Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds

Franco P. Preparata, *Jeffrey Scott Vitter*,
and *Mariette Yvinec*

Technical Report No. CS–89–50
revised August 1990

Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds[1]

Franco P. Preparata[2]
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Jeffrey Scott Vitter[3]
Department of Computer Science
Brown University
Providence, R. I. 02912–1910

Mariette Yvinec[4]
LIENS, URA CNRS 1327
Ecole Normale Supérieure
45, rue d'Ulm
75230 Paris, France

August 31, 1990

[1]An extended abstract of this work appeared in *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT '90)*, edited by J. R. Gilbert and R. Karlsson, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Volume 447, July 1990, 71–84.

[2]Support was provided in part by NSF research grant CCR–8906419. Part of this research was done while visiting Ecole Normale Supérieure in Paris, France.

[3]Support was provided in part by NSF Presidential Young Investigator Award CCR–8947808 with matching funds from an IBM research contract and by NSF research grant DCR–8403613. Part of this research was done while visiting Ecole Normale Supérieure in Paris, France.

[4]Support was provided by CNRS.

Abstract

We present a new hidden-line elimination technique for displaying the perspective view of a scene of three-dimensional isothetic parallelepipeds (3D-rectangles). We assume that the 3D-rectangles are totally ordered based upon the dominance relation of occlusion. The perspective view is generated incrementally, starting with the closest 3D-rectangle and proceeding away from the viewpoint. Our algorithm is scene-sensitive and uses $O((n+d)\log n \log\log n)$ time, where n is the number of 3D-rectangles and d is the number of edges of the display. This improves over the heretofore best known technique. The primary data structure is an efficient alternative to dynamic fractional cascading for use with augmented segment and range trees when the universe is fixed beforehand. It supports queries in $O((\log n + k)\log\log n)$ time, where k is the size of the response, and insertions and deletions in $O(\log n \log\log n)$ time, all in the worst case.

1 Introduction

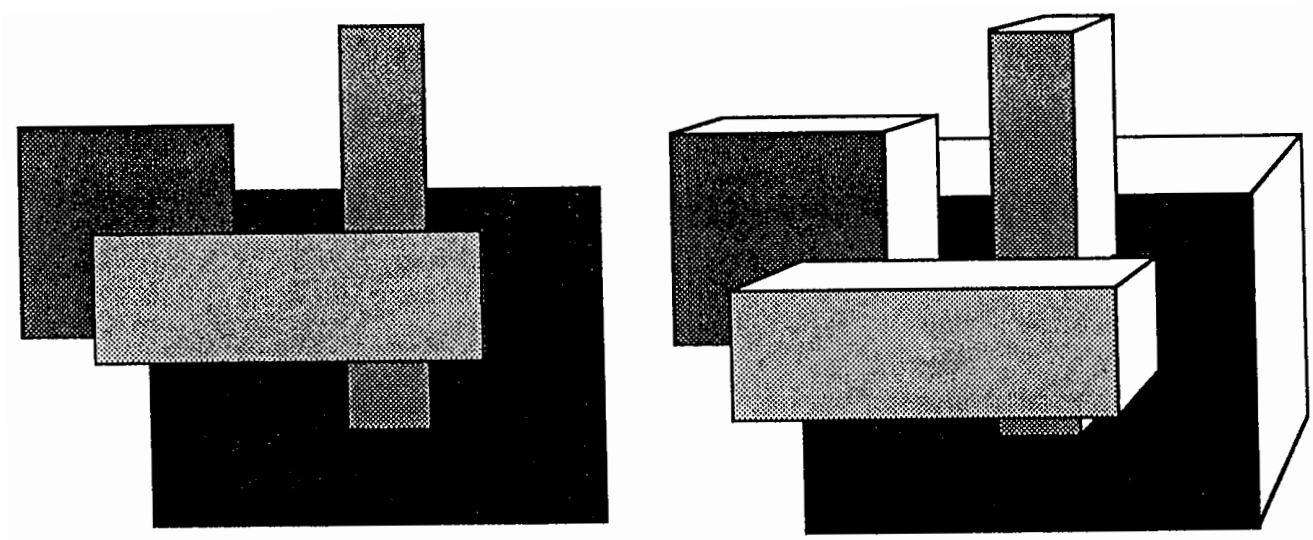
Substantial attention has been devoted to the generation of the planar display of a three-dimensional scene, a problem that is of central importance in areas such as computer graphics, animation, and flight simulation [9,18]. The scene is normally given as a collection of polyhedral objects. The goal is to produce a perspective view of the scene in which the hidden portions of the objects are eliminated and not displayed.

Known approaches to this important problem can be conveniently classified into two main types: *intersection-sensitive*, where the performance depends upon the number of intersections of the planar projections of the individual objects; and *output-sensitive*, where the performance depends upon the size of the display (considered as a planar graph). Output-sensitive approaches are obviously more attractive, since the size of the display is never larger than the size of the underlying intersection graph and is frequently much smaller. For that reason we adopt the output-sensitive approach in this paper.

Research on output-sensitive approaches is still in its infancy, and sophisticated capabilities to deal with general polyhedral scenes have yet to be developed. Known algorithms concern applications in which the scene polyhedra obey substantially restrictive constraints. Reif and Sen consider monotone polyhedral terrains [22]. Our starting point is the different model and approach of Güting and Ottmann [13], who considered the case of the axial view of a collection of n isothetic rectangles in space (that is, the view from infinity along the z-axis of rectangles whose sides are parallel to the x- and y-axes), as shown in Figure 1(a). Their algorithm makes use of augmented segment trees and requires $O((n+d)\log^2 n)$ execution time, where d is the complexity of the resulting display. Their approach is easily extended to c-oriented objects, which are polygons orthogonal to the z-axis whose sides have a fixed number c of different orientations.

The algorithm in [13] uses a "priority" sweep, that is, the rectangles are processed starting with the rectangle closest to the point of view and proceeding away from the viewpoint. It is trivial to establish that such a total ordering of the rectangles exists. In fact, the total order exists even when the objects are *3D-rectangles* considered in the axial view. A 3D-rectangle is defined to be an isothetic parallelepiped whose sides are oriented according to the x, y, and z axes. The image of each 3D-rectangle in the axial view is simply a (two-dimensional) rectangle, and the order in this case is determined by a total ordering of their frontal faces. However, in the nonaxial view of 3D-rectangles, it is possible that there is no total ordering of the 3D-rectangles; the "occlusion" relation may have cycles (see [12]).

The solution to the problem of the axial view of 3D-rectangles was recently improved upon by Preparata, Vitter, and Yvinec [21] with an $O(n\log^2 n + d\log n)$-time algorithm. The algorithm also uses a priority sweep, but with a very practical and efficient implementation of semidynamic augmented segment trees, based upon "contracted binary trees." Bern [3] and Näher, Mehlhorn, and Uhrig [17] obtained the same time bound using a radically different technique; their algorithms perform a planar sweep of the display and use the depth of the 3D-rectangles in the display plane (which in the axial view corresponds to their z-coordinates) to determine the visible portions of the 3D-rectangles being scanned. Fractional cascading can be used to improve their running times to $O(n\log n\log\log n + d\log n)$; for practical purposes, though, fractional cascading has a high overhead and is not routinely recommended for actual implementation. The running time can be improved by a different technique to



(a) (b)

Figure 1: (a) The axial view of a set of isothetic 3D-rectangles. (b) The perspective view.

$O((n+d)\log n)$ [3,17]. Atallah and Goodrich proposed an $O(n^{3/2}+d)$-time algorithm [1], and this was recently improved by Goodrich, Atallah, and Overmars using a priority sweep technique to $O(t(n^{1+2/t}+d))$ time, for any $t \geq 0$, which provides a continuous spectrum between $O(n^{3/2}+d)$ time and $O((n+d)\log n)$ time [11].

The next natural question to consider is how to generate the perspective view, as shown in Figure 1(b). We first considered extensions of the two available major approaches, namely, the priority sweep approach of [11,13,21] and the plane sweep approach of [3,17]. In both cases, it seems prudent to make the preliminary assumption that the object-occlusion relation is acyclic; this restriction does not exclude the very important case where the objects have disjoint projections on a plane, a subcase that occurs, for example, when all objects rest on a ground plane.

We encountered substantial difficulties in trying to extend the plane sweep method of [3, 17] to the perspective view. The plane sweep method relies upon managing the depth information during the generation of the visible portions of edges. This can be done easily for the axial view of 3D-rectangles, in which all edges are horizontal or vertical and the depth of each edge corresponds simply to its fixed z-coordinate. But in the perspective view of 3D-rectangles, the depth of each edge can change continuously, and this apparently cannot be handled by adapting the algorithms of [3,17].

On the other hand, the priority sweep approach of [13,21] can be extended to perspective views of 3D-rectangles. The priority sweep uses the distance of an object from the observer

in a rather crude way; it basically identifies an object with its projection from the observer onto a plane parallel to the display plane. We should note, however, that the sophisticated coherence technique and amortized time arguments used in [21] to get an algorithm faster than the one in [13] makes crucial use of the fact the the segments have only *two* distinct orientations; as a result this technique is not applicable to the case at hand.

In this paper we start with the basic priority sweep of [13] and introduce appropriate coherence devices to take advantage of the available prior knowledge of the individual scene objects. In the next section we review the basic principles behind the display algorithms of [13] and [21] for generating the display of a scene of n 3D-rectangles. The novelty of our algorithm comes from the very efficient data structures we use to store the edges and vertices of the silhouette. The data structures are efficient semidynamic implementations of augmented segment trees and range trees, in which the secondary data structure used to store the catalog at each node consists of a *contracted binary tree* (CBT) and a *contracted stratified tree* (CST). The data structures are of independent interest and can be thought of as practical alternatives to dynamic fractional cascading when the underlying universe is finite: their code is simpler to implement and faster to execute, and the time bounds are worst-case, not amortized. Normal fractional cascading, either in the static or dynamic setting, often involves too much overhead to be practical.

In Section 3 we give an overview of the data structures and discuss how they can be used to generate the special case of the axial view. In Section 4 we discuss the CBT data structure, and in Section 5 we introduce the CST data structure. We show how to generate the parallel and perspective views in Sections 6 and 7. The resulting display algorithm gives us our main result:

Theorem 1 *The perspective view from an arbitrary viewpoint of a set of n isothetic parallelepipeds (3D-rectangles) can be obtained in $O((n+d)\log n \log\log n)$ time using $O((n+d)\log n \log\log n)$ storage space, where d is the size of the display.*

2 The Display Algorithm

Our objective is to display the perspective view of the scene of n 3D-rectangles, bearing in mind that each 3D-rectangle is opaque and that hidden parts should not be displayed. The 3D-rectangles are processed successively, starting with the 3D-rectangles closest to the viewpoint and proceeding away from the viewpoint. We assume that each 3D-rectangle does not occlude any of the 3D-rectangles processed earlier. Initially, the display is empty and the first 3D-rectangle is trivially added.

