Online Point Location in Planar Arrangements and Its Applications*

Sariel Har-Peled† Micha Sharir‡

August 17, 2002

Abstract

Recently, Har-Peled [HP99b] presented a new randomized technique for online construction of the zone of a curve in a planar arrangement of arcs. In this paper, we present several applications of this technique, which yield improved solutions to a variety of problems. These applications include: (i) an efficient mechanism for performing online point location queries in an arrangement of arcs; (ii) an efficient algorithm for computing an approximation to the minimum-weight Steiner-tree of a set of points, where the weight is the number of intersections between the tree edges and a given collection of arcs; (iii) a subquadratic algorithm for cutting a set of pseudo-parabolas into pseudo-segments; (iv) an algorithm for cutting a set of line segments ('rods') in 3-space so as to eliminate all cycles in the vertical depth order; and (v) a near-optimal algorithm for reporting all bichromatic intersections between a set R of red arcs and a set B of blue arcs, where the unions of the arcs in each set are both connected.

1 Introduction

Let S be a set of n x-monotone arcs in the plane, each pair of which intersect in at most t points. Computing the whole (or parts of the) arrangement $\mathcal{A}(S)$, induced by the arcs of S, is one of the fundamental problems in computational geometry, and has received a lot of attention in recent years [SA95]. One of the basic techniques used for such constructions is based on *randomized incremental* construction of the *vertical decomposition* of the arrangement (see [Mul94, BY98]).

*This work has been supported by a grant from the U.S.–Israeli Binational Science Foundation. Work by Micha Sharir has also been supported by NSF Grant CCR-97-32101, by a grant from the Israeli Academy of Sciences for a Center of Excellence in Geometric Computing at Tel Aviv University, by the ESPRIT IV LTR project No. 21957 (CGAL), and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

†Department of Computer Science, University of Illinois, Urbana, IL, 61801, USA; sariel@cs.uiuc.edu; http://www.uiuc.edu/~sariel/

‡School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA; sharir@math.tau.ac.il; http://www.math.tau.ac.il/~sharir/

Some applications of planar arrangements require the construction of only parts of the arrangement (e.g., a single face or a zone of some curve), and, usually, those parts have smaller combinatorial complexity than that of the whole arrangement. For example, consider the *zone* of a curve γ, which is the collection of all the faces of $\mathcal{A}(S)$ that γ crosses. The complexity of such a zone is $O(\lambda_{t+2}(n+m))$, where m is the number of intersections between γ and the arcs of S and $\lambda_q(r)$ is the maximum length of a Davenport-Schinzel sequence of order q having r symbols ($\lambda_q(r)$ is nearly linear in r for any fixed q; see [SA95]). Hence, if γ has simple shape (e.g., it intersects each arc of S in a constant number of points), then m is small and the complexity of the zone is significantly smaller than that of the whole arrangement. Furthermore, under reasonable assumptions, the zone can be computed in $O(\lambda_{t+2}(n+m)\log n)$ randomized expected time [CEG+93, dBDS95].

A somewhat harder variant of this theme is the following *online* scenario: We start from a point $p = p(0) \in \mathbb{R}^2$, and we find the face f of $\mathcal{A}(S)$ that contains $p(0)$. Now the point p starts moving and traces a connected curve $\{p(t)\}_{t\geq 0}$. As this walk continues, we wish to keep track of the face of $\mathcal{A}(S)$ that contains the current point $p(t)$. The collection of these faces constitutes the zone of the curve $p(t)$. However, the function $p(t)$ is not assumed to be known in advance, and it may change when we cross into a new face or abruptly change direction in the middle of a face (see [BDH99] for an application where such a scenario arises).

Recently, Har-Peled [HP99b] gave a near optimal randomized algorithm for this online variant (a somewhat slower but deterministic solution for this problem, for the case involving lines, was recently given by Chan [Cha99]). The main idea behind this algorithm, which we call `CompZoneOnline`, as in [HP99b], relies on simulating an offline randomized incremental algorithm (which is allowed to consult an oracle for determining whether a given trapezoid lies in the zone). In fact, the technique used in `CompZoneOnline` can construct, in an online manner, any portion of the arrangement. More specifically, the algorithm can receive as input (in an online manner) any sequence of points, and it constructs the collection of vertical trapezoids (or faces) that contain the given points. Moreover, this construction is efficient, in the sense that its cost is comparable with the smallest complexity of a *connected* portion of the arrangement that contains all the points. See below for more details.

In this paper we present several applications of the new approach that are based on the properties just noted. All the new algorithms are faster than those previously known, and in some cases they provide new insights into the problem at hand. The applications that we present are:

- For a set S of n x-monotone arcs, each pair of which intersect in at most t points, the algorithm of [HP99b] can be made into an online point-location mechanism: Given any set P of m query points, in an online manner, the algorithm computes for each point the trapezoid of the vertical decomposition of $\mathcal{A}(S)$ that contains it. The overall expected cost is $O(\lambda_{t+2}(n+w)\log n)$, where $w = w(P,S)$ is the smallest number of intersections between the arcs of S and a (Steiner) tree that connects the query points. For the worst distribution of points, one has $w = O(n\sqrt{m})$ [Aga91], so the algorithm takes $O(\lambda_{t+2}(n\sqrt{m})\log n)$ expected time.

- We present an algorithm that computes a Steiner tree of a set P of m points so that the expected number of intersections between the tree edges and a prescribed set S

of n arcs, as above, is $O(\lambda_{t+2}(n + w(P, S)) \log n)$. Namely, the algorithm is "output-sensitive" in the *weight* $w(P, S)$ of the optimal Steiner tree, and its running time is always subquadratic in m and n, because, as already mentioned, $w = O(n\sqrt{m})$ in the worst case. To our knowledge, no similar "output-sensitive" algorithm for this problem was previously known. (However, a tree of weight $\Theta(n\sqrt{m})$ can be computed, in subquadratic time, using cuttings [Aga91].)

- Let S be a given set of n *pseudo-parabolas* (namely, x-monotone curves, each pair of which intersect at most twice). We present a randomized subquadratic algorithm that cuts these curves into *pseudo-segments* (namely, each pair of the new pieces intersect at most once). The motivation for this problem comes from the analysis of Tamaki and Tokuyama [TT98], who showed that this can always be done with no more than $O(n^{5/3})$ cuts. The expected number of cuts performed by the algorithm is at most $O(\lambda_4(n\sqrt{\mu}) \log n)$, where μ is the minimum number of cuts that are needed. (Actually, the expected number of cuts is $O(\lambda_4(n + w(P, S)) \log n)$, where P is any set of cutting points of the minimum size μ.) The expected running time of the algorithm is $O(\lambda_4(n + w(P, S)) \log^4 n) = O(\lambda_4(n\sqrt{\mu}) \log^4 n)$. As shown by Tamaki and Tokuyama [TT98], one always has $\mu = O(n^{5/3})$, so our algorithm is indeed subquadratic. In Appendix A, we also present an alternative 'greedy' algorithm that applies to more general arcs and performs only $O(\mu \log n)$ cuts, but requires superquadratic time; this alternative solution is not related to our basic point-location technique.

- Let S be a set of n disjoint nonvertical line segments ('rods') in 3-space. A rod $s \in S$ lies below another rod $s' \in S$ if there is a vertical line that passes through both of them and meets s at a point below its intersection with s'. The transitive closure of this relation is known as the *depth order* of S. In general, this terminology is misleading, because this relation can have cycles. In several applications in computer graphics and related areas, it is desirable to cut the given rods into smaller pieces, so that the depth order of the new set has no cycles. We present an algorithm that receives S as input and cuts its rods into smaller pieces, so that no cycles remain in the depth order of the new rods. The previous algorithm, due to Solan [Sol98], performs $O(n^{1+\varepsilon}\sqrt{\mu})$ cuts, for any $\varepsilon > 0$, where μ is the minimum number of cuts needed to eliminate all cycles. Our algorithm performs $O(n\sqrt{\mu}\alpha(n) \log n)$ expected number of cuts. Its expected running time is $O(n^{4/3+\varepsilon}\mu^{1/3})$, for any $\varepsilon > 0$ (same as the bound for Solan's algorithm). The new algorithm is conceptually simpler than the previous result, which relied on recursive application of cuttings.

- Let R be a set of 'red' arcs and B be a set of 'blue' arcs in the plane, where the total number of arcs is n and they satisfy the same properties as above. Suppose, in addition, that the union of the red arcs and the union of the blue arcs are both connected. We present an algorithm for reporting all k bichromatic intersections between the arcs of R and of B, with expected running time $O(\lambda_{t+2}(n + k) \log n)$. This improves the best known algorithms [BGR96, Cha99] by one or two logarithmic factors.

