VARIATIONS ON RAY SHOOTING

by

R. Bar-Yehuda and S. Fogel

Technical Report #639

July 1990.
Variations on Ray Shooting

Reuven Bar Yehuda* and Sergio Fogel
Computer Science Department
Technion - IIT, Haifa 32000, Israel

Abstract

We solve some problems related to Ray shooting in the plane, such as finding the first object hit by a query ray or counting their number. Our main results are an algorithm for finding the first hit when the objects are segments, and an algorithm for the case when the objects are lines. Both algorithms improve over the previous algorithms in preprocessing and space, while for segments, we improve also the query time. Moreover, our algorithms are simple and therefore of practical use.

1 Introduction

One of the most important problems in computer graphics is, given a set of objects and a point of view, to represent on a screen the image seen from the given point when looking in a certain direction. One way of solving this problem is to compute for every pixel the object seen from the given point in the direction determined by the pixel. In other words, we want to be able to shoot a light ray from a given point in a given direction, and find the lighted object. This is known as the Ray Shooting problem. Since we will usually want to ray shoot from different points and in different directions, it is worth spending some time in preprocessing the objects so that we can answer queries efficiently. The number of objects is usually large, and they have to be stored for long periods of time, so we are only interested in data structures taking linear or quasi linear (i.e. up to log factors) space. The parameters for evaluating the efficiency of a data structure are the preprocessing time, the space required to store the structure and the query time. We say that a data structure has complexity \( (P(n), S(n), Q(n)) \) if the preprocessing time is \( O(P(n)) \), the space \( O(S(n)) \) and the query time \( O(Q(n)) \).

In this paper we concentrate on ray shooting in the plane, when the objects may be either straight

*This research was partially supported by the NY Metropolitan Research Fund
In this case we are interested only in the number of hits and not in the objects themselves. The query which has been extensively studied in the literature ([Wil82, EW86, HW87, CW89, MW89]).

Sometimes we are interested not only in computing the first object hit, but also the second one, and so forth. For example, if some of the objects are partially transparent, and the query ray hit one of these objects, we want to go on and find the next object hit by a query ray. Of course, we could ray shoot again from the point of the first hit, but, as we will see, we can save time by making use of the work that we spent in finding the first hit.

Another related problem is the **Counting Problem**: counting the number of objects hit by the ray. In this case we are interested only in the number of hits and not in the objects themselves. The query time will not be dependent on the number of objects hit. An interesting problem for this case is when the objects are lines and the query ray is vertical. This is the dual of the halfplane counting problem, which has been extensively studied in the literature ([Wil82, FW85, HW87, CW89, MW89]).

All the solutions that we give for all the above problems are based on **ES-trees**, a structure proposed by Matoušek and Welzl [MW89]. The use of ES-trees makes the data structures simple enough to be

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ES-trees</td>
</tr>
<tr>
<td>2</td>
<td>Ray shooting among lines</td>
</tr>
<tr>
<td>3.1</td>
<td>Vertical rays, First &amp; hits</td>
</tr>
<tr>
<td>3.2</td>
<td>Counting</td>
</tr>
<tr>
<td>3.3</td>
<td>First hit</td>
</tr>
<tr>
<td>3.4</td>
<td>First &amp; hits</td>
</tr>
<tr>
<td>4</td>
<td>Ray shooting among rays — Counting</td>
</tr>
<tr>
<td>5</td>
<td>Ray shooting among segments</td>
</tr>
<tr>
<td>5.1</td>
<td>Counting</td>
</tr>
<tr>
<td>5.2</td>
<td>First &amp; Hits, non-intersecting segments</td>
</tr>
<tr>
<td>5.3</td>
<td>First Hit</td>
</tr>
</tbody>
</table>

