A robust and efficient implementation for the segment Voronoi diagram

Menelaos I. Karavelas
University of Notre Dame, Computer Science and Engineering Department
Notre Dame, IN 46556, U.S.A.
mkaravel@cse.nd.edu

Abstract

In this paper we present an efficient algorithm for the computation of the segment Voronoi diagram in two dimensions. Our algorithm can handle not only disjoint segments or segments that share endpoints, but also segments that may intersect at their interior. It is incremental and the expected cost of inserting n (possibly intersecting) sites (points or segments) is $O((n+m)\log^2 n)$, where m is the number of points of intersection of the (open) segments in the input site set. Finally, we describe the implementation of our algorithm, that uses techniques such as geometric filtering, and present experiments that show the robustness, efficiency and scalability of our implementation.

Keywords: Segment Voronoi diagram; Voronoi diagram hierarchy; geometric filtering

1. Introduction

The Euclidean Voronoi diagram for a set of linear segments is one of the most well studied structures in computational geometry. Applications include biology, computer graphics, pattern recognition, motion planning, shape representation, mesh generation and NC machining (cf. [Kir79, Lee82, BY98, Hel01] and the references therein).

The algorithms proposed so far assume that the segments are either disjoint or that they are allowed to intersect only at endpoints and use a variety of algorithmic paradigms. Drysdale and Lee [DL78] show how to compute the segment Voronoi diagram in time $O(n\log^2 n)$, where n is the number of input segments. Kirkpatrick [Kir79], Lee [Lee82] and Yap [Yap87] present worse-case optimal $O(n\log n)$ divide-and-conquer algorithms for this problem. Another worst-case optimal algorithm, using the sweep-line paradigm, is described by Fortune [For87]. Boissonnat et al. [BDS+92] and Klein et al. [KMM93] provide $O(n\log n)$ randomized incremental algorithms for computing the segment Voronoi diagram. The algorithm in [BDS+92] was later on made dynamic by Dobrindt and Yvinec [DY93]. Finally, Alt and Schwarzkopf [AS95] describe a randomized incremental algorithm for constructing the Voronoi diagram of a set of planar curve segments, a special case of which are line segments. Its expected running time is $O(n\log n)$, but it requires the computation of a Voronoi diagram for points, one per curve (line) segment to be inserted in the diagram.

In contrast to the above-mentioned algorithms, which assume that numerical computations are performed exactly, Imai [Ima96], Sugihara et al. [SIII00] and Held [Hel01] present algorithms and implementations for computing the segment Voronoi diagram using floating-point arithmetic. The resulting diagram may not be the exact one, due to the numerical errors introduced by the floating-point arithmetic, but it is guaranteed to have the correct topology of a Voronoi diagram. The only exact implementation that we are aware of for computing the segment Voronoi diagram is that by M. Seel [See]. It requires the input to be homogeneous points of integer coordinates and it is essentially the adaptation to the case of line segments of the algorithm in [KMM93] for abstract Voronoi diagrams. Burnikel et al. [BMS94] present a detailed study of the numerical precision required for the `incircle` test of this algorithm using exact arithmetic, while the remaining predicates are analyzed by Burnikel in his thesis [Bur96].

In this paper we present a new randomized incremental algorithm for computing the Voronoi diagram for a set $\mathcal{S}$ of points and segments. It extends the generic paradigm presented in Karavelas

