Storing line segments in partition trees

Mark H. Overmars, Haijo Schipper and Micha Sharir

RUU-CS-89-17
August 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Storing line segments in partition trees

Mark H. Overmars, Haijo Schipper and Micha Sharir

Technical Report RUU-CS-89-17
August 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

Storing line segments in partition trees

Mark H. Overmars* Haijo Schipper* Micha Sharir†

July 1989

Abstract

We design two variants of planar partition trees, called *segment partition trees* and *interval partition trees*, that can be used for storing arbitrarily oriented line segments in the plane in an efficient way. The raw structures use $O(n \log n)$ and $O(n)$ storage, respectively, and their construction time is $O(n \log n)$. In our applications we augment these structures by certain (simple) auxiliary structures, which may increase the storage and preprocessing time by a polylogarithmic factor. It is shown how to use these structures for solving line segment intersection queries, triangle stabbing queries and ray shooting queries in reasonably efficient ways. If we use the conjugation tree of [10] as the underlying partition tree, the query time for all problems is $O(n^\gamma \log n)$, where $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$. The techniques are fairly simple and easy to understand.

1 Introduction

Partition trees have originally been designed for storing points and answering half-planar range queries in which, given a set S of n points in the plane, we wish to count or to report efficiently all points lying in a query half–plane (or in a query triangle). Partition trees are based on a method of recursively partitioning the plane into convex subregions such that (1) any given line intersects only a small number of the regions, and (2) the partitioning induces a partitioning of the given set of points into subsets of progressively decreasing size. In this way it is possible to

*Department of Computer Science, University of Utrecht, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. Research of the first author was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (Project ALCOM).

†Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, N.Y. 10012, U.S.A., and School of Mathematical Sciences, Tel Aviv University, 69978 Tel Aviv, Israel. Work by this author has been supported in part by Office of Naval Research Grant N00014-87-K-0129, by National Science Foundation Grants DCR-83-20085 and CCR-89-01484, and by grants from the Digital Equipment Corporation, the IBM Corporation, the U.S.-Israeli Binational Science Foundation, the NCRD - the Israeli National Council for Research and Development, and the Fund for Basic Research in Electronics, Computers and Communication, administered by the Israeli Academy of Sciences.

identify large subsets of S that lie on one side of the given line and, hence, lie either completely inside or outside the query half-plane.

The first type of partition tree was introduced by Willard [25]. His structure has a query time of $O(n^{\log_8 4}) = O(n^{.774})$ and a preprocessing time of $O(n^2)$. These results were improved by Edelsbrunner and Welzl [10] who defined a so-called *conjugation tree*. This tree has a query time of $O(n^\gamma)$, where $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$. In section 2 we give a brief description of conjugation trees. Since then the results have been improved by Haussler and Welzl [15] but their methods are not deterministic and use randomization. This has recently been remedied by Matoušek [16] and Agarwal [1]. Welzl [23] (see also Chazelle and Welzl [6]) has devised a different kind of partition tree, not related to a matching convex decomposition of the plane; this structure achieves query time bounds that almost attain the theoretically optimal $O(\sqrt{n})$ bound (if only linear storage is allowed; see Chazelle [3] for a matching lower bound). However, the preprocessing time of these methods is still relatively high. See also [1, 2, 9, 17] for a variety of recent related results.

Although partition trees were designed for storing sets of points, it has been realized that they can also be used for solving problems concerning sets of line segments or other geometric objects (see Dobkin and Edelsbrunner [7], Guibas, Overmars and Sharir [12, 13] and Agarwal [2]). In this paper we present two new data structures based on partition trees that can be used for solving a number of query problems concerning line segments in an easy way. Both consist of a partition tree on the set of endpoints of the line segments with segments stored at particular nodes in the tree. In our descriptions we will base our new structures on the conjugation tree although the trees of Haussler and Welzl [15] could be used as well. As a matter of fact, our techniques should apply to any underlying partition tree that is induced by an appropriate convex decomposition of the plane. The query time should become $O(C(n)\log n)$, where $C(n)$ is the maximum number of partition tree nodes whose corresponding regions in the plane decomposition are crossed by a line. For example, using the technique of [15], we obtain query time $O(n^{2/3+\epsilon}\log n) = O(n^{2/3+\epsilon})$, for any $\epsilon > 0$. (The optimal structure by Welzl [23] is not suited for our purpose because it is not based on such a decomposition as the others are.)

The first structure we present is a *segment partition tree*, which bears some resemblance to the normal segment tree (see e.g. [20]). It uses $O(n\log n)$ storage because every segment is stored at $O(\log n)$ nodes of the tree. We use this structure for line segment intersection queries and for triangle stabbing queries. In the first type of query, we wish to preprocess a collection of line segments so that, given a query line or segment, we can efficiently report (or count) all the given segments it intersects; in the second type of query, we wish to preprocess a collection of triangles so that, given a query point, we can efficiently report (or count) all the given triangles containing it. The second structure that we present is the *interval partition tree*, which is a kind of two-dimensional variant of the normal interval tree (see e.g. [20]). This structure is used to answer ray shooting queries, where we are given a set of line segments and wish to preprocess it so that, given a query ray,

we can efficiently determine the first intersection of the ray with the given segments (if one exists at all). We use an interval partition tree if the given segments are non–intersecting; if they do intersect, we design a more sophisticated technique in which we again use the segment partition tree. We believe that both structures will be useful for solving other problems concerning line segments as well.

The results obtained in this paper are not the best possible. They all achieve query times of $O(n^\gamma \log n)$ with a preprocessing time of $O(n \log^i n)$ where i is 1, 2 or 3. Most of these results can also be achieved in an equally efficient way by using the techniques of Dobkin and Edelsbrunner [7] or of Guibas et al. [12, 13], but their methods are complicated. Moreover, some of the results have already been improved by Agarwal [2] but the preprocessing time of his methods is much higher and, again, the techniques are complicated. The power of the (majority of the) methods presented here is their relative simplicity, which makes them easy to understand and, probably, also easier to implement.

The paper is organized as follows:

In Section 2 we give a brief description of the conjugation tree that will be used as an underlying data structure for our techniques and review some important properties of this structure.

In Section 3 we describe the segment partition tree and show that it requires $O(n \log n)$ storage and construction time.

In Section 4 we concentrate on intersection queries. As a first instance we consider the problem of storing a set V of non-intersecting line segments such that for a given query line or query line segment s the segments in V that intersect s can be reported or counted efficiently. Next we look at the case in which the line segments in V can intersect.

In Section 5 we apply the results in Section 4 to the triangle stabbing problem: Given a set of triangles in the plane, store them such that for a given query point p we can efficiently determine the (number of) triangles that contain p.

In Section 6 we describe the interval partition tree, which uses only linear storage, and prove some of its properties.

In Section 7 we use the interval partition tree to solve the ray shooting problem in a set of non-intersecting line segments: given a point p and a direction d, determine the first line segment we hit when shooting from p in direction d. We also consider the case where the given segments can intersect, and develop a more sophisticated procedure for efficient ray shooting in such collections.

Finally, in Section 8, we give some conclusions and directions for further research.

2 Preliminaries

In this section we give a brief introduction to conjugation trees. For more details see Edelsbrunner and Welzl [10].

A conjugation tree is a binary tree that stores a set of points V in the plane. For a directed line l we denote by $Left(l)$ the set of points to the left of l, by $Right(l)$

the set of points to the right of l, and by $On(l)$ the set of point lying on the line l.

Definition 2.1 *Let V be a set of n points in the plane. A directed line l is called a* bisector *of V if $|Left(l)| \leq \lceil n/2 \rceil$ and $|Right(l)| \leq \lceil n/2 \rceil$. Another line l' is called a* conjugate *of l if l' is a bisector of both $Left(l)$ and $Right(l)$.*

Definition 2.2 *Let V be a set of n points and let l be a bisector of V. Let l' be a conjugate of l. If V is not empty, a* conjugation tree *for V (and l) is a binary tree with a root δ storing $l_\delta = l$ and $On(l)$ in some form, whose left subtree is a conjugation tree of $Left(l)$ and l', and whose right subtree is a conjugation tree of $Right(l)$ and l'. If V is empty, the conjugation tree for V is also empty.*

Because both $Left(l)$ and $Right(l)$ have size at most $\lceil n/2 \rceil$ the depth of a conjugation tree is bounded by $O(\log n)$. Each point is stored in exactly one $On(l)$ structure for some node. The form of these structures depends on the type of problem one wants to solve. In our applications the structure will be completely unneccessary and, hence, is removed from the tree. The tree clearly has $O(n)$ nodes. In [10] it is shown that a conjugation tree can be constructed in time $O(n \log n)$.

Theorem 2.1 *A conjugation tree for a set of n points uses $O(n)$ storage and can be constructed in time $O(n \log n)$.*

One can view a conjugation tree as a way of partitioning the plane. The root of the tree corresponds to the complete plane. Each son corresponds to the part on one side of the bisector l, etc. In this way it is easy to see that each node δ corresponds to a convex cell c_δ that has at most $O(\log n)$ edges. The main property of a partition tree is that any line intersects the cells of only a limited number of nodes in the conjugation tree. This is based on the fact that for each node whose cell is intersected by the line at least one of its four grandsons is not intersected by the line. For details and proof of the following theorem see [10].

Theorem 2.2 *Let P be a conjugation tree containing n points and let l be any line. There are at most $O(n^\gamma)$ nodes δ in P where l intersects c_δ, with $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$.*

In the sequel, when talking about partition trees we assume the structure is actually a conjugation tree. (As discussed in the introduction, the reason for talking about partition trees in general is that the methods apply also to other types of partition trees.)

3 Segment Partition Trees

To store line segments in a partition tree we will add an associated data structure for each node in the partition tree that stores some number of segments. The type of structure will depend on the type of problem we want to solve using the structure.

The first way of doing this is what we call a *segment partition tree* because it bears close resemblance to normal segment trees. As a first step we construct a partition tree P containing the endpoints of all the line segments. Next we search with each line segment s in the partition tree. At each node δ of P we have the following possibilities:

1. s completely intersects c_δ (the cell of node δ), i.e., no endpoint of s lies in the interior of c_δ. In this case we store the part of s inside the cell in a structure T_δ associated with δ.

2. s lies on l_δ (the dividing line of the cell). In this case we store s in an ordinary segment tree S_δ. As S_δ will only contain the segments lying on l_δ, we can indeed represent it by an ordinary 1-dimentional segment tree.

3. s lies completely to the left of l_δ (with possibly one endpoint on l_δ) and does not intersect c_δ completely. In this case we continue the search at the left son of δ. Similarly, when s lies completely to the right of l_δ we continue searching at the right son of δ.

4. s properly intersects l_δ and has at least one endpoint in the interior of c_δ. In this case we continue the search at both sons of δ.

Hence, for each node δ we get two structures: S_δ is a normal segment tree of all segments that lie on l_δ, and T_δ is a structure containing all segments that completely cut through c_δ but did not completely cut through $c_{father(\delta)}$. Note that all segments stored in T_δ intersect the dividing line $l_{father(\delta)}$.

There is a slight problem here. How do we determine in case 1 whether a segment completely intersects c_δ and how do we determine the part inside c_δ? This can be solved in the following way. Rather than continuing the search recursively with the whole segment we always continue with only the part inside the current cell. Hence, in case 4 we pass to each son only the part of the segment lying on the appropriate side of l_δ. Moreover, we mark endpoints of the segments when they lie on a dividing line. This can easily be done as part of the appropriate actions in cases 3 and 4. Now case 1 simply checks whether both endpoints are marked, and the part inside the cell is the part of the segment that is left over.

Assuming that an associated T-structure of n points can be constructed in time bounded by $B_T(n)$ and that it uses at most $M_T(n)$ storage, we get the following theorem (assuming B_T and M_T satisfy the functional equation $F(a)+F(b) \leq F(a+b)$):

Theorem 3.1 *A segment partition tree on n line segments uses $O(M_T(n)\log n)$ storage and can be constructed in time $O(B_T(n)\log n)$.*

Proof. The partition tree itself uses $O(n)$ storage and can be constructed in time $O(n \log n)$. When a segment is stored in a segment tree S_δ it must have at least one endpoint on l_δ. It immediately follows that each segment is stored in at

most two segment trees S_δ. Hence, in total, the segment trees will use $O(n \log n)$ storage and can be constructed in time $O(n \log n)$.

When a segment s is stored in a structure T_δ it (or rather its full original version) must have an endpoint in $c_{father(\delta)}$ (unless δ is the root). Hence, on each level of the segment partition tree there can be at most 2 nodes where s is stored in T_δ. As a result all T-structures on one level of the partition tree use a total of $O(M_T(n))$ storage and can be constructed in time $O(B_T(n))$. As the depth of the partition tree is bounded by $O(\log n)$, the bounds follow. □

In fact, as we shall see below, the preprocessing time can sometimes be reduced by maintaining structural information about the set of line segments when adding them to the tree.