In the general step, the display plane is partitioned into two portions: the *opaque* portion $\mathcal{F}$, consisting of the union of the projections onto the display plane of the 3D-rectangles already processed; and its complement $\overline{\mathcal{F}}$, the *transparent* portion. If r denotes the projection of the 3D-rectangle to process next, we must add to the display the visible parts of r (namely, $r \cap \overline{\mathcal{F}}$) and update the opaque portion to be $r \cup \mathcal{F}$. We refer to the boundary of the opaque portion $\mathcal{F}$ as the *silhouette* of $\mathcal{F}$. The silhouette is represented as a collection of directed polygons; the orientation on each polygon is chosen so that the interior of $\mathcal{F}$ lies to the left of each edge. The edges and vertices of the silhouette are stored in data structures that we shall describe in detail in the following sections.

The four steps described below are performed in sequence for each new 3D-rectangle r that is processed. For simplicity, we use the same notation r to denote both the 3D-rectangle and its projection onto the display plane. The first two steps generate the contribution of r to the display, while the subsequent steps update the data structures storing the silhouette.

Step 1. For each side s of the current 3D-rectangle r, we use the data structure storing the silhouette edges to intersect s with the current silhouette and obtain an ordered list of intersections.

Step 2. We generate the contribution of the current 3D-rectangle r to the display. To find the visible portions of side s of r, we determine if its first extreme point is internal or external to the silhouette. This is readily accomplished with the aid of the list of intersections obtained in Step 1. If the list is empty or if its first edge is rightward-directed, then p is external; otherwise p is internal. We can generate the visible portions of s by processing the list of intersections and alternating between external and internal segments of s.

Step 3 We use the data structure storing the silhouette vertices to find the edges of the silhouette that are internal to r.

Step 4. We update the data structures storing the vertices and edges of the silhouette by following three substeps:

a) We update the silhouette edges that are intersected by r, which were determined in Step 1. This involves deleting edges and reinserting the remaining parts back into the edge data structure.

b) We insert into the silhouette data structures the vertices and edges contributed by r, which were determined in Step 2.

c) We delete from the silhouette data structures the edges (and their vertices) of the silhouette that are internal to r, which we determined in Step 3.

In the next section we give an overview of the data structures we use to store the vertices and edges of the silhouette, and in the remaining sections we discuss their implementation and analysis. The data structures we use to store the vertices and edges require $O((n + d)\log n \log\log n)$ storage space, where d is the size of the final display. By the analysis we present in Theorems 3 and 4, denoting by k the number of items updated at each step, Steps 1 and 3 take $O((\log n + k)\log\log n)$ time, Step 2 takes $O(k)$ time, and each substep of Step 4 takes $O((k + 1)\log n \log\log n)$ time. Theorem 1 follows from these bounds, since the sum of k taken over all 3D-rectangles is $O(d)$.

3 Data Structures for the Axial View

In this section we give an overview of data structures we use to store the edges and vertices of the silhouette for the special case of generating the axial view, in which the image of each 3D-rectangle is a (two-dimensional) rectangle. The extensions to general parallel and perspective views will be discussed in Sections 6 and 7.

3.1 Segment Trees for Storing Silhouette Edges

The edges of the silhouette are stored in a dynamic data structure designed to answer intersection queries like those posed in Step 1 of the display algorithm: Given a side of a projected rectangle, find the list of silhouette edges that the side intersects, ordered according to their occurrence along the side. In addition, the data structure must be able to support insertions and deletions of edges, as done in Step 4.

In the case of axial views, the silhouette has only horizontal and vertical edges, and we use two symmetrical components $\mathcal{T}_x$ and $\mathcal{T}_y$ to store the edges. The component $\mathcal{T}_x$ maintains the set of horizontal edges of the silhouette and is queried in Step 1 with the vertical sides of the processed rectangles, while $\mathcal{T}_y$ maintains the vertical edges of the silhouette and is queried in Step 1 with horizontal sides of the rectangles. From now on, we restrict ourselves to the description of the data structure for the horizontal silhouette edges; the data structure for vertical edges is identical.

The data structure $\mathcal{T}_x$ is an *augmented segment tree*. The outer level of the data structure is a segment tree (see for example [20]), whose skeleton is built upon the set $X = \{x_l(r), x_r(r) : r \in R\}$ of the $2n$ x-coordinates of input rectangles. As is well-known, each node V of the segment tree, which we call a *primary node*, is associated with a standard interval $range(V)$ of the x-axis, which is the union of the elementary standard intervals associated with the leaves of the subtree rooted at V. Each horizontal silhouette edge is stored in the segment tree as $O(\log n)$ fragments, each of which is attributed to a given primary node. The nodes of $\mathcal{T}_x$ to which fragments of an edge e are attributed are said to be *allocated to* e, and the edge e is said to be *recorded at* those nodes. The set of nodes allocated to an edge e, which we call the *allocation set* of edge e, can be characterized as follows: A node V is allocated to e if and only if the horizontal range of e spans $range(V)$ but does not span $range(father(V))$. The reader is referred to [20] for further background on range trees and segment trees.

Each node V of the segment tree has attached to it a secondary structure designed to store the set $A(V)$ of edges recorded at the segment tree node. This sequence $A(V)$ of edges is called a *catalog* and is ordered by increasing y-coordinate. In order to insert and delete edges and perform intersection queries in an efficient way, the secondary structures we use are more sophisticated than the ordinary balanced trees. We discuss this further in Section 3.3 and later sections.

In Step 1, given a query $[x; y_1, y_2)$,[1] which is the vertical side of a rectangle, we can find the silhouette edges that it intersects by searching the catalog $A(V)$ of each of the $O(\log n)$ primary nodes V in $\mathcal{T}_x$ such that $x \in range(V)$. The nodes V form a path from the root to x's leaf in $\mathcal{T}_x$. In each catalog $A(V)$, the horizontal edges with y-coordinates in the range $[y_1, y_2)$ are returned in sorted order. To obtain the entire sorted collection of horizontal edges that are returned, it suffices to merge together the $O(\log N)$ sorted lists of edges; the merging process takes $O(k \log\log n)$ extra time, where k is the total number of horizontal edges satisfying the query.

[1] In this paper we use the notation $[p_1, p_2)$ or more explicitly $[x; y_1, y_2)$ to denote the semiclosed vertical segment whose bottom and top endpoints are $p_1 = (x, y_1)$ and $p_2 = (x, y_2)$, respectively. Similarly we use $[p_1, p_2)$ or more explicitly $[x_1, x_2; y)$ to denote the horizontal segment whose left and right endpoints are $p_1 = (x_1, y)$ and $p_2 = (x_2, y)$, respectively.

3.2 Range Trees for Storing Silhouette Vertices

The dynamic data structure we use to store the vertices of the silhouette answers queries of the following form: Given a rectangle, find the set of vertices of the silhouette contained in that rectangle. This type of query can be used to find the edges internal to the query rectangle, as we need to do in Step 3. The data structure must also be able to support the insertion and deletion of vertices, as needed in Step 4.

For the case of axial views, we use an *augmented range tree* data structure $\mathcal{R}_x$. The outer level of $\mathcal{R}_x$ is a range tree whose skeleton is built upon the set $X = \{x_l(r), x_r(r) : r \in R\}$ of the $2n$ x-coordinates of input rectangles. The range tree $\mathcal{R}_x$ is dual to the segment tree $\mathcal{T}_x$ we used in Section 3.1: Range trees store vertices, while segment trees store edges. The primary nodes in $\mathcal{R}_x$ allocated to a stored vertex correspond to the primary nodes in $\mathcal{T}_x$ accessed during a query, whereas the primary nodes in $\mathcal{T}_x$ allocated to a stored edge correspond to the primary nodes in $\mathcal{R}_x$ accessed during a query.

Each vertex $p = (x, y)$ is stored in $\mathcal{R}_x$ with $O(\log n)$ representatives. These representatives are recorded at the leaf node corresponding to x and at all the ancestors of the leaf node. Each node V of $\mathcal{R}_x$ has attached to it a secondary structure that stores the catalog $A(V)$ of vertices recorded at node V, ordered by y-coordinate. (There may be several vertices in a catalog that have the same y-coordinate; these are stored together as an unordered list.) The secondary structures used are, again, more sophisticated than the ordinary balanced trees and will be discussed further in the next section.

In Step 3, given a query rectangle $[x_1, y_1) \times [x_2, y_2)$, we can find the silhouette vertices it contains by searching the catalog $A(V)$ of each primary node V in $\mathcal{R}_x$ accessed during the search, which corresponds to a node allocated in $\mathcal{T}_x$ to the horizontal edge $[x_1, x_2; y_1)$. The vertices with y-coordinates in the range $[y_1, y_2)$ are returned. Once the vertices internal to r are found, it is easy to find the edges internal to r: they are the ones with both endpoints internal to r.

3.3 Overview of the Secondary Data Structures

In order to insert and delete edges and perform intersection queries in an efficient way, each secondary structure attached to a segment tree node embeds at the same time two related substructures, the *contracted binary tree* (CBT) and the *contracted stratified tree* (CST).

The CBT handles insertions and deletions very efficiently. Each edge has to be inserted into or deleted from the catalogs of $O(\log n)$ segment tree nodes. The main idea is to provide bridges that link together entries of different catalogs, thus customizing to our purpose the principles behind fractional cascading, which was introduced by Chazelle and Guibas [6], building on the works of [5,7,8,14], and adapted by Mehlhorn and Näher [16] to fully dynamic operations. In our data structure, the bridging between different catalogs is obtained by the introduction into the catalogs of additional dummy elements called *witnesses*.

While witnesses are crucial to the efficiency of insertions and deletions, they unfortunately pollute the answers to intersection queries, and filtering them out can drastically slow down the handling of these queries. The CST substructure attached to each segment tree node is designed to remedy this shortcoming. The CST is inspired by the priority queue structure of van Emde Boas, Kaas, and Zijlstra [4].

The CBT and CST data structures are efficient and easy-to-implement, and they allow practical realizations of augmented segment trees and augmented range trees in a dynamic setting when the underlying universe of keys is finite. We explore their structures and properties in detail in the following Sections 4 and 5.

4 The Contracted Binary Tree (CBT)

A *contracted binary tree* (CBT) [21] is a tree structure that implements a dynamic dictionary L of distinct elements belonging to a finite totally ordered universe U known *a priori*. Without significant loss of generality, universe U is assumed to have cardinality $2n$, where n is a power of 2, and is viewed as the set of integers $\{0, 1, \ldots, 2n-1\}$. The following dictionary operations can be performed on L: *insert*(y) causes the new element y to be inserted; *delete*(y) removes y; and *search*(y), where $y \in U$, returns the smallest element in L that is larger than y.

The CBT is best described in terms of its underlying "shadow structure," which is not actually implemented but is useful to consider for the purpose of exposition. (See Figure 2.)

Definition 1 The *shadow structure* $\mathcal{SS}$ is a complete (balanced) binary tree with $2n$ leaves ordered from left to right in the usual way and associated with the keys 0, 1, ..., $2n-1$.

Definition 2 The CBT $\mathcal{B}_L$ implementing the dictionary L is a binary tree built on the *active* nodes of $\mathcal{SS}$. The active nodes are defined recursively as follows:

1. The root is active.
2. A leaf is active if it belongs to L.
3. An internal node is active if both of its subtrees contain active nodes.

The tree structure of the CBT $\mathcal{B}_L$ is defined as follows: The parent of a node v in $\mathcal{B}_L$ is either the root or v's lowest active ancestor in the shadow structure $\mathcal{SS}$. In addition, the nodes of $\mathcal{B}_L$ are linked in a doubly linked list via *next* and *prev* pointers, corresponding to an inorder traversal of the tree.