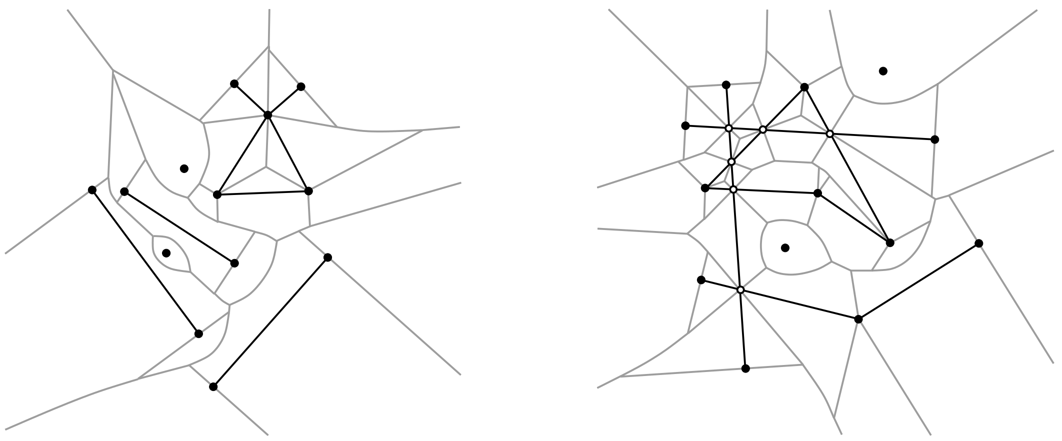

Figure 1. The Voronoi diagram (gray) for a set of 8 closed segments and 2 points (black). Left: weakly intersecting sites. Right: strongly intersecting sites; the points with white interior are points of intersection.

and Yvinec [KY03], where the Voronoi diagram for a set of (piecewise) smooth disjoint convex objects is computed using a hierarchy of Voronoi diagrams as a location data structure. The fact that we want to allow segments to intersect at endpoints or at their interior, makes the extension from this generic paradigm non-trivial, since we have to build the hierarchy of Voronoi diagrams in a consistent way. As a result, analyzing the hierarchy and the cost of insertion of a single site becomes much more complicated.

When we allow the input segments to intersect at their interior the resulting Voronoi diagram is no longer an instance of an abstract Voronoi diagram (the bisectors are no longer homeomorphic to a line), and the classical Voronoi diagram theory does not apply. Moreover, the algorithm in [KY03] requires a preprocessing step, thus it is no longer incremental. The key idea is to consider the points and segments in the arrangement $\mathcal{A}(\mathcal{S})$ of $\mathcal{S}$: the segments in $\mathcal{A}(\mathcal{S})$ are now disjoint or intersect only at their endpoints. Our algorithm exploits this fact. It maintains the invariant that the Voronoi diagram of the sites that have already been inserted is in fact the Voronoi diagram of the sites in the arrangement of the input. This is achieved by appropriately splitting, in an on-line fashion, the input segments, as well as the segments already inserted in the diagram, into subsegments, using their points of intersection (cf. Fig. 1). The expected running time of our algorithm is $O((n+m)\log^2 n)$, where n is the size of the input set $\mathcal{S}$ and $m = O(n^2)$ is the complexity of the arrangement $\mathcal{A}(\mathcal{S})$.

Unlike the case of segments that intersect only at endpoints, the question of how to represent the sites in the diagram becomes important. If points of intersection are represented by their coordinates, their bit complexity can increase exponentially with respect to the size of the input. To this effect, we describe how to represent the sites in the Voronoi diagram in a consistent manner that takes into consideration knowledge about the history of construction of the sites in the diagram. Moreover, this representation guarantees that the bit complexity of every site in the diagram is a constant multiple of the bit complexity of the input, thus independent of the size of the input. This representation is coupled with a technique we call *geometric filtering* (Wein [Wei02] has referred to this approach as “high-level” filtering, in contrast to “floating-point” or “arithmetic” filtering). Geometric filtering consists of exploiting the representation of the sites in order to filter out seemingly degenerate configurations without resulting to exact arithmetic.

One final novel aspect of this work is the implementation of our algorithm. To the best of our knowledge, this is the first truly generic and realistically efficient implementation for the segment Voronoi diagram based on the exact computation paradigm.

The rest of the paper is structured as follows. In Section 2 we provide some basic definitions. In Section 3 we describe our algorithm, and in Section 4 we give a brief complexity analysis. Section 5 discusses how points and segments are represented and how this representation is coupled with our filtering technique. Finally, in Section 6 we present our implementation and experimental results obtained by it.

2. Definitions

Let $\mathcal{S}$ be a set of distinct disjoint sites, where a *site* $S \in \mathcal{S}$ is either a closed linear segment t or a point p. We refer to the interior of a segment t, denoted by t°, as an *open segment*. The interior of a point is the empty set. We say that two sites, open or closed, *intersect* if they share at least one common point. In this paper we only deal with sites such that each pair has at most one common point. The case of pairs of intersecting sites having more than one common point can be dealt with within our framework easily, but due to space limitations we will not discuss it. We say that two closed intersecting sites *weakly intersect* if their common point does not lie in the interior of any of the two sites. Finally, we say that two closed intersecting sites *strongly intersect* if their common point lies in the interior of at least one of the two sites.