Before closing this section, we observe the following useful property of segment partition trees:

Lemma 3.2 *Assume that no two of the given segments have an endpoint in common. Then the number of segments stored in a node δ at level i of the partition tree is $O(n/2^i)$.*

Proof. Let s be a segment that is stored in δ. Then s must have an endpoint inside $c_{father(\delta)}$. The number of such points is $O(n/2^i)$ and the above assumption completes the proof. □

If k segments share an endpoint, we can modify the construction by treating this point as k distinct points that happen to coincide. With some care, it is possible to modify the construction of the conjugation tree so that it handles such cases correctly.

4 Intersection Queries

In this section we will show how a number of variants of the line segment intersection query problem can be solved using segment partition trees. We start with a restricted version of the problem: Let V be a set of n non-intersecting line segments in the plane; we want to store them such that for a given query line l we can efficiently report the line segments intersected by l.

To solve this problem we store the set V of line segments in a segment partition tree. For each node δ we store in T_δ the line segments ordered by intersection with $l_{father(\delta)}$ (we use any convenient balanced tree structure to represent T_δ). Now assume we want to perform a query with line l. First we determine all nodes δ for which l intersects c_δ. For each such node δ we determine the intersection of l with $l_{father(\delta)}$. If l does intersect $l_{father(\delta)}$ inside the cell we search with the intersection in T_δ. Assume it lies between the segments s_i and s_{i+1}. We check whether l intersects s_i or s_{i+1} (inside the cell). It clearly cannot intersect both because the line segments stored in T_δ do not intersect each other and extend to the cell boundary, and the cell is convex. If l intersects none of them we are finished at this node. Otherwise,

assume l intersects s_i. We report s_i and check whether l intersects s_{i-1}, s_{i-2}, etc. until we find a segment that is not intersected by l. It is easy to see that in this way all segments in T_δ that intersect l (inside the cell) are correctly reported.

If l does not intersect $l_{father(\delta)}$ inside the cell, let l' be the part of l that lies inside c_δ. (l' can be maintained, at constant cost per cell, during the search with l in the partition tree.) We search with l' in T_δ in the following way. Compare l' with the segment s_r stored in the root of T_δ. If l' intersects s_r report the intersection and continue in both sons. Otherwise, if l' lies left of s_r continue in the left son, otherwise continue in the right son. (Because l' lies inside the cell, s_r cuts through the cell, and the cell is convex, it is easy to decide on which side l' lies: simply check l' with respect to the line l_{s_r} that is the extension of s_r.) Since the segments in T_δ do not intersect, this correctly reports all segments intersected by l inside the cell. The search obviously takes time $O(\log n)$ plus the number of intersected segments found.

It remains to treat the segments stored in S_δ. If l intersects l_δ we perform a simple "stabbing" query on S_δ to report all segments in S_δ that contain the intersection point between l and l_δ (see e.g. [20]). If l overlaps l_δ, the entire S_δ is reported.

Theorem 4.1 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the k segments intersecting a given query line l can be reported in time $O(k + n^\gamma \log n)$.*

Proof. The storage and a weaker $O(n \log^2 n)$ bound on the preprocessing time immediately follow from Theorem 3.1, noting that $M_T(n) = O(n)$ and $B_T(n) = O(n \log n)$. The query time follows from the fact that l intersects $O(n^\gamma)$ cells. In each cell we have to perform a query on the T-structure and a query on the S-structure. Both queries take time $O(\log n)$ plus the number of intersected segments found.

We next show that the preprocessing time can in fact be improved to $O(n \log n)$. What we need is to obtain the line segments to be stored at a node δ in an ordered way. Note that the ordering along $l_{father(\delta)}$ is the same as (or rather the reverse of) the ordering along the rest of the boundary of c_δ. Note also that, at the moment we reach node δ all these line segments must be intersecting the rest of the boundary of c_δ. So we can use the following technique. We insert all line segments simultaneously into the partition tree P. When we reach a node δ we have a set V^δ of line segments that (partially) lie inside c_δ. This set consists of two parts: the subset V_1^δ of line segments that completely lie inside the cell and the subset V_2^δ of line segments that already intersect the boundary. We keep V_2^δ ordered by intersection with the boundary of the cell (more precisely, we maintain a circular list of all endpoints of segments in V_2^δ that lie on the boundary of c_δ, sorted in circular order along that boundary). Now we proceed as follows. We determine the line segments in V_2^δ that now completely intersect the cell. We remove them from V_2^δ and build T_δ out of them. Since they are already (circularly) ordered, this takes linear time. We also determine the segments that have to go in the S_δ structure. The remaining segments

are passed to the sons of δ as follows. Find those segments in V_1^δ that intersect l_δ, and sort them in their order along that line. Determine which segments of V_2^δ should be passed to which son, and extract the two corresponding sublists out of V_2^δ. Those segments that are passed to both sons also intersect l_δ, but their intersection order along that line can be determined in linear time from their order along the boundary of c_δ. Finally, we merge the new list of segments of V_1^δ that intersect l_δ with each of the sublists into which V_2^δ has been decomposed; the remaining segments of V_1^δ are partitioned among the V_1 lists of the two sons of δ. This reduces the preprocessing time to $O(n \log n)$ because any segment is only involved once in sorting (when it intersects a dividing line for the first time). □

Now assume our query object is a line segment s rather than a line. We can use exactly the same structure. To perform a query we determine all cells that are intersected by s. For each corresponding node δ we do the following. We restrict s to the cell c_δ (this can be done, as above, during the searching with s, at constant cost per cell). We search with this restricted line segment s' in T_δ as follows. We compare s' with the segment s_r in the root of T_δ. If s' lies left of s_r we continue in the left son. If s' lies to the right of s_r we continue in the right son. Otherwise we report the intersection and continue in both sons. (Note that the segments reported by this procedure necessarily form a contiguous portion of T_δ.) This clearly takes time $O(\log n)$ plus the number of segments reported. We also check whether s intersects l_δ and, if so, perform a stabbing query in S_δ with the intersection point. Similarly, if s partially overlaps l_δ, it is also easy to report all segments in S_δ that partially overlap s. The following result is now immediate:

Theorem 4.2 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the k segments intersecting a given query line segment s can be determined in time $O(k + n^\gamma \log n)$.*

The result can easily be adapted to the situation in which we only want to count the number of segments intersected by s. Both the T_δ and the S_δ structures can also be used to count the number of intersections rather than report the intersections themselves. This leads to the following result:

Theorem 4.3 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the number of segments intersecting a query line l or a query line segment s can be determined in time $O(n^\gamma \log n)$.*

Now let us consider the case in which our set V consists of possibly intersecting line segments and our query object is again a line l. Again we construct a segment partition tree on the set of endpoints of the line segments in V. For a node δ, each of the segments to be stored in T_δ intersects the boundary of c_δ twice—once on $l_{father(\delta)}$ and once on the rest of the boundary. We sort the intersections on $l_{father(\delta)}$ and also sort the intersections on the rest of the boundary. Thus we can represent each line

segment s as a pair (i_s, i'_s) where i_s is the index of its intersection along $l_{father(\delta)}$ and i'_s is the index of its intersection along the rest of the boundary. Both sets of intersections are stored in appropriate balanced search trees. In addition, we store the double indices (i_s, i'_s) in an appropriate structure Q_δ for performing orthogonal range queries; see below for details of this structure.

To perform a query with line l we first determine all the cells intersected by l. For each corresponding node δ we proceed as follows. First we check whether l intersects l_δ and, if so, perform a stabbing query on the segment tree S_δ with the intersection point (if l overlaps l_δ, the entire S_δ is reported). Next we determine the two intersections of l with the boundary of the cell c_δ (this can be done, as above, during the search with l at constant cost per cell). There are two possible cases: either one intersection lies on $l_{father(\delta)}$ or none lies on $l_{father(\delta)}$. If no intersection lies on $l_{father(\delta)}$ we search in the structure storing the intersection points of the segments in T_δ with the portion of the boundary of c_δ outside $l_{father(\delta)}$, to determine the segments having an endpoint that lies between the two intersections of l with the boundary. These are precisely the segments in T_δ that intersect l (see figure 1). This obviously takes time $O(\log n)$ plus the number of segments found. Otherwise, if l does intersect $l_{father(\delta)}$ we determine the location of the intersections of l with $l_{father(\delta)}$ and with the rest of the boundary among the intersections of the segments of T_δ with the boundary. Assume that on $l_{father(\delta)}$ the intersection lies between i_s and i_s+1 and on the rest of the boundary between $i'_{s'}$ and $i'_{s'}+1$. (See figure 2.) Now it is easy to see that the segments t intersecting l correspond to the points (i_t, i'_t) lying in the range $[-\infty : i_s] \times [i'_{s'}+1 : \infty]$ or in the range $[i_s+1 : \infty] \times [-\infty : i'_{s'}])$. Hence, we simply have to perform two orthogonal range queries on Q_δ. Since both range queries are half-infinite, we can represent Q_δ by two priority search trees (see McCreight [18]). This allows us to perform an orthogonal range query in time $O(\log n)$ plus the number of segments found; the storage for Q_δ is $O(n)$ and the preprocessing is $O(n \log n)$. This leads to the following result:

Theorem 4.4 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the k segments intersecting a given query line l can be determined in time $O(k + n^\gamma \log n)$.*

To solve the counting version of the problem, in which we want to count the number of segments intersecting l, we can no longer use priority search trees. Instead we can use the 2-dimensional range counting structure developed by Chazelle [4]. This structure has a query time of $O(\log n)$, and requires $O(n)$ storage and $O(n \log n)$ preprocessing time. Otherwise, processing a query is done exactly as above, leading to the following result:

Theorem 4.5 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the number of segments intersecting a given query line l can be determined in time $O(n^\gamma \log n)$.*

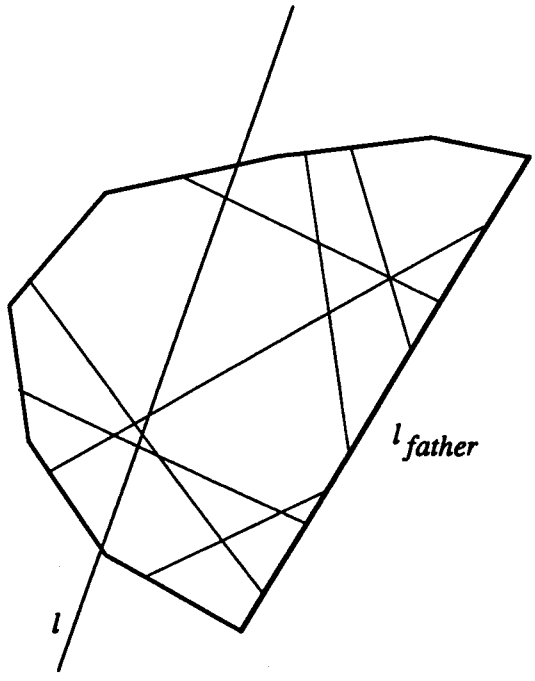

Figure 1: l does not intersect $l_{father(\delta)}$.

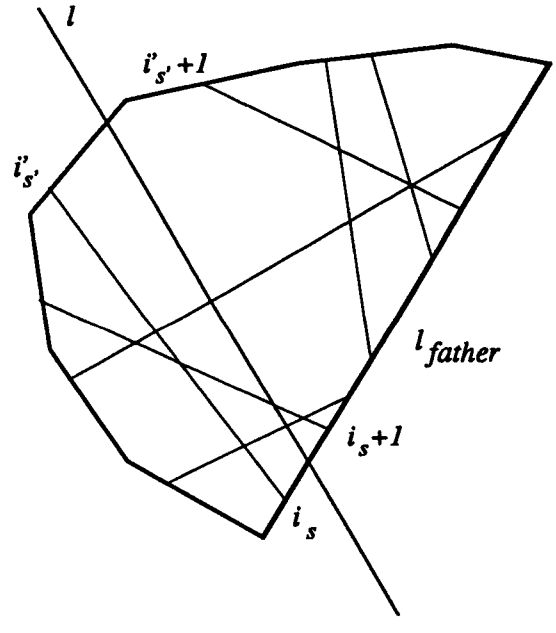

Figure 2: l does intersect $l_{father(\delta)}$.