For convenience, each leaf of the shadow structure $\mathcal{SS}$ may be thought as labeled with the binary representation of its key value. Each internal node is labeled with the common prefix of the labels of its leaf descendants. The following lemma follows immediately from the above definition of active nodes:

Lemma 1 *If L_1 and L_2 are two subsets of the universe such that $L_1 \subseteq L_2$, then for each node in $\mathcal{B}_{L_1}$ there is a node with the same label in $\mathcal{B}_{L_2}$.*

Two other notions are useful for characterizing our algorithms. Let y be an element in U that is not in L; y corresponds to a non-active leaf of $\mathcal{SS}$. The *neighbor* of y in $\mathcal{B}_L$ is the leaf in $\mathcal{B}_L$ whose label has the longest common prefix with the binary representation of y. Let z be an element in U. An ancestor u in $\mathcal{B}_L$ is said to be a *right ancestor* (respectively, *left ancestor*) of z if z is in the right (respectively, left) subtree of u in $\mathcal{SS}$. The *companion nodes* v and u of z are defined as follows: The first companion node v is the lowest active

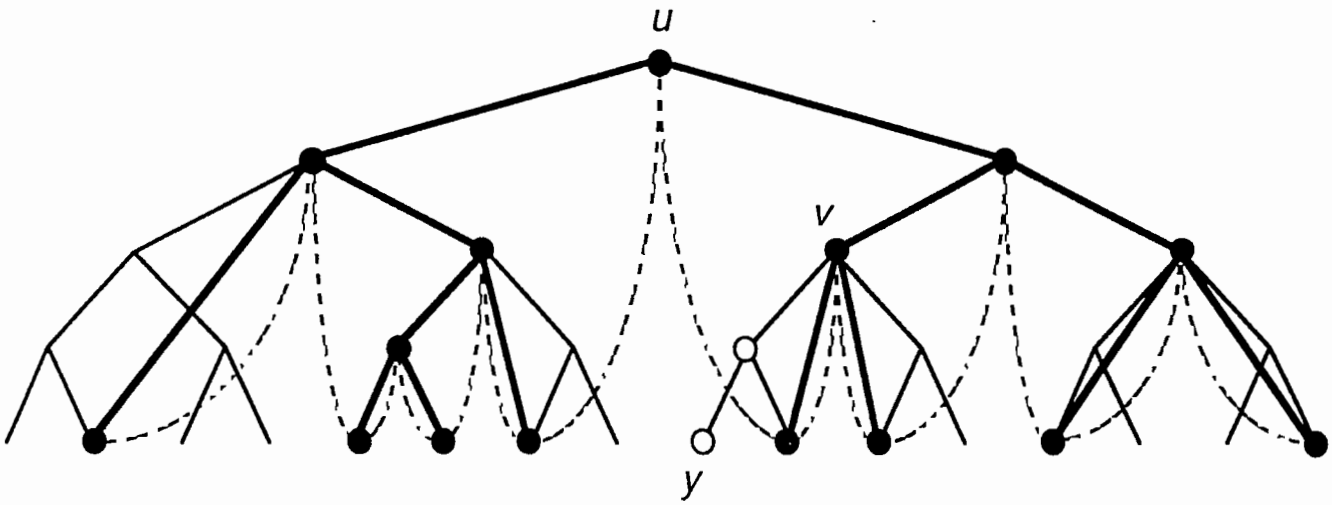


Figure 2: A contracted binary tree (CBT) and its underlying shadow structure. The active nodes are represented by black circles. The *parent*, *left*, and *right* links are shown as heavy lines, and the *next* and *prev* pointers are shown as dashed lines. The "companion nodes" v and u of the new leaf y are used to insert y in constant time.

ancestor of z in $\mathcal{SS}$. If v is a left ancestor, then the second companion node u is defined to be the lowest active right ancestor of z in $\mathcal{SS}$; otherwise, u is defined to be the lowest active left ancestor of z in $\mathcal{SS}$.

Lemma 2 *The CBT $\mathcal{B}_L$ uses $O(|L|)$ space and supports the operations of insert(y), delete(y), and search(y) in $O(\log n)$ time, where $2n$ is the size of the universe. In addition delete(y) can be performed in constant time if we are given a pointer to the leaf storing y in the CBT, and insert(y) can be done in constant time if we are given a pointer to y's neighbor in $\mathcal{B}_L$ or to its companion nodes.*

We refer the reader to [21] for the proof of Lemma 2, along with a complete discussion of the CBT algorithms and their analyses. In Sections 4.1 and 4.2, we build further upon the CBT data structure.

4.1 CBTs in Segment Trees

In the segment tree $\mathcal{T}_x$, which we use to store the horizontal edges of the silhouette, the edges present in the catalog $A(V)$ of a primary node V have distinct y-coordinates, belonging to the universe $U = \{y_b(r), y_t(r) : r \in R\}$, which consists of the $2n$ y-coordinates of the input rectangles. Each catalog $A(V)$ is stored in a CBT $\mathcal{B}_{A(V)}$. When a horizontal edge e is added to the silhouette, an entry for this edge has to be added to the catalog $A(V)$ of each primary node V allocated to e.

To insert these entries efficiently, we add to the data structure some links between entries in different catalogs. This system of links is described in detail later in this section. The links are analogous to the bridges between different catalogs used in the dynamic implementation

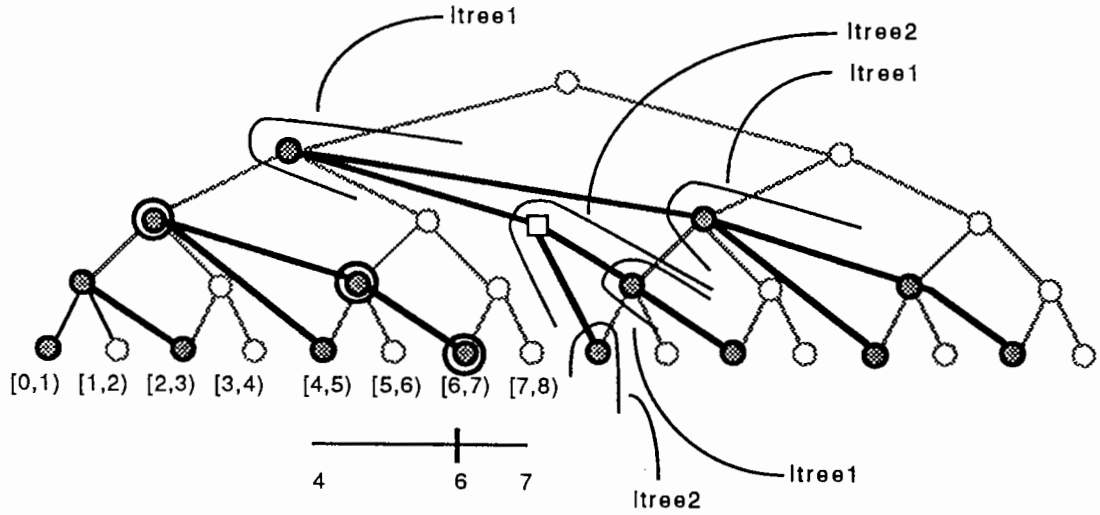


Figure 3: The left allocation forest G_l. For edge $e = (4, 7; y)$, the left allocation path includes the nodes allocated for e (which span x-ranges $[4, 6)$ and $[6, 7)$) plus the extra node that spans the x-range $[0, 4)$. Only one dummy node, shown as a square, occurs in this example.

of fractional cascading [16]. Our system of linked CBTs is however notably simpler than general dynamic fractional cascading, because it takes full advantage of the fact than the universe is finite and fixed in advance. In particular, no rebalancing of the structure is required when insertions or deletions are performed. The result is a simple implementation and fast performance.

Another original aspect of our data structure is that the idea of fractional cascading is applied (but with little overhead) to the insertion or deletion of an edge. This requires that we link together catalogs attached to primary nodes that are likely to appear together in the allocation set of an edge. These primary nodes are not connected by a simple path in the segment tree, so we construct a new structure, called the *allocation forest* G (see also [19]), which is a forest of binary trees whose nodes are the segment tree nodes (less the root) and additional "dummy" nodes. Graph G consists of two entirely symmetrical components, which we call the *left allocation forest* G_l and the *right allocation forest* G_r. For concreteness, we describe G_l, which is pictured in Figure 3; the description of G_r is obtained by substituting "right" to "left", and by modifying the definitions in an obvious way.

A non-root primary node of the segment tree is called a *left node* if it is the left child of its parent. The *left path* of a primary node V, which we denote $lpath(V)$, is defined to be the maximal path of left nodes distinct from V in the subtree rooted at V; we say that $lpath(V)$ issues from V. The construction of G_l can be explained easily in two steps: (i) First, we construct G'_l, a forest on the left nodes of $\mathcal{T}_x$, where the children of (left node) V are all the nodes on $lpath(V')$, V' being the sibling of V; the trees in G'_l are nonbinary trees rooted at the nodes of $\vec{lpath}(root)$, where *root* denotes the root of $\mathcal{T}_x$; (ii) Next, we transform G'_l into a binary forest G_l by introducing dummy nodes, as shown in Figure 3. Note that each binary tree in G_l has depth at most $\log n - 1$, and that the number of dummy nodes is a fraction

of the number of primary nodes.

More formally, for each primary node V, let $g(V)$ denote the left child of the sibling of V. We use the notation $T_l * V * T_r$ to denote the binary tree having T_l, V, and T_r as its left subtree, root, and right subtree, respectively. If V is either **nil** or is a leaf, we define $\mathit{ltree1}(V) := \mathit{ltree2}(V) := V$; otherwise, we denote the left child of V by $\mathit{lchild}(V)$, and we define

$$\begin{aligned} \mathit{ltree1}(V) &:= \mathit{ltree2}(\mathit{lchild}(g(V))) * V * \mathit{ltree1}(g(V)); \\ \mathit{ltree2}(V) &:= \mathit{ltree2}(\mathit{lchild}(V)) * W * \mathit{ltree1}(V), \end{aligned}$$

where W is a dummy node. The forest G_l is defined to be the union of the trees $\mathit{ltree1}(V)$, for each V on $\mathit{lpath}(\mathit{root})$. Each primary node contains a pointer to the root of its subtree in G_l.

We extend the notion of primary node to include the dummy nodes, and we store the catalog of each dummy node in a CBT as well. We also modify our definition of "recording an edge" so that each edge $e = [x_1, x_2; y)$ is recorded not only at the nodes allocated to e in $\mathcal{T}_x$, but also at the ancestors of these nodes in G, as shown in Figure 3. By so doing, e is recorded at the nodes of two simple paths in G, which we call its *left allocation path* in G_l and its *right allocation path* in G_r. We say that e is a *witness* in catalog $A(V)$ if e is recorded at V and V is not an allocation node of e (we use a boolean parameter to identify witnesses). Note that, in all cases, e is recorded at $O(\log n)$ primary nodes. The stated policy insures that $A(V') \subseteq A(V)$ whenever V' is a child of V in G, and therefore the condition of Lemma 1 is established for the corresponding CBTs.

