

Segment Intersection Searching Problems in General Settings*

Vladlen Koltun †

June 27, 2002

Abstract

We consider segment intersection searching amidst (possibly intersecting) algebraic arcs in the plane. We show how to preprocess n arcs in time $O(n^{2+\varepsilon})$ into a data structure of size $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$, such that the k arcs intersecting a query segment can be counted in time $O(\log n)$ or reported in time $O(\log n + k)$. This problem was extensively studied in restricted settings (e.g., amidst segments, circles or circular arcs), but no solution with comparable performance was previously presented for the general case of possibly intersecting algebraic arcs. Our data structure for the general case matches or improves (sometimes by an order of magnitude) the size of the best previously presented solutions for the special cases.

As an immediate application of this result, we obtain an efficient data structure for the triangular windowing problem, which is a generalization of triangular range searching. As another application, the first substantially sub-quadratic algorithm for a red-blue intersection counting problem is derived. We also describe simple data structures for segment intersection searching among disjoint arcs, and for ray shooting among possibly intersecting arcs.

1 Introduction

Intersection searching problems typically take the following form. Given a collection Γ of data objects in \mathbb{R}^d , we wish to preprocess Γ such that for a query object γ , we can efficiently report or count the subset $\Delta \subseteq \Gamma$ of objects intersecting γ .

The above definition of intersection searching is quite general, and some of the most widely studied problems in computational geometry can be formulated in this setting. One example is the range searching problem [3], where Γ is constrained to consist of points. At the other extreme end there is a variant of the point location problem [22], in which γ is a point, Γ is composed of closed objects, and we wish to efficiently report or count the number of objects of Γ that contain γ . Besides range searching and point location, other intersection searching problems have received considerable attention over the years. In these problems both the query and the data objects are more complex than points.

Segment intersection searching problems arise when our query objects are constrained to be segments. One of the simplest and the most extensively studied instances of segment

*Work on this paper has been supported by a grant from the Israel Science Fund (for a Center of Excellence in Geometric Computing). A preliminary version of the paper has appeared in the Proceedings of the 17th ACM Symposium on Computational Geometry, 2001.

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. vladlen@tau.ac.il

intersection searching is when the data objects are segments in the plane. In this setting, Agarwal and Sharir [6] have obtained query time $O(n^{1+\varepsilon}/\sqrt{m})$, after $O(m^{1+\varepsilon})$ preprocessing¹, for $n \leq m \leq n^2$. Segment intersection searching problems among lines, simple polygons, and other types of data objects have also been considered. For more details, the reader is referred to Agarwal and Erickson [3].

The most general planar segment intersection searching data structures we are aware of were presented by Agarwal, van Kreveld and Overmars [8]. For segment intersection searching among circles, a data structure of size $O(n^{2+\varepsilon})$ with logarithmic query time was presented. For segment intersection searching among circular arcs, a near-linear data structure with query time $O(n^{2/3} + k)$ was presented, where k is the size of the output. A space-time trade-off based on the above results is reported in [3]. Specifically, a data structure of size m , for $n \leq m \leq n^3$, is reported, that can count the intersections between a query segment and n circular data arcs in time $O(\frac{n}{m^{1/3}} \text{polylog}(n))$. If we are interested in obtaining polylogarithmic query time, the size of this trade-off data structure becomes near-cubic.

In this paper, we show how to preprocess n possibly intersecting algebraic arcs, defined by polynomials of constant maximum degree, in time $O(n^{2+\varepsilon})$ into a data structure of size $O(n^{2+\varepsilon})$, such that the k arcs intersecting a query segment can be counted in time $O(\log n)$ or reported in time $O(\log n + k)$. The segment intersection searching problem in this general setting has been recognized as an interesting open problem for some time now [2]. Besides being of interest in its own right, it sometimes arises as a sub-problem in other applications (see Section 7). Nevertheless, to our knowledge, no solutions with comparable performance were previously presented. Our data structure for the general setting matches, and in some cases improves by an order of magnitude, the best previously presented solutions for the special cases of the problem. For the special case of circular arcs, our result leads to a significantly improved space-time trade-off.

Ray shooting. We also consider ray shooting amidst (possibly intersecting) algebraic arcs in the plane. Ray shooting has received considerable attention in computational geometry primarily due to its connection to ray-tracing in computer graphics. Due to the astounding body of works on this problem, we only mention the most relevant results, and the reader is referred to Agarwal and Erickson [3] and Pellegrini [19] for more information. A general approach to ray shooting problems, which reduces the problem to *segment-emptiness*, was presented by Agarwal and Matoušek [4], and a number of subsequent results were obtained using this approach.

A data structure of size $O(n^2)$ with query time $O(\log n)$ for ray shooting among disjoint convex sets was presented by Pocchiola and Vegter [20]. For ray shooting among segments, the best known data structure having logarithmic query time is due to Agarwal [1]; its size is $O(n^2 \alpha^2(n))$. Agarwal, van Kreveld and Overmars [8] have obtained a near-linear data structure for performing ray shooting among disjoint Jordan arcs in time $O(\sqrt{n} \log n)$; they have also presented a data structure of near-linear size that handles ray shooting queries among circular arcs in time $O(n^{2/3+\varepsilon})$. Ray shooting with circular arcs (inside simple polygons in the plane) has also been studied [7, 11].

In this paper, we describe a data structure of size $O(\lambda_{s+2}^2(n))$ that can answer a ray shooting query among n possibly intersecting algebraic arcs in time $O(\log n)$ (where $\lambda_q(n)$ is the maximal length of Davenport-Schinzel sequences of order q on n symbols, and s is the maximum degree of the polynomials that define the data arcs). Furthermore, we give a significantly improved

¹Throughout this paper, a complexity bound of the form $f(n) = O(n^{q+\varepsilon})$ means that, for an arbitrarily small positive ε , there exists a constant c_ε , such that $f(n) \leq c_\varepsilon n^{q+\varepsilon}$.

space-time trade-off for ray shooting among circular arcs.

Disjoint arcs. We also consider segment intersection searching and ray shooting in the restricted setting of disjoint algebraic arcs. A data structure of size $O(n^2)$ is described that allows reporting the k arcs intersected by a query segment in $O(\log n + k)$ time. A data structure of size $O(n^2 \log n)$ allows counting the intersected arcs in $O(\log n)$ time.

Triangular windowing. As an application of our main segment intersection searching result, we derive an efficient data structure that handles triangular windowing, which is the following instance of intersection searching. Given a set Γ of algebraic arcs, preprocess Γ into a data structure such that for a query triangle φ , we can efficiently report the arcs intersecting the closure of the interior of φ . The triangular windowing problem is a generalization of triangular range searching that allows the data objects to be general arcs, not only points.

The triangular windowing problem is motivated by its connection to problems arising in industry and in other disciplines, e.g. clipping in computer graphics. We are not aware of any previously presented solutions to this problem in general, although a simpler problem, rectangular windowing, has been studied extensively. Rectangular windowing is essentially a special case of triangular windowing, since we can represent an axis-parallel rectangle as a union of two triangles. In general, our data structure for triangular windowing can be easily extended to support *clipping* a set of arcs to a polygonal query window of arbitrary shape and topology.

Bichromatic intersection counting. A widely studied problem in computational geometry is counting (or reporting) all pairwise intersections in a collection of input objects. When designing divide-and-conquer algorithms for such intersection counting (reporting) problems, the problem of *red-blue* intersection counting (reporting) arises as a natural sub-problem. In red-blue intersection counting (reporting), we are given a set R of red objects and a set B of blue objects, and we wish to count (respectively, report) all the intersections between the red and the blue objects, disregarding red-red and blue-blue intersections.