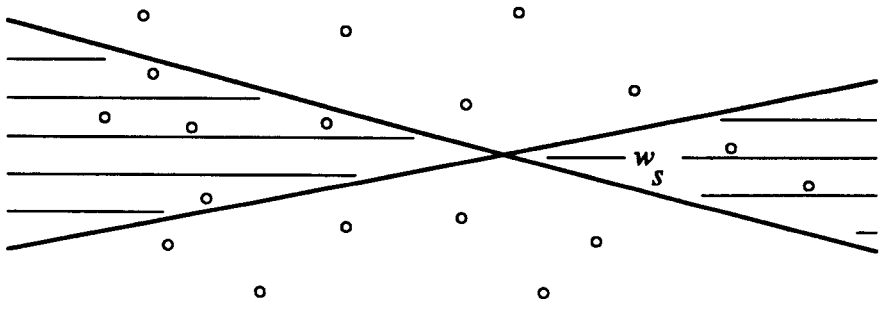

Figure 3: The dual form of the segment intersection query for a collection of intersecting segments.

Finally, we consider the case where the query object is a segment and the given segments can intersect. Let s be the query segment and let l denote the line containing s. We can essentially apply the procedures given above to the line l, but in addition keep track of the location of the endpoints of s along l. Thus, if l crosses a cell c_δ of the partition tree and no endpoint of s lies in c_δ, then either s completely cuts through c_δ, in which case the above procedures can be carried out unchanged, or s is disjoint from c_δ, in which case we simply ignore c_δ.

The case which requires special treatment is when one or two endpoints of s lie inside c_δ. However, this can happen in at most two nodes in each level of our partition tree, so that there are only $O(\log n)$ cells that require special treatment. Let c_δ be one of these cells. Handling the segments stored in S_δ is easy, so consider only the segments stored in T_δ. Since they cut completely through c_δ and c_δ is convex, we can extend each of these segments to a full line without affecting the set of intersections of these objects with s within c_δ. If we now dualize the problem (as in [8]), the (extended) data lines become points in the dual plane, and the (portion within c_δ of the) query segment s becomes a double wedge w_s. See figure 3.

The set of segments in T_δ that intersect s within c_δ is easily seen to be the same as the set of dual points contained in the wedge w_s. Thus to report or to count these segments, we need to perform a double–wedge range query in a given set of points in the dual plane. We can solve this problem by constructing a (conjugation) partition tree on this set of points, as in [10]. This requires extra $O(m)$ storage and $O(m \log m)$ preprocessing time, where $m = |T_\delta|$, and allows us to perform a double–wedge query in time $O(k_\delta + m^\gamma)$ for reporting the k_δ segments in T_δ intersected by s, or in time $O(m^\gamma)$ for counting those segments. However, there are at most two nodes δ in each level of the segment partition tree that need to be queried by s in this manner, and, by Lemma 3.2, the size of T_δ for a node δ at level i of the tree is only $O(n/2^i)$. Thus the total cost of these special queries is

$$O(k + \sum_{j=1}^{O(\log n)} (n/2^j)^\gamma) = O(k + n^\gamma)$$

for reporting queries, and, by the same argument, $O(n^\gamma)$ for counting queries. We thus obtain

Theorem 4.6 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the k segments intersecting a given query segment s can be reported in time $O(k + n^\gamma \log n)$, or counted in time $O(n^\gamma \log n)$.*

5 Triangle Stabbing

The triangle stabbing problem is the following: Let V be a set of n triangles in the plane; store V such that those triangles that contain a given query point p can be determined (reported or counted) efficiently. We will show how to reduce these problems to that of answering segment intersection queries, and use the techniques from the previous section to solve them.

As a first step we cut each triangle in two halves with a vertical line through the middle vertex. Hence, from now on we can assume that each triangle has one edge parallel to the y-axis. We project all triangles on the x-axis. In this way we obtain a set of intervals. On this set of intervals we construct a standard segment tree (see e.g. [20]); we will refer to this tree as the *primary segment tree*. Each node δ of the primary segment tree corresponds to a vertical slab $slab_\delta$ in the plane. The intervals that are stored at δ correspond to triangles that completely cut through $slab_\delta$ (have no vertex in the interior of $slab_\delta$). When we want to perform a query with a point p we search with the x–coordinate of p in the segment tree. It is easy to see that all triangles that might contain p must be stored at some node δ on the search path of p. Hence, for each of these $O(\log n)$ nodes we have to search in the corresponding set of triangles.

Now consider a node δ with slab $slab_\delta$. Let V_δ be the set of triangles that are stored at δ. Each triangle $t \in V_\delta$ is extended to a double wedge w_t formed between the two lines containing the non-vertical bounding edges of t. See figure 4 for an illustration. Whenever we search in the set V_δ with a point p we know that p lies inside $slab_\delta$. Hence, we can as well search in the corresponding set of double wedges — the wedges w_t containing p will correspond to the triangles t containing p. So we are left with the following problem: Given a set of double wedges, determine those wedges containing a given point p. To solve this problem we dualize it (see [8] for a discussion on dualization). The double wedges become line segments and the point p becomes a line l_p. Hence, we want to know which of these line segments are intersected by the line l_p. This problem was solved in Theorems 4.4 and 4.5 using a segment partition tree.

So the complete data structure looks as follows. We build a primary segment tree on the x-projections of the (vertically split) triangles. For each node δ we extend the set of triangles V_δ to double wedges, dualize them to line segments and build a segment partition tree of the set of line segments, as used in Theorems 4.4 and 4.5. A query with a point p is performed by searching with the x-projection of p in

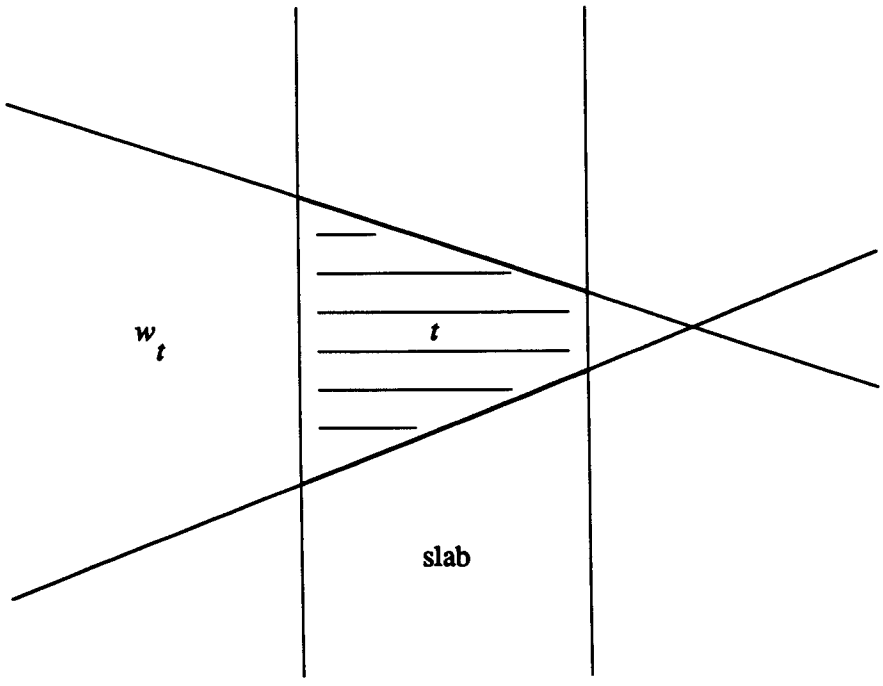

Figure 4: Extending triangles to wedges.

the primary segment tree and then by searching, for each node δ along the search path in the primary tree, with the dual line l_p in the corresponding partition tree to report or to count the dual line segments intersecting l_p. These line segments, collected over all nodes δ of the primary tree as above, correspond to the triangles containing p.

Theorem 5.1 *Given a set of n triangles in the plane, they can be stored in a data structure that uses $O(n \log^2 n)$ storage and can be constructed in $O(n \log^3 n)$ preprocessing time, such that the k triangles that contain a given query point p can be reported in time $O(k + n^\gamma \log n)$, or counted in time $O(n^\gamma \log n)$.*

Proof. Note that in a segment tree, each interval occurs at most twice in each level. Hence, all associated structures on one level of the primary segment tree contain $O(n)$ line segments altogether, and, hence, can be constructed in time $O(n \log^2 n)$ according to Theorem 4.4 or 4.5. So the total construction time is $O(n \log^3 n)$. The amount of storage required is similarly bounded.

To prove the bound on query time, note that the number of segments stored in an associated partition tree at level i of the primary segment tree is bounded by $O(n/2^i) = O(2^{\log n - i})$. Hence, the total reporting query time is bounded by

$$O(\sum_{j=1}^{\log n} (k_j + (2^j)^\gamma \log 2^j))$$

(where k_j is the number of triangles found at this node), which is bounded by $O(k + n^\gamma \log n)$. A similar argument applies to the counting version of the problem. □

6 Interval Partition Trees

In this section we describe a basically different way of storing line segments in partition trees, which we call an *interval partition tree* because it bears close resemblance to the ordinary interval tree for storing a set of segments on a line. A main advantage of the interval partition tree is that it uses only linear storage.

Again we construct a partition tree P on the set of endpoints of the line segments. Now for each line segment s we determine the lowest node δ in P such that s lies completely in the interior of the cell c_δ corresponding to δ. Let l_δ be the dividing line in δ. If s lies on l_δ we store it in an interval tree I_δ associated with δ. Otherwise we store at each son of δ the part of s that lies in the corresponding cell in some structure T depending on the type of query we want to answer (note that s will always intersect l_δ). Hence, each line segment is either stored in an interval tree I_δ or in two associated T structures. Defining $B_T(n)$, $M_T(n)$ as in Section 3, the following theorem is immediate:

Theorem 6.1 *An interval partition tree on n line segments uses $O(M_T(n))$ storage and can be constructed in time $O(B_T(n) + n \log n)$.*

Note that the line segments stored in a T_δ structure are all anchored at the dividing line $l_{father(\delta)}$ at the father of δ and that their other endpoint lies inside c_δ. See figure 5.

7 Ray Shooting

The ray shooting problem is the following: Given a set of n line segments in the plane, store them such that, for a given point p and a direction d, the first line segment hit when shooting from p in direction d can be determined efficiently. When the line segments form the edges of a simple polygon the shooting problem is known to be solvable with a query time of $O(\log n)$, using a structure that requires $O(n)$ storage and can be constructed in time $O(n\text{loglog}\, n)$ (see [5] and [11], using the triangulation method of [22]).

In this section we provide efficient techniques for solving more general instances of the ray shooting problem. We first consider the situation in which the set V consists of n non-intersecting line segments. We store these line segments in an interval partition tree. Now consider a node δ of the partition tree. All segments stored at δ are anchored at the dividing line $l_{father(\delta)}$ of the father of δ. Take the line $l_{father(\delta)}$ together with the parts of the line segments lying in the cell c_δ. Since the part of the complement of their union that lies on the c_δ-side of $l_{father(\delta)}$ is simply

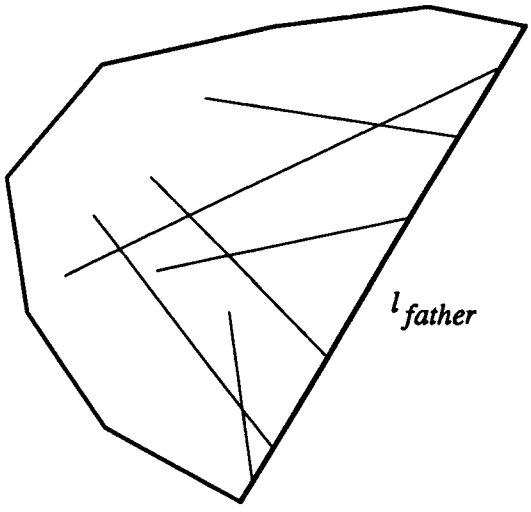

Figure 5: Line segments stored in T_δ.

connected (see figure 6), it is well known that we can apply to it the ray shooting techniques of [5, 11] as if it were a simple polygon. Let T_δ denote the resulting shooting structure. It immediately follows from Theorem 6.1 that the complete interval partition tree structure requires preprocessing time of $O(n \log n)$ and uses $O(n)$ storage.

Now assume we want to perform a ray shooting query from point p in direction d. Let ρ_d be the ray that runs from p in direction d. First determine all nodes δ in the partition tree such that ρ_d (partially) intersects c_δ. Because the underlying structure is a conjugation tree, the number of cells intersected by ρ_d is $O(n^\gamma)$ and they can be determined in $O(n^\gamma)$ time. For each of these nodes δ we perform a separate ray shooting query, as follows. If p lies inside c_δ we perform a shooting query from p in direction d in T_δ. Otherwise we determine the intersection of ρ_d with the boundary of c_δ which is nearest to p, and shoot from that intersection point in direction d into T_δ. In both cases we either hit a segment stored at T_δ, or we hit $l_{father(\delta)}$, or we don't hit anything. Only in the case we hit an actual line segment we consider this as a real hit. We also check whether ρ_d intersects l_δ and, if so, query I_δ with the intersection point to see whether any line segment on l_δ is hit (there can be at most one such segment). In this way we collect a total of at most $O(n^\gamma)$ candidate segments. The one among them nearest to p (in direction d) is the output to our shooting query.