The edges of the allocation forest G define the linking structure between the different catalogs. To implement such a linking, each node in a CBT contains three pointers *alloc_plink*, *alloc_rlink*, and *alloc_llink*. Let V be a primary node and let V' be a child of V in the allocation forest G. Since $A(V') \subseteq A(V)$, it follows by Lemma 1 that for each node v' in the CBT storing $A(V')$ there is an identically labeled node v in the CBT storing $A(V)$. We set the pointer values accordingly:

$$\begin{aligned} \mathit{alloc\_plink}(v') &:= v; \\ \mathit{alloc\_llink}(v) &:= v', \qquad \text{if } V' \text{ is the left child of } V \text{ in } G; \\ \mathit{alloc\_rlink}(v) &:= v', \qquad \text{if } V' \text{ is the right child of } V \text{ in } G. \end{aligned}$$

Undefined pointers are set to the value **nil** as usual. With this data structure we can efficiently insert and delete edges:

Theorem 2 *The total time required for the insertion or deletion of an edge in the CBTs attached to the primary nodes on its left and right allocation paths is* $O(\log n)$.

Proof: In order to insert an edge e into the data structure, we execute in $O(\log n)$ time the standard segment tree visit to locate a primary node of the left and right allocation paths of e in G (whichever exists). For each such allocation path, by following a pointer we find in constant time the root V of the path. Edge e is recorded at V by finding in $O(\log n)$ time the neighbor of e in $\mathcal{B}_{A(V)}$ and then inserting e into it in constant time (see Lemma 2). As a byproduct of the insertion of e, we find the companion nodes v and u of e in $\mathcal{B}_{A(V)}$, as defined in Section 4. Let V' be the child of V in the left allocation path, and without

loss of generality, suppose that V' is the *left* child of V. Since $A(V') \subseteq A(V)$, in order to find the companion nodes v' and u' of e in $\mathcal{B}_{A(V')}$, we walk upward from v toward the root in $\mathcal{B}_{A(V)}$ until we find a node w such that $alloc\_llink(w) \neq \textbf{nil}$; we set $v' := alloc\_llink(w)$ since this v' is the lowest active ancestor of e in $\mathcal{B}_{A(V')}$. If v' is a left (right) ancestor of e, we find the other companion node u' by walking upward from u until we find a node w such that $alloc\_llink(w) \neq \textbf{nil}$ and $alloc\_llink(w)$ is a right (left) ancestor of e; we then set u' to be $alloc\_llink(w)$. Using v' and u', we insert e in constant time into $\mathcal{B}_{A(V')}$, by Lemma 2. We iterate this process until e has been recorded at all the primary nodes of the chosen allocation path. Symmetrically we record e at all nodes of its other allocation path. The deletion of an edge is straightforward and completed in time $O(\log n)$, since the entries in the CBTs for a given edge are linked together.

We now show that the total time needed to insert or delete an edge is $O(\log n)$. Binary searches are performed at the root nodes of the right and left allocation paths in $O(\log n)$ time. The total time spent finding the companion nodes of edge e in the other primary nodes of its allocation paths is $O(\log n)$ by the following argument: Each time we traverse a CBT to find the companion nodes, we walk upwards towards the root. In terms of the shadow structure of the CBTs, the visited nodes lie along two upward paths. The height of the shadow structure is $\log n + 1$, so the total time needed to traverse the CBTs is $O(\log n)$. Once e's companion nodes are known for a CBT, the insertion of e can be performed in constant time. □

A separate bridge structure is needed to speed up queries. Let us recall that when we process a query $e = [x; y_1, y_2)$ we traverse a path in $\mathcal{T}_x$ from the root to the leaf associated with x; at each segment tree node V we search the catalog $A(V)$ and return all edges whose y-coordinates are in the range $[y_1, y_2)$. In order to accelerate the searching, we need extra witnesses that link together corresponding entries in the CBTs along this search path.

The solution is as follows: Whenever we record an edge e at a primary node V using the process outlined above in Theorem 2, we also record e as a witness at all ancestors of V in $\mathcal{T}_x$. We maintain a set of pointers similar to those defined above for linking together catalog entries. Each CBT node contains three additional pointers: *search_plink*, *search_rlink*, and *search_llink*. Let V be any segment tree node and let V' be a child of V in $\mathcal{T}_x$. The addition of the new witnesses guarantees that $A(V') \subseteq A(V)$. Hence, by Lemma 1, for each node v' in $\mathcal{B}_{A(V')}$ there is an identically labeled node v in $\mathcal{B}_{A(V)}$. We set

$$\begin{aligned} search\_plink(v') &:= v; \\ search\_llink(v) &:= v', \quad \text{if } V' \text{ is the left child of } V \text{ in } \mathcal{T}_x; \\ search\_rlink(v) &:= v', \quad \text{if } V' \text{ is the right child of } V \text{ in } \mathcal{T}_x. \end{aligned}$$

Undefined pointers are set to the value **nil** as usual.

The insertion and deletion procedures outlined in Theorem 2 remain valid even with these extra witnesses, since the crucial subset property remains valid. The following lemma shows that the total number of witnesses introduced to realize the two bridge structures described above is not too large:

Lemma 3 *Each inserted edge causes $O(\log n)$ witnesses to be introduced.*

Proof: The bridge structure for the queries requires that an edge e be recorded as a witness at all primary nodes that are ancestors of the nodes on its allocation paths. As is well-known (see [20], for example), the union of such ancestors consists of the $O(\log n)$ nodes not allocated for e that are visited during the insertion process. The bridge structure for insertion and deletion requires that an edge be recorded at all nodes of its left and right allocation paths in G, each having at most $\log n$ nodes. Since witnesses of e appear at a subset of these nodes, the result follows. □

Given a query vertical rectangle side $e = [x; y_1, y_2)$, we process the query by starting at the root *root* of $\mathcal{T}_x$ and performing a binary search for y_1 in the CBT $\mathcal{B}_{A(root)}$. As a byproduct we also find the companion nodes u and v of y_1 and the neighbor of y_1 in $\mathcal{B}_{A(root)}$. We then repeat the following process for each node V on the path from *root* to the leaf node associated with x in $\mathcal{T}_x$: The entries with y-coordinate in the range $[y_1, y_2)$ in $A(V)$ are found via a linear walk through $\mathcal{B}_{A(V)}$, starting at y_1's neighbor. (Recall that the CBT entries are linked together in y-sorted order.) If V is a leaf in $\mathcal{T}_x$, the query halts. Otherwise, we let V' be the child of V in $\mathcal{T}_x$ on the query path. We can find the companion nodes u' and v' of y_1 in $\mathcal{B}_{A(V')}$ by an upward walk from u and v in $\mathcal{B}_{A(V)}$, as in the proof of Theorem 2. Given u' and v', the neighbor of y_1 can be found in constant time.

Unfortunately, this algorithm does not perform queries in $O(\log n + k)$ time, where k is the number of intersected horizontal edges of the silhouette. The reason is that there might be many witnesses returned as the result of a query. Since witnesses are really "dummy" entries, they have to be filtered out during the query process. The resulting time bound is $O(\log n + k + w)$, where w, the numbered of witnesses encountered, might be significantly greater than $\log n + k$.

The solution is to use a complementary data structure at each segment tree node that allows us to "jump over" witnesses and find the truly recorded edge e having the next-larger y-coordinate. The data structure has to be dynamic and allow the insertion and deletion of witnesses and truly recorded edges. The contracted stratified tree (CST) data structure discussed in Section 5 allows us to do the jumping in $O(\log \log n)$ time (see Theorem 6). This gives us the following performance:

Theorem 3 *Given a query vertical side, an ordered list (by y-coordinate) of the horizontal edges of the silhouette that it intersects can be found in* $O((\log n + k) \log \log n)$ *time, where* k *is the number of intersected edges. The time required to insert or delete an edge is* $O(\log n \log \log n)$.

The method described above finds the k intersected horizontal edges, but not in sorted order. The edges can be sorted in $O(k \log \log n)$ extra time, as explained at the end of Section 3.1, which can be absorbed into the time bound of Theorem 3. We should note that the time for insertion and deletion increases from the $O(\log n)$ bound quoted in Theorem 2 to $O(\log n \log \log n)$ because of the updating required for the CST.

4.2 CBTs in Range Trees

We augment the primary nodes of the range tree $\mathcal{R}_x$ used to store the silhouette vertices in the same way we did to the segment trees $\mathcal{T}_x$ and $\mathcal{T}_y$ in Section 4.1. That is, we add

dummy nodes so that the resulting allocation forest consists of a forest of binary trees. We use bridges as before also.

As noted in Section 3.2, the way we use the range tree $\mathcal{R}_x$ is dual to how we use $\mathcal{T}_x$, and this reflects itself in how we use the allocation forests. The primary nodes accessed during a query correspond to allocation paths. Each vertex $v = (x, y)$ stored in $\mathcal{R}_x$ is recorded at the range tree nodes along a path from the leaf associated with that vertex to the root. We also record v as a witness at all primary nodes on the left and right allocation paths for the edges $(-\infty, x; y)$ and $[x, +\infty; y)$, as defined in the last section. An analysis similar to that in Section 4.1 gives us the following performance bounds:

Theorem 4 *Given a query rectangle, the vertices in the silhouette that it contains can be found in $O((\log n + k) \log\log n)$ time, where k is the number of interior vertices. The time required to insert or delete a vertex is $O(\log n \log\log n)$.*

5 The Contracted Stratified Tree (CST)

The bridge data structures described in Section 4 enable us to find the elements within a specified range of the accessed expanded catalogs. By expanded catalog, we mean the catalog consisting of witnesses as well as truly recorded entries. Let us refer to the truly recorded entries as *marked*. In order to filter out the witnesses during the query, we need to be able to quickly "jump over" the unmarked entries in a list of entries and find the next-larger marked entry.

The problem can be formally stated as follows: We define the *expanded catalog* A to be a subset of a totally ordered universe U, and we define the *catalog* (or *marked set*) to be a subset C of A. Both A and C are dynamic sets. Modifications to A are called *insert* and *delete*, and modifications to C are called *mark* and *unmark*. Given $x \in A$, a search (called *find*(x)) returns the smallest element y in C that is $\geq x$.

The problem is a classical one and has been considered in various settings by several authors. A straightforward use of the CBT to solve the above search problem results in $O(\log n)$-time performance, which exceeds our allowance. Our goal is to construct a practical data structure that can perform the above tasks in $O(\log\log n)$ time.

The prototypical solution is offered by the stratified tree structure of van Emde Boas, Kaas, and Zijlstra [4]. Similar ideas are used in the priority queue data structure of Johnson [15], which implements a different set of primitives. The data structure is semidynamic, in that it is designed for a universe known *a priori*, as is our set U. However, the stratified tree of [4] must be fully initialized and uses $O(n \log\log n)$ storage space irrespective of the number $|A|$ of items it actually stores. This does not suit our needs, since each node in the segment tree requires such an auxiliary structure, many of which store relatively few items. The stratified tree data structure of [4] also provides a more powerful repertory of operations than needed in our application. In our case, the crucial operation *insert*(x), which we call the *guided insertion* of x into A, begins with full knowledge of the two consecutive elements x_1 and x_2 in A such that $x_1 \leq x < x_2$.

Mehlhorn and Näher [16] recently proposed a structure substantially inspired by [4], but with memory requirement $O(|A| \log\log |A|)$ rather than $O(|U| \log\log |U|)$. It is designed to solve a generalized version of guided insertion, where the totally ordered set U need not be

finite. In such cases, the primitive operations are comparisons in U rather than the more efficient manipulations of the normalized representations of the elements of U (the integers $0, 1, \ldots, |U| - 1$ represented in binary). This added generality considerably complicates the implementation of the *insert*, *delete*, and *find* operations, since rebalancing is required in order to achieve the time bound of $O(\log\log|A|)$, which moreover holds only in the amortized sense.