Unlike red-blue intersection reporting, where a number of near-optimal solutions have recently been proposed for a fairly general instance of the problem (in which the union of the red objects and the union of the blue objects are both connected; see [16] and the references therein), red-blue intersection counting is far from being sufficiently understood. While solutions have been devised for cases where R and B are composed of segments [1], or of circles or circular arcs [5], no significantly sub-quadratic algorithm was previously given for a setting in which one (let alone both) of the sets consists of general arcs. We show in Section 7.2 that such an algorithm can be designed using our main segment intersection searching result.

Organization. In Section 2 we outline and demonstrate the general idea of our solution, and introduce the necessary geometric transforms. In Section 3 we obtain the main result of the paper by attacking the general segment intersection searching problem. A space-time trade-off is then presented in Section 4. The special case of disjoint arcs is considered in Section 5, and ray shooting among intersecting arcs is treated in Section 6. Finally, Section 7 discusses two applications of the main result.

2 Preliminaries

2.1 Overview

The basic idea behind the presented algorithms is to re-parametrize the problem domain, transforming relatively complex queries in the plane into well-understood queries such as point location in space.

Consider the problem of ray shooting. We are given a collection $\Gamma : \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ of n possibly intersecting algebraic arcs, defined by polynomials of constant maximum degree, and we wish to preprocess this collection, so that we can efficiently report the first arc γ_i hit by a query ray r . The idea is to transform Γ into a collection $\tilde{\Gamma} : \{\tilde{\gamma}_1, \tilde{\gamma}_2, \dots, \tilde{\gamma}_n\}$ of n surfaces² in \mathbb{R}^3 , such that a surface $\tilde{\gamma}_j \in \tilde{\Gamma}$, for $1 \leq j \leq n$, corresponds to the arc $\gamma_j \in \Gamma$. Moreover, there is a one-to-one correspondence between every ray r in \mathbb{R}^2 and a point p in \mathbb{R}^3 , such that the first arc γ_i hit by r corresponds to the first surface $\tilde{\gamma}_i$ hit by a downward ray emanating from p (in other words, the surface that lies immediately below p).

Thus, ray shooting queries in the plane are essentially mapped to point location queries in 3-space.³ Similarly, segment intersection searching queries in the plane are mapped to *vertical* segment intersection searching queries in 3-space. We show that point location and vertical segment intersection searching in the special class of 3-dimensional arrangements that we consider can be performed in time $O(\log n)$ using a data structure of size $O(n^{2+\epsilon})$. This compares favorably with point location among general surfaces in 3-space, which requires cubic or super-cubic storage in the worst case.

2.2 The Transform

We first introduce some notation. A segment s having end-points p and q is denoted by $s : (p, q)$. A ray r emanating from a point p is denoted by $r : (p, \alpha)$, where $-\pi < \alpha \leq \pi$ is its angle with the *negative* direction of the y -axis. For example, a ray r emanating from the origin O in the positive direction of the x -axis is denoted by $r : (0, \frac{\pi}{2})$. The arrangement [21] of a collection Γ of geometric objects is denoted by $\mathcal{A}(\Gamma)$. The vertical decomposition [9, 18] of $\mathcal{A}(\Gamma)$ is denoted by $\mathcal{V}(\Gamma)$. $\mathcal{V}(\Gamma)$ is a decomposition of $\mathcal{A}(\Gamma)$ into *elementary cells* of constant description complexity (that is, defined by a constant number of polynomial equations and inequalities of constant maximum degree), each of which is contained in a cell of $\mathcal{A}(\Gamma)$. Abusing the notation slightly, we will also refer to the set of elementary cells of $\mathcal{V}(\Gamma)$ by $\mathcal{V}(\Gamma)$. Thus, $v \in \mathcal{V}(\Gamma)$ denotes an elementary cell.

We proceed to define a geometric transform \mathcal{T} that maps points in \mathbb{R}^2 to 1-dimensional curves in \mathbb{R}^3 , and arcs in \mathbb{R}^2 to 2-dimensional surfaces in \mathbb{R}^3 . To facilitate understanding, we first provide an intuitive description of the transform, and only then give the formal definition. The motivation behind the transform, and its properties, are discussed afterwards.

Let P be the vertical plane $y = 0$ in 3-space. The general idea behind the transform is to “lift” all features in P to \mathbb{R}^3 by continuously moving P in a direction orthogonal to it, while rotating it around its origin. Imagine continuously moving P in the positive direction of the y -axis. The trajectory traced by a specific point $(x, z) \in P$ during this movement is a y -parallel ray, originating at $(x, 0, z)$. Similarly, a circle in P traces a (bottomless) semi-infinite cylinder.

²We use the term ‘surface’ in a fairly loose sense, referring to surfaces that are not necessarily totally defined.

³Point location in this case means computing the surface that lies immediately below the query point.

Now, let us add an extra “twist” by rotating P around its origin while moving it. Suppose the rotation is performed at a constant speed. The trajectory traced by a point in P is now a helix. Its radius is the distance of the point from the origin (in P), and its central axis is the y -axis. Similarly, a curve in P is translated to a helix-like (or spiral-like) surface in \mathbb{R}^3 that is the collection of helices that correspond to the points on the curve.

The definition that follows is a formalization of the above. It is slightly complicated by the fact that we would like \mathcal{T} to transform semi-algebraic sets (in \mathbb{R}^2) to semi-algebraic sets (in \mathbb{R}^3). The above-described “lifting”, where the rotation speed is constant, does not meet this demand; points are transformed to helices, which are known not to have any description as semi-algebraic sets. Therefore, we resort to the trick of expressing $\sin \theta$ and $\cos \theta$ in terms of $\tan \frac{\theta}{2}$. (In the context of the above description, this means that the rotation speed is a trigonometric function.) Also, the following transform is defined for all values of y , not only for $y \geq 0$ as above.

Definition 2.1. (a) Let p be a point $(x, y) \in \mathbb{R}^2$. p is mapped to the following 1-dimensional curve $\mathcal{T}(p)(\theta)$ in \mathbb{R}^3 :

$$\mathcal{T}(p)(\theta) : (x \cos \theta + y \sin \theta, \tan \frac{\theta}{2}, -x \sin \theta + y \cos \theta), \theta \in (-\pi, \pi) \quad (1)$$

(b) Let $\gamma(t)$ be an arc $(f_x(t), f_y(t)), t \in [0, 1]$, in \mathbb{R}^2 . $\gamma(t)$ is mapped to the following 2-dimensional surface $\mathcal{T}(\gamma)(t, \theta)$ in \mathbb{R}^3 :

$$\mathcal{T}(\gamma)(t, \theta) : (f_x(t) \cos \theta + f_y(t) \sin \theta, \tan \frac{\theta}{2}, -f_x(t) \sin \theta + f_y(t) \cos \theta), t \in [0, 1], \theta \in (-\pi, \pi) \quad (2)$$

(c) Let $\Gamma : \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ be a set of arcs in \mathbb{R}^2 . Define $\mathcal{T}(\Gamma)$ to be the following set of surfaces in \mathbb{R}^3 :

$$\mathcal{T}(\Gamma) : \{\mathcal{T}(\gamma_1), \mathcal{T}(\gamma_2), \dots, \mathcal{T}(\gamma_n)\} \quad (3)$$

We will use $\tilde{\Gamma}$ to denote $\mathcal{T}(\Gamma)$, and $\tilde{\gamma}$ to denote $\mathcal{T}(\gamma)$, for $\gamma \in \Gamma$.