Table 1: Organization of the paper

<table>
<thead>
<tr>
<th>Section</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ES-trees</td>
</tr>
<tr>
<td>2</td>
<td>Ray shooting among lines</td>
</tr>
<tr>
<td>3.1</td>
<td>Vertical rays, First &amp; hits</td>
</tr>
<tr>
<td>3.2</td>
<td>Counting</td>
</tr>
<tr>
<td>3.3</td>
<td>First hit</td>
</tr>
<tr>
<td>3.4</td>
<td>First &amp; hits</td>
</tr>
<tr>
<td>4</td>
<td>Ray shooting among rays — Counting</td>
</tr>
<tr>
<td>5</td>
<td>Ray shooting among segments</td>
</tr>
<tr>
<td>5.1</td>
<td>Counting</td>
</tr>
<tr>
<td>5.2</td>
<td>First &amp; Hits, non-intersecting segments</td>
</tr>
<tr>
<td>5.3</td>
<td>First Hit</td>
</tr>
</tbody>
</table>

Table 1: Organization of the paper
Figure 1: A right sheaf with respect to $p$.

The ES-tree is a data structure that was proposed by Matoušek and Welzl [MW89]. This data structure stores a set $H$ of lines in the plane in such a way that certain queries can be answered efficiently. In this paper we look at an ES-tree as a "black box" which, given a query point $p$, outputs a partition of the lines of $H$ into $O(\sqrt{n})$ subsets complying with certain requirements. The reader interested in details of ES-trees should refer to [MW89]. An improvement of the construction of ES-trees appears in [BF90].

The main concept behind ES-trees is the sheaf: We say that a set of lines $F \subseteq H$ is a right (left) sheaf with respect to a point $p$ if no two lines of $F$ intersect to the right (left) of $p$ (see figure 1).

An ES-tree for a set of $n$ lines $H$ is a data structure that holds inside of it a set of sheaves $\mathcal{H} = \{H_1, H_2, \ldots, H_h\}$, $H_i \subseteq H$ and has the following properties:

1. Every sheaf of $\mathcal{H}$ has size $O(\sqrt{n})$.
2. Every line $l \in H$ appears in $O(\log n)$ sheaves of $\mathcal{H}$.
3. Given a query point $p$, it is possible to find a set of $O(\sqrt{n})$ indices $I \subseteq \{1, \ldots, h\}$ such that $\{H_i\}_{i \in I}$ is a partition of $\mathcal{H}$. Furthermore, $I$ can be found in $O(\sqrt{n})$ time.
In order to solve ray shooting problems, we will augment every sheaf of an ES-tree with some additional data structures. Suppose that the data structure stored at each sheaf takes $O(s(n))$ space per line. Then, by property 2, the total space required by the augmented ES-tree is $O(s(n)n \log n)$.

In [BF90] we prove that an ES-tree can be built in time $O(n^{1.5})$. If the time required for building the structure stored at each sheaf of the ES-tree is $O(p(n))$ per line, then by property 2 the total time required for preprocessing is $O(n^{1.5} + p(n)n \log n)$. A more detailed analysis of ES-trees reveals that for $p(n) = O(\sqrt{n})$ the preprocessing time is still $O(n^{1.5})$.  

3 Ray Shooting Among Lines

For all the problems of ray shooting among lines we use a similar technique. We first build an ES-tree for the input lines, and augment each sheaf with some data structures. In order to answer a query for a ray $p$ starting from a point $p$, we obtain from the ES-tree a partition of the input into $O(\sqrt{n})$ sheaves with respect to $p$. The problem is solved independently for each sheaf, and then the partial results are combined.

3.1 Vertical rays

For this problem, each sheaf is stored as a linear array ordered by the slopes of the lines. Finding the first hit or counting the number of hits in a sheaf is accomplished by a binary search. This is the use for which ES-trees were proposed in [MW89]. The complexity for counting or finding the first hit is clearly $(n^{1.5}, n \log n, \sqrt{n} \log n)$.

In order to find the first $k$ hits, we find the first hit in each of the sheaves and build a priority queue of the resulting distances. Every time we report a hit that belongs to a sheaf $H_i$, we delete the reported line from the priority queue, and insert into the priority queue the next line of $H_i$. The first phase takes $O(\sqrt{n} \log n)$ time, and the second one $O(\log n)$ time per reported point, yielding a total complexity of $(n^{1.5}, n \log n, \sqrt{n} \log n + k \log n)$.