Informally, the point-location mechanism of [HP99b] can be applied either to an unknown set of points (as in the applications of cutting pseudo-parabolas and cutting rods in space)

or to a set of points that are revealed only during the execution of the algorithm (as in the case of computing bichromatic intersections). Some of these situations can also be handled by recursively constructing cuttings of the given arrangement of arcs. However, (i) this does not always work (e.g., in the case of bichromatic intersections), and (ii) when this alternative technique can be applied, the resulting algorithms are considerably more cumbersome and less efficient.

The paper is organized as follows. In Section 2 we review the technique of [HP99b]. In Sections 3–7 we present the above applications. Concluding remarks are given in Section 8.

2 Review of the Technique

Let S be a set of n x-monotone arcs in the plane, as above, so that any pair of arcs of S intersect at most t times (for some fixed constant t). Let $\mathcal{A}(S)$ denote the arrangement of S; namely, the partition of the plane into faces, edges, and vertices as induced by the arcs of S (see [SA95] for details). For the sake of simplicity of exposition, we assume that S is in *general position*, meaning that no three arcs of S have a common point, and that the x-coordinates of the intersections and endpoints of the arcs of S are pairwise distinct. The *vertical decomposition* of $\mathcal{A}(S)$, denoted by $\mathcal{A}_{\mathcal{VD}}(S)$, is the partition of the plane into vertical pseudo-trapezoids, obtained by erecting two vertical segments up and down from each vertex of $\mathcal{A}(S)$ (i.e., each point of intersection between a pair of arcs and each endpoint of an arc), and by extending each of them until it either reaches an arc of S, or otherwise all the way to infinity. See, e.g., [BY98, SA95] for more details concerning vertical decompositions. To simplify (though slightly abuse) the notation, we refer to the cells of $\mathcal{A}_{\mathcal{VD}}(S)$ as *trapezoids*.

Computing the decomposed arrangement $\mathcal{A}_{\mathcal{VD}}(S)$ can be done as follows. Pick a random permutation $\langle S\rangle = \langle s_1, \ldots, s_n\rangle$ of S. Put $S_i = \langle s_1, \ldots, s_i\rangle$, for $i = 1, \ldots, n$. We compute incrementally the decomposed arrangements $\mathcal{A}_{\mathcal{VD}}(S_i)$, by inserting the i-th arc s_i of $\langle S\rangle$ into $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$, for each $i = 1, \ldots, n$, starting with an empty arrangement. To do so, we compute the *zone* Z_i of s_i in $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$, which is the set of all trapezoids in $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$ that constitute the faces of $\mathcal{A}(S_{i-1})$ that are crossed by s_i. We split each trapezoid of Z_i into at most $O(t)$ trapezoids, such that no trapezoid intersects s_i in its interior, as in [SA95]. Finally, we perform a pass over all the newly created trapezoids, and merge vertical trapezoids that are adjacent and have identical top and bottom arcs. The merging step guarantees that the resulting decomposition is $\mathcal{A}_{\mathcal{VD}}(S_i)$, independently of the insertion order of elements in S_i; see [dBvKOS97, BY98].

We can further augment this algorithm to produce on the fly a *history directed acyclic graph (DAG)* (as in [SA95]), whose nodes are the trapezoids created by the algorithm and where each trapezoid destroyed during the execution of the algorithm points to the trapezoids that were created from it. Let $H_{DAG}(S_i)$ denote this structure after the i-th iteration of the algorithm. Note that the out-degree of each node of H_{DAG} is bounded by a constant that depends on t. Each node v of H_{DAG} is associated with a trapezoid Δ_v, and Δ_v uniquely defines v once the permutation $\langle S\rangle$ used by the algorithm is fixed.

However, in many applications, $H_{DAG}(S)$ is by far too large, since it contains redundant information about parts of the arrangement that are of no interest for the application. In [HP99b], an online technique was described for computing only relevant parts of H_{DAG}. The

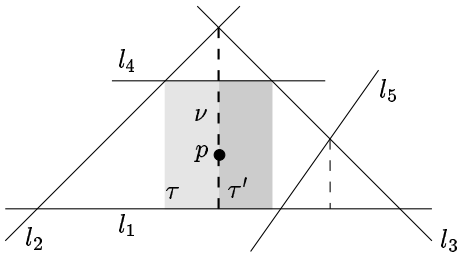

Figure 1: For $\langle S \rangle = \langle l_1, l_2, l_3, l_4, l_5 \rangle$, the trapezoid τ is transient and thus `Expand`(τ) will be called, resulting in the final (gray) trapezoid $\nu = \tau \cup \tau'$.

algorithm starts by computing a random permutation $\langle S \rangle$ of S, and by initializing a partial history DAG $\mathcal{T}$ to a DAG containing a single node that corresponds to the root of H_{DAG} (which represents the entire plane). Two basic operations on the partial history DAG $\mathcal{T}$ are provided.

1. `Split`(v) takes a leaf node of the partial history DAG that corresponds to a node of H_{DAG} and computes its children, into which Δ_v is split by inserting the next arc in the permutation that crosses it. Because of the merging step in the offline algorithm, some of those children may be *transient* (that is, they are to be merged into larger trapezoids by the offline algorithm, right at the end of the current insertion step).

2. `Expand`(v) takes a leaf v of $\mathcal{T}$, and computes the node that corresponds to it in H_{DAG}. This operation is null, unless the trapezoid currently associated with v is transient, in which case `Expand` is non-trivial—it has to compute (or retrieve) several other nodes in the partial history DAG and merge them to form the new "final" (i.e., non-transient) trapezoid, which is the desired node of H_{DAG}. This in turn may require computing other substantial portions of H_{DAG} that do not yet exist in $\mathcal{T}$. See Figure 1. The `Expand` operation is carried out by performing a sequence of (recursive) point-location queries in $\mathcal{T}$; see [HP99b] for details.

Each node v of $\mathcal{T}$ stores a conflict list $\mathrm{cl}(v)$, which is the subsequence of all the arcs of $\langle S \rangle$ that cross the interior of Δ_v. The *weight* of v is $|\,\mathrm{cl}(v)|$. The weight of $\mathcal{T}$ is the sum of the weights of its nodes.

Let us illustrate how the technique works for the case, studied in [HP99b], of computing the zone of a curve γ. The online algorithm, denoted `CompZoneOnline`, works as follows. At each step it obtains the next 'critical' point p along the curve γ (it is typically the point where γ exits from the current trapezoid of $\mathcal{A}_{\mathcal{VD}}(S)$), and performs a point-location query with p in $\mathcal{T}$. In general, the query mechanism follows a path in $\mathcal{T}$ and reaches a node v of $\mathcal{T}$ that is not a final trapezoid in $\mathcal{A}_{\mathcal{VD}}(S)$. The algorithm then expands below v the path in H_{DAG} that leads to the final leaf trapezoid that contains p, by executing a sequence of (recursive) calls to `Split` and `Expand`, until it obtains the final trapezoid of $\mathcal{A}_{\mathcal{VD}}(S)$ that contains p. (As noted, the algorithm may generate during this expansion many additional nodes of $\mathcal{T}$, in order to complete all the paths in $\mathcal{T}$ that lead to the desired final trapezoid.)

As shown in [HP99b], the overall cost of this construction is proportional to the total weight of $\mathcal{T}$. More precisely:

Theorem 2.1 ([HP99b]) *Let $\mathcal{T}$ be a history DAG computed by a sequence of operations of* `Split` *and* `Expand`*. Let m be the number of nodes in $\mathcal{T}$, and let w be the total weight of these nodes. The overall expected time spent on computing $\mathcal{T}$ is $O(w + m \log n)$ (where the expectation is with respect to the choice of the permutation $\langle S \rangle$.*

Lemma 2.2 ([Mul94]) *With high probability, for any query point p, the number of trapezoids of H_{DAG} that contain p is $O(\log n)$. Hence, the number of nodes of $\mathcal{T}$ that are visited when performing a point-location query with p (ignoring nodes visited in recursive calls to* `Split` *and* `Expand`*) is also $O(\log n)$ with high probability.*

Thus, if we use the above technique only as an engine for answering point-location queries, we can bound the expected running time, by estimating the weight of the resulting history DAG, and each point-location that we perform incurs an additional overhead of $O(\log n)$ expected time.

Lemma 2.3 ([HP99b]) *Let γ be any curve in the plane that intersects the arcs of S in a total of m points. Let $\mathcal{T}$ be the partial history DAG produced by the algorithm* `CompZoneOnline` *as it constructs the zone of γ in $\mathcal{A}_{\mathcal{VD}}(S)$. Then the expected number of nodes in $\mathcal{T}$ is $O(\lambda_{t+2}(n+m))$, and the total expected weight of the nodes of $\mathcal{T}$ is $O(\lambda_{t+2}(n+m)\log n)$.*

Remark 2.4 In the above results, the (expected) space needed by the algorithms is proportional to the weight of the computed *DAG*. Thus, the expected storage is asymptotically the same as the expected running time of the algorithms. The same holds for all the results in this paper, and we thus omit any explicit statement of the space bounds.

