 \geq y_j(v) \geq y$. The definition of $y_j(v)$ ensures that $y \geq y_1$. Therefore $(k_1, k_2; y_1, y_2)$

contains (k, y). Now suppose there is a $(k_1, k_2; y_1, y_2)$ in $W(v)$ which satisfies $k_1 \le k \le k_2$ and contains (k, y). The only scenario in which $(k_1, k_2; y_1, y_2)$ does not satisfy $y_1 < y_j(v) \le y_2$ is $y_j(v) = y = y_1$. This is not possible given the assumption that y can not be the y-coordinate of any horizontal edge[1]. □

Lemma 5.1 shows that the section B of $V_j(v)$ which corresponds to the stripe containing (k, y) indicates correctly the Cartesian trees in $\{C_{i,j} | 2 \le i < j \le c - 1\}$ which should be visited. Using a look-up table of size $O(n)$, similar to the one described in [21], we can transform B into a list of integers $(m, I_1, I_2, \ldots, I_m)$, where m is the number of 1-bits in B and I_l is the index of a unique Cartesian $C_{i,j}(v)$, for $l = 1, 2, \ldots, m$. Then we simply visit these Cartesian trees one by one.

Searching the sorted lists associated with non-root nodes can be done using fast fractional cascading. The correctness proof and complexity analysis for this part is similar to that in Section 4 and thus is omitted here.

Theorem 5.1. *There exists a linear-space algorithm to handle the rectangle point enclosure queries in* $O(\log n / \log\log n + f)$ *time, where* f *is the number of segments reported.*

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of Latin American Theoretical Informatics*, pages 88–94, 2000.

[3] S. Bespamyatnikh and J. Snoeyink. Queries with segments in Voronoi diagrams. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 122–129, Baltimore, MD, 1999.

[4] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, Aug. 1986.

[5] B. Chazelle and L. J. Guibas. Fractional Cascading: I. A data structure technique. *Algorithmica*, 1(2):133–162, 1986.

[6] B. Chazelle and L. J. Guibas. Fractional Cascading: II. Applications. *Algorithmica*, 1(2):163–191, 1986.

[7] S.-W. Cheng, H. Everett, O. Cheong, and R. van Oostrum. Hierarchical vertical decompositions, ray shooting, and circular arc queries in simple polygons. In *Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 227–236, Miami Beach, FL, 1999.

[1] This assumption might seem to be crucial to the correctness of the Lemma. However, without this assumption, the only case that needs a special care is when (k, y) is contained in no rectangle of the form of $(k_1, k_2; y_1, y_2)$ except those satisfying $y_2 = y$. This can be fixed by modifying $L(v)$ to make sure that this special case is not missed.

[8] R. Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–220, 1986.

[9] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[10] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[11] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[12] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 135–143, Washington, DC, 1984.

[13] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[14] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[15] W. Lipski and F. P. Preparata. Segments, rectangles, contours. *Journal of Algorithms*, 2(1):63–76, 1981.

[16] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[17] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[18] F. P. Preparata and M. I. Shamos. *Computational Geometry – An Introduction*. Springer-Verlag, 1985.

[19] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[20] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[21] Q. Shi and J. JaJa. Fast algorithms for 3-d dominance reporting and counting. Technical Report CS-TR-4437, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[22] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2), 1979.

[23] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[24] D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion three. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.