In fact the preprocessing time is given by the recursive formula $P(n) \leq O(n)p(n) + 2P(\frac{n}{2})$, when $p(n)$ is the preprocessing time per point. For the case $p(n) = O(\sqrt{n})$, $P(n) = O(n^{1.5})$.
3.2 Counting

In order to count hits in a sheaf $H_i$, we again store $H_i$ in a linear array sorted by slope. Suppose without loss of generality that $H_i$ is a right sheaf. If the query ray $p$ is directed rightwards\(^2\) then the lines of $H_i$ intersecting $p$ form a contiguous subsequence of $H_i$ (see figure 2). In this case the number of hits can be computed by a binary search in time $O(\log n)$. If $p$ is directed leftwards, we count hits between $p'$ and $H_i$, when $p'$ is the ray opposite to $p$. Then we subtract the number found from $|H_i|$. The complexity is therefore $(n^{1.5}, n \log n, \sqrt{n} \log n)$.

This problem can be seen as the dual of a special case of the triangle counting problem, which has been solved in [MW89].

3.3 First hit

In this case we hold for each sheaf $H_i$ the description of all the unbounded faces of the arrangement of $H_i$ (see e.g. [Ed87]). Each face is stored as a linear array of its vertices, sorted by the order of their appearance along the face.

Observation 1. The total number of edges of all the unbounded faces of an arrangement of $n$ lines is $O(n)$.

\(^2\)A ray is said to be directed rightwards if it intersects a vertical line that is to the right of its starting point.
Proof: In [EOS83] it is proved that the total number of edges in the faces of an arrangement of \( n \) lines cut by any straight line is \( O(n) \). It is always possible to draw two lines that intersect all the unbounded faces of any arrangement, so the number of edges in those faces is linear. □

Observation 1 implies that the average space needed per point in each sheaf is \( O(1) \), which means that the total space remains \( O(n \log n) \).

In order to answer a query in a sheaf for a ray \( p \) starting in a point \( P \), we will first locate the unbounded face containing \( p \), and then ray shoot inside that face (see figure 3). Since the face is convex, ray shooting in it requires only \( O(\log n) \) time. Locating the unbounded face can be done in time \( O(\log n) \) by a binary search.

We still have to show how the unbounded faces can be computed efficiently. One possibility is to build the entire arrangement in time \( O(|H_i|^2) \) as in [EOS83] and then find the unbounded faces. Since we can always ask for sheaves of size \( O(\sqrt{n}) \), the preprocessing time per line is \( O(\sqrt{n}) \). From the discussion of section 2 follows that the total preprocessing time remains \( O(n^{1.5}) \).

In fact, there is no need to spend quadratic time in each sheaf. In appendix A we give an \( O(|H_i| \log |H_i|) \) algorithm for computing all the unbounded faces of the arrangement of \( H_i \).

Remark: We can use the same technique in order to solve other queries. For instance, we can find the line closest to a query point by just storing together with every unbounded face the Voronoi diagram of the lines of the face.
3.4 First $k$ hits

Let $p$ be the query ray and $p'$ its starting point. We now describe how to find the first $k$ lines hit by $p$. We start by using an ES-tree in order to get a partition of the input lines into $O(\sqrt{n})$ sheaves with respect to $p$. We store the first line of each sheaf hit by $p$ in a priority queue sorted by the order of the hits. Then we repeat the following process $k$ times: Report the line $\ell$ stored at the head of the priority queue; delete $\ell$ from the priority queue; let $F$ be the sheaf to which $\ell$ belongs; find $\ell'$, the next line in $F$ hit by $p$ and insert $\ell'$ into the priority queue.

The only problem is how to find $\ell'$. In order to do this efficiently we augment every sheaf of the ES-tree with an additional data structure. This data structure supports the following two operations: Finding the first line hit by a query ray and deleting a line. The complexity of this data structure for a sheaf $F$ is $(|F| \log^2 |F|, |F| \log |F|, \log^3 |F|)$, which implies that the complexity for finding the first $k$ hits is $(n^{1.5}, n \log^2 n, \sqrt{n} \log n + k \log^2 n)$.