3 Online Point-Location Queries

As described in Section 2, one can use the partial history DAG mechanism to perform (online) point-location queries in an arrangement of arcs. In this section we analyze the performance of this technique. The material in this section is an extension of the analysis presented in [HP99b].

Definition 3.1 For a point set P, and a set of arcs S, let $\mathcal{M}(P, S)$ denote a connected polygonal set, such that: (i) $P \subseteq \mathcal{M}(P, S)$, and (ii) the number of intersections between $\mathcal{M}(P, S)$ and the arcs of S is the smallest possible. Let $w(P, S)$ denote this minimum number of intersections.

The set $\mathcal{M}(P, S)$ can be interpreted as the minimum-weight Steiner-tree of P, under the metric that measures the distance between two points a, b by the minimum number of arcs of S that have to be crossed by a path from a to b.

Theorem 3.2 *Given a set S of n arcs in the plane, as above, one can answer point-location queries for a set P of m points, where the points are given in an online manner, such that the overall expected time to answer those queries is $O((\lambda_{t+2}(n+w(P,S))+m)\log n)$, and the expected weight of the resulting history DAG is $O(\lambda_{t+2}(n + w(P, S))\log n)$. Here the output of a query is the trapezoid of $\mathcal{A}_{\mathcal{VD}}(S)$ that contains the query point. No preprocessing is needed, except for choosing a random permutation of S.*

Proof: We precompute a random permutation $\langle S \rangle$ of S, and perform the point-location queries by computing the relevant parts of the history DAG of $\mathcal{A}_{\mathcal{VD}}(S)$, as described in Section 2. (Note that here there is no "continuity", as in the case of constructing the zone of a curve: consecutive point location queries may explore far away portions of the plane. Nevertheless, since the point-location algorithm does not make any explicit assumption concerning continuity, its execution will not be affected by this lack of continuity; this will only affect the analysis of the expected running time of the algorithm.)

We claim that, by the time the algorithm terminates, each internal node of the partial history DAG $\mathcal{T}$, and each leaf that does not represent a transient trapezoid, is contained in $\mathcal{HT}_\gamma(S)$, which is the partial history DAG computed by `CompZoneOnline`(γ, S), for $\gamma = \mathcal{M}(P, S)$.

Indeed, consider two 'off-line' randomized incremental algorithms A_P, A_γ that compute, respectively, the zone of P and the zone of γ in $\mathcal{A}(S)$. Both algorithms use the same permutation $\langle S \rangle$, and are allowed to consult with an oracle $\mathcal{O}$ to decide whether or not a specific trapezoid lies inside the relevant zone. Let $\mathcal{HD}_P, \mathcal{HD}_\gamma$ denote the two resulting history DAGs produced by these two respective algorithms. Since $P \subseteq \gamma$, it easily follows that $\mathcal{HD}_P$ is a substructure of $\mathcal{HD}_\gamma$. In [HP99b, Lemma 3.7], it was shown that all the final nodes of $\mathcal{T}$ appear in $\mathcal{HD}_P$. On the other hand, all the nodes of $\mathcal{HD}_\gamma$ appear in $\mathcal{HT}_\gamma(S)$ [HP99b, Lemma 3.8]. Hence all the final nodes of $\mathcal{T}$ appear in $\mathcal{HT}_\gamma(S)$.

Furthermore, the total weight of the transient leaves in $\mathcal{T}$ is proportional to the total weight of their parents (which are final, and are thus included in the above analysis). This implies that the expected total weight of the trapezoids of $\mathcal{HT}_\gamma(S)$ is $O(\lambda_{t+2}(n + w(P, S)) \log(n))$, by Lemma 2.3.

As for the expected running time, it is bounded by the total weight of $\mathcal{T}$, plus the time spent directly on the point-location queries, and is thus $O((\lambda_{t+2}(n + w(P, S)) + m) \log n)$, by Lemma 2.2. ∎

Remark 3.3 The algorithm of Theorem 3.2, can be modified to output the whole face that contains the query point. Since the face description is not necessarily of constant size the output of the algorithm is a pointer to a data-structure that describes the face (i.e., a list of trapezoids forming the face, with adjacency information stored with each trapezoid). This modification has no effect on the overall asymptotic running times, and the bounds stated in Theorem 3.2 holds also for this case.

Lemma 3.4 *There exists a Steiner-tree $\mathcal{M}'$ of P, so that the total number of crossings between the arcs of S and the edges of $\mathcal{M}'$ is $w(P, S) = O(n\sqrt{m})$, and this is tight, for $m \leq n^2$, in the worst case (even when the arcs are lines).*

Proof: This is a folklore result, and we only sketch the rather easy proof. Compute a $(1/\sqrt{m})$-cutting of S of size $O(m)$ [HP00]. Let R' be the union of the boundaries of the pseudo-trapezoids of the cutting. Connect each point of P to R' by a vertical segment. Let R be the resulting connected set in the plane. It is easy to verify that, after deforming R slightly, it has at most $O(m \cdot (n/sqrtm)) = O(n\sqrt{m})$ crossings with the arcs of S, and it spans the points of S. The tightness of the bound follows from a simple grid construction that uses $n/2$ horizontal lines and $n/2$ vertical lines, and places the m points at evenly spaced grid cells that form a $\sqrt{m}$-by-$\sqrt{m}$ subgrid. ∎

Corollary 3.5 *The overall expected time to answer m online point-location queries in an arrangement of n arcs, as above, is $O((\lambda_{t+2}(n\sqrt{m}) + m)\log n)$.*

Remark 3.6 The result of Theorem 3.2 is somewhat disappointing since, in the worst case, $w(P,S) = \Theta(nm^{1/2})$ (Lemma 3.4), while for the case of lines or segments, m faces can be computed in, roughly, $O(n^{2/3}m^{2/3})$ time [AMS98] (a task that clearly subsumes our point location queries). Currently, for the case of general arcs, no bound better than $O\left(m^{1/2}\lambda_{t+2}(n)\right)$ is known for the complexity of m faces in an arrangement of n arcs (see [EGP+92, HP99a]). We leave the problem of improving the performance of the algorithm of Theorem 3.2, either for the general case or just for the case of lines, as an open problem for further research.

The algorithm of Theorem 3.2 is simple and online, and it has the additional favorable property of being adaptive. Namely, if $w(P,S)$ is small (i.e., the query points are "close together") the overall query time improves. Furthermore, if there are many queries close together, the first query may be slow, but the subsequent ones will be faster (since those queries will use parts of paths that already exist in the partial history DAG).

To our knowledge, no other algorithm have this adaptiveness property. In particular, if $w(P,S)$ is near linear, the overall time required to answer the online point-location is linear. This property is one of the key ingredients in the recent algorithm of Har-Peled and Indyk [HPI00] (see below).

4 Computing a Steiner-Tree with a Small Crossing Number

Since $w(P,S)$ can be considerably smaller than the bound provided by Lemma 3.4, it is useful, in several applications, to compute $\mathcal{M}(P,S)$ or even an approximation of it. A typical application is in range searching: Given S and P, we wish to count the number of arcs that lie above each point of P. Having a tree $\mathcal{T}$ that connects the points of P and has a small crossing number, allows us to compute the desired information by simply tracing the edges of $\mathcal{T}$ and updating the number of arcs above the traced point in time proportional to the total weight of the tree. (Technically speaking, we should replace each arc by the curve bounding the region lying below the arc—this is the curve formed by the original arc with two downward-directed vertical rays attached to the arc endpoints, and compute the tree for this set of curves.)

If one is not allowed to add Steiner points, then one needs to compute the minimum spanning tree of the points under the intersection distance induced by S (i.e., one wants to minimize the number of intersections between the MST and the arcs of S). Asano et al. [AdBC+99] have established worst-case tight bounds on the weight of such an MST, for the case where S is a set of segments. However, if one allows Steiner points in the MST, their bounds are easily obtained via cuttings (in a manner similar to the proof of Lemma 3.4).

In this section, we present an algorithm that computes a Steiner-tree for P so that its weight approximates well the minimum-weight Steiner-tree weight $w(P,S)$.

Theorem 4.1 *Let S be a set of n x-monotone arcs, as above, and P a set of m points in the plane. Then one can compute, in expected $O((\lambda_{t+2}(n+w)+m)\log n)$ time, a Steiner-tree*

$\hat{\mathcal{M}}$ of P, so that the expected weight of $\hat{\mathcal{M}}$ is $O(\lambda_{t+2}(n+w)\log n)$, where $w = w(P,S)$. Each edge of the tree is either a vertical segment or a portion of some arc of S.