The correctness of this method is easily established. For the time complexity note that we query at most $O(n^\gamma)$ T_δ and I_δ structures. Each of these queries costs $O(\log n)$ time. This leads to the following result.

Theorem 7.1 *Given a set of n non-intersecting line segments in the plane, they can be stored in a data structure that uses $O(n)$ storage and can be constructed in time*

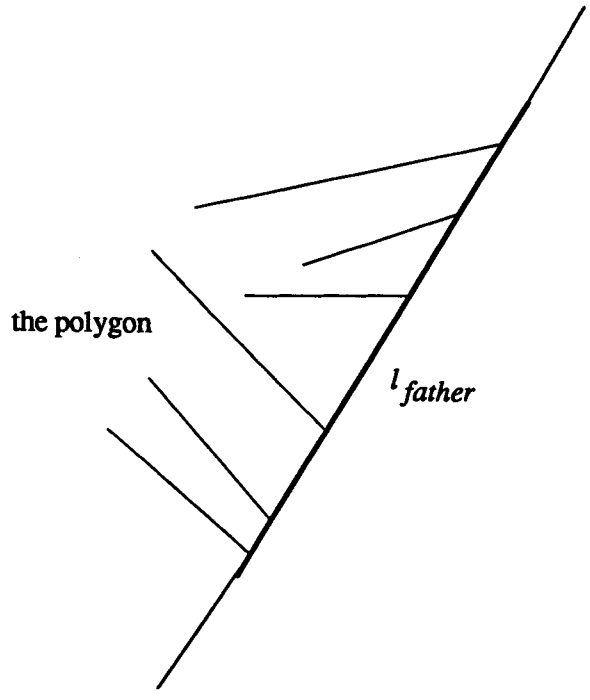

Figure 6: A polygon arising at a node δ.

$O(n \log n)$, such that ray shooting queries into the given segments can be answered in time $O(n^\gamma \log n)$.

Note that for the above method it is unneccessary to use the complicated triangulation technique of [22]. Any $O(n \log n)$ triangulation algorithm will do as well, without asymptotically increasing the time bounds. Also note that after the first query, any subsequent segment that ρ_d hits can be determined in time $O(\log n)$. To this end we sort all the candidate intersections computed by the preceding procedure, in increasing distance from p. When we want to find the next segment to be hit by ρ_d, we only have to perform a new shooting query in the cell in which the previous intersection point was found, thus requiring only $O(\log n)$ additional time. The new intersection (if it exists) is added to the sorted list of candidate intersections, and the next point in order is reported. Details are left to the reader.

Next consider the case where the given segments can intersect. This introduces several new technical difficulties, which can be solved at the expense of making our structures more complicated, which somewhat defeats our purpose of using as simple a technique as possible. Still we include a brief sketch of this solution for the sake of completeness, and also because it is somewhat different from other known techniques (such as that of [2]).

We use a segment partition tree, instead of an interval partition tree, to solve this more general problem. Observe first that shooting from outside a cell is relatively easy. Specifically, if ρ_d crosses a cell c_δ but p lies outside c_δ, we need to shoot, as

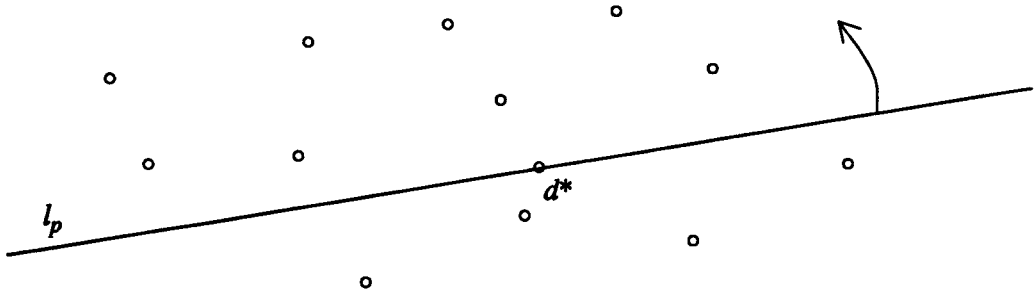

Figure 7: The dual version of ray shooting in an arrangement of lines

above, from the intersection, nearest to p, of ρ_d with the boundary of c_δ. To this end we compute the unbounded cell of the arrangement of the (portions within c_δ of the) segments stored at δ, and intersect it with c_δ. It is easily verified that this intersection consists of a linear number of simply connected regions, whose boundary consists of $O(m\alpha(m))$ subsegments (where $m = |T_\delta|$ and $\alpha(m)$ is the functional inverse of Ackermann's function), and that all these regions can be computed in time $O(m\alpha(m)\log^2 m)$ (see [14, 19, 24]). We next preprocess each of these regions for logarithmic–time ray shooting, as above, in total time $O(m\alpha(m)\log m)$. Now shooting into c_δ from the outside can be easily accomplished in $O(\log m)$ time by locating the appropriate region and shooting inside it.

Shooting from the inside of a cell is more difficult, however. Fortunately, there is a single such cell in each level of the partition tree, so we need to solve only $O(\log n)$ inside–shooting queries. Let c_δ be one of the cells containing p. Extend all segments stored at δ into full lines, and do the shooting in the arrangement of these lines. If the nearest hit lies inside c_δ, it coincides with the nearest hit of ρ_d with the unextended segments in T_δ; if it lies outside c_δ, we simply ignore it.

The problem thus reduces to that of ray–shooting from a point p in direction d in an arrangement of m lines. Dualize the lines to obtain a set S of m points. The query ray is transformed into the pair (l_p, d^*), where l_p is the line dual to p and d^* is the point on l_p dual to the line containing ρ_d; the query itself becomes: find the first point of S, if any, hit as we rotate l_p about d^* in counterclockwise direction until it becomes vertical; see figure 7.

We solve this problem as follows. Construct a (conjugation) partition tree P on the points of S. For each cell c_ζ of this auxiliary partition tree maintain the convex hull H_ζ of the points of S in c_ζ. Now, given the query (l_p, d^*), search with l_p in P and collect all cells that l_p misses (as in [10]) — there are $O(m^\gamma)$ such cells. For each of these cells c_ζ compute, in $O(\log m)$ time, the tangents from d^* to H_ζ. Collecting all such tangents, and choosing the one with the smallest slope that is larger than the slope of l_p, we obtain the (dual of the) solution to our shooting query. Thus the overall cost of this inside–shooting query is $O(m^\gamma \log m)$.

Putting everything together, we obtain the following result:

Theorem 7.2 *Given a set of n possibly intersecting line segments in the plane, they can be stored in a data structure that uses $O(n \log^2 n)$ storage and can be constructed in time $O(n\alpha(n)\log^3 n)$, such that ray shooting queries into the given segments can be answered in time $O(n^\gamma \log n)$.*

Proof. The storage and preprocessing time bounds follow from Theorem 3.1, noting that storing all convex hulls H_ζ, for a fixed node δ of the main partition tree, requires $O(m \log m)$ storage, where $m = |T_\delta|$, and this dominates the $O(m\alpha(m))$ storage requires to store the unbounded cell of the segments in T_δ; similarly, for preprocessing time, the time $O(m\alpha(m)\log^2 m)$ needed to compute the unbounded cell dominates the time needed to compute the convex hulls H_ζ.

The query time is analyzed as in the proofs of Theorems 4.6 and 5.1. Specifically, it suffices to bound the cost of the inside–shooting queries. Since we need to perform only one such query at each level of the main partition tree, it follows from Lemma 3.2 that the total cost of these queries is

$$O\Big(\sum_{j=1}^{O(\log n)} (n/2^j)^\gamma \log(n/2^j)\Big) = O(n^\gamma \log n)\,.$$

□

8 Conclusions and Open Problems

In this paper we have presented two data structures based on partition trees for storing line segments in the plane, which can be used for answering different types of queries such as segment intersection and ray shooting queries. Although we based our results on conjugation trees it is also possible to adapt our techniques and use, for example, the partition trees of Haussler and Welzl [15]. This would improve the query times for the different problems to $O(n^{2/3+\epsilon})$ for arbitrary small ϵ at the cost of having to use randomization (and increasing the complexity of the structure). Although the same or even improved query times can be obtained using the techniques of Dobkin and Edelsbrunner [7] and Agarwal [2] we believe that our methods have the desired simplicity to make them implementable and potentially practical.

We think that the two basic types of partition trees introduced can be used for solving other query problems involving line segments as well, using an appropriate type of associated T-structure. In an acompanying paper ([21]) we show that it is also possible to obtain dynamic versions of the structures presented here.

A number of open problems remain. It would be interesting to study whether the techniques presented can also be used with the most recent partition tree of Welzl [23] as an underlying structure. Alternatively, the ultimate goal is to design a partition tree structure that is induced by a convex decomposition of the plane and has the properties that (a) it can be constructed in close to linear time, and (b) only about $\sqrt{n}$ cells of the decomposition are crossed by any line. Our techniques should

be applicable to such a partition tree structure. Finally, concerning ray shooting, we note that no significant lower bound is known for that problem (even when we are allowed only roughly linear storage), so query time bounds like $O(n^\gamma \log n)$, or even close to $\sqrt{n}$, may still be far worse than optimal. In our approach, for instance, we can do the shootings into the structures T_δ in the order in which the cells c_δ are crosses by ρ_d, and stop as soon as a real hit is found. Can this be used to obtain a faster query time?

References

[1] Agarwal, P.K., An efficient deterministic algorithm for partitioning arrangements of lines and its applications, *Proc. 5th ACM Symp. on Computational Geometry,* 1989, pp. 11–22.

[2] Agarwal, P.K., Ray shooting and other applications of spanning trees with low stabbing number, *Proc. 5th ACM Symp. on Computational Geometry,* 1989, pp. 315–325.

[3] Chazelle, B., Polytope range searching and integral geometry, *Proc. 28th IEEE Symp. on Foundations of Computer Science,* 1987, pp. 1–10.

[4] Chazelle, B., A functional approach to data structures and its use in multidimensional searching, *SIAM J. Comput.* **17** (1988), pp. 427-462.

[5] Chazelle, B., and L. Guibas, Visibility and intersection problems in plane geometry, *Proc. 1st ACM Symp. on Computational Geometry,* 1985, pp. 135–146.

[6] Chazelle, B., and E. Welzl, Quasi optimal range searching in spaces with finite VC–dimension, *Discrete Comput. Geom.* **4** (1989), to appear.

[7] Dobkin, D., and H. Edelsbrunner, Space searching for intersecting objects, *J. Algorithms* **8** (1987), pp. 348–361.

[8] Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, New York, 1987.

[9] Edelsbrunner, H., L. Guibas, J. Hershberger, R. Seidel, M. Sharir, J. Snoeyink and E. Welzl, Implicitly representing arrangements of lines and of segments, *Discrete Comput. Geom.* **4** (1989), to appear.

[10] Edelsbrunner, H., and E. Welzl, Halfplanar range search in linear space and $O(n^{0.695})$ query time, *Inform. Proc. Lett.* **23** (1986), pp. 289–293.

[11] Guibas, L., J. Hershberger, D. Leven, M. Sharir and R. Tarjan, Linear time algorithms for visibility and shortest path problems in triangulated simple polygons, *Algorithmica* **2** (1987), pp. 209–233.

[12] Guibas, L., M.H. Overmars and M. Sharir, Intersecting line segments, ray shooting and other applications of geometric partitioning techniques, *Proc. SWAT 88,* Lect. Notes in Comp. Science 318, Springer-Verlag, Heidelberg, 1988, pp. 64–73.

[13] Guibas, L., M.H. Overmars and M. Sharir, Ray shooting, implicit point location, and related queries in arrangements of segments, Techn. Rep. 433, Courant Inst. of Math. Sciences, New York University, 1989.

[14] Guibas, L., M. Sharir and S. Sifrony, On the general motion planning problem with two degrees of freedom, *Discrete Comput. Geom.* **4** (1989), to appear.

[15] Haussler, D., and E. Welzl, ϵ-nets and simplex range searching, *Discrete Comput. Geom.* **2** (1987), pp. 127–151.

[16] Matoušek, J., Construction of ϵ-nets, *Proc. 5th ACM Symp. on Computational Geometry,* 1989, pp. 1–10.

[17] Matoušek, J., Spanning trees with low crossing number, manuscript, 1988.

[18] McCreight, E.M., Priority search trees, *SIAM J. Comput.* **14** (1985), pp. 257–276.

[19] Pollack, R., M. Sharir and S. Sifrony, Separating two simple polygons by a sequence of translations, *Discrete Comput. Geom.* **3** (1988), pp. 123–136.