The data structure for a sheaf $F$ is a balanced binary tree $T_F$, built as follows: The leaves of $T_F$ store the lines of $F$. Let $v$ be an internal node of $T_F$, and let $\text{leaves}(v)$ denote the set of the lines stored at the leaves of the subtree of $T_F$ rooted at $v$. Every node $v$ holds a data structure for ray shooting among $\text{leaves}(v)$. This structure will be the same as the one used in section 3.3.

Since every line of $F$ appears in $O(|F|)$ nodes of $T_F$, and at each node it occupies an average of $O(|F|)$ memory, the memory requirement for the sheaf is $O(|F| \log |F|)$. The preprocessing time for a sheaf is $O(|F| \log^3 |F|)$.

During the query we hold for every node $v$ of $T_F$ the following variables: $v.\text{visited}$ (figure 4) and $v.\text{firstHit}$.

Let $D$ be the set of all lines that have already been deleted (reported). The invariant during the query will be that $v.\text{visited}$ implies that $v.\text{firstHit}$ is the first line hit by $p$ among leaves$(v) - D$.

At the beginning of the query, $v.\text{visited}$ is false for all nodes except the root $r$, and $r.\text{firstHit}$ is computed form the structure held at $r$ in time $O(\log n)$.

Finding the first hit is done by just retrieving $r.\text{firstHit}$ since $r.\text{visited}$ is always kept true.

In order to delete a line $\ell$ we climb from the leave containing $\ell$ towards the root, and at each node $v$ we do the following: If $v$ has a child $u$ such that $u.\text{visited}$ is false, then set $u.\text{visited} = \text{true}$, and compute $u.\text{firstHit}$ form the structure held at $u$ in time $O(\log n)$. Set $u.\text{visited} = \text{true}$, and set $v.\text{firstHit}$ = the nearest between the first hits of its children.

The time complexity for a deletion is $O(\log^2 n)$ since the longest path in the tree has length...
4 Ray shooting among rays — Counting

In this section we deal with the counting problem for the case when the input if is in itself a set of rays. This problem will be useful in the next section, when the input is composed of a set of segments. We will separate the problem in two cases: When all the input rays and the query ray are directed in the same direction and when not.

4.1 All the rays are directed rightwards

Given a query ray \( p \) starting at a point \( P \), we treat independently the input rays that start to the right of \( p \), and the ones that start to the left of \( p \). Let us deal first with the rays starting to the left of \( p \).

Observation 2 \( p \) hits a ray \( r \) starting to the left of \( p \) only if \( p \) hits the line supporting \( r \) (see figure 5(A)).

This means that, after we have filtered out the rays starting to the right of \( p \), we can just use the data structure of section 3.2 for counting hits with lines. In order to leave only the rays that start to the left of \( p \) we use a tree \( T \) built as follows. The leaves of \( T \) are the rays of \( R \) sorted by the \( x \) coordinate of their starting point. Let \( v \) be an internal node of \( T \), and let \( \text{leaves}(v) \) be the set of rays stored at the leaves of the subtree rooted at \( v \). We associate with \( v \) a data structure for computing hits between a query ray and the lines supporting the rays of \( \text{leaves}(v) \).
In order to count how many rays starting to the left of $p$ are hit by the query ray $p$, we find a set of $O(\log n)$ nodes $v_1, v_2, \ldots, v_t$ of $T$ such that the union of the rays of $\text{leaves}(v_i)$ is the set of all the input rays that start to the left of $p$, and such that the $\text{leaves}(v_i)$'s, $1 \leq i \leq t$ are disjoint. All that rests is to compute the hits among each $\text{leaves}(v_i)$ (using the structure associated with the node $v_i$), and sum the results. The time complexity for the query in each $v_i$ is $O(\sqrt{|\text{leaves}(v_i)| \log |\text{leaves}(v_i)|})$. Since the sizes of the sets $\text{leaves}(v_i)$ are bounded by a geometric progression, the total complexity is $O(n \sqrt{\log n})$.