Proof: Let $\mathcal{T}$ be the history DAG computed by performing online point-location queries for the points of P. Let $G' = \bigcup_{v\in\mathcal{T}} \partial\Delta_v$. This is a connected graph formed by the boundaries of the trapezoids stored at the nodes of $\mathcal{T}$. Note that each point of P is contained in a leaf trapezoid of $\mathcal{T}$ (which has an empty conflict list), and thus can be connected to G' by a vertical segment that does not intersect any of the arcs of S. The desired Steiner-tree is obtained by taking any spanning tree of P in the graph G'', which is the union of G' with all the vertical segments connecting each point of P to the boundary of the leaf trapezoid containing it. Clearly, the weight of this tree is proportional to the total weight of $\mathcal{T}$ (the sum of the sizes of the conflict lists of its nodes).

Let $\mathcal{M} = \mathcal{M}(P,S)$ be the minimum weight Steiner-tree of P, and let $\mathcal{T}_{\mathcal{M}}$ be the history DAG that would have been obtained if we applied `CompZoneOnline` to construct the zone of (the unknown) $\mathcal{M}$ in $\mathcal{A}(S)$, using the same permutation as the one used to construct $\mathcal{T}$. By Lemma 2.3, the expected weight of $\mathcal{T}_{\mathcal{M}}$ is $O(\lambda_{t+2}(n+w)\log n)$.

Arguing as in the proof of Theorem 3.2, it follows that the zone of $\mathcal{M}$ in $\mathcal{A}(S)$ contains the zone of P in $\mathcal{A}(S)$, and all the internal nodes (as well as all the nontransient leaves) of $\mathcal{T}$ are contained in $\mathcal{T}_{\mathcal{M}}$. Thus, the total weight of $\mathcal{T}$ is at most proportional to the weight of $\mathcal{T}_{\mathcal{M}}$, since the total weight of the leaves of a history DAG is proportional to the total weight of its internal nodes.

The expected additional overhead time for the point-location queries is $O(m\log n)$ by Lemma 2.2. Overall, the expected running time is $O((\lambda_{t+2}(n+w)+m)\log n)$. ∎

Remark 4.2 Note that some edges of the resulting Steiner-tree are portions of arcs of S, and the assertion concerning crossings between the tree edges and the arcs of S has to be modified, to take also into account overlaps between the tree edges and the given arcs. Alternatively, one can slightly perturb the tree edges so that they do not overlap any arc.

Remark 4.3 The algorithm of Theorem 4.1 was recently used by Har-Peled and Indyk [HPI00] to derive an algorithm with near-linear running time for computed an approximate MST under the crossing metric of lines in the plane.

5 Cutting Pseudo-Parabolas and More General Arcs into Pseudo-Segments

For a collection S of n arcs in the plane, it is sometimes desirable to cut them into smaller pieces that constitute a collection S' of *pseudo-segments*; that is, each pair of arcs of S' have at most one intersection point. Such an application, for a collection of parabolas or pseudo-parabolas (i.e., where each arc is the graph of a totally-defined continuous function, and each pair of arcs intersect at most twice), is given by Tamaki and Tokuyama [TT98]. They show that for such an arrangement, $O(n^{5/3})$ cuts are sufficient (in the worst case, $\Omega(n^{4/3})$ cuts may be required). In this section we present a randomized algorithm that computes a set of cuts, whose expected size is $O(\lambda_4(n\sqrt{\mu})\log n)$, where μ is the minimum number of cuts that

are required. The expected running time is $O(\lambda_4(n\sqrt{\mu})\log^4 n)$. Hence, the number of cuts that the algorithm makes is at most $O(\lambda_4(n^{11/6})\log n)$, and its expected running time is at most $O(\lambda_4(n^{11/6})\log^4 n)$. To our knowledge, this is the first subquadratic algorithm for this problem.

Our algorithm is based on a fast procedure (see Lemma 5.3 below) that detects, in $O(n\log^3 n)$ time, whether a collection of n pseudo-parabolas forms a pseudo-segment arrangement inside a trapezoid τ. Intuitively, the approach of the algorithm is to apply the online point location mechanism described above to the (unknown) points of the optimal cutting set. When the algorithm splits a trapezoid, we collect the children trapezoids for which the portion of the arrangement inside them is not a pseudo-segment arrangement, and recurse the point location procedure within each of them. Upon termination of this procedure, we obtain a covering of the plane so that inside each trapezoid the arrangement is a pseudo-segment arrangement. Hence, by cutting each pseudo-parabola at its intersection points with the boundaries of the trapezoids that it crosses, the resulting collection of arcs is a family of pseudo-segments.

Note that, as in the preceding section, parts of the boundaries of the trapezoids overlap the arcs of S, which means that some of the cutting points are intersection points of the arcs. In case this is undesirable, one may shift the cutting points slightly away (in both directions) from the trapezoid boundaries.

Theorem 5.1 *Let S be a set of n pseudo-parabolas. One can cut the arcs of S into smaller pieces that constitute an arrangement of pseudo-segments, so that the expected number of cuts is $O(\lambda_4(n\sqrt{\mu})\log n)$, where μ is the minimal number of cuts needed to cut S into a collection of pseudo-segments. The expected running time of the algorithm is $O(\lambda_4(n\sqrt{\mu})\log^4 n)$.*

Proof: Let $\mathcal{T}$ be a partial history DAG computed over a random permutation of S. We initialize $\mathcal{T}$ to a single-node DAG that represents the whole plane. The algorithm maintains a queue $\mathcal{Q}$ of some of the leaves of $\mathcal{T}$, initialized to contain the root of $\mathcal{T}$. Whenever a new leaf is being created, it is added to $\mathcal{Q}$.

The algorithm stops when the queue $\mathcal{Q}$ becomes empty. Otherwise, it removes the first element v of $\mathcal{Q}$. If since its insertion, v has become an internal node of $\mathcal{T}$, the algorithm ignores it and continues to the next element of $\mathcal{Q}$. Otherwise, the algorithm checks whether the arrangement of the arcs of $\mathrm{cl}(v)$ is a pseudo-segment arrangement inside Δ_v. As we shall show below, this can be done in time $O(|\,\mathrm{cl}(v)|\log^3|\,\mathrm{cl}(v)|)$, by the algorithm of Lemma 5.3. If not, the algorithm performs `Expand`(v) (any new leaves created by this operation are added to $\mathcal{Q}$), and then `Split`(v). The new children of v are added to $\mathcal{Q}$.

By the time the algorithm terminates, all the subarrangements inside the trapezoids of the leaves of $\mathcal{T}$ are pseudo-segment arrangements. Since the leaves of $\mathcal{T}$ form a covering of the plane, as can be easily verified (see also [HP99b]), it follows that by cutting each curve of S at each of its intersections with the boundaries of the trapezoids stored at the leaves of $\mathcal{T}$, we get a collection of pseudo-segments.

The number of cuts performed by the algorithm is bounded by the total weight of $\mathcal{T}$ (upon termination). Let $\mathcal{T}'$ be the (hypothetical) history DAG that would have been produced by performing online point-location queries with the set C^* of the μ cutting points of the optimal solution (using the same random permutation). Let v be a node of $\mathcal{T}$ that was created by

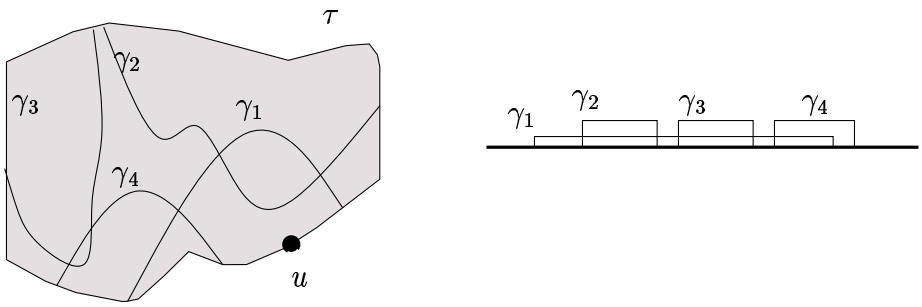

Figure 2: Mapping a set of arcs inside a trapezoid τ into a set of intervals. These arcs do not form a pseudo-segment arrangement because the pair γ_1, γ_2 (which correspond to two nested intervals) and the pair γ_3, γ_4 (which correspond to two disjoint intervals) intersect at two points each inside τ.

a call to `Expand` of the algorithm at the top level of recursion. The algorithm called this procedure because the arrangement inside the trapezoid Δ_v of v was not a pseudo-segment arrangement. This implies that C^* must have at least one point inside Δ_v. That is, the node v also exists in $\mathcal{T}'$ and was created during the (hypothetical) sequence of point-location queries that created that DAG. We conclude that all the expanded nodes of $\mathcal{T}$ exist in $\mathcal{T}'$; In particular, all the internal nodes of $\mathcal{T}$ exist in $\mathcal{T}'$, implying that the weight of $\mathcal{T}$ is at most proportional to the weight of $\mathcal{T}'$. However, by Corollary 3.5, the expected weight of $\mathcal{T}'$ is $O(\lambda_4(n\sqrt{\mu})\log n)$.