The most appropriate data structure is one that most economically fits the constraints of the application. In our case, we know *a priori* the universe U and can take advantage of guided insertion. Our proposed data structure, called the *contracted stratified tree* (CST) combines in a nontrivial and efficient way the features of the two data structures mentioned above. It performs the *insert*, *delete*, and *find* operations in $O(\log\log n)$ worst-case time using $O(|A|\log\log n)$ space.

The motivation behind the CST is germane to that of the CBT. We start with the same underlying shadow structure $\mathcal{SS}$, described in Section 4, which is a (fixed) balanced binary tree whose leaves correspond to the elements of U. We implement only those nodes of $\mathcal{SS}$ that are necessary in the execution of the desired operations. The set of nodes implemented in the CST for A will certainly contain all the nodes of the CBT for A, along with an additional set of nodes used to efficiently support the operations *find*, *mark*, and *unmark*.

For ease of presentation we assume $|U| = 2^K$, that is, the elements of U are the K-bit integers; the structure can be readily adapted to the general case. The shadow structure $\mathcal{SS}$ is the balanced binary tree with 2^K leaves; leaves are designed to store the expanded catalog A, and internal nodes contain tree-traversal information. The *level* of a node u of $\mathcal{SS}$ (denoted *level*(u)) is the number of edges in the root-to-leaf paths of the subtree rooted at u. Leaves have level 0, for example. For convenience, we label each leaf with the K-bit representation of its position in $\mathcal{SS}$, and we label each internal node with the maximal common prefix of the labels of the leaves in its subtree. Thus, the label of a node u of $\mathcal{SS}$ has $K - level(u)$ bits, and the root is labeled by the empty string.

The levels of $\mathcal{SS}$ are organized hierarchically as follows. Let $\mathcal{L}$ be a balanced (but otherwise arbitrary) segment tree on the integer set $\{0, 1, \ldots, K\}$, each integer corresponding to a level of $\mathcal{SS}$. (See Figure 4 for the case $K = 9$, where each node of $\mathcal{L}$ is pictured with its associated range.) An edge joining a node of $\mathcal{SS}$ at level i to a node of $\mathcal{SS}$ at level j, where $i < j$, can be viewed as an interval $[i, j)$; this interval (and the corresponding edge in $\mathcal{SS}$) is uniquely partitioned by $\mathcal{L}$ according to the well-known segment tree mechanism.

For any given level j, we define *outdegree*(j) (respectively, *indegree*(j)) to be the number of proper intervals that have integer j as their lower (respectively, upper) extreme. In Figure 4, we have *indegree*$(0) = 0$, *outdegree*$(0) = 4$; *indegree*$(2) =$ *outdegree*$(2) = 2$; *indegree*$(3) = 1$, *outdegree*$(3) = 2$; and so on.

Let $[i, j)$ be the range of node w in $\mathcal{L}$. Any subtree of $\mathcal{SS}$ with its root at level j and its leaves at level i is called a *canonical subtree* of $\mathcal{SS}$, with *base* at i, *top* at j, and *span* $[i, j)$. The *order* of a canonical subtree T is the distance (in terms of number of edges in $\mathcal{L}$) from the root to the node of $\mathcal{L}$ whose span is the span of T. In particular, the root has order 0, its children have order 1, and so on. Given two canonical subtrees T_1 and T_2 such that $T_1 \subseteq T_2$, we say that T_1 is a *substructure* of T_2. The "substructure" relation is clearly a partial order.

With this background we can now proceed to the following definition of the CST $\mathcal{S}_A$ for a given catalog A:

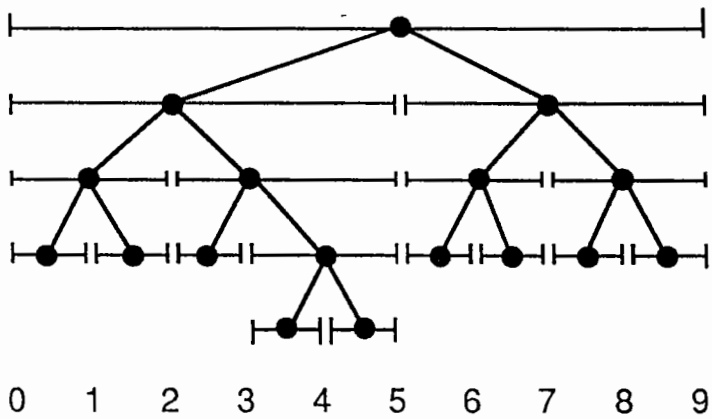


Figure 4: A segment tree $\mathcal{L}$ with base $\{0, 1, \ldots, K\}$, shown for $K = 9$. Each node of $\mathcal{L}$ is shown with its associated range.

Definition 3 Let $\mathcal{B}_A$ be the CBT for the expanded catalog A. (Recall that the root is always present.) The nodes of $\mathcal{B}_A$ are called *branching nodes*. Each edge (u, w) of $\mathcal{B}_A$ is partitioned by $\mathcal{L}$ and is replaced by a chain where a node (called a *buffer node*) is inserted at each cut-point. The branching nodes and buffer nodes are called *active nodes*, and they form the node set of the CST $\mathcal{S}_A$. Some of the nodes of the CST $\mathcal{S}_A$ are *marked*. A leaf is marked if the corresponding key belongs to the catalog, and an internal node is marked if it has a marked child.

In the Appendix to this paper we establish an interesting and insightful link between $\mathcal{S}_A$ and the stratified tree of [4].

We now show that the implantation of the buffer nodes upon the CBT provides the support for the pointers needed to implement the operations *find*, *mark*, and *unmark*. A canonical subtree of $\mathcal{SS}$ is called a *canonical subtree* of $\mathcal{S}_A$ if its root and at least one of its leaves are nodes of $\mathcal{S}_A$. For each canonical subtree T_p of $\mathcal{S}_A$ of order p having node u of $\mathcal{S}_A$ as a leaf, u has a pointer $rep_p[u]$ to the root of T_p. Conversely, for each canonical subtree T_p of $\mathcal{S}_A$ of order p having u as its root, u has pointers $min_p[u]$ and $max_p[u]$ to its marked leaves with smallest and largest values, respectively. (If none of its leaves are marked, we have $min_p[u] = max_p[u] =$ **nil**.) Referring to the mechanism that establishes canonical subtrees of $\mathcal{S}_A$, node u of $\mathcal{S}_A$ is equipped with three indexed sets of pointers $rep[h:k]$, $min[h:l]$, and $max[h:l]$, where h is the minimum order of canonical subtrees for which u is either the root or a leaf, and $k \leq h + outdegree(u)$, and $l \leq h + indegree(u)$. In other words, we must accommodate the appropriate pointers for each existing canonical subtree of which u is either the root or a leaf. Following [16] it is convenient to view node u of $\mathcal{S}_A$ as a list of *subnodes* $u_h, u_{h+1}, \ldots, u_{\max\{k,l\}}$, such that $u_{i+1} = succ(u_i)$, for $h \leq i \leq \max\{k, l\} - 1$. Each subnode is equipped with the appropriate fields *rep*, *min*, and *max*. Node u_h is also denoted $top(u)$. All subnodes of u are given the same label as u; from the implementation point of view, all subnodes have a pointer to a common instantiation of the label of u. This interpretation is

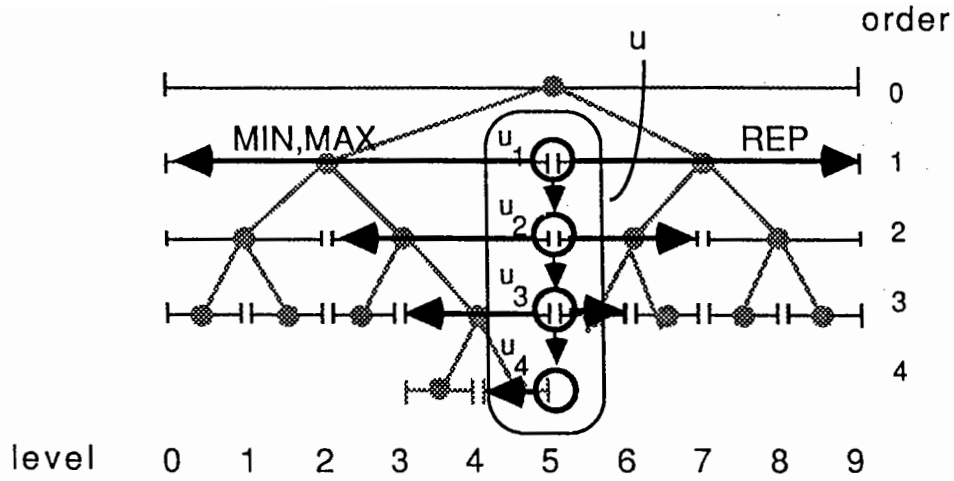


Figure 5: The subnodes u_1, u_2, u_3, and u_4 of a given node u in $\mathcal{S}_A$ on level 5. For each subnode, the levels of the nodes pointed to by its *min*, *max*, and *rep* pointers are indicated by arrows. More specifically, there are four nodes in the segment tree $\mathcal{L}$ whose span ends at 5; these spans start at 0, 2, 3, and 4. Therefore, u has a pair of *min* and *max* pointers for each level $\in \{0, 2, 3, 4\}$ for which u has a descendant in $\mathcal{S}_A$ on that level. Similarly, there are three nodes in the segment tree $\mathcal{L}$ whose span starts at 5; these spans end at 6, 7, and 9. Therefore, u has a *rep* pointer for each level $\in \{6, 7, 9\}$ for which u has an ancestor on that level.

illustrated in Figure 5, where a node u of $\mathcal{S}_A$ at level 5 is assumed to have its pointers fully utilized and is replaced by the chain of subnodes u_1, u_2, u_3, and u_4. In Figure 6 we show an example of a CST $\mathcal{S}_A$, complete with the specification of its *min* and *max* pointers. Each pointer is expressed with as many bits as in the label of its destination.