We still have to take care of the rays that start to the right of $p$. Let $\alpha$ be the slope of $p$.

**Observation 3** A ray $r$ starting to the right of $p$ intersects $p$ if either $r$ starts from above $p$ and the slope of $r$ is smaller than $\alpha$ or $r$ starts from below $p$ and the slope of $r$ is greater than $\alpha$ (see figure 5(B)).

From the above observation, we can count the number of rays starting to the right of $p$ and hitting $p$ as follows. We use a tree $T$ as before, but this time sorted on the slopes of the input rays. With each node $v_i$ of $T$ we associate a data structure for counting points in triangles, when the points stored are the starting points of the input rays. The structure for counting points in triangles of [MW89] has complexity $(n^{1.5}, n \log n, n \sqrt{\log n})$. Therefore the space complexity of our structure is $O(n \sqrt{\log n})$ and the preprocessing time is $O(n^{1.5})$.

In order to answer a query, we find a set of $O(\log n)$ nodes $v_1, v_2, \ldots, v_t$ of $T$ such that the union of the rays of $\text{leaves}(v_i)$ is the set of all the input rays with slopes smaller than $\alpha$. For each $v_i$ we count how many rays of $\text{leaves}(v_i)$ have their starting point above $p$ and to the right of $p$. This is
accomplished using the data structure associated with \( u_i \) in order to count how many starting points are inside the (unbounded) triangle determined by \( p \) and the vertical line passing through \( p \). In the same way we count how many rays start below \( p \) and have slope greater than \( \alpha \). The query time is then \( O(E_{iv} \sqrt{\log n}) \).

4.2 General Case

First we separate the input rays into two sets \( H_l \) and \( H_r \), according to the direction of the rays. Let \( H'_l \) and \( H'_r \) be the rays opposite to \( H_l \) and \( H_r \), respectively and let \( H_l \) and \( H_r \) be their supporting lines. We preprocess \( H_l \), \( H_r \), \( H'_l \) and \( H'_r \) as in the previous subsection. In addition, we preprocess \( H_l \) and \( H_r \) as in section 3.2.

Assume that \( p \) is directed to the right. Then counting hits with \( H_r \) is straightforward. In order to count hits with \( H_l \), we count hits with \( H_l \) and subtract from the number found the number of hits with \( H'_l \). The time complexity of the query is then \( O(\sqrt{\log n}) \). The space is clearly \( O(n \log^2 n) \) and the preprocessing time \( O(n^{1.5}) \).

5 Ray shooting among segments

5.1 Counting

Every segment can be represented as the subtraction of two rightwards directed rays. Let \( H_L \) be the set of rays determined by the left endpoints of the segments and \( H_R \) the rays determined by the right endpoints. All we have to do is to compute, using the data structure of section 4.1, the number of hits between the query ray \( p \) and \( H_L \), and subtract from the number found the number of hits between \( p \) and \( H_R \). The complexity will be \( (n^{1.5}, n \log^2 n, \sqrt{n} \log n) \).

5.2 First \( k \) hits among non intersecting segments

Let us first briefly discuss duality. A thorough revision of duality can be found in [Eds87]. The dual transform maps points into lines and lines into points, and preserves the basic relations between them. For instance, if a point \( p \) is above a line \( \ell \), then the dual line \( \hat{\ell} \) is above the dual point \( \hat{p} \).

The dual of an ES-tree for a set of points \( P \) is a data structure which is fed with a line \( \ell \), and gives as a partition of \( P \) into \( O(\sqrt{n}) \) subsets, which are dual sheaves. Let \( \hat{P} \) be a dual sheaf. If we sort the points of \( \hat{P} \) on increasing \( x \) coordinate, and connect the points in the given order by a polygonal path
In order to ray shoot among segments, we will build a dual ES-tree for the left endpoints of the segments. When given a ray $\rho$, we will feed the ES-tree with $\beta$, the line supporting $\rho$, and ask it for $O(\sqrt{n})$ dual sheaves with respect to $\beta$. Then we will ray shoot in each dual sheaf, and combine the results.