As for the expected running time, we note that it is dominated by the time it takes to check whether the arrangement inside each node of the history DAG is a pseudo-segment arrangement. As we will show next, this test, at a node v, takes expected $O(|\operatorname{cl}(v)|\log^3|\operatorname{cl}(v)|)$ time. Summing over all nodes, and denoting by $w(\mathcal{T}')$ the overall weight of $\mathcal{T}'$, we conclude that the expected running time is $O(w(\mathcal{T}')\log^3 n) = O(\lambda_4(n\sqrt{\mu})\log^4 n)$. ∎

Remark 5.2 Of course, when the points of C^* are close together, $w(\mathcal{T}')$ may be much smaller than the upper bound $O(\lambda_4(n\sqrt{\mu})\log n)$. In such a case, the algorithm will make fewer cuts and will run faster.

5.1 Testing for the Pseudo-Segment Property

To complete the description and analysis of our algorithm, we next show how to determine efficiently whether the given arrangement forms a pseudo-segment arrangement within a given trapezoid.

Let S be a collection of n pseudo-parabolas. Let τ be a trapezoid, and let S_τ denote the set of all arcs of S that cross τ. We clip each $\gamma \in S_\tau$ within τ, and consider separately each connected component of $\gamma \cap \tau$. Let S_τ^* denote the resulting collection of subarcs. Our goal is to determine whether S_τ^* is a collection of pseudo-segments, and we accomplish it as follows.

Let C denote the boundary of τ. Fix a point $u \in C$, not lying on any arc of S_τ, and consider C as an open arc, rather than a closed loop, that starts and ends at u and is oriented counterclockwise; see Figure 2. The endpoints of each arc $\gamma \in S_\tau^*$ lie on C. We refer to the first of these endpoints along the oriented C as the *initial point* of γ and to the second

endpoint as the *terminal point*. We denote these points as $i(\gamma)$ and $t(\gamma)$, respectively. Let I and T denote the sets of initial points and terminal points, respectively. We sort the points in $I \cup T$ in their order along C, and use the notation $a < b$, for $a, b \in I \cup T$, to mean that a precedes b along C. With each $\gamma \in S_\tau^*$ we associate an interval $e(\gamma)$, which is the subarc of C delimited by $i(\gamma)$ and $t(\gamma)$.

Observation: Let $\gamma, \gamma' \in S_\tau^*$. Then γ and γ' intersect exactly once (within τ) if and only if their intervals $e(\gamma)$, $e(\gamma')$ interleave. In other words, γ and γ' intersect at either zero or two points (within τ) if and only if their intervals are either disjoint or nested.

Hence, to check whether S_γ^* is a collection of pseudo-segments, it suffices to test every pair of arcs whose intervals are either disjoint or nested for intersection. S_τ^* is a collection of pseudo-segments if and only if there is no intersection between any of these pairs. Our approach is to take the collection of all these pairs, and decompose it into a disjoint union of complete bipartite subgraphs. We then test whether there exists an intersection within each of these subgraphs.

Consider the collection $\mathcal{N}$ of all ordered pairs (γ, γ') of arcs whose intervals are nested, that is, $e(\gamma) \subset e(\gamma')$. This condition can be expressed by the two inequalities $i(\gamma) > i(\gamma')$ and $t(\gamma) < t(\gamma')$. We construct a minimum-height binary tree Q on the set I, whose elements are stored at the leaves of Q in sorted order. Each node v of Q is associated with the subset $S^{(v)}$ of all arcs whose initial points are stored at the leaves of the subtree rooted at v. For each such node v, we construct a secondary tree $Q_v^{(2)}$ on the set of all terminal points of the arcs in $S^{(v)}$. For each internal node v of Q and for each internal node w of $Q_v^{(2)}$, we construct the complete bipartite graph $A_{vw} \times B_{vw}$, where A_{vw} (resp. B_{vw}) is the set of all arcs whose initial points are stored at the right (resp. left) subtree of Q rooted at v and whose terminal points are stored at the left (resp. right) subtree of $Q_v^{(2)}$ rooted at w. It is easily verified that each nested pair of intervals appears in exactly one of these graphs.

Put $n_\tau = |S_\tau^*|$. The above decomposition consists of $O(n_\tau \log n_\tau)$ graphs, so that the overall number of vertices of these graphs is $O(n_\tau \log^2 n_\tau)$.

The decomposition of the collection $\mathcal{D}$ of all ordered pairs (γ, γ') of arcs whose intervals are disjoint, that is, $e(\gamma)$ precedes $e(\gamma')$ along C, is obtained in a similar, and in fact simpler, manner. Indeed, since this condition can be expressed by the single inequality $t(\gamma) < i(\gamma')$, a single-level tree construction suffices. The tree is constructed over the full set $I \cup T$, and with each internal node v of the tree we associate a complete bipartite graph $A_v \times B_v$, where A_v (resp. B_v) is the set of arcs whose terminal (resp. initial) points are stored at the left (resp. right) subtree rooted at v. We have only $O(n_\tau)$ such graphs, whose overall number of vertices is $O(n_\tau \log n_\tau)$.

Let $A_{vw} \times B_{vw}$ be a complete bipartite graph of the first kind. It is easily seen that there exist two points $p < q \in C$, such that for each pair $(\gamma, \gamma') \in A_{vw} \times B_{vw}$, both endpoints of γ lie in the portion C_1 of C between p and q and both endpoints of γ' lie in the complement C_2, which is the union of the portions of C between u and p and between q and u. Similarly, for a graph $A_v \times B_v$ of the second kind, there exists a point $p \in C$, such that for each pair $(\gamma, \gamma') \in A_v \times B_v$, both endpoints of γ lie in the portion C_1 of C between u and p and both endpoints of γ' lie in the complementary portion C_2 between p and u.

We thus face the following situation. Viewing C again as a closed loop, we have a partition of C into two complementary arcs C_1, C_2, and two subsets of arcs $A, B \subset S_\tau^*$, such that both endpoints of each $\gamma \in A$ lie in C_1 and both endpoints of each $\gamma' \in B$ lie in C_2. Our goal is to determine whether there exists any intersecting pair of arcs in $A \times B$. This is done as follows.

Let p, q denote the common endpoints of C_1 and C_2. For each $\gamma \in A$, let K_γ denote the region enclosed between γ and C_1. Similarly, for each $\gamma' \in B$, let $K_{\gamma'}$ denote the region enclosed between γ' and C_2. Define $U_1 = \bigcup_{\gamma \in A} K_\gamma$ and $U_2 = \bigcup_{\gamma' \in B} K_{\gamma'}$. By appropriate slight perturbations of the boundary portions of these regions along C, we can turn each of the collections $\{K_\gamma\}_{\gamma \in A}$ and $\{K_{\gamma'}\}_{\gamma' \in B}$ into a collection of *pseudodisks* (namely, the boundaries of each pair cross at most twice). Hence (see [KLPS86, MMP+91], the complexity of U_1 is $O(|A|)$ and it can be constructed in randomized expected time $O(|A| \log |A|)$ (by a standard randomized incremental construction, see [Mul94], as the set original pseudo parabolas are x-monotone, and thus this amounts to the computation of upper/lower envelope), and similarly for U_2. We now test, in time $O((|A| + |B|) \log(|A| + |B|))$, whether ∂U_1 and ∂U_2 intersect. It is clear that $A \times B$ contains an intersecting pair of arcs if and only if these boundaries intersect.

Applying this procedure to every pair of graphs in our decomposition, we thus obtain:

Lemma 5.3 *One can determine, in randomized expected time $O(n_\tau \log^3 n_\tau)$, whether S_τ^* is a collection of pseudo-segments.*

Remark 5.4 The preceding analysis relied heavily on the fact that the given arcs are pseudo-parabolas. If the maximum number of intersections between any pair of arcs is $t > 2$ then two issues need to be addressed:

(1) The pseudodisk property of the regions $K(\gamma)$ no longer holds. However, this is not a real problem: Because the arcs are x-monotone, one can construct the unions U_1, U_2 efficiently and test for their intersection using lower and upper envelopes. We omit the easy details.

(2) A more serious problem is that now we also have to test for intersections between pairs of arcs whose intervals interleave. More precisely, each pair of such arcs intersect an odd number of times, and we need to determine whether there is a pair with more than one intersection. At the moment, we do not have any efficient procedure for doing this.