A very useful property of the transform is that, assuming γ is a semi-algebraic set in \mathbb{R}^2 , $\tilde{\gamma}$ is also a semi-algebraic set. $\tilde{\gamma}$ can be described in terms of polynomial equalities and inequalities by, for example, setting $a = \tan \frac{\theta}{2}$, and using the standard formulae $\sin \theta = 2a/(1 + a^2)$ and $\cos \theta = (1 - a^2)/(1 + a^2)$. Assuming γ is an algebraic arc as above, $\tilde{\gamma}$ can be partitioned into a constant number of xy -monotone surface patches, and standard primitive operations, such as determining whether a point lies above/below it, can be performed on $\tilde{\gamma}$ in constant time.

To gain some more intuition about the transform and the motivation behind its introduction, notice that *slicing* $\mathcal{A}(\tilde{\Gamma})$ with the xz -parallel plane $y = 0$ produces the arrangement $\mathcal{A}(\Gamma)$. Moreover, slicing $\mathcal{A}(\tilde{\Gamma})$ with the xz -parallel plane $y = \tan \frac{\beta}{2}$, for $-\pi < \beta < \pi$, produces a 2-dimensional arrangement that is identical to $\mathcal{A}(\Gamma)$ rotated around the origin at angle β .

Consider a ray $r : (p, \alpha)$. The intersection of $\mathcal{T}(r)$ and the plane $y = \tan \frac{\alpha}{2}$ is a downward-oriented ray emanating from the point $\mathcal{T}(p)(\alpha)$. This is easily seen to imply that the first arc in Γ hit by r corresponds to the surface of $\tilde{\Gamma}$ that lies immediately below $\mathcal{T}(p)(\alpha)$. We now formulate a somewhat more general lemma, the correctness of which is immediate from the definition of the transform.

Lemma 2.1. Let $s : (p, q)$ be a segment lying on an oriented line that defines an angle $-\pi < \alpha < \pi$ with the negative direction of the y -axis. Let $\Delta \subseteq \Gamma$ be the set of arcs intersected by s . Let $\Delta^* \subseteq \tilde{\Gamma}$ be the set of surfaces intersected by the z -vertical segment $s' : (\mathcal{T}(p)(\alpha), \mathcal{T}(q)(\alpha))$. Then $\Delta^* \equiv \tilde{\Delta}$.

Remark. Since every segment s lies on two oriented lines, there are two segments s' as above that correspond to s .

Therefore, the problem of finding the arcs in Γ intersected by a query segment can be reduced to finding the surfaces in $\tilde{\Gamma}$ intersected by a corresponding z -vertical segment. The following lemma will be instrumental in the efficient construction of search structures on $\tilde{\Gamma}$.

Lemma 2.2. *Let $\Gamma : \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ be a collection of algebraic arcs, defined by polynomials of maximum degree s . The complexity of the vertical decomposition $\mathcal{V}(\tilde{\Gamma})$ of the three-dimensional arrangement $\mathcal{A}(\tilde{\Gamma})$ is $O(\lambda_{s+2}^2(n))$.*

Proof. It is sufficient to bound the number of *vertical visibility events* in $\mathcal{A}(\tilde{\Gamma})$ [21, Section 8.3]. Each such event corresponds to a vertical segment that is disjoint from all surfaces of $\tilde{\Gamma}$, such that both its end-points lie either on a 1-dimensional edge of $\mathcal{A}(\tilde{\Gamma})$ or a vertical tangency point on one of the surfaces of $\tilde{\Gamma}$. We can thus distinguish between edge-edge, edge-silhouette and silhouette-silhouette vertical visibility events (where *silhouette* refers to the locus of vertical tangency points on a surface). We bound the complexity of each class of events separately.

The idea is to relate vertical visibility events in $\mathcal{A}(\tilde{\Gamma})$ to certain features of $\mathcal{A}(\Gamma)$, using **(a)** the fact that 1-dimensional edges of $\mathcal{A}(\tilde{\Gamma})$ correspond to vertices of $\mathcal{A}(\Gamma)$, **(b)** the fact that each vertical segment s' in \mathbb{R}^3 corresponds to a specific segment s in \mathbb{R}^2 , in the sense that $\mathcal{T}(s) = s'$, and **(c)** the fact that for any segment s in \mathbb{R}^2 , there are at most two such vertical segments s' (as stated in the above remark). The following analysis is illustrated in Figure 1.

- **Silhouette-silhouette events.** Each such event corresponds to a *free bitangent* — a segment that is internally disjoint from Γ and is tangent to two arcs of Γ at its end-points. It is therefore sufficient to bound the number of free bitangents in Γ . Since each arc in Γ is defined by a polynomial of constant maximum degree, the number of lines that are tangent to a specific pair of arcs is bounded by a constant. Therefore, the number of free bitangents in Γ is $O(n^2)$. This implies that the number of silhouette-silhouette events in $\mathcal{A}(\tilde{\Gamma})$ is also $O(n^2)$.
- **Edge-edge events.** Each such event corresponds to a *connecting segment* — a segment that is internally disjoint from Γ and is connecting two vertices of $\mathcal{A}(\Gamma)$. A simple but crucial observation is that both vertices have to lie in the same 2-dimensional face of $\mathcal{A}(\Gamma)$. Therefore, the number of connecting segments is bounded by the number of pairs of vertices of $\mathcal{A}(\Gamma)$ that lie in the same face. This quantity is bounded by the sum-of-squares of cell complexities of $\mathcal{A}(\Gamma)$, which is $O(\lambda_{s+2}^2(n))$ [21, Theorem 5.13]. Thus, the number of edge-edge events in $\mathcal{A}(\tilde{\Gamma})$ is also $O(\lambda_{s+2}^2(n))$.
- **Edge-silhouette events.** Analogously to the previous two cases, the number of edge-silhouette events can be bounded by the number of *connecting tangents*. A connecting tangent is a segment that is internally disjoint from Γ , has one end-point lying on a vertex of $\mathcal{A}(\Gamma)$, and is tangent to an arc of Γ at its other end-point. The number of connecting tangents in $\mathcal{A}(\Gamma)$ is (asymptotically) bounded by the number of (v, e) pairs, such that v and e are a vertex and an edge of $\mathcal{A}(\Gamma)$, respectively, both belonging to the same face of $\mathcal{A}(\Gamma)$. This, in turn, is bounded by the sum-of-squares of cell complexities of $\mathcal{A}(\Gamma)$, which is $O(\lambda_{s+2}^2(n))$.

□

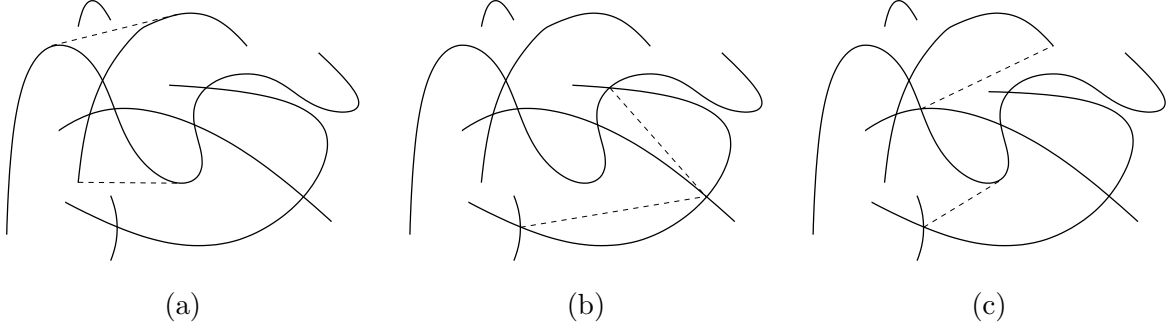


Figure 1: Events in $\mathcal{A}(\Gamma)$ that are transformed to vertical visibility events in $\mathcal{A}(\tilde{\Gamma})$. (a) shows two free bitangents in $\mathcal{A}(\Gamma)$ (dashed), corresponding to silhouette-silhouette events in $\mathcal{A}(\tilde{\Gamma})$; (b) shows two connecting segments in $\mathcal{A}(\Gamma)$ (dashed), corresponding to edge-edge events in $\mathcal{A}(\tilde{\Gamma})$; (c) shows two connecting tangents in $\mathcal{A}(\Gamma)$ (dashed), corresponding to edge-silhouette events in $\mathcal{A}(\tilde{\Gamma})$. Notice that bitangents and connecting tangents can be tangent to an arc at an end-point.