We will now show how to ray shoot in a set of segments $F$ whose left endpoints form a dual sheaf. Our aim is to add to $F$ some extra "dashed" segments and convert it into a simple polygon of zero width. The idea is to apply Chazelle and Guibas' algorithm [CG85] in order to ray shoot in the resulting simple polygon. We recall that the data structure of [CG85] answers queries in time $O(\log n)$, occupies $O(n)$ space and is built in $O(n \log n)$ time. The problem that arises is that the first hit may be one of the dashed segments. The dashed segments that we add will be part of the polygonal path determined by the left endpoints of $F$. Since those points are a dual sheaf with respect to $\beta$ (the line supporting the query ray $\rho$), $\rho$ intersects at most one dashed line. If the first segment hit by $\rho$ is dashed, we will ray shoot again from the hit point in the same direction as before. This time we can be sure that $\rho$ will not hit a dashed segment.

As we mentioned, the dashed segments will be part of the polygonal path determined by the left endpoints of $F$ (figure 6). We will not add the entire polygonal path, since this could lead to a non-simple (and too large) polygon. Instead we will use just the minimum needed to connect every segment to the ones starting to its left (figure 6(B)). Then we will duplicate each segment in order to build the desired polygon (figure 6(C)).

Since we add at most one dashed segment per original segment, the size of the polygon is linear on the size of the sheaf. The structure of [CG85] takes linear space, and therefore the total space required by the structure is $O(n \log n)$. The query time is $O(\log n)$ per sheaf, giving a total of $O(\sqrt{n} \log n)$.

All that rests is to show how to compute the left endpoints of the dashed lines. As in section 3.3, a simple quadratic time algorithm is sufficient, since the total preprocessing is dominated by the $O(n^2)$ of building the ES-tree. A more efficient algorithm can be obtained by sweeping the segments from right to left, in a very similar fashion to the one used for computing all intersections between a set of segments. We leave to the reader the proof that in this way the dashed segments can be computed in time $O(n \log n)$.

In order to find the first $k$ hits, all we need is a priority queue, since we can find the next hit in a sheaf by just ray shooting again in the sheaf. The query time will be $O(\sqrt{n} \log n + k \log n)$.

Remark: We may use spanning paths with low stabbing number [CW89] instead of sheaves in order to connect all the segments into a polygon. In this case, the query ray may hit $O(\sqrt{n})$ dashed
Figure 6: (A) The polygonal path determined by the left endpoints. (B) Adding dashed lines in order to connect the segments. (C) The resulting polygon.
segments before hitting a real one. The problem with paths with low stabbing number is the high complexity needed for building them. If we use paths with low stabbing number, the dashed lines have to be found using the second algorithm shown, since otherwise the preprocessing time becomes excessively high.

5.3 First Hit (Intersecting Segments)

In order to ray shoot among (possibly) intersecting segments we will use the data structure of Agarwal [Aga89]. The data structure of [Aga89] is a tree which holds at each node some data structures: one for planar point location, one for ray shooting among non intersecting segments and one for ray shooting among lines. These data structures are used as “black boxes”. If we change the black boxes for ray shooting among lines and among non intersecting segments by the ones described in sections 3.3 and 5.1, we improve the complexity from \( (n^{1.5} \log^{4.32} n, n \alpha(n) \log^4 n, \sqrt{n \alpha(n)} \log^2 n) \) to \( ((n \alpha(n))^{1.5}, n \alpha(n) \log^2 n, \sqrt{n \alpha(n)} \log n) \).