[20] Preparata, F., and M. Shamos, *Computational Geometry: An Introduction,* Springer-Verlag, Heidelberg, 1985.

[21] Schipper, H., and M.H. Overmars, Dynamic partition trees, Techn. Rep., Dept. of Comp. Science, University of Utrecht, 1989, to appear.

[22] Tarjan, R., and C. Van Wyk, An $O(n\log\log n)$ algorithm for triangulating simple polygons, *SIAM J. Comput.* **17** (1988), pp. 143–178.

[23] Welzl, E., Partition trees for triangle counting and other range searching problems, *Proc. 4th ACM Symp. on Computational Geometry,* 1988, pp. 23–33.

[24] Wiernik, A., and M. Sharir, Planar realizations of nonlinear Davenport–Schinzel sequences by segments, *Discrete Comput. Geom.* **3** (1988), pp. 15–47.

[25] Willard, D.E., Polygon retrieval, *SIAM J. Comput.* **11** (1982), pp. 149–165.

Storing line segments in partition trees

Mark H. Overmars, Haijo Schipper and Micha Sharir

RUU-CS-89-17
August 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Storing line segments in partition trees

Mark H. Overmars, Haijo Schipper and Micha Sharir

Technical Report RUU-CS-89-17
August 1989

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

Storing line segments in partition trees

Mark H. Overmars* Haijo Schipper* Micha Sharir†

July 1989

Abstract

We design two variants of planar partition trees, called *segment partition trees* and *interval partition trees*, that can be used for storing arbitrarily oriented line segments in the plane in an efficient way. The raw structures use $O(n \log n)$ and $O(n)$ storage, respectively, and their construction time is $O(n \log n)$. In our applications we augment these structures by certain (simple) auxiliary structures, which may increase the storage and preprocessing time by a polylogarithmic factor. It is shown how to use these structures for solving line segment intersection queries, triangle stabbing queries and ray shooting queries in reasonably efficient ways. If we use the conjugation tree of [10] as the underlying partition tree, the query time for all problems is $O(n^\gamma \log n)$, where $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$. The techniques are fairly simple and easy to understand.

1 Introduction

Partition trees have originally been designed for storing points and answering half-planar range queries in which, given a set S of n points in the plane, we wish to count or to report efficiently all points lying in a query half-plane (or in a query triangle). Partition trees are based on a method of recursively partitioning the plane into convex subregions such that (1) any given line intersects only a small number of the regions, and (2) the partitioning induces a partitioning of the given set of points into subsets of progressively decreasing size. In this way it is possible to

*Department of Computer Science, University of Utrecht, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. Research of the first author was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (Project ALCOM).

†Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, N.Y. 10012, U.S.A., and School of Mathematical Sciences, Tel Aviv University, 69978 Tel Aviv, Israel. Work by this author has been supported in part by Office of Naval Research Grant N00014-87-K-0129, by National Science Foundation Grants DCR-83-20085 and CCR-89-01484, and by grants from the Digital Equipment Corporation, the IBM Corporation, the U.S.-Israeli Binational Science Foundation, the NCRD - the Israeli National Council for Research and Development, and the Fund for Basic Research in Electronics, Computers and Communication, administered by the Israeli Academy of Sciences.

identify large subsets of S that lie on one side of the given line and, hence, lie either completely inside or outside the query half-plane.

The first type of partition tree was introduced by Willard [25]. His structure has a query time of $O(n^{\log_6 4}) = O(n^{.774})$ and a preprocessing time of $O(n^2)$. These results were improved by Edelsbrunner and Welzl [10] who defined a so-called *conjugation tree*. This tree has a query time of $O(n^\gamma)$, where $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$. In section 2 we give a brief description of conjugation trees. Since then the results have been improved by Haussler and Welzl [15] but their methods are not deterministic and use randomization. This has recently been remedied by Matoušek [16] and Agarwal [1]. Welzl [23] (see also Chazelle and Welzl [6]) has devised a different kind of partition tree, not related to a matching convex decomposition of the plane; this structure achieves query time bounds that almost attain the theoretically optimal $O(\sqrt{n})$ bound (if only linear storage is allowed; see Chazelle [3] for a matching lower bound). However, the preprocessing time of these methods is still relatively high. See also [1, 2, 9, 17] for a variety of recent related results.

Although partition trees were designed for storing sets of points, it has been realized that they can also be used for solving problems concerning sets of line segments or other geometric objects (see Dobkin and Edelsbrunner [7], Guibas, Overmars and Sharir [12, 13] and Agarwal [2]). In this paper we present two new data structures based on partition trees that can be used for solving a number of query problems concerning line segments in an easy way. Both consist of a partition tree on the set of endpoints of the line segments with segments stored at particular nodes in the tree. In our descriptions we will base our new structures on the conjugation tree although the trees of Haussler and Welzl [15] could be used as well. As a matter of fact, our techniques should apply to any underlying partition tree that is induced by an appropriate convex decomposition of the plane. The query time should become $O(C(n)\log n)$, where $C(n)$ is the maximum number of partition tree nodes whose corresponding regions in the plane decomposition are crossed by a line. For example, using the technique of [15], we obtain query time $O(n^{2/3+\epsilon}\log n) = O(n^{2/3+\epsilon})$, for any $\epsilon > 0$. (The optimal structure by Welzl [23] is not suited for our purpose because it is not based on such a decomposition as the others are.)

The first structure we present is a *segment partition tree*, which bears some resemblance to the normal segment tree (see e.g. [20]). It uses $O(n\log n)$ storage because every segment is stored at $O(\log n)$ nodes of the tree. We use this structure for line segment intersection queries and for triangle stabbing queries. In the first type of query, we wish to preprocess a collection of line segments so that, given a query line or segment, we can efficiently report (or count) all the given segments it intersects; in the second type of query, we wish to preprocess a collection of triangles so that, given a query point, we can efficiently report (or count) all the given triangles containing it. The second structure that we present is the *interval partition tree*, which is a kind of two-dimensional variant of the normal interval tree (see e.g. [20]). This structure is used to answer ray shooting queries, where we are given a set of line segments and wish to preprocess it so that, given a query ray,

we can efficiently determine the first intersection of the ray with the given segments (if one exists at all). We use an interval partition tree if the given segments are non–intersecting; if they do intersect, we design a more sophisticated technique in which we again use the segment partition tree. We believe that both structures will be useful for solving other problems concerning line segments as well.

The results obtained in this paper are not the best possible. They all achieve query times of $O(n^\gamma \log n)$ with a preprocessing time of $O(n \log^i n)$ where i is 1, 2 or 3. Most of these results can also be achieved in an equally efficient way by using the techniques of Dobkin and Edelsbrunner [7] or of Guibas et al. [12, 13], but their methods are complicated. Moreover, some of the results have already been improved by Agarwal [2] but the preprocessing time of his methods is much higher and, again, the techniques are complicated. The power of the (majority of the) methods presented here is their relative simplicity, which makes them easy to understand and, probably, also easier to implement.

The paper is organized as follows:

In Section 2 we give a brief description of the conjugation tree that will be used as an underlying data structure for our techniques and review some important properties of this structure.

In Section 3 we describe the segment partition tree and show that it requires $O(n \log n)$ storage and construction time.

In Section 4 we concentrate on intersection queries. As a first instance we consider the problem of storing a set V of non-intersecting line segments such that for a given query line or query line segment s the segments in V that intersect s can be reported or counted efficiently. Next we look at the case in which the line segments in V can intersect.

In Section 5 we apply the results in Section 4 to the triangle stabbing problem: Given a set of triangles in the plane, store them such that for a given query point p we can efficiently determine the (number of) triangles that contain p.

In Section 6 we describe the interval partition tree, which uses only linear storage, and prove some of its properties.

In Section 7 we use the interval partition tree to solve the ray shooting problem in a set of non-intersecting line segments: given a point p and a direction d, determine the first line segment we hit when shooting from p in direction d. We also consider the case where the given segments can intersect, and develop a more sophisticated procedure for efficient ray shooting in such collections.

Finally, in Section 8, we give some conclusions and directions for further research.

2 Preliminaries

In this section we give a brief introduction to conjugation trees. For more details see Edelsbrunner and Welzl [10].

A conjugation tree is a binary tree that stores a set of points V in the plane. For a directed line l we denote by $Left(l)$ the set of points to the left of l, by $Right(l)$

the set of points to the right of l, and by $On(l)$ the set of point lying on the line l.

Definition 2.1 *Let V be a set of n points in the plane. A directed line l is called a* bisector *of V if $|Left(l)| \leq \lceil n/2 \rceil$ and $|Right(l)| \leq \lceil n/2 \rceil$. Another line l' is called a* conjugate *of l if l' is a bisector of both $Left(l)$ and $Right(l)$.*

Definition 2.2 *Let V be a set of n points and let l be a bisector of V. Let l' be a conjugate of l. If V is not empty, a* conjugation tree *for V (and l) is a binary tree with a root δ storing $l_\delta = l$ and $On(l)$ in some form, whose left subtree is a conjugation tree of $Left(l)$ and l', and whose right subtree is a conjugation tree of $Right(l)$ and l'. If V is empty, the conjugation tree for V is also empty.*

Because both $Left(l)$ and $Right(l)$ have size at most $\lceil n/2 \rceil$ the depth of a conjugation tree is bounded by $O(\log n)$. Each point is stored in exactly one $On(l)$ structure for some node. The form of these structures depends on the type of problem one wants to solve. In our applications the structure will be completely unneccessary and, hence, is removed from the tree. The tree clearly has $O(n)$ nodes. In [10] it is shown that a conjugation tree can be constructed in time $O(n \log n)$.

Theorem 2.1 *A conjugation tree for a set of n points uses $O(n)$ storage and can be constructed in time $O(n \log n)$.*

One can view a conjugation tree as a way of partitioning the plane. The root of the tree corresponds to the complete plane. Each son corresponds to the part on one side of the bisector l, etc. In this way it is easy to see that each node δ corresponds to a convex cell c_δ that has at most $O(\log n)$ edges. The main property of a partition tree is that any line intersects the cells of only a limited number of nodes in the conjugation tree. This is based on the fact that for each node whose cell is intersected by the line at least one of its four grandsons is not intersected by the line. For details and proof of the following theorem see [10].

Theorem 2.2 *Let P be a conjugation tree containing n points and let l be any line. There are at most $O(n^\gamma)$ nodes δ in P where l intersects c_δ, with $\gamma = \log(1+\sqrt{5}) - 1 \approx 0.695$.*

In the sequel, when talking about partition trees we assume the structure is actually a conjugation tree. (As discussed in the introduction, the reason for talking about partition trees in general is that the methods apply also to other types of partition trees.)

3 Segment Partition Trees

To store line segments in a partition tree we will add an associated data structure for each node in the partition tree that stores some number of segments. The type of structure will depend on the type of problem we want to solve using the structure.

The first way of doing this is what we call a *segment partition tree* because it bears close resemblance to normal segment trees. As a first step we construct a partition tree P containing the endpoints of all the line segments. Next we search with each line segment s in the partition tree. At each node δ of P we have the following possibilities:

1. s completely intersects c_δ (the cell of node δ), i.e., no endpoint of s lies in the interior of c_δ. In this case we store the part of s inside the cell in a structure T_δ associated with δ.

2. s lies on l_δ (the dividing line of the cell). In this case we store s in an ordinary segment tree S_δ. As S_δ will only contain the segments lying on l_δ, we can indeed represent it by an ordinary 1-dimentional segment tree.

3. s lies completely to the left of l_δ (with possibly one endpoint on l_δ) and does not intersect c_δ completely. In this case we continue the search at the left son of δ. Similarly, when s lies completely to the right of l_δ we continue searching at the right son of δ.

4. s properly intersects l_δ and has at least one endpoint in the interior of c_δ. In this case we continue the search at both sons of δ.

Hence, for each node δ we get two structures: S_δ is a normal segment tree of all segments that lie on l_δ, and T_δ is a structure containing all segments that completely cut through c_δ but did not completely cut through $c_{father(\delta)}$. Note that all segments stored in T_δ intersect the dividing line $l_{father(\delta)}$.

There is a slight problem here. How do we determine in case 1 whether a segment completely intersects c_δ and how do we determine the part inside c_δ? This can be solved in the following way. Rather than continuing the search recursively with the whole segment we always continue with only the part inside the current cell. Hence, in case 4 we pass to each son only the part of the segment lying on the appropriate side of l_δ. Moreover, we mark endpoints of the segments when they lie on a dividing line. This can easily be done as part of the appropriate actions in cases 3 and 4. Now case 1 simply checks whether both endpoints are marked, and the part inside the cell is the part of the segment that is left over.