6 Eliminating Cycles of Rods in Space

Let S be a set of n pairwise-disjoint nonvertical line segments ('rods') in 3-space. A rod $s \in S$ lies below a rod $s' \in S$ if there exists a vertical line ℓ passing through s and s', so that the point $\ell \cap s$ lies below the point $\ell \cap s'$. (The transitive closure of) this relation is called the *depth order* of S. In general, it may contain cycles, and the problem we face, which arises in several applications, is to cut some of the rods of S into smaller pieces so as to obtain a new set S' of rods whose depth order contains no cycles. The goal is to make the smallest number of cuts needed to eliminate all cycles. This is a hard problem, so we seek any solution so

that the number of cuts that it makes approximates, or depends, on the minimum number of cuts. Recently, Solan [Sol98] presented an algorithm that makes $O(n^{1+\varepsilon}\sqrt{\mu})$ cuts, where μ is the smallest number of cuts, for any $\varepsilon > 0$. Using our point-location mechanism, we obtain an improved solution that makes only $O(n\sqrt{\mu}\alpha(n)\log n)$ expected number of cuts. It is easy to verify that $O(n^2)$ cuts are sufficient in general, and there are examples where $\Omega(n^{3/2})$ cuts are needed [Sha94]. However, the number of cuts needed might be considerably smaller. See also the earlier paper [CEG+92] for related results.

The algorithm works as follows. Let S^* denote the set of xy-projections of the rods in S. We construct a DAG $\mathcal{T}$ for S^* incrementally; initially, $\mathcal{T}$ consists of a single node, the root, that represents the whole plane. For each leaf v of $\mathcal{T}$, we take each rod $s \in S$ whose xy-projection crosses the trapezoid Δ_v of v, and clip s by intersecting it with the vertical prism erected over Δ_v. Let $S(v)$ denote the resulting set of rods. We run the algorithm of [dBOS94] for detecting depth-order cycles, on $S(v)$. The algorithm runs in time $O(|S(v)|^{4/3+\varepsilon})$, for any $\varepsilon > 0$.

If no cycle is detected, we do not expand $\mathcal{T}$ below v. Otherwise, we expand it, using the mechanism described above. We keep expanding $\mathcal{T}$ in this manner, until all leaves v of $\mathcal{T}$ are such that their corresponding sets $S(v)$ are cycle-free. Since the leaves of $\mathcal{T}$ always form a cover of the plane, it follows that the union of the sets $S(v)$, over the leaves v of $\mathcal{T}$, is a cutting of the rods of S in which all cycles are eliminated.

Theorem 6.1 *Let S be a set of n rods in 3-space, for which all depth-order cycles can be eliminated by making μ cuts. Then one can compute a cutting set for S of expected size $O(n\sqrt{\mu}\alpha(n)\log n)$, in expected time $O(n^{4/3+\varepsilon}\mu^{1/3})$, for any $\varepsilon > 0$.*

Proof: We first bound the expected number of cuts that the above algorithm makes. By the preceding discussion, it suffices to bound the overall expected weight of $\mathcal{T}$ upon termination. Let C be a set of μ cutting points on the rods of S that eliminate all cycles, and let C^* denote its xy-projection. Note that the trapezoid of each internal node v of $\mathcal{T}$ was either: (i) split because it contains at least one point of C^* (i.e., the arrangement induced inside it contained a cycle), or (ii) was generated by the execution of a call to `Expand`(u) by the top-level of `CompZoneOnline`, with an adjacent node u so that the arrangement induced inside Δ_u contained a cycle. In this latter case, v must lie on a path in $\mathcal{T}$ that connects the root of $\mathcal{T}$ with u (see [HP99b] for the proof of this claim). It follows that $\mathcal{T}$ is contained in the DAG $\mathcal{T}_0$ that would have been produced if we performed point-location queries with the (unknown) points of C^* in $\mathcal{A}(S^*)$. As argued above, the total expected weight of $\mathcal{T}_0$ is $O(\lambda_{t+2}(n + w(C^*, S^*) + |C^*|)\log n)$. Hence this is also an upper bound on the number of cuts that our algorithm makes. As noted earlier, we always have $w(C^*, S^*) = O(n\sqrt{|C|}) = O(n\sqrt{\mu})$, which implies the first assertion of the theorem.

As is easily verified, the expected running time of the algorithm is dominated by the bound on the time required to test whether a clipped set $S(v)$ contains a cycle, summed over all nodes v of $\mathcal{T}$. To bound this quantity, we interpret the history DAG $\mathcal{T}$ as the history DAG that would have been generated by an off-line randomized incremental algorithm that uses an oracle for determining which of the new trapezoids that it generates contain cycles. The set of trapezoids maintained by this algorithm fits the settings of the analysis by de Berg et al. [dBDS95] and by Agarwal et al. [AMS98] (these papers consider an extension of

the standard setting of randomized incremental geometric constructions due to Clarkson and Shor [CS89]). In particular, after the i-th iteration of the algorithm, the expected number of 'active' trapezoids (leaves of the DAG) maintained by the algorithm is $O(\min(i^2, \lambda_{t+2}(i + (i/n)w)))$, where w is the minimum weight of a Steiner-tree connecting the points of the optimal solution. Indeed, the first term is a bound on the complexity of the whole projected arrangement, whereas the second term bounds the expected complexity of the zone of the minimum Steiner-tree, which is an upper bound on the number of active trapezoids; the bound on the zone complexity follows from the fact that the expected number of crossings between the spanning tree and the first i arcs is $(i/n)w$, which implies that the expected number of trapezoids in the zone of the spanning tree in $\mathcal{A}(S_i)$ is $O(\lambda_{t+2}(i + (i/n)w))$.

By [dBDS95, AMS98], the overall expected work involved in checking for each of those trapezoids whether it contains a cycle is

$$O\left(\min\left(i^2, \lambda_{t+2}(i + (i/n)w)\right) \cdot \left(\frac{n}{i}\right)^{4/3+\varepsilon}\right).$$

Thus, using backward analysis [Sei93], the expected number of trapezoids created in the i-th iteration is $O(\min(i, \lambda_{t+2}(i + (i/n)w)/i))$, and the expected work required by this iteration is

$$O\left(\min\left(i, \frac{\lambda_{t+2}(i + iw/n)}{i}\right) \cdot \left(\frac{n}{i}\right)^{4/3+\varepsilon}\right).$$

Thus, the expected overall running time of the algorithm is

$$O\left(\sum_{i=1}^{w/n} i \cdot \left(\frac{n}{i}\right)^{4/3+\varepsilon} + \sum_{i=w/n+1}^{n} \frac{\lambda_{t+2}(i + iw/n)}{i} \cdot \left(\frac{n}{i}\right)^{4/3+\varepsilon}\right) = O\left(n^{4/3+\varepsilon} + n^{2/3+\varepsilon} w^{2/3+\varepsilon}\right),$$

as can be easily verified. Since w is at most $O(n\sqrt{\mu})$, we obtain that the expected running time of the algorithm is $O(n^{4/3+\varepsilon}\mu^{1/3+\varepsilon}) = O(n^{4/3+\varepsilon}\mu^{1/3})$, since $\mu = O(n^2)$. This completes the proof. ∎

Remark 6.2 (a) Theorem 6.1 improves the bound on the number of cuts over the previous algorithm of Solan [Sol98], and both algorithms have the same asymptotic expected running time. Moreover, the number of cuts performed by the algorithm is a nearly-linear function of $w = w(C^*, S^*)$, which may be considerably smaller than the estimate used in Theorem 6.1. The expected running time of the algorithm is also a function of w (that is, it is $O\left(n^{4/3+\varepsilon} + n^{2/3+\varepsilon} w^{2/3+\varepsilon}\right)$), so the algorithm will be faster when w is small.

(b) As in the preceding sections, the structure produced by the algorithm is somewhat degenerate, because portions of the trapezoid boundaries that it generates are projections of portions of the given rods. In particular, some of the cutting points may lie directly above or below another rod. In case this is undesirable, an appropriate slight perturbation of these points can be applied.

7 Computing Bichromatic Intersections

Let R and B be two sets of 'red' and 'blue' x-monotone arcs, as above, so that the union $\mathcal{U}_R$ of the arcs of R and the union $\mathcal{U}_B$ of the arcs of B are both connected. Let t be the

maximum number of intersections between any pair of arcs of $R \cup B$, and let n denote the overall number of arcs. Let $R_{\mathcal{VD}} = \mathcal{A}_{\mathcal{VD}}(R)$ and $B_{\mathcal{VD}} = \mathcal{A}_{\mathcal{VD}}(B)$ denote, respectively, the vertical decompositions of the arrangements $\mathcal{A}(R)$ and $\mathcal{A}(B)$. A trapezoid $\Delta \in R_{\mathcal{VD}}$ (resp. $\Delta \in B_{\mathcal{VD}}$) is called *hot* if the interior of Δ intersects one of the blue (resp. red) arcs of B (resp. R). Let k denote the number of bichromatic intersections between the red and the blue arcs.

To compute all the k bichromatic intersections, we perform a simultaneous sweep of the blue and red arrangements, and maintain hot trapezoids along the y-structure of the sweep. To detect hot trapezoids, we will use the data-structure of Section 3 for online point-locations in these two arrangements. Our algorithm can be viewed as a straightforward adaption of the algorithm of Basch et al. [BGR96], so that it uses our online data-structure and thereby achieves better performance (see the introduction and Remark 7.6 below for comparison with existing works).