Appendix A: Computing the unbounded faces of an arrangement in time \( O(n \log n) \)

Let \( H \) be a set of \( n \) lines. We want to compute efficiently the description of all the unbounded faces of \( A(H) \). The face to which a point \( p \) belongs is determined by the set of lines that can be reached by moving \( p \) without touching any other line. Let us consider the dual scenario. We have a set of points \( \tilde{H} \) dual to the lines of \( H \), and a line \( \tilde{p} \) dual to the point \( p \). The dual of the face to which \( p \) belongs is determined by the set of points that can be reached by translating and rotating \( \tilde{p} \) without touching any other point, and without making \( \tilde{p} \) vertical. (Crossing the vertical would mean wrapping around infinity and getting back from the other side). Note that, if \( p \) is inside an unbounded face, it is possible to reach a vertical line without touching any point. Every face unbounded to the right (left) can therefore be defined by translating and rotating clockwise (counterclockwise) the corresponding vertical line. This idea is similar to that of [EGH*89].

Let \( \ell_1, \ldots, \ell_r \) be a set of vertical lines so that \( \ell_i \) has \( i \) points of \( \tilde{H} \) to its left. Let \( \tilde{h}_1, \ldots, \tilde{h}_n \) be the points of \( \tilde{H} \) ordered by their \( x \) coordinate. Let \( \tilde{C}_i \) be the convex hull of the points \( \tilde{h}_1, \ldots, \tilde{h}_i \) and \( \tilde{C}_i \) the convex hull of \( \tilde{h}_i, \ldots, \tilde{h}_n \), and let \( \tilde{t}_i \) and \( \tilde{t}_i' \) be the inner common tangents of \( \tilde{C}_i \) and \( \tilde{C}_{i+1} \). The \( i \)-th face unbounded to the left (right) of \( A(H) \) for \( 1 \leq i < n \) is determined by the portion of \( \tilde{C}_i \) between \( \tilde{t}_i (\tilde{t'}_i) \) and \( f_i \) and the portion of \( \tilde{C}_{i+1} \) between \( f_{i+1} \) and \( \tilde{t}_i (\tilde{t'}_i) \). The first and last faces are simply the upper and lower hulls of \( \tilde{H} \).
In order to compute those sets of points, we will use the beneath-beyond (or incremental) algorithm for computing convex hulls [PS85], and run it step by step. We recall that the beneath-beyond algorithm starts with an empty convex hull, and increments it by incorporating the points sorted by their x coordinate. At each step a point is incorporated to the hull, as possibly some other points are deleted. Since each point is incorporated once and deleted at most once, the number of insertions and deletions made by the algorithm is linear.

We are interested not only in the final result of the algorithm but also in all the intermediate results. Moreover, we need them organized in such a way that search operations can be efficiently accomplished. In order to do this we will store all the intermediate results in a persistent data structure ([ST86]). For instance, we could use a persistent red-black tree ([Tar83]). At every step of the beneath-beyond algorithm we will incorporate all changes to the convex hull to the red-black tree. This will allow us later to ask questions about any $C_i$ without having to store all of them explicitly. We then have the following algorithm:

**Finding the unbounded faces of an arrangement**

**Input:** A sequence $H = h_1, h_2, \ldots, h_n$ of lines in the plane sorted by their slope.

**Output:** The description of all the unbounded faces of $A(H)$.

1. Find $\hat{H} = \hat{h}_1, \ldots, \hat{h}_n$, the dual of the lines of $H$ (They are sorted by increasing $x$ coord.)
2. Initialize a persistent red-black tree $T$ as empty.
3. Apply the incremental convex hull algorithm of [PS85] from left to right. At each stage of the algorithm update $T$ so that it contains the description of $CH(h_1, \ldots, h_i)$.
4. For every $i$ from $n - 1$ downto 1 do:
   a) Compute $C_{i+1}$ from $C_{i+2}$ as in the incremental algorithm.
   b) Compute in $\log n$ time the inner common tangents $t_i$ and $t_i'$ between $C_i$ and $C_{i+1}$
   c) Report all the points between $t_i$ and $h_i$ and those between $h_{i+1}$ and $t_i$.
   d) Report all the points between $t_i'$ and $h_i$ and those between $h_{i+1}$ and $t_i'$.
5. Report the upper and lower convex hulls of $\hat{H}$.

The algorithm has complexity $O(n \log n)$ and requires linear space.

**References**