We now analyze the storage used by the pointer structure introduced above. Since $\mathcal{S}_A$ has $|A|$ leaves, the number of branching nodes in the CST is either $|A|-1$ or $|A|$ (it is $|A|$ when the root has degree 1) and the total number of nodes is $O(|A| \log\log |U|)$. Each node of $\mathcal{S}_A$ can have as many as *indegree*(u) *min* and *max* pointers and *outdegree*(u) *rep* pointers. Although both *indegree*(u) and *outdegree*(u) are $O(\log\log |U|)$, one should not hastily conclude that space $O(|A|(\log\log |U|)^2)$ is necessary. We prove the following:

Lemma 4 *When adding a leaf x to A (via insert(x)) the updating of the pointers rep, min, and max for the newly created nodes takes $O(\log\log |U|)$ time and requires $O(\log\log |U|)$ additional storage.*

Proof: The insertion of a leaf x is correctly viewed as the splitting of an existing chain from u to v at a branching node w in its interior, and the addition of a chain from x to w (see Figure 7). We first consider the effect of inserting w into the chain between u and v (Figure 7(b)). If w coincides with one of the existing buffer nodes, then no additional *rep* pointers are necessary. In the opposite case, w falls between two original buffer nodes w_1 and w_2, which are the extremes of the range of a node of $\mathcal{L}$. The original edge (w_1, w_2) is fragmented by the insertion of w according to the structure of $\mathcal{L}$ (Figure 7(c)), and the

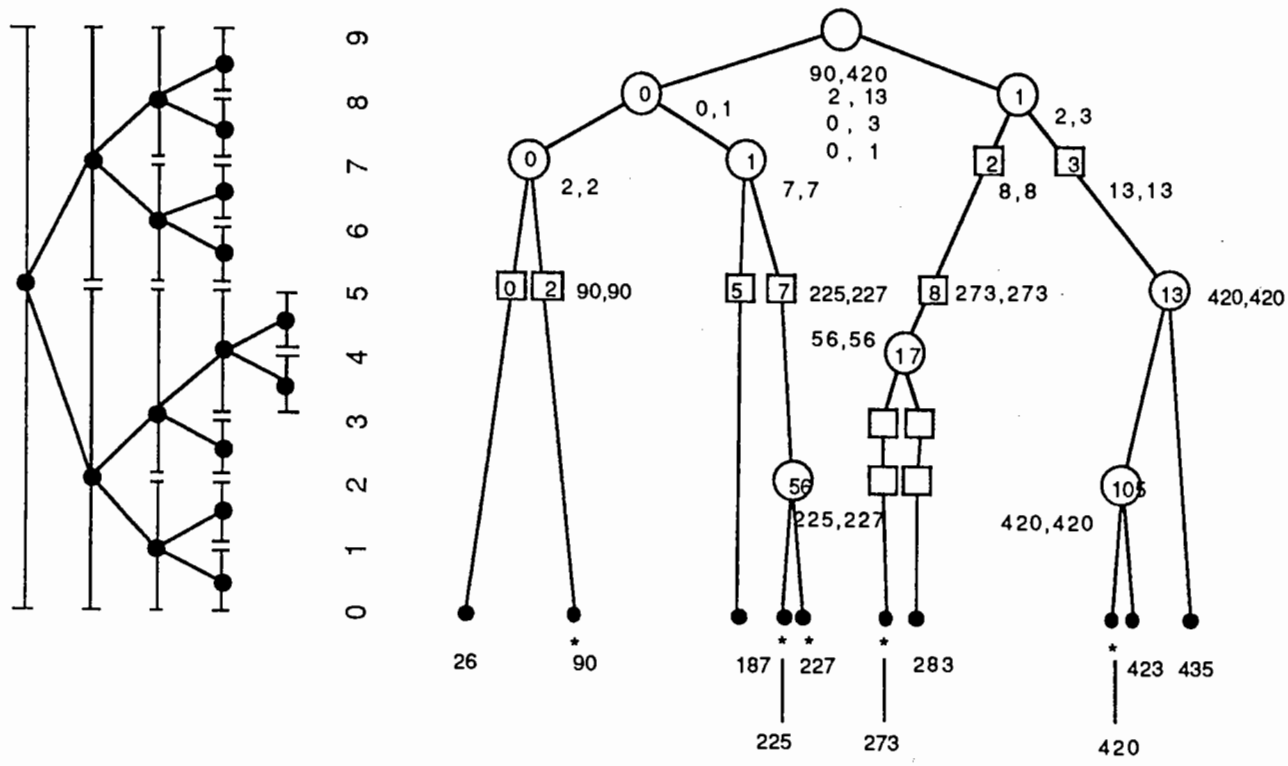
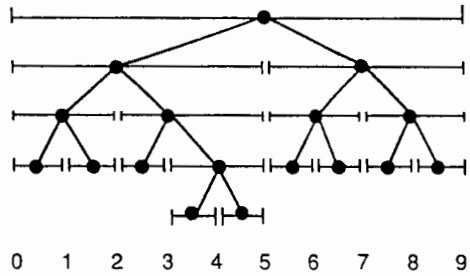


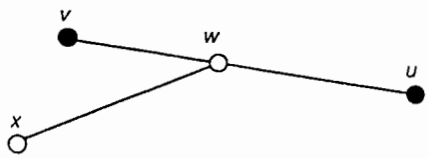
Figure 6: The CST $\mathcal{S}_A$ for $A \subseteq \{0, 1, \ldots, 511\}$. The marked elements C are designated by "*". Each node is labeled with the decimal interpretation of its bit string. Only the *min* and *max* pointers that are not **nil** are shown; each pointer is represented by the label value it points to. Shown on the left is the underlying segment tree $\mathcal{L}$. For example, there are four nodes of $\mathcal{L}$ whose spans end at 9; these spans start at levels 0, 5, 7, and 8. respectively; thus each node in $\mathcal{S}_A$ at level 9 (in this case, there is only one such node, namely, the root of $\mathcal{S}_A$) can serve as the root of four canonical subtrees with leaves at 0, 5, 7, and 8, respectively. The root of $\mathcal{S}_A$ has four pairs of *min* and *max* pointers, one for each of four canonical subtrees that hang below it. Similarly, there are four nodes of $\mathcal{L}$ whose spans start at 0; these spans end at 1, 2, 5, and 9. The node labeled 187 in $\mathcal{S}_A$ on level 0 has no ancestors on levels 1 or 2, but it does have ancestors on levels 5 and 9, and thus it is a leaf of canonical subtrees rooted at levels 5 and 9. Its *rep* pointers (not shown) point to the node labeled 5 on level 5 and to the root on level 9.

additional required pointers are confined to the interior of this range (Figure 7(d)). Referring to the segment tree mechanism, edge (w_1, w_2) is recursively split and a buffer node w' is created at each level where splitting occurs, until $w' = w$. Each of the $O(\log\log|U|)$ nodes that are introduced requires the creation of three subnodes and six pointers; each subnode can be initialized in constant time, since it simply extends the existing chain of subnodes of a given node.

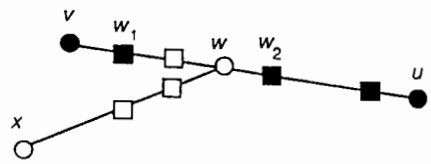
A minor modification of the preceding argument allows us to analyze the insertion of the chain from leaf x to branching node w. The insertion of this chain can be viewed as the splitting at node w of a *single* edge from x to the root of $\mathcal{S}_A$, in which case we do not add any buffer nodes on the resulting chain from w to the root. Since the splitting of the edge (x, w) creates $O(\log\log|U|)$ buffer nodes, it follows from the preceding argument that $O(\log\log|U|)$



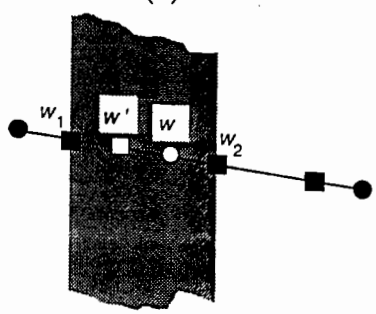
(a)



(b)



(c)



(d)

Figure 7: Illustration of the insertion of leaf x into the CST, as explained in the proof of Lemma 4. (a) The underlying segment tree $\mathcal{L}$ for the CST. Note in particular that x is on level 0 since it is a leaf. In (b)–(d), existing nodes are depicted as filled, and nodes being inserted are depicted as unfilled. Squares represent buffer nodes, and circles represent braching nodes. (b) The corresponding insertion into the CBT represented by the branching nodes. (c) The corresponding insertion into the CST. (d) The range of levels in the pre-update CST where alterations due to the insertion may occur.

time and storage are used. Combining the two arguments establishes the lemma. □

This leads to the following theorem:

Theorem 5 *The storage used by CST $\mathcal{S}_A$ is $O(|A| \log\log |U|)$.*

Proof: The theorem follows immediately from Lemma 4 and the following fact: The structure of $\mathcal{S}_A$ is unique, and its construction is achieved by a sequence of $|A|$ insertions, beginning with a tree containing only the root. □

We now consider the performance of the other operations on $\mathcal{S}_A$. By analogy with the preceding analysis, the deletion of a leaf x of A and the updating of the pertinent nodes and pointers can be done in time $O(\log\log |U|)$. We now consider the performance of the other operations on $\mathcal{S}_A$.

Let us consider next the management of the marked set C and the execution of the operations *find*, *mark*, and *unmark*. To gain some intuition, if $y = find(x)$ is defined (that is, if $y \neq$ **nil**) there are two paths in $\mathcal{S}_A$ issuing, respectively, from leaves x and y (with x to the left of y) and converging at their lowest common ancestor $lca(x, y)$. The objective of *find* is to "home in" on $lca(x, y)$, specifically, on the unique maximal (that is, having minimum order) canonical substructure T' of $\mathcal{S}_A$ having $lca(x, y)$ as its root.

This search may be viewed as the traversal of a sequence of nested canonical substructures of $\mathcal{S}_A$, starting with the entire CST $\mathcal{S}_A$ and ending with T' (see Figure 8 for an example). Each of these substructures corresponds to a node of $\mathcal{L}$, and the history of a *find* operation is correctly interpreted as the traversal of a path in $\mathcal{L}$ starting at the root. The advancing mechanism of *find* performs the test for termination at T', and when required it selects the "next" substructure within the "current" substructure in constant time, by making use of the pointers *min* and *max*. This is illustrated by the following recursive algorithm, where x, y, and w denote subnodes of $\mathcal{S}_A$; with a minor abuse of notation, we associate each subnode with the value of its label, so that we can formally use the subnode itself in comparisons.

```
function find(x);
  begin
    w := rep(x);
    if succ(w) = nil then find y by inspection;
    { In this case the canonical substructure rooted at w has at most two leaves }
    while (max(w) ≠ nil) and (min(w) < x < max(w)) do
      begin
        x := succ(x);
        w := rep(x)
      end;
    if (max(w) = nil) or (max(w) ≤ x) then
      if w is the root of SS then y := nil
      else y := min(top(find(w)))
    else y := min(w);
    return(y)
  end;
```