Lemma 7.1 *The number of hot trapezoids in $B_{\mathcal{VD}}$ and $R_{\mathcal{VD}}$ is $O(\lambda_{t+2}(n+k))$.*

Proof: Let B' be the set of arcs generated from the arcs of B, by breaking each blue arc into several subarcs, at its intersection points with the red arcs. Clearly, $|B'| = O(n+k)$. All the red arcs now lie in a single face f of $\mathcal{A}(B')$, and the number of hot trapezoids in $B_{\mathcal{VD}}$ is at most proportional to the complexity of this face. Indeed, the collection of hot trapezoids covers f (viewed as a collection of faces of $\mathcal{A}(B)$), so their number is proportional to the complexity of these faces (in $\mathcal{A}(B)$), which is smaller than or equal to the complexity of f (as a single face of $\mathcal{A}(B')$). The lemma thus follows from the fact that the complexity of this face is $O(\lambda_{t+2}(n+k))$; see [SA95].

Clearly, the same bound also holds for the number of hot trapezoids in $R_{\mathcal{VD}}$. ∎

Definition 7.2 Two arcs $\beta \in B, \rho \in R$ are *visible* at x_0, if the two arcs intersect the vertical line $\ell : \ x = x_0$, and no other arc of $R \cup B$ intersects ℓ between those two arcs. A pair of arcs (β, ρ) are visible, if there is a value of x so that β and ρ are visible at x.

A triple (β, ρ, x_0), for $\beta \in B$, $\rho \in R$, $x \in \mathbb{R}$ is *a visibility triple* if β and ρ are visible at x_0, and are not visible immediately to the left of x_0.

Lemma 7.3 *The number of visibility triples is $O(\lambda_{t+2}(n+k))$.*

Proof: We charge each visibility triple to a hot trapezoid of either $B_{\mathcal{VD}}$ or $R_{\mathcal{VD}}$, or to a bichromatic intersection point. By Lemma 7.1, the number of charged entities is $O(\lambda_{t+2}(n+k))$.

For a visibility triple (β, ρ, x_0), if the vertical segment connecting β and ρ along the line $x = x_0$ is contained in either the right or the left side of a trapezoid of $B_{\mathcal{VD}}$ or $R_{\mathcal{VD}}$, we can charge this triple to the relevant trapezoid, since there are at most 2 visibility triples involving the same trapezoid at the same x-coordinate.

Otherwise, β and ρ became visible (as we sweep from left to right) because one of the arcs became "suddenly" visible to the other arc; namely, to the left of x_0, either β was occluded by a red arc from seeing ρ, or ρ was occluded by a blue arc from seeing β. In either case, the only way a visibility triple can be created is by a bichromatic intersection involving the occluded arc. (The other possibilities, where the occluding arc starts or ends at x_0, or where

one of the arcs, say β, is occluded from the other arc by an arc β' of the same color and the occlusion stops when β and β' intersect, have already been treated, because the endpoint of the occluding arc or the monochromatic intersection of the occluding and occluded arc both induce a vertical side of a hot trapezoid at x_0.) Thus, we charge this visibility triple to this bichromatic intersection point. It is easy to verify that, under a general position assumption, each bichromatic intersection is charged only $O(1)$ times. ∎

Observation 7.4 *Let $p = \beta \cap \rho = (x_p, y_p)$ be a bichromatic intersection of $\beta \in B, \rho \in R$. Then there exists a visibility triple (β, ρ, x_0), so that $x_0 < x_p$.*

We will compute all the bichromatic intersections by sweeping $\mathcal{A}(B), \mathcal{A}(R)$ simultaneously with a vertical line from left to right, and by maintaining the visibility information along the vertical sweeping line. It is easy to verify that the visibility information changes either at the x-coordinate of a visibility triple, or at a bichromatic intersection.

To maintain all the visible pairs of arcs β, ρ at the current location of the sweeping line, we maintain online data-structures $\mathcal{D}_B, \mathcal{D}_R$ for point location in $\mathcal{A}(B)$ and $\mathcal{A}(R)$, respectively, using the technique described in Sections 2 and 3.

Given a point $p \in \mathcal{U}_B$, we can check whether it is vertically visible from an arc of R as follows: Perform an upward and downward vertical ray-shooting queries from p in $\mathcal{D}_R$, and let $q_{top}, q_{bot} \in \mathcal{U}_R$ be the points returned. (To answer these queries, we simply perform point location with p in $\mathcal{A}(R)$; the top and bottom arcs of the output trapezoid that contains p are the answers to the queries.) Now perform a downward vertical ray-shooting query from q_{top} in $\mathcal{D}_B$ (using the same mechanism), and let $q'_{top} \in \mathcal{U}_B$ be the returned point. Then p is visible from a red arc from above if and only if $q'_{top} = p$. Similarly, if we perform an upward vertical ray-shooting query from q_{bot}, and q'_{bot} is the returned point, then p is visible from a red arc from below if and only if $q'_{bot} = p$.

To perform the sweep, we insert all the endpoints of the arcs of $B \cup R$ into the x-structure event queue. For a visible pair of arcs, we maintain the corresponding pair of vertical trapezoids of $B_{\mathcal{VD}}, R_{\mathcal{VD}}$ that contain the vertical connecting segment between the arcs. For each such trapezoid, we insert its bottom right vertex into the event queue. For any visible pair of arcs, we check whether they have any bichromatic intersections to the right of the current sweepline, and if so we insert these intersections into the event queue.

Maintaining the visibility during the sweep is now straightforward. Each event in the x-structure queue is either an endpoint of an arc, a bichromatic intersection, or the right side of a hot trapezoid. In each of these cases we need to construct new hot trapezoids that materialize to the right of the event, which we can do efficiently by performing appropriate point-location queries in $\mathcal{D}_R, \mathcal{D}_B$. We omit the somewhat tedious though straightforward details.

The key to the above algorithm are the structures $\mathcal{D}_R, \mathcal{D}_B$, which provide "cheap" tracking of the changes in the visibility information. Without these data-structures one may have to work much harder, as in [BGR96].

We note that, by a careful implementation of the algorithm, all the point-location queries in $\mathcal{D}_B$ (resp. $\mathcal{D}_R$) were performed by points lying in $\mathcal{U}_R$ (resp. in $\mathcal{U}_B$), namely, in the zone of $\mathcal{U}_R$ (resp. $\mathcal{U}_B$) in $\mathcal{A}(B)$ (resp. $\mathcal{A}(R)$). Let P_R denote the set of query point-locations performed in $\mathcal{D}_B$. Furthermore, each point-location can be charged to a visibility triple, and

such a triple is charged only a constant number of times. Thus, by Lemma 7.3, we conclude that $|P_R| = O(\lambda_{t+2}(n+k))$.

Let $\mathcal{M}_R = \mathcal{M}(P_R, B)$ be the minimum weight Steiner-tree of the points of P_R in the arrangement $\mathcal{A}(B)$. Clearly, $w(\mathcal{M}_R) \leq k$, as $\mathcal{U}_R$ is a connected set having k intersections with $\mathcal{U}_B$. Define $\mathcal{M}_B = \mathcal{M}(P_B, R)$ in complete analogy; its weight is also at most k.

We therefore conclude that the expected time required to execute those $O(\lambda_{t+2}(n+k))$ point-location queries in $\mathcal{D}_B$ is $O(\lambda_{t+2}(n+k)\log n)$, by Theorem 3.2 and Lemma 2.2. As is easy to verify, all the other operations carried out by the algorithm also require $O(\lambda_{t+2}(n+k)\log n)$ expected time. We conclude:

Theorem 7.5 *Let R and B be two sets of a total of n red and blue x-monotone arcs, as above, so that the union of the arcs of R and the union of the arcs of B are both connected. Then one can compute, in $O(\lambda_{t+2}(n+k)\log n)$ randomized expected time, all the k bichromatic intersections between the red and blue arcs.*

Remark 7.6 Previously, an $O((n+k)\log^3 n)$-time algorithm for the case of segments was given by [BGR96]. Chan [Cha99] mentions that his data-structure for the maintenance of intersection of half-planes can be used to get an algorithm with $O((n+k)\log^{2+\varepsilon} n)$ (deterministic) running time for the above red-blue intersection problem. For the case general arcs, an $O(\lambda_{t+2}(n+k)\log^3 n)$-time algorithm is presented in [BGR96].

8 Conclusions

In this paper we have presented several new applications of the technique of [HP99b]. In all the cases studied here the new solutions are faster than what was previously known. We believe that the technique of [HP99b] can be used for other applications; in particular, we believe it can be extended to three dimensions.

Underlining the analysis of all these applications is the 'invisible' parameter, which is the minimum weight of a Steiner-tree of a set of points in a planar arrangement of arcs, under the arc-intersection distance defined above. The performance of our algorithm and its applications depends, in an almost-linear fashion, on this parameter. In fact, in some applications, such as that of eliminating cycles in a set of rods, or of cutting pseudo-parabolas into pseudo-segments, even the points that define this tree are 'invisible', and yet the algorithm performs as if it knew where these points are. We believe that a better understanding of the structure and behavior of this minimum-weight Steiner-tree could lead to further progress on the problems studied and on other related problems.