Assuming that an associated T-structure of n points can be constructed in time bounded by $B_T(n)$ and that it uses at most $M_T(n)$ storage, we get the following theorem (assuming B_T and M_T satisfy the functional equation $F(a)+F(b) \leq F(a+b)$):

Theorem 3.1 *A segment partition tree on n line segments uses $O(M_T(n)\log n)$ storage and can be constructed in time $O(B_T(n)\log n)$.*

Proof. The partition tree itself uses $O(n)$ storage and can be constructed in time $O(n \log n)$. When a segment is stored in a segment tree S_δ it must have at least one endpoint on l_δ. It immediately follows that each segment is stored in at

most two segment trees S_δ. Hence, in total, the segment trees will use $O(n \log n)$ storage and can be constructed in time $O(n \log n)$.

When a segment s is stored in a structure T_δ it (or rather its full original version) must have an endpoint in $c_{father(\delta)}$ (unless δ is the root). Hence, on each level of the segment partition tree there can be at most 2 nodes where s is stored in T_δ. As a result all T-structures on one level of the partition tree use a total of $O(M_T(n))$ storage and can be constructed in time $O(B_T(n))$. As the depth of the partition tree is bounded by $O(\log n)$, the bounds follow. □

In fact, as we shall see below, the preprocessing time can sometimes be reduced by maintaining structural information about the set of line segments when adding them to the tree.

Before closing this section, we observe the following useful property of segment partition trees:

Lemma 3.2 *Assume that no two of the given segments have an endpoint in common. Then the number of segments stored in a node δ at level i of the partition tree is $O(n/2^i)$.*

Proof. Let s be a segment that is stored in δ. Then s must have an endpoint inside $c_{father(\delta)}$. The number of such points is $O(n/2^i)$ and the above assumption completes the proof. □

If k segments share an endpoint, we can modify the construction by treating this point as k distinct points that happen to coincide. With some care, it is possible to modify the construction of the conjugation tree so that it handles such cases correctly.

4 Intersection Queries

In this section we will show how a number of variants of the line segment intersection query problem can be solved using segment partition trees. We start with a restricted version of the problem: Let V be a set of n non-intersecting line segments in the plane; we want to store them such that for a given query line l we can efficiently report the line segments intersected by l.

To solve this problem we store the set V of line segments in a segment partition tree. For each node δ we store in T_δ the line segments ordered by intersection with $l_{father(\delta)}$ (we use any convenient balanced tree structure to represent T_δ). Now assume we want to perform a query with line l. First we determine all nodes δ for which l intersects c_δ. For each such node δ we determine the intersection of l with $l_{father(\delta)}$. If l does intersect $l_{father(\delta)}$ inside the cell we search with the intersection in T_δ. Assume it lies between the segments s_i and s_{i+1}. We check whether l intersects s_i or s_{i+1} (inside the cell). It clearly cannot intersect both because the line segments stored in T_δ do not intersect each other and extend to the cell boundary, and the cell is convex. If l intersects none of them we are finished at this node. Otherwise,

assume l intersects s_i. We report s_i and check whether l intersects s_{i-1}, s_{i-2}, etc. until we find a segment that is not intersected by l. It is easy to see that in this way all segments in T_δ that intersect l (inside the cell) are correctly reported.

If l does not intersect $l_{father(\delta)}$ inside the cell, let l' be the part of l that lies inside c_δ. (l' can be maintained, at constant cost per cell, during the search with l in the partition tree.) We search with l' in T_δ in the following way. Compare l' with the segment s_r stored in the root of T_δ. If l' intersects s_r report the intersection and continue in both sons. Otherwise, if l' lies left of s_r continue in the left son, otherwise continue in the right son. (Because l' lies inside the cell, s_r cuts through the cell, and the cell is convex, it is easy to decide on which side l' lies: simply check l' with respect to the line l_{s_r} that is the extension of s_r.) Since the segments in T_δ do not intersect, this correctly reports all segments intersected by l inside the cell. The search obviously takes time $O(\log n)$ plus the number of intersected segments found.

It remains to treat the segments stored in S_δ. If l intersects l_δ we perform a simple "stabbing" query on S_δ to report all segments in S_δ that contain the intersection point between l and l_δ (see e.g. [20]). If l overlaps l_δ, the entire S_δ is reported.

Theorem 4.1 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the k segments intersecting a given query line l can be reported in time $O(k + n^\gamma \log n)$.*

Proof. The storage and a weaker $O(n \log^2 n)$ bound on the preprocessing time immediately follow from Theorem 3.1, noting that $M_T(n) = O(n)$ and $B_T(n) = O(n \log n)$. The query time follows from the fact that l intersects $O(n^\gamma)$ cells. In each cell we have to perform a query on the T-structure and a query on the S-structure. Both queries take time $O(\log n)$ plus the number of intersected segments found.

We next show that the preprocessing time can in fact be improved to $O(n \log n)$. What we need is to obtain the line segments to be stored at a node δ in an ordered way. Note that the ordering along $l_{father(\delta)}$ is the same as (or rather the reverse of) the ordering along the rest of the boundary of c_δ. Note also that, at the moment we reach node δ all these line segments must be intersecting the rest of the boundary of c_δ. So we can use the following technique. We insert all line segments simultaneously into the partition tree P. When we reach a node δ we have a set V^δ of line segments that (partially) lie inside c_δ. This set consists of two parts: the subset V_1^δ of line segments that completely lie inside the cell and the subset V_2^δ of line segments that already intersect the boundary. We keep V_2^δ ordered by intersection with the boundary of the cell (more precisely, we maintain a circular list of all endpoints of segments in V_2^δ that lie on the boundary of c_δ, sorted in circular order along that boundary). Now we proceed as follows. We determine the line segments in V_2^δ that now completely intersect the cell. We remove them from V_2^δ and build T_δ out of them. Since they are already (circularly) ordered, this takes linear time. We also determine the segments that have to go in the S_δ structure. The remaining segments

are passed to the sons of δ as follows. Find those segments in V_1^δ that intersect l_δ, and sort them in their order along that line. Determine which segments of V_2^δ should be passed to which son, and extract the two corresponding sublists out of V_2^δ. Those segments that are passed to both sons also intersect l_δ, but their intersection order along that line can be determined in linear time from their order along the boundary of c_δ. Finally, we merge the new list of segments of V_1^δ that intersect l_δ with each of the sublists into which V_2^δ has been decomposed; the remaining segments of V_1^δ are partitioned among the V_1 lists of the two sons of δ. This reduces the preprocessing time to $O(n \log n)$ because any segment is only involved once in sorting (when it intersects a dividing line for the first time). □

Now assume our query object is a line segment s rather than a line. We can use exactly the same structure. To perform a query we determine all cells that are intersected by s. For each corresponding node δ we do the following. We restrict s to the cell c_δ (this can be done, as above, during the searching with s, at constant cost per cell). We search with this restricted line segment s' in T_δ as follows. We compare s' with the segment s_r in the root of T_δ. If s' lies left of s_r we continue in the left son. If s' lies to the right of s_r we continue in the right son. Otherwise we report the intersection and continue in both sons. (Note that the segments reported by this procedure necessarily form a contiguous portion of T_δ.) This clearly takes time $O(\log n)$ plus the number of segments reported. We also check whether s intersects l_δ and, if so, perform a stabbing query in S_δ with the intersection point. Similarly, if s partially overlaps l_δ, it is also easy to report all segments in S_δ that partially overlap s. The following result is now immediate:

Theorem 4.2 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the k segments intersecting a given query line segment s can be determined in time $O(k + n^\gamma \log n)$.*

The result can easily be adapted to the situation in which we only want to count the number of segments intersected by s. Both the T_δ and the S_δ structures can also be used to count the number of intersections rather than report the intersections themselves. This leads to the following result:

Theorem 4.3 *Given a set of n non-intersecting line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log n)$, such that the number of segments intersecting a query line l or a query line segment s can be determined in time $O(n^\gamma \log n)$.*

Now let us consider the case in which our set V consists of possibly intersecting line segments and our query object is again a line l. Again we construct a segment partition tree on the set of endpoints of the line segments in V. For a node δ, each of the segments to be stored in T_δ intersects the boundary of c_δ twice—once on $l_{father(\delta)}$ and once on the rest of the boundary. We sort the intersections on $l_{father(\delta)}$ and also sort the intersections on the rest of the boundary. Thus we can represent each line

segment s as a pair (i_s, i'_s) where i_s is the index of its intersection along $l_{father(\delta)}$ and i'_s is the index of its intersection along the rest of the boundary. Both sets of intersections are stored in appropriate balanced search trees. In addition, we store the double indices (i_s, i'_s) in an appropriate structure Q_δ for performing orthogonal range queries; see below for details of this structure.

To perform a query with line l we first determine all the cells intersected by l. For each corresponding node δ we proceed as follows. First we check whether l intersects l_δ and, if so, perform a stabbing query on the segment tree S_δ with the intersection point (if l overlaps l_δ, the entire S_δ is reported). Next we determine the two intersections of l with the boundary of the cell c_δ (this can be done, as above, during the search with l at constant cost per cell). There are two possible cases: either one intersection lies on $l_{father(\delta)}$ or none lies on $l_{father(\delta)}$. If no intersection lies on $l_{father(\delta)}$ we search in the structure storing the intersection points of the segments in T_δ with the portion of the boundary of c_δ outside $l_{father(\delta)}$, to determine the segments having an endpoint that lies between the two intersections of l with the boundary. These are precisely the segments in T_δ that intersect l (see figure 1). This obviously takes time $O(\log n)$ plus the number of segments found. Otherwise, if l does intersect $l_{father(\delta)}$ we determine the location of the intersections of l with $l_{father(\delta)}$ and with the rest of the boundary among the intersections of the segments of T_δ with the boundary. Assume that on $l_{father(\delta)}$ the intersection lies between i_s and i_s+1 and on the rest of the boundary between $i'_{s'}$ and $i'_{s'}+1$. (See figure 2.) Now it is easy to see that the segments t intersecting l correspond to the points (i_t, i'_t) lying in the range $[-\infty : i_s] \times [i'_{s'}+1 : \infty]$ or in the range $[i_s+1 : \infty] \times [-\infty : i'_{s'}])$. Hence, we simply have to perform two orthogonal range queries on Q_δ. Since both range queries are half-infinite, we can represent Q_δ by two priority search trees (see McCreight [18]). This allows us to perform an orthogonal range query in time $O(\log n)$ plus the number of segments found; the storage for Q_δ is $O(n)$ and the preprocessing is $O(n \log n)$. This leads to the following result:

Theorem 4.4 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the k segments intersecting a given query line l can be determined in time $O(k + n^\gamma \log n)$.*

To solve the counting version of the problem, in which we want to count the number of segments intersecting l, we can no longer use priority search trees. Instead we can use the 2-dimensional range counting structure developed by Chazelle [4]. This structure has a query time of $O(\log n)$, and requires $O(n)$ storage and $O(n \log n)$ preprocessing time. Otherwise, processing a query is done exactly as above, leading to the following result:

Theorem 4.5 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the number of segments intersecting a given query line l can be determined in time $O(n^\gamma \log n)$.*

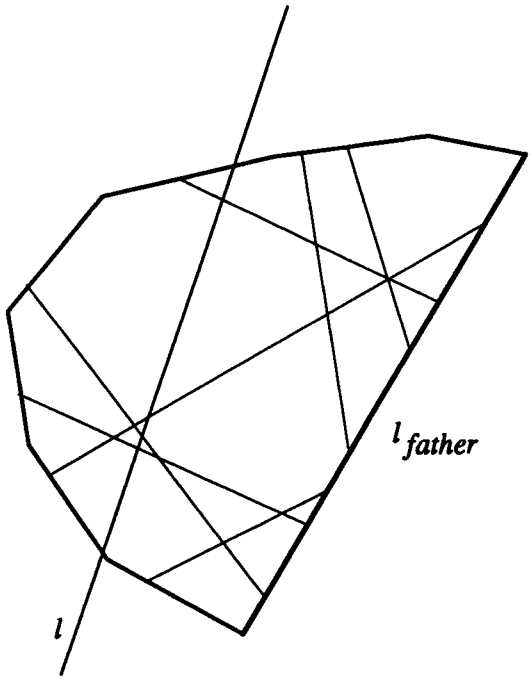

Figure 1: l does not intersect $l_{father(\delta)}$.