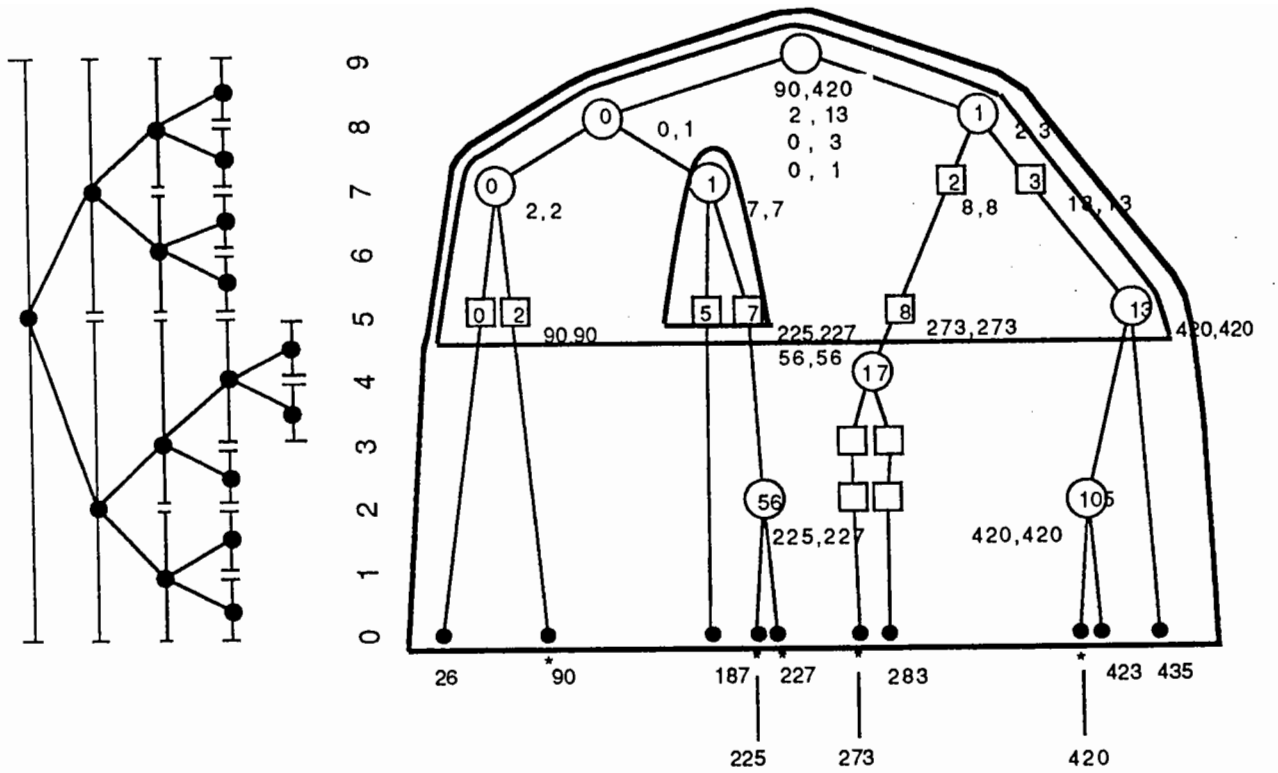


Figure 8: The sequence of nested canonical substructures traversed by *find*(187). Note that *lca*(187, 225) is the node labeled 1 at level 7.

The above algorithm deals with subnodes; therefore the function that finds the marked leaf y immediately to the right of leaf x is implemented by a call $find(top(x))$, which returns $top(y)$. The number of canonical substructures it traverses is $O(\log\log|U|)$, the depth of $\mathcal{L}$. Stepping from the current substructure to the next takes constant time, so the overall running time of the procedure is $O(\log\log|U|)$ in the worst case.

Note that whenever we have either $max = min$ or $max = min =$ **nil** for a canonical substructure T (that is, when T contains at most one node at its base belonging to paths leading to marked elements), the search does not proceed to canonical substructures of T. This has the following consequence: Let x be a leaf of T and w its root (both viewed as subnodes of actual nodes of S_A) so that $w = rep(x)$ and $min(w) = max(w) = x$; then for the subnodes $rep(succ(x))$, $rep(succ(succ(x)))$, etc., the *min* and *max* pointers are set to **nil**.

We now present pseudo-code versions of the recursive procedures *mark* and *unmark*, to be applied to the *top* subnode of the leaf being processed. As usual, x, y, and w are subnodes, as defined earlier.

```
    procedure mark(x);
      begin
1.      w := rep(x);
2.      while (min(w) ≠ max(w)) and (succ(w) ≠ nil) do
          begin  { succ(x) ≠ nil since succ(w) ≠ nil }
3.          if x < min(w) then min(w) := x;
4.          if x > max(w) then max(w) := x;
5.          x := succ(x);
6.          w := rep(x)
          end;
7.      if succ(w) = nil then mark w according to its two leaves
8.      else if min(w) = max(w) = nil then
          begin
9.          min(w) := max(w) := x;
10.         mark(w)
          end
        else
          begin
11.         mark(succ(x))
12.         if x < min(w) then
              begin
13.             if rep(succ(min(w))) ≠ rep(succ(x)) then mark(succ(min(w)));
14.             min(w) := x
              end;
15.         if x > max(w) then
              begin
16.             if rep(succ(max(w))) ≠ rep(succ(x)) then mark(succ(max(w)));
17.             max(w) := x
              end;
          end
      end;
```

To analyze the performance of *mark*, we first note that the **while**-loop in lines 2–6 is followed either by a marking at line 7 (which takes constant time and ends the procedure) or by a recursive call to *mark* at line 10 or by a recursive call at line 11 (which might be followed by another one either at lines 13 or 16). If the **while**-loop is executed t_1 times, for some $0 \leq t_1 \leq \lceil \log\log |U| \rceil$, the recursive calls are applied to canonical substructures of order $t_1 + 1$. The recursive calls at lines 10 and 11 are mutually exclusive, and the recursive call at line 11 is trivial if either line 13 or line 16 is executed. Thus, at most one nontrivial recursive call is made on a canonical substructure of order $t_1 + 1$. Since $t_1 \leq \log\log |U| + 1$, it follows by induction that *mark* runs in $O(\log\log |U|)$ time in the worst case.

```
   procedure unmark(x);
     begin
1.     w := rep(x);
2.     while (min(w) < x < max(w)) and (succ(w) ≠ nil) do
         begin  { succ(x) ≠ nil since succ(w) ≠ nil }
3.         x := succ(x);
4.         w := rep(x)
         end;
5.     if succ(w) = nil then mark w according to its leaves
6.     else if min(w) = max(w) then
         begin
7.         min(w) := max(w) := nil;
8.         unmark(w)
         end
       else
         begin
9.         unmark(succ(x));
10.        t := rep(succ(x));
11.        if min(w) = x then
12.          if min(t) ≠ nil then min(w) := min(t)
13.          else begin unmark(t); min(w) := min(min(rep(t))) end
14.        else if max(w) = x then
15.          if max(t) ≠ nil then max(w) := max(t)
16.          else begin unmark(t); max(w) := max(max(rep(t))) end
         end
     end;
```

The analysis of *unmark* is analogous to that of *mark*. Let us assume that the **while**-loop in lines 2–4 is executed $0 < t_2 \leq \lceil \log\log|U| \rceil$ times. The **while**-loop is followed either by a recursive call at line 8 or else by a recursive call at line 9, followed if necessary by one or two mutually exclusive code segments (lines 11–13 and lines 14–16). Both recursive calls at lines 8 and 9 are applied to canonical substructures of order $t_2 + 1$; the call at line 9, however, is trivial if either line 13 or line 16 is executed. Thus, at most one nontrivial recursive call is made on a canonical substructure of order $t_2 + 1$. Repeating the above argument developed for procedure *mark*, we obtain the same time bound $O(\log\log|U|)$ in the worst case. Combining the preceding analysis with Theorem 5, we get the following performance bound:

Theorem 6 *The CST $\mathcal{S}_A$ for catalog A and universe U uses space $O(|A|\log\log|U|)$ and can perform the operations insert, delete, find, mark, and unmark in time $O(\log\log|U|)$ in the worst case.*

6 Parallel View from a Viewpoint at Infinity

In the preceding sections we considered the axial view, in which the point of view was the point at infinity along one of the coordinate axes; the two-dimensional image of each 3D-

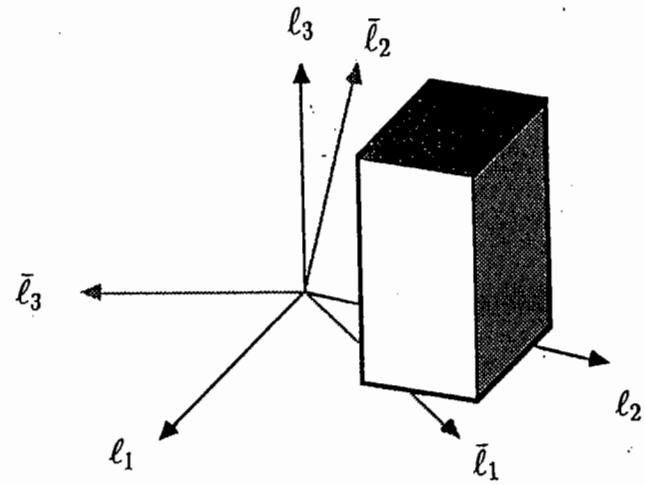


Figure 9: In the parallel view from infinity, each 3D-rectangle appears as a hexagon with pairs of parallel sides oriented along three directions ℓ_1, ℓ_2, and ℓ_3. Its image can be naturally partitioned into three parallelograms for purposes of finding interior edges of the silhouette.

rectangle was an isothetic rectangle. In this section we show how to implement our display algorithm of Section 2 for the case in which the point of view is at infinity, but no longer along one of the axes. In this case the two-dimensional image of each 3D-rectangle is a hexagon, whose opposite sides are parallel. All the hexagons share the same three orientations.

Let us denote the three orientations of the sides of each hexagon by the directions ℓ_1, ℓ_2, and ℓ_3. We denote by $\overline{\ell_i}$ the direction orthogonal to ℓ_i, for each i, as shown in Figure 9. We let S_i denote the set of edges of the current silhouette parallel to the ℓ_i direction.

We replace the segment tree data structures $\mathcal{T}_x$ and $\mathcal{T}_y$ used for the axial view by six analogous segment trees $\mathcal{T}_{1,2}$, $\mathcal{T}_{1,3}$, $\mathcal{T}_{2,1}$, $\mathcal{T}_{2,3}$ $\mathcal{T}_{3,1}$, and $\mathcal{T}_{3,2}$. For each $1 \leq i, j \leq 3$, $i \neq j$, segment tree $\mathcal{T}_{i,j}$ is used for finding intersections between a query segment in the ℓ_i direction and the stored edges in the ℓ_j direction; $\mathcal{T}_{i,j}$ stores the edges of S_j projected onto the $\overline{\ell_i}$ axis, using the $\overline{\ell_j}$ dimension for discriminating in the secondary data structure. The edges of S_j that are stored in $\mathcal{T}_{i,j}$ might not start or end at $\overline{\ell_i}$ ordinates. In such cases, the edge is "trimmed" at each end to the nearest ordinate and only the remainder of the edges is stored in $\mathcal{T}_{i,j}$; the pieces that are trimmed off cannot generate any intersections with query segments in the ℓ_i direction.

The following procedure suffices in Step 1 of the display algorithm to find the intersections of a query segment in the ℓ_1 direction with the S_2 and S_3 edges of the silhouette: First we find the intersections between the query segment and S_2 using the $\mathcal{T}_{1,2}$ segment trees and between the query segment and S_3 using the $\mathcal{T}_{1,3}$ segment trees. Then we merge the intersections together in a linear fashion. The crucial fact that makes this merging work is that the edges in the silhouette do not cross each other.

In a similar way we replace the range tree data structure $\mathcal{R}_x$ used for the axial view by three range trees $\mathcal{R}_{1,2}$, $\mathcal{R}_{1,3}$, and $\mathcal{R}_{2,3}$. For each $1 \leq i < j \leq 3$, range tree $\mathcal{R}_{i,j}$ is used for finding the vertices of the silhouette that are internal to a query parallelogram oriented in the ℓ_i and ℓ_j directions; $\mathcal{R}_{i,j}$ stores the vertices of the silhouette projected onto the $\overline{\ell_i}$ axis, using the $\overline{\ell_j}$ dimension for discriminating between vertices in the secondary data structure.