2.3 Point Location Among Surfaces

A central role in our construction is played by the point location structure of Chazelle et al. [9]. We provide a brief overview of a variant of the data structure that is most relevant to the application we have in mind. For more information, the reader is referred to [21, Section 8.3] (see also Clarkson [12] for an earlier application of top-down sampling to the problem of point location). Given a collection \mathcal{F} of n algebraic surfaces (of constant maximum degree), denote the vertical decomposition of $\mathcal{A}(\mathcal{F})$ by $\mathcal{V}(\mathcal{F})$, and its complexity by $|\mathcal{V}(\mathcal{F})|$. Denote the described point location structure on \mathcal{F} by $\mathcal{Q}(\mathcal{F})$, and its size by $|\mathcal{Q}(\mathcal{F})|$.

$\mathcal{Q}(\mathcal{F})$ is a tree structure. Its height and branching factor are determined by a parameter r . If $r = n^\varepsilon$, for some $\varepsilon > 0$, the height of the tree is a constant that depends on ε , and the degree of each node is $n^{O(\varepsilon)}$. The size of $\mathcal{Q}(\mathcal{F})$ is determined by the number of elementary cells in $\mathcal{V}(\mathcal{F})$. Specifically, if $|\mathcal{V}(\mathcal{F})| = O(g(n))$ then $\mathcal{Q}(\mathcal{F})$ has size $O(g(n)n^{\varepsilon'})$ and can be constructed in time $O(g(n)n^{\varepsilon'})$, where ε' depends on ε and can be made arbitrarily small by bringing ε sufficiently close to 0. Each node v of $\mathcal{Q}(\mathcal{F})$ is associated with a set of surfaces, denoted \mathcal{F}_v . For the root u of $\mathcal{Q}(\mathcal{F})$, $\mathcal{F}_u \equiv \mathcal{F}$.

The data structure is constructed in a recursive fashion, using the notion of $1/r$ -nets [17]. If $|\mathcal{F}| < r$, $\mathcal{V}(\mathcal{F})$ is computed, and its elementary cells become children of the root u and leaves of $\mathcal{Q}(\mathcal{F})$; in this case the tree has depth 1. Otherwise, the construction of $\mathcal{Q}(\mathcal{F})$ starts by computing a $1/r$ -net R_u of \mathcal{F} of size $O(r \log r)$. Let t_1, \dots, t_m be the elementary cells of $\mathcal{V}(R_u)$. For $1 \leq i \leq m$, t_i is intersected by at most n/r surfaces of \mathcal{F} . For $1 \leq i \leq m$, a child node v_i of u is created, and \mathcal{F}_{v_i} is set to be the subset of \mathcal{F} composed of the surfaces that intersect t_i . The elementary cell t_i is also associated with v_i . The construction then proceeds recursively in v_i with the set of surfaces \mathcal{F}_{v_i} .

An important property of $\mathcal{Q}(\mathcal{F})$ that is immediate from the construction is the following.

Lemma 2.3. *Let w_1, \dots, w_k be a sequence of nodes that make up a path from the root of $\mathcal{Q}(\mathcal{F})$ to one of the leaves. Then $\mathcal{F} \equiv \mathcal{F}_{w_1} \supset \mathcal{F}_{w_2} \supset \dots \supset \mathcal{F}_{w_k} \equiv \phi$.*

A query in $\mathcal{Q}(\mathcal{F})$ is performed by traversing a path from the root to a leaf, through a sequence of $O(1)$ nodes, such that each of the cells that correspond to these nodes contains the query

point. (Notice that in the sequence of these cells, a cell does not necessarily lie entirely inside its predecessor.) During the construction of $\mathcal{Q}(\mathcal{F})$, a point location structure on R_v is stored at each node v of $\mathcal{Q}(\mathcal{F})$. When in node v , the query algorithm uses this structure to determine the next node it should proceed to. Overall, the query algorithm visits $O(1)$ nodes, spending $O(\log n)$ time in each. The query time is thus $O(\log n)$.

A crucial property of $\mathcal{Q}(\mathcal{F})$ is that certain auxiliary data structures can be stored in each node without increasing the asymptotic storage space or the construction time of the data structure; moreover, the point location algorithm can query these second-level structures without increasing the query time. This is easily proved using the two Recurrence Lemmas [13, Lemmas 2.8 and 2.9].

Lemma 2.4. *Storing $O(|\mathcal{Q}(\mathcal{F}_v)|)$ additional information, in $O(|\mathcal{Q}(\mathcal{F}_v)|)$ time, in each node v of $\mathcal{Q}(\mathcal{F})$, does not increase the asymptotic storage size or the construction time of $\mathcal{Q}(\mathcal{F})$. Moreover, spending $O(\log n)$ additional time in each node visited by the query algorithm does not increase the query time.*

In particular, Lemma 2.4 implies that if $|\mathcal{Q}(\mathcal{F})| = O(n^{2+\epsilon})$, we can store up to a constant number of additional data structures of near-quadratic size (and logarithmic query time) in each node of $\mathcal{Q}(\mathcal{F})$, without affecting the storage and preprocessing bounds. Moreover, during the point location query, we can query these data structures in each visited node, without affecting the overall $O(\log n)$ query time. This is the form of Lemma 2.4 that will be used throughout this paper.

2.4 Demonstration: Ray Shooting

We now have the necessary machinery to exemplify our approach by designing a simple data structure that handles ray shooting queries. (A more efficient solution will be presented in Section 6.) Let Γ be as in Lemma 2.2. Lemma 2.1 implies that we can answer a ray shooting query in Γ by identifying the surface of $\tilde{\Gamma}$ that lies immediately below the point $\mathcal{T}(p)(\alpha)$, where $r : (p, \alpha)$ is the query ray (see also the discussion preceding Lemma 2.1 in Section 2.2).

We preprocess $\tilde{\Gamma}$ by constructing the point location structure $\mathcal{Q}(\tilde{\Gamma})$. Lemma 2.2 implies that this can be done in time $O(n^{2+\epsilon})$. The query algorithm queries $\mathcal{Q}(\tilde{\Gamma})$ with the point $\mathcal{T}(p)(\alpha)$. Consider the sequence of cells that corresponds to the sequence of nodes of $\mathcal{Q}(\tilde{\Gamma})$ traversed by this query. If all these cells are unbounded from below, the algorithm reports that the ray does not hit any arc. Otherwise, the algorithm computes the intersection point of the downward ray emanating from $\mathcal{T}(p)(\alpha)$ with (the floor of) each of the cells in the sequence, and selects the highest of those points. The answer to the ray shooting query is the arc in Γ that corresponds to the surface in $\tilde{\Gamma}$ that contains this highest point.

We have thus shown how to preprocess Γ into a data structure of size $O(n^{2+\epsilon})$, that can report the first arc in Γ hit by a query ray in time $O(\log n)$.

Remark. The point $\mathcal{T}(p)(\alpha)$, for any $p \in \mathbb{R}^2$, is undefined for $\alpha = \pm\pi$. Consequently, the ray shooting algorithm doesn't handle upward-oriented rays. Such (degenerate) rays can easily be handled either by perturbation or by preprocessing Γ into a simple upward ray shooting structure.