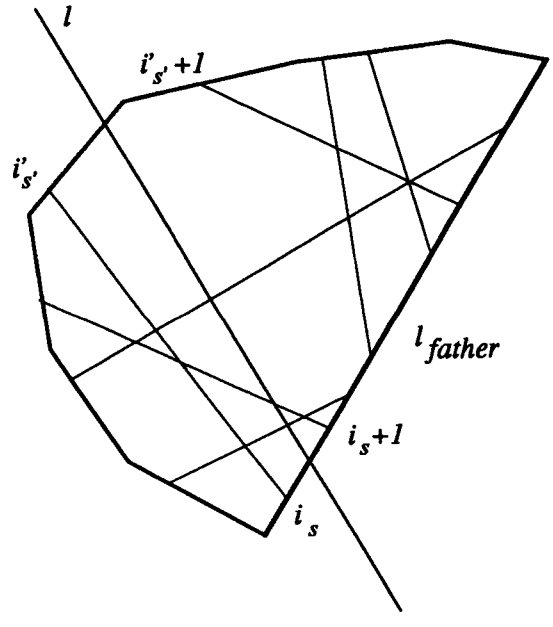

Figure 2: l does intersect $l_{father(\delta)}$.

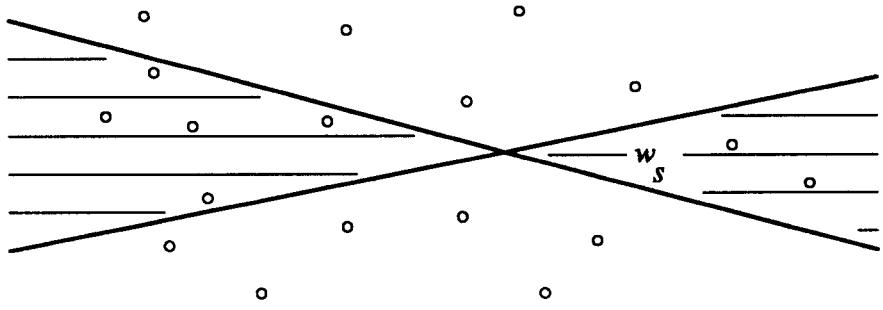

Figure 3: The dual form of the segment intersection query for a collection of intersecting segments.

Finally, we consider the case where the query object is a segment and the given segments can intersect. Let s be the query segment and let l denote the line containing s. We can essentially apply the procedures given above to the line l, but in addition keep track of the location of the endpoints of s along l. Thus, if l crosses a cell c_δ of the partition tree and no endpoint of s lies in c_δ, then either s completely cuts through c_δ, in which case the above procedures can be carried out unchanged, or s is disjoint from c_δ, in which case we simply ignore c_δ.

The case which requires special treatment is when one or two endpoints of s lie inside c_δ. However, this can happen in at most two nodes in each level of our partition tree, so that there are only $O(\log n)$ cells that require special treatment. Let c_δ be one of these cells. Handling the segments stored in S_δ is easy, so consider only the segments stored in T_δ. Since they cut completely through c_δ and c_δ is convex, we can extend each of these segments to a full line without affecting the set of intersections of these objects with s within c_δ. If we now dualize the problem (as in [8]), the (extended) data lines become points in the dual plane, and the (portion within c_δ of the) query segment s becomes a double wedge w_s. See figure 3.

The set of segments in T_δ that intersect s within c_δ is easily seen to be the same as the set of dual points contained in the wedge w_s. Thus to report or to count these segments, we need to perform a double–wedge range query in a given set of points in the dual plane. We can solve this problem by constructing a (conjugation) partition tree on this set of points, as in [10]. This requires extra $O(m)$ storage and $O(m \log m)$ preprocessing time, where $m = |T_\delta|$, and allows us to perform a double–wedge query in time $O(k_\delta + m^\gamma)$ for reporting the k_δ segments in T_δ intersected by s, or in time $O(m^\gamma)$ for counting those segments. However, there are at most two nodes δ in each level of the segment partition tree that need to be queried by s in this manner, and, by Lemma 3.2, the size of T_δ for a node δ at level i of the tree is only $O(n/2^i)$. Thus the total cost of these special queries is

$$O(k + \sum_{j=1}^{O(\log n)} (n/2^j)^\gamma) = O(k + n^\gamma)$$

for reporting queries, and, by the same argument, $O(n^\gamma)$ for counting queries. We thus obtain

Theorem 4.6 *Given a set of n (possibly intersecting) line segments in the plane, they can be stored in a segment partition tree that uses $O(n \log n)$ storage and can be constructed in time $O(n \log^2 n)$, such that the k segments intersecting a given query segment s can be reported in time $O(k + n^\gamma \log n)$, or counted in time $O(n^\gamma \log n)$.*

5 Triangle Stabbing

The triangle stabbing problem is the following: Let V be a set of n triangles in the plane; store V such that those triangles that contain a given query point p can be determined (reported or counted) efficiently. We will show how to reduce these problems to that of answering segment intersection queries, and use the techniques from the previous section to solve them.

As a first step we cut each triangle in two halves with a vertical line through the middle vertex. Hence, from now on we can assume that each triangle has one edge parallel to the y-axis. We project all triangles on the x-axis. In this way we obtain a set of intervals. On this set of intervals we construct a standard segment tree (see e.g. [20]); we will refer to this tree as the *primary segment tree*. Each node δ of the primary segment tree corresponds to a vertical slab $slab_\delta$ in the plane. The intervals that are stored at δ correspond to triangles that completely cut through $slab_\delta$ (have no vertex in the interior of $slab_\delta$). When we want to perform a query with a point p we search with the x–coordinate of p in the segment tree. It is easy to see that all triangles that might contain p must be stored at some node δ on the search path of p. Hence, for each of these $O(\log n)$ nodes we have to search in the corresponding set of triangles.

Now consider a node δ with slab $slab_\delta$. Let V_δ be the set of triangles that are stored at δ. Each triangle $t \in V_\delta$ is extended to a double wedge w_t formed between the two lines containing the non-vertical bounding edges of t. See figure 4 for an illustration. Whenever we search in the set V_δ with a point p we know that p lies inside $slab_\delta$. Hence, we can as well search in the corresponding set of double wedges — the wedges w_t containing p will correspond to the triangles t containing p. So we are left with the following problem: Given a set of double wedges, determine those wedges containing a given point p. To solve this problem we dualize it (see [8] for a discussion on dualization). The double wedges become line segments and the point p becomes a line l_p. Hence, we want to know which of these line segments are intersected by the line l_p. This problem was solved in Theorems 4.4 and 4.5 using a segment partition tree.

So the complete data structure looks as follows. We build a primary segment tree on the x-projections of the (vertically split) triangles. For each node δ we extend the set of triangles V_δ to double wedges, dualize them to line segments and build a segment partition tree of the set of line segments, as used in Theorems 4.4 and 4.5. A query with a point p is performed by searching with the x-projection of p in

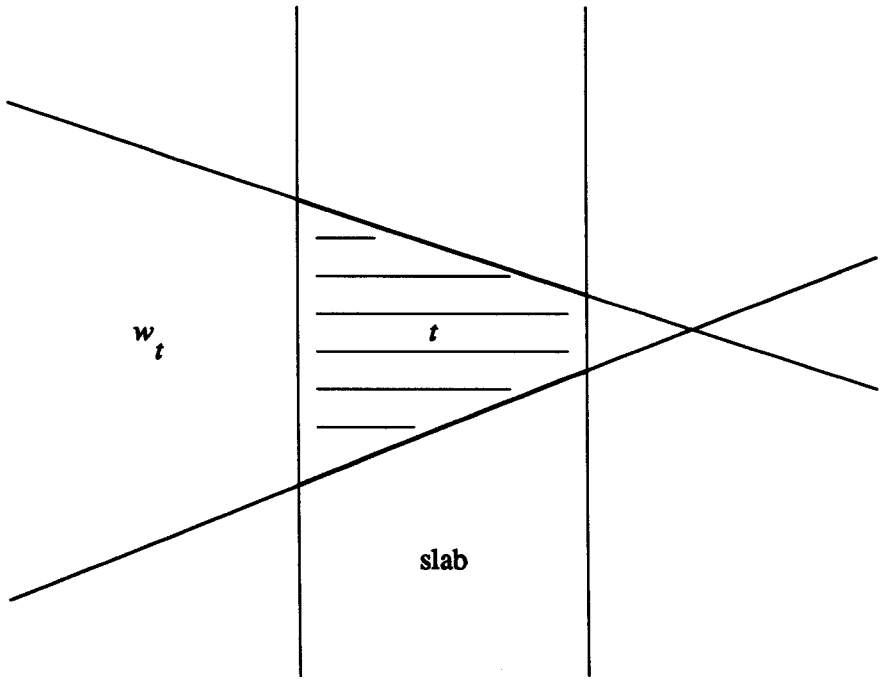

Figure 4: Extending triangles to wedges.

the primary segment tree and then by searching, for each node δ along the search path in the primary tree, with the dual line l_p in the corresponding partition tree to report or to count the dual line segments intersecting l_p. These line segments, collected over all nodes δ of the primary tree as above, correspond to the triangles containing p.

Theorem 5.1 *Given a set of n triangles in the plane, they can be stored in a data structure that uses $O(n \log^2 n)$ storage and can be constructed in $O(n \log^3 n)$ preprocessing time, such that the k triangles that contain a given query point p can be reported in time $O(k + n^\gamma \log n)$, or counted in time $O(n^\gamma \log n)$.*

Proof. Note that in a segment tree, each interval occurs at most twice in each level. Hence, all associated structures on one level of the primary segment tree contain $O(n)$ line segments altogether, and, hence, can be constructed in time $O(n \log^2 n)$ according to Theorem 4.4 or 4.5. So the total construction time is $O(n \log^3 n)$. The amount of storage required is similarly bounded.

To prove the bound on query time, note that the number of segments stored in an associated partition tree at level i of the primary segment tree is bounded by $O(n/2^i) = O(2^{\log n - i})$. Hence, the total reporting query time is bounded by

$$O(\sum_{j=1}^{\log n} (k_j + (2^j)^\gamma \log 2^j))$$

(where k_j is the number of triangles found at this node), which is bounded by $O(k + n^\gamma \log n)$. A similar argument applies to the counting version of the problem. □

6 Interval Partition Trees

In this section we describe a basically different way of storing line segments in partition trees, which we call an *interval partition tree* because it bears close resemblance to the ordinary interval tree for storing a set of segments on a line. A main advantage of the interval partition tree is that it uses only linear storage.

Again we construct a partition tree P on the set of endpoints of the line segments. Now for each line segment s we determine the lowest node δ in P such that s lies completely in the interior of the cell c_δ corresponding to δ. Let l_δ be the dividing line in δ. If s lies on l_δ we store it in an interval tree I_δ associated with δ. Otherwise we store at each son of δ the part of s that lies in the corresponding cell in some structure T depending on the type of query we want to answer (note that s will always intersect l_δ). Hence, each line segment is either stored in an interval tree I_δ or in two associated T structures. Defining $B_T(n)$, $M_T(n)$ as in Section 3, the following theorem is immediate:

Theorem 6.1 *An interval partition tree on n line segments uses $O(M_T(n))$ storage and can be constructed in time $O(B_T(n) + n \log n)$.*

Note that the line segments stored in a T_δ structure are all anchored at the dividing line $l_{father(\delta)}$ at the father of δ and that their other endpoint lies inside c_δ. See figure 5.

7 Ray Shooting

The ray shooting problem is the following: Given a set of n line segments in the plane, store them such that, for a given point p and a direction d, the first line segment hit when shooting from p in direction d can be determined efficiently. When the line segments form the edges of a simple polygon the shooting problem is known to be solvable with a query time of $O(\log n)$, using a structure that requires $O(n)$ storage and can be constructed in time $O(n\text{loglog}\, n)$ (see [5] and [11], using the triangulation method of [22]).

In this section we provide efficient techniques for solving more general instances of the ray shooting problem. We first consider the situation in which the set V consists of n non-intersecting line segments. We store these line segments in an interval partition tree. Now consider a node δ of the partition tree. All segments stored at δ are anchored at the dividing line $l_{father(\delta)}$ of the father of δ. Take the line $l_{father(\delta)}$ together with the parts of the line segments lying in the cell c_δ. Since the part of the complement of their union that lies on the c_δ–side of $l_{father(\delta)}$ is simply

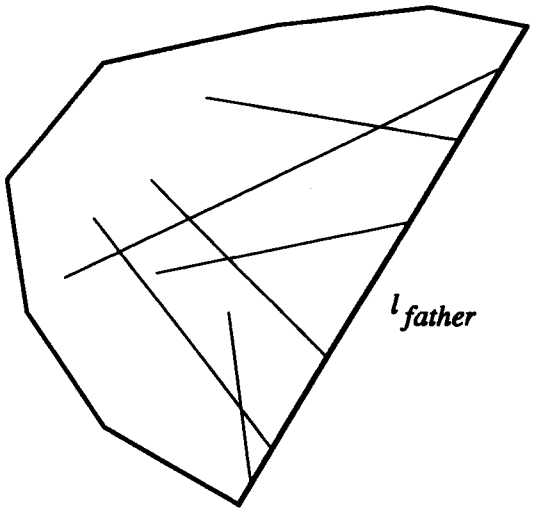

Figure 5: Line segments stored in T_δ.

connected (see figure 6), it is well known that we can apply to it the ray shooting techniques of [5, 11] as if it were a simple polygon. Let T_δ denote the resulting shooting structure. It immediately follows from Theorem 6.1 that the complete interval partition tree structure requires preprocessing time of $O(n \log n)$ and uses $O(n)$ storage.