In order to find interior vertices in a query hexagon in Step 4 of the display algorithm,

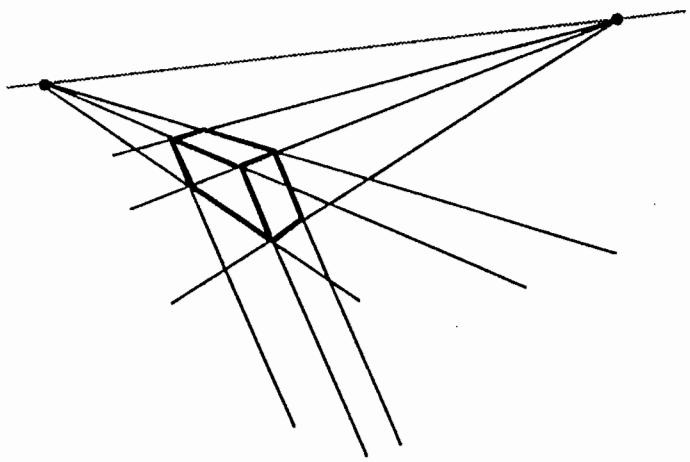


Figure 10: The perspective view of a 3D-rectangle.

we first subdivide the hexagon into three parallelograms, as shown in Figure 9 and then use each of the three range tree data structures for finding interior vertices in the respective parallelograms.

7 Perspective View

In the previous section we considered the parallel view from infinity, in which the sides of the hexagon belong to lines forming three distinct pencils, in the ℓ_1, ℓ_2, and ℓ_3 directions. The pole of each of these pencils is a point lying on the line at infinity. Assuming that not all three pencil poles are at infinity transforms the parallel view into a perspective view, as shown in Figure 10.

The technique described in the previous section requires only small changes to adapt to the more general case of the perspective view. Specifically, segment trees and range trees are now built on sets of polar angles rather than on sets of cartesian coordinates. With this single modification, the entire discussion of the previous section applies to the generation of the display of the perspective view.

8 Conclusions

We have presented a new algorithm for hidden-line elimination in the perspective view of a scene of three-dimensional isothetic parallelepipeds (3D-rectangles). The algorithm runs in $O((n+d)\log n \log\log n)$ time, where n is the number of 3D-rectangles and d is the number of edges of the display. It is scene-sensitive, in that its running time depends on the complexity of the resulting display and not on the potentially much larger set of underlying intersections.

The main component of our algorithm is a fast and relatively easy-to-program implementation of augmented segment and range trees in a semidynamic setting. The principles of fractional cascading are incorporated into the algorithm, but without the customary overhead.

An interesting research problem is to see if our techniques can be extended to significantly more general polyhedral scenes. Reif and Sen consider hidden-line elimination in monotone polyhedral terrains. [22]. Polyhedral terrains are more general in some respects, but more restrictive in others, than the scenes we consider in this paper. It would be interesting to combine our techniques with those of [22] to enlarge the class of scenes we can handle.

We made the assumption in this paper that the 3D-rectangles in the scene can be ordered in some way consistent with the dominance relation based on occlusion; in other words, we assumed that the dominance relation is acyclic. Very recently, de Berg and Overmars [2] have announced a clever algorithm for our problem that does not require acyclicity. Their algorithm makes use of static fractional cascading and runs in $O((n+d)\log n)$ time, with the customary overhead due to fractional cascading; their data structures support visibility and ray shooting queries in $O(\log n)$ time.

An interesting open problem is whether there is an $o(n^2)$ algorithm to determine if the dominance relation on a set of 3D-rectangles is acyclic. If the dominance relation is not acyclic, then we can split some 3D-rectangles until the dominance relation on the resulting set of 3D-rectangles becomes acyclic [10]. Another open problem is whether finding a minimal set of 3D-rectangles to split is NP-hard.

References

[1] M. J. Atallah & M. T. Goodrich, "Output-sensitive Hidden Surface Elimination for Rectangles," Department of Computer Science, The Johns Hopkins University, Technical Report 88–13, 1988.

[2] M. de Berg & M. H. Overmars, "Hidden Surface Removal for Axis-Parallel Polyhedra," Utrecht University, Technical Report RUU–CS–90–21, May 1990.

[3] M. Bern, "Hidden Surface Removal for Rectangles," *Journal of Computer and System Sciences* (1990), 49–69.

[4] P. van Emde Boas, R. Kaas & E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory* 10 (1977), 1977.

[5] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science* (November 1983), 122–132.

[6] B. M. Chazelle & L. J. Guibas, "Fractional Cascading I: A Data Structuring Technique," *Algorithmica* 1 (1986), 133–162.

[7] R. Cole, "Searching and Storing Similar Lists," *Journal of Algorithms* 7 (1986), 202–220.

[8] H. Edelsbrunner, L.J. Guibas & J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM Journal on Computing* 15 (1986), 317–340.

[9] J. D. Foley, A. van Dam, S. K. Feiner & J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

[10] H. Fuchs, Z. M. Kedem & B. Naylor, "On Visible Surface Generation by A Priori Structures," *Computer Graphics* 14 (1980), 124–133.

[11] M. T. Goodrich, M. J. Atallah & M. H. Overmars, "Output-Sensitive Methods for Rectilinear Hidden-Surface Removal," Department of Computer Science, The Johns Hopkins University, Technical Report JHU–90/11, 1989.

[12] L. J. Guibas & F. F. Yao, "Translating a Set of Rectangles," in *Advances in Computing Research, Volume 1: Computational Geometry*, F. P. Preparata, ed., JAI Press Inc., 1983, 61–78.

[13] R. H. Güting & T. H. Ottmann, "New Algorithms for Special Cases of the Hidden Line Elimination Problem," *Computer Vision, Graphics, and Image Processing* 40 (1987).

[14] H. Imai & T. Asano, "Dynamic Orthogonal Segment Intersection Search," *Journal of Algorithms* 8 (1987), 1–18.

[15] D. B. Johnson, "A Priority Queue in Which Initialization and Queue Operations Take $O(\log \log D)$ Time," *Mathematical Systems Theory* 15 (1982), 295–309.

[16] K. Mehlhorn & S. Näher, "Dynamic Fractional Cascading," *Algorithmica* 5 (1990), 215–242.

[17] S. Näher, K. Mehlhorn & C. Uhrig, Universität des Saarlandes, Technical Report A 02/90, 1990.

[18] W. M. Newman & R. E. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.

[19] F. P. Preparata & D. T. Lee, "Parallel Batch Planar Point Location on the CCC," *Information Processing Letters* 33 (December 1989), 175–179.

[20] F. P. Preparata & M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[21] F. P. Preparata, J. S. Vitter & M. Yvinec, "Computation of the Axial View of a Set of Isothetic Parallelepipeds," *ACM Transactions on Graphics* 9 (July 1990), 278–300.

[22] J. H. Reif & S. Sen, "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and its Parallelization," *Proceedings of the Fourth Annual ACM Symposium on Computational Geometry* (June 1988), 193–200.

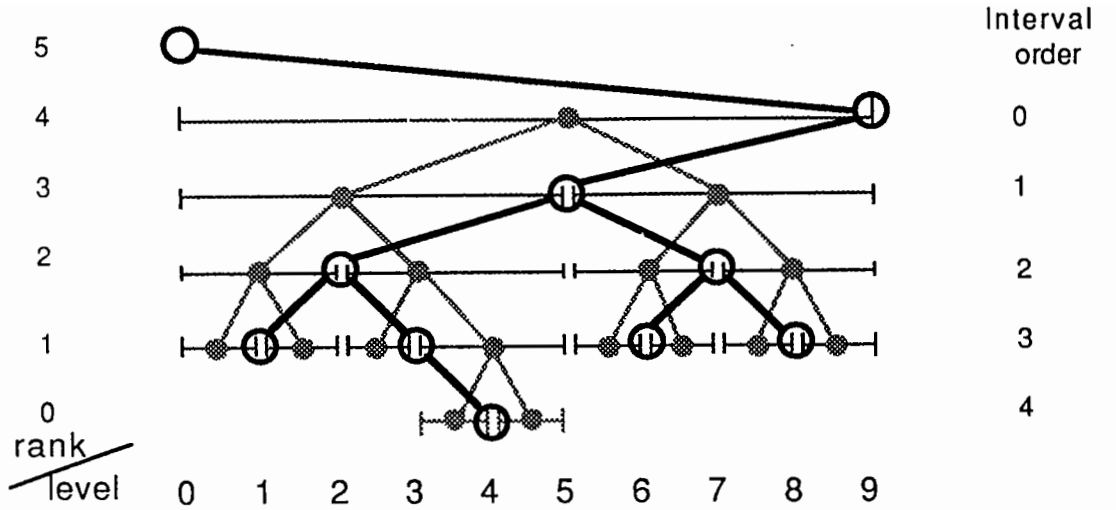


Figure 11: The definition of the rank function on the levels implicitly induces a binary tree (shown with heavy lines) on the levels themselves.

Appendix

In this Appendix we give an interesting connection between the CST $\mathcal{S}_A$ and the stratified tree (ST) of [4]. We begin by noting that the choice of the segment tree $\mathcal{L}$ automatically induces a "rank" function on the levels of $\mathcal{SS}$, analogous to the one used in [4]. Specifically, let M be the maximum order of the nodes of $\mathcal{L}$; we have $rank(0) = M + 1$, $rank(K) = M$, and $rank(i) = M - 1 - order(w)$ if i splits the range of node w in $\mathcal{L}$ into the ranges of its children (see Figure 11). This permits us to redefine the notion of "reach" of a node in terms of the canonical substructures determined by $\mathcal{L}$. Specifically, let T_1 and T_2 be the maximal canonical substructures having u as root and as a leaf level, respectively; the *reach* of u is the union of the following subtrees of $\mathcal{SS}$:

(i) $T_D(u)$, consisting of T_1 minus its leaves;

(ii) $T_U(u)$, consisting of the maximal proper subtree of T_2 containing u.

Theorem 7 *The nodes of the CST $\mathcal{S}_A$ are exactly the "active" nodes in the stratified tree (ST) of [4].*

Proof: First of all, the active leaves and the branching nodes (which are the least common ancestors of pairs of active leaves) are in both trees. So we must consider the nonbranching (buffer) nodes. In the CST, the buffer nodes are produced by the fragmentation of the CBT edges effected by the segment tree $\mathcal{L}$. In the ST, these nodes are specified by the "properness condition" [4, page 108] and are those on a path from a branching node to a leaf such that their reach contains a branching node. (In the language of [4], all nodes on a path from a branching node are called "present"; those satisfying the properness condition are called "active.") Let u be an active branching node in the ST, with reach $T_D(u) \cup T_U(u)$. Since u is active, there is at least one branching node w either in $T_U(u)$ or in $T_D(u)$. In the first case,

their least common ancestor $w' = lca(w, u)$ (not necessarily distinct from w) is a branching node on the path from u to the root of $T_U(u)$; thus there is a path of "present nodes" in the ST starting at level $j < level(u)$, containing u, and terminating at w'. This path corresponds to an edge of the CBT, and by the mechanism of the segment tree, a buffer node is placed at $level(u)$, namely, u itself. In the second case, there is a path of present nodes starting at w, passing by u, and continuing toward the root. Again, the edge of the CBT corresponding to this path is cut by the segment tree at u (a buffer node). Therefore, if u is active in the ST it also exists in the CST. The argument is clearly reversible. □