Note that this ray shooting data structure can easily be used to report the k arcs intersecting a query segment in time $O(k \log n)$, simply by shooting successive rays along the query segment,

starting from one of its end-points. In the next section, we are going to utilize more advanced machinery that will allow us not only to perform segment intersection reporting in reduced time $O(\log n + k)$, but also to count the number of arcs intersecting a query segment in time $O(\log n)$.

3 Intersection Searching

We turn to deriving the main result of this paper — an efficient algorithm for segment intersection searching among a collection Γ of possibly intersecting algebraic arcs as above.

According to Lemma 2.1, it is sufficient to design a data structure that handles intersection searching with vertical segments, among $\tilde{\Gamma}$. To this end, we build a three-level data structure based on $\mathcal{Q}(\tilde{\Gamma})$. For a query vertical segment s' , one of the levels selects the (parts of) surfaces of $\tilde{\Gamma}$ that lie below the top end-point of s' . Another level selects among those (parts of) surfaces the ones that lie above the bottom end-point of s' . In this fashion, the surfaces of $\tilde{\Gamma}$ intersecting s' are selected in a small number of canonical subsets, allowing them to be counted or reported efficiently.

Let us introduce some notation that will be used throughout this section. Let v be a node of $\mathcal{Q}(\tilde{\Gamma})$. Define the *bottom shaft* of v to be the locus of points in \mathbb{R}^3 that are strictly below the elementary cell associated with v . In other words, an upward ray emanating from any point inside the bottom shaft intersects the elementary cell. Define the *top shaft* of v symmetrically. The bottom and top shafts of v are pairwise disjoint, are disjoint from v , and the union of the shafts with v forms an infinite vertical prism.

For a surface $\tilde{\gamma}$, $\tilde{\gamma}^{vb}$ denotes the portion of $\tilde{\gamma}$ that lies inside the bottom shaft of v , and $\tilde{\gamma}^{vt}$ denotes the portion of $\tilde{\gamma}$ that lies inside the top shaft of v . (Note that $\tilde{\gamma}^{vb}$ and $\tilde{\gamma}^{vt}$ may have more connected components than $\tilde{\gamma}$.) Let $\{\tilde{\gamma}_1, \dots, \tilde{\gamma}_m\}$ be a set of surfaces, such that each $\tilde{\gamma}_i$ is associated with the *parent* of v , but is not associated with v . (If v is the root node, this set is empty.) Define $\tilde{\Gamma}_v^b \equiv \{\tilde{\gamma}_1^{vb}, \dots, \tilde{\gamma}_m^{vb}\}$ and $\tilde{\Gamma}_v^t \equiv \{\tilde{\gamma}_1^{vt}, \dots, \tilde{\gamma}_m^{vt}\}$.

Before we proceed, we need to state the performance of a simple data structure that will prove useful in the sequel. It is easily constructed by preprocessing the arrangement of the xy -projections of the surfaces, similarly to the data structure of Theorem 5.1 (Section 5).

Lemma 3.1. *For a set of n possibly intersecting semi-algebraic sets of constant description complexity in \mathbb{R}^3 , a data structure of size $O(n^2)$, that allows reporting the k surfaces intersecting a query vertical line in time $O(\log n + k)$ or counting them in time $O(\log n)$, can be constructed in time $O(n^2 \log n)$.*

3.1 Ray Intersection Searching

We now describe a data structure that allows us to efficiently report or count the surfaces of $\tilde{\Gamma}$ that are intersected by an upward vertical query ray in \mathbb{R}^3 . This data structure will subsequently be employed as a substructure.

The following notation is used. For a point p , l_p denotes the vertical line containing p and r_p denotes the upward vertical ray emanating from p . Let v be a node in $\mathcal{Q}(\tilde{\Gamma})$. The vertical decomposition of the $1/r$ -net stored in v is denoted by \mathcal{V}_v . With a slight abuse of notation, we denote a node of $\mathcal{Q}(\tilde{\Gamma})$ and the elementary cell associated with it identically. Thus, $u \in \mathcal{V}_v$ denotes a cell of \mathcal{V}_v , as well as a child node of v . (Recall that each elementary cell of \mathcal{V}_v corresponds to a child node of v .)

The described data structure, denoted by \mathcal{Q}' , is obtained by augmenting $\mathcal{Q}(\tilde{\Gamma})$ as follows. For each node v of $\mathcal{Q}(\tilde{\Gamma})$, we preprocess $\tilde{\Gamma}_v^t$ for vertical-line intersection searching, as described in Lemma 3.1, and store the resulting structure (denoted by L_v) at v . Lemmas 2.4 and 3.1 imply that the size and construction time of \mathcal{Q}' are identical to the size and construction time of $\mathcal{Q}(\tilde{\Gamma})$, respectively.

The counting query algorithm on \mathcal{Q}' is given in pseudo-code below. It receives the origin point p of the upward vertical query ray r_p , and counts the number of surfaces of $\tilde{\Gamma}$ intersecting r_p , by querying the second-level line intersection searching structures in each of the nodes of \mathcal{Q}' that contain p (**QueryLine**(v, l) stands for “Query L_v with l ”). In fact, the path traced by the algorithm (in the top-level data structure) is exactly the path traversed by the standard point location query algorithm of $\mathcal{Q}(\tilde{\Gamma})$. The algorithm can be easily modified for the purpose of intersection reporting.

Function **QueryRay**(\mathcal{Q}', p)

1. set *Counter* to 0.
2. return **QueryRayInner**(root(\mathcal{Q}'), p).

End **QueryRay**

Function **QueryRayInner**($node, p$)

1. if $node$ is a leaf then return *Counter*.
2. locate the cell v of \mathcal{V}_{node} that contains p .
3. set *Counter* to *Counter* + **QueryLine**(v, l_p).
4. return **QueryRayInner**(v, p).

End **QueryRayInner**

Lemma 3.2. *Given a set Γ of algebraic arcs, defined by polynomials of constant maximum degree, we can preprocess the set $\tilde{\Gamma}$ in time $O(n^{2+\varepsilon})$, into a data structure of size $O(n^{2+\varepsilon})$, such that the k surfaces of $\tilde{\Gamma}$ intersected by a query upward vertical ray can be reported in time $O(\log n + k)$ or counted in time $O(\log n)$.*

Proof. The size and construction time of \mathcal{Q}' were analyzed above. We are left to prove the correctness of the query algorithm, and analyze its running time. We concentrate on the counting algorithm, and show that all surfaces that are intersected by r_p are counted exactly once. This is sufficient to prove correctness, since it is easy to see that a surface that is *not* intersected by r_p cannot be counted. The proof can easily be modified to suite the context of reporting.

Let $\tilde{\gamma} \in \tilde{\Gamma}$ be a surface that is intersected by r_p . Consider a node v that is traversed by the algorithm, such that $\tilde{\gamma}$ is associated with the parent of v , but not with v . Lemma 2.3 implies that such a node always exists, and is unique. Since there is a point on $\tilde{\gamma}$ that is hit by an upward ray emanating from p (which is inside v , since v is traversed by the algorithm), $\tilde{\gamma}$ intersects the top shaft of v . Thus, the part of $\tilde{\gamma}$ intersecting the top shaft of v is in $\tilde{\Gamma}_v^t$, and is one of the surfaces that are preprocessed for the vertical-line intersection searching data structure L_v . It will therefore be counted (exactly once) during Stage 3 of **QueryRayInner**, when this function visits the parent node of v .