Now assume we want to perform a ray shooting query from point p in direction d. Let ρ_d be the ray that runs from p in direction d. First determine all nodes δ in the partition tree such that ρ_d (partially) intersects c_δ. Because the underlying structure is a conjugation tree, the number of cells intersected by ρ_d is $O(n^\gamma)$ and they can be determined in $O(n^\gamma)$ time. For each of these nodes δ we perform a separate ray shooting query, as follows. If p lies inside c_δ we perform a shooting query from p in direction d in T_δ. Otherwise we determine the intersection of ρ_d with the boundary of c_δ which is nearest to p, and shoot from that intersection point in direction d into T_δ. In both cases we either hit a segment stored at T_δ, or we hit $l_{father(\delta)}$, or we don't hit anything. Only in the case we hit an actual line segment we consider this as a real hit. We also check whether ρ_d intersects l_δ and, if so, query I_δ with the intersection point to see whether any line segment on l_δ is hit (there can be at most one such segment). In this way we collect a total of at most $O(n^\gamma)$ candidate segments. The one among them nearest to p (in direction d) is the output to our shooting query.

The correctness of this method is easily established. For the time complexity note that we query at most $O(n^\gamma)$ T_δ and I_δ structures. Each of these queries costs $O(\log n)$ time. This leads to the following result.

Theorem 7.1 *Given a set of n non-intersecting line segments in the plane, they can be stored in a data structure that uses $O(n)$ storage and can be constructed in time*

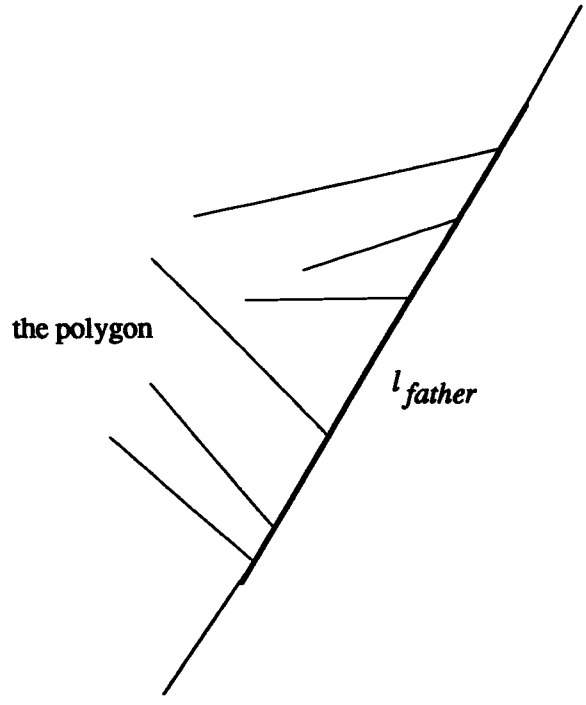

Figure 6: A polygon arising at a node δ.

$O(n \log n)$, such that ray shooting queries into the given segments can be answered in time $O(n^{\gamma} \log n)$.

Note that for the above method it is unneccessary to use the complicated triangulation technique of [22]. Any $O(n \log n)$ triangulation algorithm will do as well, without asymptotically increasing the time bounds. Also note that after the first query, any subsequent segment that ρ_d hits can be determined in time $O(\log n)$. To this end we sort all the candidate intersections computed by the preceding procedure, in increasing distance from p. When we want to find the next segment to be hit by ρ_d, we only have to perform a new shooting query in the cell in which the previous intersection point was found, thus requiring only $O(\log n)$ additional time. The new intersection (if it exists) is added to the sorted list of candidate intersections, and the next point in order is reported. Details are left to the reader.

Next consider the case where the given segments can intersect. This introduces several new technical difficulties, which can be solved at the expense of making our structures more complicated, which somewhat defeats our purpose of using as simple a technique as possible. Still we include a brief sketch of this solution for the sake of completeness, and also because it is somewhat different from other known techniques (such as that of [2]).

We use a segment partition tree, instead of an interval partition tree, to solve this more general problem. Observe first that shooting from outside a cell is relatively easy. Specifically, if ρ_d crosses a cell c_δ but p lies outside c_δ, we need to shoot, as

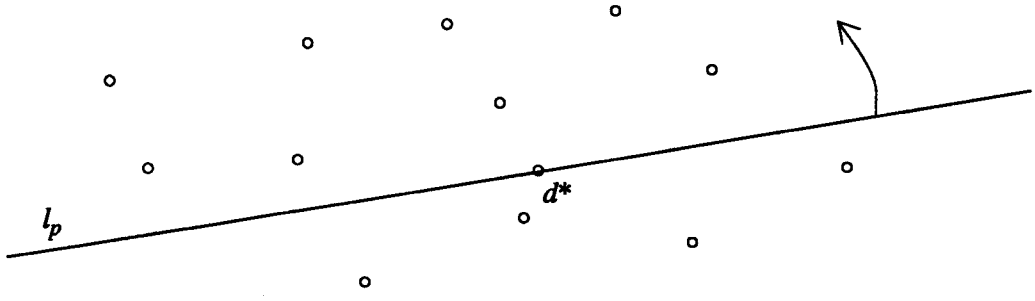

Figure 7: The dual version of ray shooting in an arrangement of lines

above, from the intersection, nearest to p, of ρ_d with the boundary of c_δ. To this end we compute the unbounded cell of the arrangement of the (portions within c_δ of the) segments stored at δ, and intersect it with c_δ. It is easily verified that this intersection consists of a linear number of simply connected regions, whose boundary consists of $O(m\alpha(m))$ subsegments (where $m = |T_\delta|$ and $\alpha(m)$ is the functional inverse of Ackermann's function), and that all these regions can be computed in time $O(m\alpha(m)\log^2 m)$ (see [14, 19, 24]). We next preprocess each of these regions for logarithmic–time ray shooting, as above, in total time $O(m\alpha(m)\log m)$. Now shooting into c_δ from the outside can be easily accomplished in $O(\log m)$ time by locating the appropriate region and shooting inside it.

Shooting from the inside of a cell is more difficult, however. Fortunately, there is a single such cell in each level of the partition tree, so we need to solve only $O(\log n)$ inside–shooting queries. Let c_δ be one of the cells containing p. Extend all segments stored at δ into full lines, and do the shooting in the arrangement of these lines. If the nearest hit lies inside c_δ, it coincides with the nearest hit of ρ_d with the unextended segments in T_δ; if it lies outside c_δ, we simply ignore it.

The problem thus reduces to that of ray–shooting from a point p in direction d in an arrangement of m lines. Dualize the lines to obtain a set S of m points. The query ray is transformed into the pair (l_p, d^*), where l_p is the line dual to p and d^* is the point on l_p dual to the line containing ρ_d; the query itself becomes: find the first point of S, if any, hit as we rotate l_p about d^* in counterclockwise direction until it becomes vertical; see figure 7.

We solve this problem as follows. Construct a (conjugation) partition tree P on the points of S. For each cell c_ζ of this auxiliary partition tree maintain the convex hull H_ζ of the points of S in c_ζ. Now, given the query (l_p, d^*), search with l_p in P and collect all cells that l_p misses (as in [10]) — there are $O(m^\gamma)$ such cells. For each of these cells c_ζ compute, in $O(\log m)$ time, the tangents from d^* to H_ζ. Collecting all such tangents, and choosing the one with the smallest slope that is larger than the slope of l_p, we obtain the (dual of the) solution to our shooting query. Thus the overall cost of this inside–shooting query is $O(m^\gamma \log m)$.

Putting everything together, we obtain the following result:

Theorem 7.2 *Given a set of n possibly intersecting line segments in the plane, they can be stored in a data structure that uses $O(n \log^2 n)$ storage and can be constructed in time $O(n\alpha(n) \log^3 n)$, such that ray shooting queries into the given segments can be answered in time $O(n^\gamma \log n)$.*

Proof. The storage and preprocessing time bounds follow from Theorem 3.1, noting that storing all convex hulls H_ζ, for a fixed node δ of the main partition tree, requires $O(m \log m)$ storage, where $m = |T_\delta|$, and this dominates the $O(m\alpha(m))$ storage requires to store the unbounded cell of the segments in T_δ; similarly, for preprocessing time, the time $O(m\alpha(m) \log^2 m)$ needed to compute the unbounded cell dominates the time needed to compute the convex hulls H_ζ.

The query time is analyzed as in the proofs of Theorems 4.6 and 5.1. Specifically, it suffices to bound the cost of the inside–shooting queries. Since we need to perform only one such query at each level of the main partition tree, it follows from Lemma 3.2 that the total cost of these queries is

$$O\Big(\sum_{j=1}^{O(\log n)} (n/2^j)^\gamma \log(n/2^j)\Big) = O(n^\gamma \log n)\,.$$

□

8 Conclusions and Open Problems

In this paper we have presented two data structures based on partition trees for storing line segments in the plane, which can be used for answering different types of queries such as segment intersection and ray shooting queries. Although we based our results on conjugation trees it is also possible to adapt our techniques and use, for example, the partition trees of Haussler and Welzl [15]. This would improve the query times for the different problems to $O(n^{2/3+\epsilon})$ for arbitrary small ϵ at the cost of having to use randomization (and increasing the complexity of the structure). Although the same or even improved query times can be obtained using the techniques of Dobkin and Edelsbrunner [7] and Agarwal [2] we believe that our methods have the desired simplicity to make them implementable and potentially practical.

We think that the two basic types of partition trees introduced can be used for solving other query problems involving line segments as well, using an appropriate type of associated T-structure. In an acompanying paper ([21]) we show that it is also possible to obtain dynamic versions of the structures presented here.

A number of open problems remain. It would be interesting to study whether the techniques presented can also be used with the most recent partition tree of Welzl [23] as an underlying structure. Alternatively, the ultimate goal is to design a partition tree structure that is induced by a convex decomposition of the plane and has the properties that (a) it can be constructed in close to linear time, and (b) only about $\sqrt{n}$ cells of the decomposition are crossed by any line. Our techniques should

be applicable to such a partition tree structure. Finally, concerning ray shooting, we note that no significant lower bound is known for that problem (even when we are allowed only roughly linear storage), so query time bounds like $O(n^\gamma \log n)$, or even close to $\sqrt{n}$, may still be far worse than optimal. In our approach, for instance, we can do the shootings into the structures T_δ in the order in which the cells c_δ are crosses by ρ_d, and stop as soon as a real hit is found. Can this be used to obtain a faster query time?

References

[1] Agarwal, P.K., An efficient deterministic algorithm for partitioning arrangements of lines and its applications, *Proc. 5th ACM Symp. on Computational Geometry,* 1989, pp. 11–22.

[2] Agarwal, P.K., Ray shooting and other applications of spanning trees with low stabbing number, *Proc. 5th ACM Symp. on Computational Geometry,* 1989, pp. 315–325.

[3] Chazelle, B., Polytope range searching and integral geometry, *Proc. 28th IEEE Symp. on Foundations of Computer Science,* 1987, pp. 1–10.

[4] Chazelle, B., A functional approach to data structures and its use in multidimensional searching, *SIAM J. Comput.* **17** (1988), pp. 427-462.

[5] Chazelle, B., and L. Guibas, Visibility and intersection problems in plane geometry, *Proc. 1st ACM Symp. on Computational Geometry,* 1985, pp. 135–146.

[6] Chazelle, B., and E. Welzl, Quasi optimal range searching in spaces with finite VC–dimension, *Discrete Comput. Geom.* **4** (1989), to appear.

[7] Dobkin, D., and H. Edelsbrunner, Space searching for intersecting objects, *J. Algorithms* **8** (1987), pp. 348–361.

[8] Edelsbrunner, H., *Algorithms in Combinatorial Geometry,* Springer-Verlag, New York, 1987.

[9] Edelsbrunner, H., L. Guibas, J. Hershberger, R. Seidel, M. Sharir, J. Snoeyink and E. Welzl, Implicitly representing arrangements of lines and of segments, *Discrete Comput. Geom.* **4** (1989), to appear.

[10] Edelsbrunner, H., and E. Welzl, Halfplanar range search in linear space and $O(n^{0.695})$ query time, *Inform. Proc. Lett.* **23** (1986), pp. 289–293.

[11] Guibas, L., J. Hershberger, D. Leven, M. Sharir and R. Tarjan, Linear time algorithms for visibility and shortest path problems in triangulated simple polygons, *Algorithmica* **2** (1987), pp. 209–233.