Recently, using the results in this paper, Har-Peled and Indyk [HPI00] showed how to ε-approximate the minimum spanning tree of a set of points under the intersection metric defined by a set of lines in the plane. A critical component in achieving a near-linear running time was the linear dependency on the 'invisible' minimum tree weight mentioned above.

We conclude by mentioning the following open problems:

- Can one compute all k bichromatic intersections in the setup of Section 7 for the case of segments, in $O(n\log n + k)$ time?

- Can one use the hitting set technique of [BG95] (see also [PA95] and Appendix A) to get a better approximation for the problem of eliminating cycles of rods in space?
- Can one speed up the algorithms presented in this paper by using Chazelle's hierarchical cuttings [Cha93] instead of using the technique of [HP99b]?

Acknowledgments

The authors wish to thank Pankaj Agarwal and Danny Halperin for helpful discussions concerning the problems studied in this paper and related problems. Finally, the authors thank the anonymous referees for numerous useful comments.

References

[AdBC+99] T. Asano, M. de Berg, O. Cheong, L.J. Guibas, J. Snoeyink, and H. Tamaki. Spanning trees crossing few barriers. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 41–48, 1999.

[Aga91] P. K. Agarwal. *Intersection and Decomposition Algorithms for Planar Arrangements*. Cambridge University Press, New York, NY, 1991.

[AMS98] P. K. Agarwal, J. Matoušek, and O. Schwarzkopf. Computing many faces in arrangements of lines and segments. *SIAM J. Comput.*, 27(2):491–505, 1998.

[BDH99] K.-F. Böhringer, B. Donald, and D. Halperin. The area bisectors of a polygon and force equilibria in programmable vector fields. *Discrete Comput. Geom.*, 22(2):269–285, 1999.

[BG95] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete Comput. Geom.*, 14:263–279, 1995.

[BGR96] J. Basch, L.J. Guibas, and G.D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proc. 4th Annu. European Sympos. Algorithms*, volume 1136 of *Lecture Notes Comput. Sci.*, pages 302–319. Springer-Verlag, 1996.

[BY98] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by H. Brönnimann.

[CEG+92] B. Chazelle, H. Edelsbrunner, L.J. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink. Counting and cutting cycles of lines and rods in space. *Comput. Geom. Theory Appl.*, 1:305–323, 1992.

[CEG+93] B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments. *SIAM J. Comput.*, 22:1286–1302, 1993.

[Cha93] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.

[Cha99] T.M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 92–99, 1999.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[dBDS95] M. de Berg, K. Dobrindt, and O. Schwarzkopf. On lazy randomized incremental construction. *Discrete Comput. Geom.*, 14:261–286, 1995.

[dBOS94] M. de Berg, M. Overmars, and O. Schwarzkopf. Computing and verifying depth orders. *SIAM J. Comput.*, 23:437–446, 1994.

[dBvKOS97] M. de Berg, M. van Kreveld, M. H. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[EGP+92] H. Edelsbrunner, L. J. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir. Arrangements of curves in the plane: Topology, combinatorics, and algorithms. *Theoretical Computer Science*, 92:319–336, 1992.

[HP99a] S. Har-Peled. Multicolor combination lemma. *Comput. Geom. Theory Appl.*, 12:155–176, 1999.

[HP99b] S. Har-Peled. Taking a walk in a planar arrangement. In *Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci.*, 1999. 100-110.

[HP00] S. Har-Peled. Constructing planar cuttings in theory and practice. *SIAM J. Comput.*, 29(6):2016–2039, 2000.

[HPI00] S. Har-Peled and P. Indyk. When crossings count - approximating the minimum spanning tree. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 166–175, 2000.

[KLPS86] K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Comput. Geom.*, 1:59–71, 1986.

[Lov75] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Math.*, 13:383–390, 1975.

[MMP+91] J. Matoušek, N. Miller, J. Pach, M. Sharir, S. Sifrony, and E. Welzl. Fat triangles determine linearly many holes. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 49–58, 1991.

[Mul94] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[PA95] J. Pach and P.K. Agarwal. *Combinatorial Geometry.* John Wiley & Sons, New York, NY, 1995.

[SA95] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.

[Sei93] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.

[Sha94] M. Sharir. On joints in arrangements of lines in space and related problems. *Journal of Combinatorial Theory*, 67(1):89–99, 1994.

[Sol98] A. Solan. Cutting cylces of rods in space. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 135–142, 1998.

[TT98] H. Tamaki and T. Tokuyama. How to cut pseudo-parabolas into segments. *Discrete Comput. Geom.*, 19:265–290, 1998.

A Cutting Pseudo-Parabolas and More General Arcs into Pseudo-Segments—A Slower Algorithm that Produces a Near-Optimal Solution for General Arcs

Let S be a collection of n general x-monotone arcs, each pair of which intersect at most t times. In this subsection we present an alternative algorithm, not based on the online point-location mechanism, that constructs a cutting set for S that turns it into a pseudo-segment arrangement, whose size is nearly optimal. The algorithm is a variant of the greedy algorithm for the hitting set problem, and a naive implementation of it will require $O(n^3)$ time. Our more careful implementation achieves $O(n^2 \log n)$ running time.

Intersect each curve $s \in S$ with all the other curves. For a pair of points $p, q \in s$, with p to the left of q, denote by $s[p, q]$ the subarc of s delimited by p and q. For any other $s' \in S$, let $p_1, \ldots, p_k$, for $k \leq t$, denote the points of intersection of s and s', sorted from left to right. If $k \geq 2$, we associate with s' the $k-1$ subarcs $s[p_i, p_{i+1}]$, for $i = 1, \ldots, k-1$, along s. If $k = 1$ we ignore the interaction between s and s'. Repeating this process over all s, s', we obtain a system $\mathcal{I}_s$ of subarcs along each curve $s \in S$. Define the *depth* of a point p lying on a curve $s \in S$ to be the number of subarcs of $\mathcal{I}_s$ that contain p.

We construct a cutting set for S using the following greedy algorithm:

(a) Find the deepest point p over all curves of S.

(b) Cut the curve s containing p at p.

(c) For each subarc $\gamma = s[a, b] \in \mathcal{I}_s$ that contains p, do the following: (i) Delete γ from $\mathcal{I}_s$. (ii) Let s' be the other curve that induces γ; delete $s'[a, b]$ (the 'sibling' subarc induced by s along s') from $\mathcal{I}_{s'}$.

(d) Repeat steps (a)–(c) until all sets $\mathcal{I}_s$ become empty.

It is clear that the algorithm produces a cutting set that creates a pseudo-segment arrangement. Using well known results on the performance of the greedy algorithm for the hitting set problem in hypergraphs (see, e.g., [PA95, Theorem 15.2], [Lov75]), it follows that the size of the cutting set produced by this algorithm is $O(\mu \log d)$, where μ is the size of the minimum cutting set and where d is the maximum depth of a point.

To derive a reasonably efficient (near-quadratic) implementation of this algorithm, we proceed as follows. The initial construction of the sets $\mathcal{I}_s$, for $s \in S$, can be done in $O(n^2 \log n)$ time. For each $s \in S$, store the subarcs of $\mathcal{I}_s$ in a segment tree T_s, and augment the tree so that each node v of T_s stores the deepest point on s (and its depth) which is contained in the subinterval associated with the subtree of T_s rooted at v. The initial construction of all these augmented segment trees can be done in $O(n^2 \log n)$ time. Finally, the roots of all the trees T_s are stored in a heap, ordered by the keys $depth(s)$ = the depth of the deepest point on s, for $s \in S$.

Steps (a) and (b) are easy to perform in $O(\log n)$ time. Let p and s be the point found in (a) and the curve containing it, respectively. The set of all subarcs of $\mathcal{I}_s$ that contain p can be obtained as the disjoint union of $O(\log n)$ lists, stored at the nodes of T_s on the path from the root to p. We pick up all these lists and iterate over each of their subarcs, deleting it from T_s and deleting its sibling subarc from the appropriate tree $T_{s'}$. The deletion of an arc from a tree can be done in $O(\log n)$ time, including the update of the depth pointers at the relevant nodes of the tree (there are only $O(\log n)$ nodes to update, and their parents lie on two paths of the tree). Since the overall initial size of all the sets $\mathcal{I}_s$ is $O(n^2)$ and we only perform deletions, the total running time of the algorithm is $O(n^2 \log n)$ time. We thus obtain the following result.

Theorem A.1 *Given a set S of n arcs, as above, one can construct, in $O(n^2 \log n)$ time, a cutting set for S that turns the arcs into an arrangement of pseudo-segments, whose size is $O(\mu \log d)$, where μ is the minimum size of such a cutting set and where d is the maximum depth of a point.*