The time spent at Stage 1 of **QueryRayInner** is bounded by a constant, the time spent at Stage 2 is $O(\log n)$ by the construction of $\mathcal{Q}(\tilde{\Gamma})$, and Lemma 3.1 implies that the time spent at Stage 3 is also $O(\log n)$. The query time thus follows from Lemma 2.4. \square

3.2 Segment Intersection Searching

Consider augmenting $\mathcal{Q}(\tilde{\Gamma})$ as follows. For each node v of $\mathcal{Q}(\tilde{\Gamma})$, we preprocess $\tilde{\Gamma}_v^b$ for upward-ray intersection searching, as described in the previous subsection, and store the resulting two-level data structure (denoted by R_v) at v . This results in a three-level data structure, denoted by \mathcal{Q}'' , which we use for counting or reporting the surfaces of $\tilde{\Gamma}$ that are intersected by a vertical query segment.

The counting query algorithm on \mathcal{Q}'' is given in pseudo-code below. It receives a vertical segment s , the top and bottom end-points of which are denoted by s_t and s_b , respectively. The algorithm can be easily modified for the purpose of intersection reporting.

Function **QuerySegment**(\mathcal{Q}'' , s)

1. set *Counter* to 0.
2. return **QuerySegmentInner**(root(\mathcal{Q}''), s).

End **QuerySegment**

Function **QuerySegmentInner**(*node*, s)

1. if *node* is a leaf then return *Counter*.
2. locate the cell v of \mathcal{V}_{node} that contains s_t .
3. set *Counter* to *Counter* + **QueryRay**(R_v , s_b).
4. return **QuerySegmentInner**(v , s).

End **QuerySegmentInner**

Theorem 3.3. *A collection of n possibly intersecting algebraic arcs, defined by polynomials of constant maximum degree, can be preprocessed in time $O(n^{2+\epsilon})$ into a data structure of size $O(n^{2+\epsilon})$, such that the k arcs intersecting a query segment can be counted in time $O(\log n)$ or reported in time $O(\log n + k)$.*

Proof. We concentrate our attention on the problem of counting, noting that the proof can easily be modified to suite the context of reporting. Correctness of the preprocessing and query algorithms of \mathcal{Q}'' is proved similarly to the proof of Lemma 3.2, while the query time of \mathcal{Q}'' follows from Lemma 2.4, since the time spent at Stage 1 of **QuerySegmentInner** is bounded by a constant, the time spent at Stage 2 is $O(\log n)$ by the construction of $\mathcal{Q}(\tilde{\Gamma})$, and Lemma 3.2 implies that the time spent at Stage 3 is also $O(\log n)$.

We are left to prove that the size and construction time of \mathcal{Q}'' are $O(n^{2+\epsilon})$, which will easily follow from Lemmas 2.4 and 3.2, provided that the data structure R_v has the same performance as the data structure \mathcal{Q}' (for any node v of $\mathcal{Q}(\tilde{\Gamma})$). This will follow if we prove that the ray intersection searching algorithm of the previous section, and its analysis, are applicable to $\tilde{\Gamma}_v^b$. For a node v of $\mathcal{Q}(\tilde{\Gamma})$, $\tilde{\Gamma}_v^b$ is the part of $\tilde{\Gamma}_v$ that is contained in the bottom shaft of v , and it is not equivalent to $\tilde{\Gamma}$. The question is whether Lemma 3.2 still holds if we replace $\tilde{\Gamma}$ by $\tilde{\Gamma}_v^b$ in its text.

We now prove that $|\mathcal{V}(\tilde{\Gamma}_v^b)| = O(\lambda_{s+2}^2(n))$. According to the discussion in Section 2.3, this implies that $|\mathcal{Q}(\tilde{\Gamma}_v^b)| = O(n^{2+\epsilon})$, which makes the performance analysis in the previous subsection valid. Observe that

$$|\mathcal{V}(\tilde{\Gamma}_v^b)| = O(|\mathcal{V}(\tilde{\Gamma}_v \cup W)|),$$

where W denotes the vertical “walls” of the bottom shaft of v . (Intuitively, since the “ceiling” of the shaft is defined by a surface that belongs to $\tilde{\Gamma}_v$, the “walls” of the shaft are the only thing

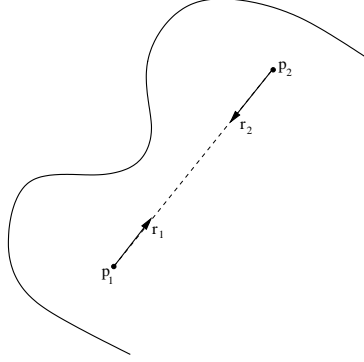


Figure 2: An arc (shown solid) that intersects the rays r_1 and r_2 does not necessarily intersect the segment (p_1, p_2) .

that might make $|\mathcal{V}(\tilde{\Gamma}_v^b)|$ larger than $|\mathcal{V}(\tilde{\Gamma}_v)|$.) In turn,

$$|\mathcal{V}(\tilde{\Gamma}_v \cup W)| = O(|\mathcal{V}(\tilde{\Gamma}_v)| + |W \cap \mathcal{A}(\tilde{\Gamma}_v)|),$$

since the walls W are vertical. Lemma 2.2 states that $|\mathcal{V}(\tilde{\Gamma}_v)| = O(\lambda_{s+2}^2(n))$. $W \cap \mathcal{A}(\tilde{\Gamma}_v)$ is a 2-dimensional arrangement of $O(n)$ algebraic arcs, and thus $|W \cap \mathcal{A}(\tilde{\Gamma}_v)| = O(n^2)$. Hence,

$$|\mathcal{V}(\tilde{\Gamma}_v^b)| = O(\lambda_{s+2}^2(n) + n^2) = O(\lambda_{s+2}^2(n)).$$

Therefore, the data structure R_v possesses the necessary performance bounds, which implies the asserted size and construction time bounds on \mathcal{Q}'' . \square

Some remarks. It is worth pointing out a property that complicates the design of algorithms for segment intersection searching among arcs. Consider a query segment $s : (p_1, p_2)$, and two rays, one emanating from p_1 toward p_2 and another emanating from p_2 toward p_1 . Any object that intersects s has to intersect both rays. In fact, any *segment* that intersects both rays intersects s as well. On the other hand, as shown in Figure 2, the latter statement does not hold for general arcs — an arc that intersects both rays may fail to intersect s . Thus, the segment intersection searching problem cannot be simply solved by a two-level ray intersection searching data structure in \mathbb{R}^2 . In our solution, this problem is treated by constructing the second-level data structure R_v only for the parts of the surfaces of $\tilde{\Gamma}$ that lie inside the bottom shaft of v .

Note that if we just want to count the number of intersection *points* between the arcs and a query segment (as opposed to counting the number of intersected *arcs*), a more naive data structure, that does not require the use of shafts, suffices. We leave the details as an exercise to the reader, noting that in this setting we can decompose the surfaces $\tilde{\Gamma}$ into xy -monotone patches, and operate on the collection of these patches.

We also remark that the data structure of Theorem 3.3 can be modified to treat (the more general) colored segment intersection searching [15]. We omit the easy details.

4 A Trade-Off Between Space and Query Time

As mentioned in the introduction, a space-time trade-off was previously reported [3] for the problem of segment intersection searching among circular arcs. In particular, a data structure

of size m , for $n \leq m \leq n^3$, that can count the intersections between a query segment and n circular arcs in time $O(\frac{n}{m^{1/3}} \text{polylog}(n))$, was reported, as well as a data structure with identical performance for ray shooting among circular arcs.

Theorem 7 in [3] provides a general technique for combining two data structures in order to obtain a space-time trade-off. This technique can be easily applied to combine the segment intersection searching data structure of Theorem 3.3 with the near-linear segment intersection searching (among circular arcs) data structure of [8], and to combine the ray shooting data structure of Section 2.4 with the near-linear ray shooting (among circular arcs) data structure of [8]. This leads to the following theorem.

Theorem 4.1. *A collection of n possibly intersecting circular arcs can be preprocessed into a data structure of size $O(m)$, for any $n \leq m \leq n^{2+\varepsilon}$, that allows reporting the k arcs intersecting a query segment in time $O((n^{2+\varepsilon}/m)^{2/3} + \log \frac{m}{n} + k)$, counting them in time $O((n^{2+\varepsilon}/m)^{2/3} + \log \frac{m}{n})$, as well as performing ray shooting amidst the arcs in time $O((n^{2+\varepsilon}/m)^{2/3} + \log \frac{m}{n})$.*

5 Disjoint Arcs

It was shown in Section 3 that segment intersection searching queries in general settings can be handled efficiently by a data structure of size $O(n^{2+\varepsilon})$. The goal of this section is to show that slightly smaller data structures are possible for the restricted setting of disjoint arcs.

Theorem 5.1. *For a set Γ of n disjoint algebraic arcs, defined by polynomials of constant maximum degree, a data structure of size $O(n^2)$ that allows reporting the k arcs intersecting a query segment in time $O(\log n + k)$, as well as performing ray shooting in time $O(\log n)$, can be constructed in time $O(n^2 \log n)$. A data structure of size $O(n^2 \log n)$ that also allows counting the number of intersected arcs in time $O(\log n)$ can be constructed in time $O(n^2 \log^2 n)$.*

Proof. $\tilde{\Gamma}$ is a collection of disjoint algebraic surfaces. It can be partitioned into $O(n^2)$ vertically unbounded *slabs* (infinite vertical prisms), by erecting infinite vertical walls from vertical tangency points on all the surfaces of $\tilde{\Gamma}$. All surfaces contained in a certain slab are totally defined in it and are disjoint. Therefore, for each slab, a total vertical order on the portions of the surfaces of $\tilde{\Gamma}$ contained in the slab exists. Moreover, the sets of surfaces contained in adjacent slabs differ by at most a constant number of surfaces⁴.

Partitioning each surface of $\tilde{\Gamma}$ into (a constant number of) xy -monotone patches results in $O(n)$ xy -monotone patches overall. The xy -projection of the boundary of each patch is an algebraic curve. Let Ψ be the collection of these projections. $\mathcal{A}(\Psi)$ corresponds to the planar decomposition obtained by intersecting the boundaries of all the vertical slabs described above with an xy -parallel plane. We can thus identify the slab containing a certain point by performing point location in $\mathcal{A}(\Psi)$.

We first describe the data structure used for answering ray shooting and segment intersection reporting queries. The data structure used for segment intersection counting queries is more involved and is described afterwards. To construct the former data structure, we first preprocess $\mathcal{A}(\Psi)$ into a point location structure that has size $O(n^2)$ and query time $O(\log n)$. This can be done in time $O(n^2 \log n)$, using any optimal point location algorithm [22].

⁴Assuming general position. Otherwise, the end-points of $\Omega(n)$ arcs can be collinear, resulting in the existence of two adjacent slabs, one of which contains $\Omega(n)$ surfaces of $\tilde{\Gamma}$ that are not contained in the other. It can be proved that this would not affect the complexity of the algorithm, or the size of the data structure.

With each face v of $\mathcal{A}(\Psi)$ we associate a binary search tree on the vertically-ordered list of xy -monotone portions of surfaces that intersect $S(v)$, where $S(v)$ denotes the vertical slab that corresponds to v . Since the lists associated with adjacent faces differ by at most a constant number of members, persistency can be employed to efficiently store them. There are $O(n^2)$ faces in $\mathcal{A}(\Psi)$; thus, all the binary search trees can be stored persistently using space $O(n^2)$ in time $O(n^2 \log n)$, using the node-copying method [14].

We now describe the query algorithms. Lemma 2.1 implies that ray shooting in Γ can be reduced to locating the surface of $\tilde{\Gamma}$ that lies immediately below some point $p:(x, y, z)$. The ray shooting query procedure first locates the face v_p of $\mathcal{A}(\Psi)$ that contains the point (x, y) , and then queries the binary search tree associated with v_p to obtain the required surface of $\tilde{\Gamma}$. The overall query time is $O(\log n)$.

Segment intersection reporting queries are conducted similarly. By Lemma 2.1, it suffices to report the surfaces of $\tilde{\Gamma}$ that intersect a vertical segment $s:((x, y, z_1), (x, y, z_2))$. The query algorithm first locates the face v_p of $\mathcal{A}(\Psi)$ that contains the point (x, y) , and then queries the binary search tree associated with v_p for all the (portions of) surfaces located between the two end-points. The overall query time is $O(\log n + k)$, where k is the number of reported surfaces.

Some care should be taken, since a surface of $\tilde{\Gamma}$ may be reported more than once. This is due to the fact that up to a constant number of xy -monotone portions in $S(v)$ may actually be parts of one surface of $\tilde{\Gamma}$. Double reporting can be easily avoided by associating a binary flag with each surface of $\tilde{\Gamma}$. This flag is initially turned off, and is turned on by the algorithm when the surface is first reported. A surface is reported only if the flag is not yet turned on.

This query algorithm can be altered in a straightforward fashion to count the number of *portions* of surfaces of $\tilde{\Gamma}$ that are intersected by the vertical query segment, in $O(\log n)$ time. This corresponds to counting the number of intersections between the arcs in Γ and a query segment in the plane. In order to be able to count the number of *arcs* in Γ that are intersected by the query segment we have to design a data structure that allows counting the number of surfaces of $\tilde{\Gamma}$ that intersect a query vertical segment. To this end, we modify the data structure that is associated with each vertical slab $S(v)$ (where v is a face of $\mathcal{A}(\Psi)$).

The situation inside a particular vertical slab is essentially 1-dimensional. It can be likened to the following. There are $O(n)$ points on the real line, each having a specific color. Every color is shared by $O(1)$ points. Our data structure has to be able to efficiently count the number of distinct colors of points contained in a query interval (as opposed to counting the number of points contained in a query interval).

Lemma 5.2 ([15]). *A set of n colored points on the real line can be preprocessed into a data structure of size $O(n \log n)$ that can count the number of distinct colors of points contained in a query interval in time $O(\log n)$.*

The lemma was stated in a 1-dimensional setting for the sake of clarity. The construction and query algorithms of the described data structure can be easily carried over to the setting of our original problem, involving totally-defined disjoint portions of surfaces inside a vertical slab. Points having the same color correspond to portions of surfaces belonging to the same surface of $\tilde{\Gamma}$, and query intervals correspond to vertical query segments.

To ensure the stated size and preprocessing time bounds, we save the data structures associated with the faces of $\mathcal{A}(\Psi)$ persistently, using the node-copying method [14], and relying on the fact that adjacent slabs differ by at most a constant number of surface portions. The two-

level data structure of [15] that is associated with each face is more involved than that used for segment intersection reporting and ray shooting queries, which was simply a binary search tree. In particular, the removal or addition of one surface portion from the slab can affect $O(\log n)$ second-level data structures. Thus, although the node-copying method is still applicable in our case, an extra logarithmic factor is added to the space and preprocessing time bounds. \square

6 Ray Shooting

We now employ the results from the previous section to devise a simple and efficient data structure for ray shooting amidst possibly intersecting algebraic arcs. Our approach is similar to the approach used by Agarwal [1] in the special case of ray shooting amidst segments.

Theorem 6.1. *A collection Γ of n possibly intersecting algebraic arcs, defined by polynomials of maximum degree s , can be preprocessed in time $O(\lambda_{s+2}^2(n) \log n)$ into a data structure of size $O(\lambda_{s+2}^2(n))$, such that ray shooting queries can be answered in time $O(\log n)$.*

Proof. Preprocess $\mathcal{A}(\Gamma)$ into any optimal point location data structure. Consider a face f of $\mathcal{A}(\Gamma)$, and denote its complexity by N_f . Its (not necessarily connected or closed) boundary consists of at most N_f interior-disjoint arcs. Preprocess these arcs into the ray shooting data structure of the previous section, and associate it with the face f . The size of this data structure is $O(N_f^2)$.

This process is repeated for all faces in $\mathcal{A}(\Gamma)$. The overall required storage space is thus (asymptotically) bounded by the sum-of-squares of cell complexities $\sum_f N_f^2$, which is $O(\lambda_{s+2}^2(n))$ [21, Theorem 5.13]. Similarly, the overall preprocessing time is easily seen to be bounded by $O(\lambda_{s+2}^2(n) \log n)$.

Given a query ray, the query algorithm first locates the face of $\mathcal{A}(\Gamma)$ that contains the ray's origin point. It then performs ray shooting inside this face, using the ray shooting data structure associated with this face. \square

7 Applications

7.1 Triangular Windowing

Theorem 7.1. *A collection Γ of n algebraic arcs, defined by polynomials of constant maximum degree, can be preprocessed in time $O(n^{2+\varepsilon})$ into a data structure of size $O(n^{2+\varepsilon})$, such that the k arcs intersecting the closure of the interior of a query triangle can be reported in time $O(\log n + k)$.*

Proof. When queried with a triangle φ , the data structure has to return the union of the following two (not necessarily disjoint) sets: the set of arcs intersected by the edges of φ , and the set of arcs completely contained in φ . We build one data structure for reporting arcs of the first type, and another data structure for reporting arcs of the second type.

For the purpose of reporting arcs of the first type, we preprocess Γ into the segment intersection searching structure of Section 3. All arcs intersecting the edges of φ can be reported by querying this structure with each of the edges of φ . Theorem 3.3 states that the size of

the segment intersection searching structure is $O(n^{2+\varepsilon})$, and so is the time required for its construction.

For the purpose of reporting arcs of the second type, we pick a point on each of the arcs of Γ . This results in a set of n points, which we preprocess for triangular range searching, using the data structure of Chazelle, Sharir and Welzl [10] that has size $O(n^{2+\varepsilon})$ and matching construction time. Each arc is associated with the point that was picked on it. If an arc γ is completely contained in φ , then any point on γ is also contained in φ . Thus, all arcs of the second type can be reported by querying the described triangular range searching structure with φ .

The overall query algorithm consists of querying the first structure with each of the three edges of φ , and querying the second structure with φ . Each of the four queries takes time $O(\log n + k)$.

Notice that an arc may be reported more than once. Double reporting can be avoided by associating a binary flag with each arc. This flag is initially turned off, and is turned on when the arc is first reported. An arc is reported only if the flag is not yet turned on. \square

Remark. This data structure has the same size and construction time as the best known triangular range searching data structures that have $O(\log n + k)$ query (reporting) time [10], but is considerably more general, as its input objects are arcs that are not restricted to be points. It can also be easily extended to support *clipping* the arcs of Γ to the interior of a polygonal query window of arbitrary shape and topology.

7.2 Bichromatic Intersection Counting

Theorem 7.2. *Given a set B of n possibly intersecting segments and a set R of n possibly intersecting algebraic arcs as above, the number of intersections between segments of B and arcs of R can be counted in time $O(n^{\frac{3}{2}+\varepsilon})$, using space $O(n)$.*

Proof. The algorithm is obtained by *batching* the segment intersection counting data structure of Theorem 3.3. The gist of the batching technique [1] is dividing the problem into a (large) number of relatively small sub-problems, and treating each sub-problem separately. Batching can be applied to the problem at hand in the following fashion.

For any $\varepsilon > 0$, partition R , arbitrarily, into $n^{\frac{1}{2}+\varepsilon}$ sets of size $n^{\frac{1}{2}-\varepsilon}$ each. Then, consider the sets sequentially, and for each set: Use Theorem 3.3 to preprocess it for segment intersection counting, and query the resulting data structure with every segment from B . The required number of intersections is the sum of the answers to all the segment intersection searching queries made during this process.

The asserted performance bounds follow from Theorem 3.3 using routine calculations. Notice, in particular, that partitioning R as above (instead of, say, into \sqrt{n} sets of size \sqrt{n}) ensures that only linear space (instead of space $O(n^{1+\varepsilon})$, as in the above example) is needed, since each stage of the algorithm uses only one data structure, of size $O(n^{(\frac{1}{2}-\varepsilon)\cdot(2+\varepsilon)}) = O(n)$. \square

Remark. The data structure of Theorem 3.3 allows counting the number of data arcs intersected by a query segment, while this application requires a data structure that allows counting the number of *intersections* between the data arcs and the query segment. It is easy to adjust the data structure of Theorem 3.3 for this purpose.

8 Conclusion

We have presented efficient data structures for a variety of intersection searching problems among algebraic arcs, and have studied two possible applications — triangular windowing and red-blue intersection counting. An interesting open problem is to design an algorithm with significantly sub-quadratic running time that counts red-blue intersections when both the red and the blue sets consist of possibly intersecting algebraic arcs. Our result, enabling one of the sets to consist of such arcs, is a step towards this goal.

Acknowledgements

The author wishes to thank Micha Sharir for his comments on this paper and for helpful discussions on the problems studied herein. The suggestions of two anonymous referees are also appreciated.

References

- [1] P. K. Agarwal. *Intersection and decomposition algorithms for planar arrangements*. Cambridge University Press, New York, NY, 1991.
- [2] P. K. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–598. CRC Press LLC, Boca Raton, FL, 1997.
- [3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.
- [4] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.
- [5] P. K. Agarwal, M. Pellegrini, and M. Sharir. Counting circular arc intersections. *SIAM J. Comput.*, 22(4):778–793, 1993.
- [6] P. K. Agarwal and M. Sharir. Applications of a new space-partitioning technique. *Discrete Comput. Geom.*, 9:11–38, 1993.
- [7] P. K. Agarwal and M. Sharir. Circle shooting in a simple polygon. *J. Algorithms*, 14:69–87, 1993.
- [8] P. K. Agarwal, M. van Kreveld, and M. Overmars. Intersection queries in curved objects. *J. Algorithms*, 15(2):229–266, 1993.
- [9] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoret. Comput. Sci.*, 84:77–105, 1991.

- [10] B. Chazelle, M. Sharir, and E. Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8:407–429, 1992.
- [11] S.-W. Cheng, O. Cheong, H. Everett, and R. van Oostrum. Hierarchical vertical decompositions, ray-shooting and circular arc queries in simple polygons. In *Proc. 15th ACM Symposium on Computational Geometry*, pages 227–236, 1999.
- [12] K. L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.
- [13] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *Lecture Notes Comput. Sci.* Springer-Verlag, Berlin, Germany, 1993.
- [14] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.
- [15] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: counting, reporting and dynamization. *J. Algorithms*, 19:282–317, 1995.
- [16] S. Har-Peled and M. Sharir. On-line point location in planar arrangements and its applications. *Discrete Comput. Geom.*, 26:19–40, 2001.
- [17] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.*, 2:127–151, 1987.
- [18] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science*, pages 56–65, 2001.
- [19] M. Pellegrini. Ray shooting and lines in space. In J. E. Goodman and J. O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 32, pages 599–614. CRC Press LLC, Boca Raton, FL, 1997.
- [20] M. Pocchiola and G. Vegter. Pseudo-triangulations: Theory and applications. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 291–300, 1996.
- [21] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.
- [22] J. Snoeyink. Point location. In J. E. Goodman and J. O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.