The distance of a point $x \in \mathbb{E}^2$ from a closed site S_i is defined as $\delta(x, S_i) = \min\{\|x - y\| : y \in S_i\}$. Let $H_{ij} = \{x \in \mathbb{E}^2 : \delta(x, S_i) \leq \delta(x, S_j)\}$. The bisector π_{ij} of S_i and S_j, i.e., the locus of points at equal distance from S_i and S_j is then the set $H_{ij} \cap H_{ji}$. Since the set $\mathcal{S}$ contains only disjoint sites, the bisectors are curves homeomorphic to the line. The *Voronoi cell* $V(S_i)$ is defined to be the set of points in $x \in \mathbb{E}^2$ that are closer (or at equal distance) to S_i than to any other site S_j in $\mathcal{S}$, i.e., $V(S_i) = \cap_{i \neq j} H_{ij}$. The connected sets of points that belong to the intersection of exactly two Voronoi cells are called *Voronoi edges*, whereas the the points that belong to at least three Voronoi cells are called *Voronoi vertices*. The subdivision of the plane into Voronoi vertices, edges and cells is called the *Voronoi diagram* $\mathcal{V}(\mathcal{S})$ of $\mathcal{S}$. The collection of Voronoi vertices and edges is called the 1-skeleton $\mathcal{V}_1(\mathcal{S})$ of $\mathcal{S}$.

If sites are allowed to weakly intersect the bisectors π_{ij} can become two-dimensional. The standard technique for avoiding two-dimensional bisectors is to consider segments not as one object, but rather as three, namely, the two endpoints and the open segment (e.g., cf. [Bur96]). For the Voronoi diagram to be well defined, we now need to define the bisector of an open segment t_i° and its endpoint p_j. In this case the bisector π_{ij} of t_i° and p_j is the line perpendicular to t_i° that passes through p_j. The set H_{ij} is the closed halfplane delimited by π_{ij} that contains t_i°. Moreover, we have to distinguish between the set $\mathcal{S}_\mathcal{I}$ of input sites and the set $\mathcal{S}$ of sites in the Voronoi diagram. $\mathcal{S}_\mathcal{I}$ will always consist of closed sites, whereas $\mathcal{S}$ consists of the points and open segments in the arrangement of $\mathcal{S}_\mathcal{I}$. Given these modifications in the definition of H_{ij} we can now define the Voronoi diagram for a set $\mathcal{S}_\mathcal{I}$ of weakly intersecting sites in exactly the same manner as for disjoint sites.

Let us now consider two weakly intersecting sites S_i, S_j. A circle tangent to both S_i and S_j is a *bitangent Voronoi circle* C_{ij}. Given three sites S_i, S_j and S_k, a circle tangent to all three of them is called a *tritangent Voronoi circle* C_{ijk}. Points on Voronoi edges are centers of bitangent Voronoi circles, whereas Voronoi vertices are centers of tritangent Voronoi circles. Let now $S \notin \mathcal{S}_\mathcal{I}$. We say that S is in conflict with a (bitangent or tritangent) Voronoi circle C if S intersects the interior of the disk bounded by C. Moreover, S is in conflict with a Voronoi edge $e \in \mathcal{V}(\mathcal{S})$, if it is in conflict with at least one of the disks bounded by the Voronoi circles centered on e. We can then define the conflict region $\mathcal{R}_\mathcal{S}(S)$ of S to be the set of points on the 1-skeleton $\mathcal{V}_1(\mathcal{S})$ corresponding to Voronoi circles that are in conflict with S.

3. The algorithm

In this section we describe our algorithm for computing the Voronoi diagram for a set of possibly intersecting sites. We start by describing our algorithm for the case of weakly and then for strongly intersecting sites. We then discuss the location data structure for weakly intersecting sites and we conclude by presenting the modifications needed in order to handle strongly intersecting sites. In the text below, $\mathcal{S}$ will denote the set of sites that have already been inserted in the diagram, and S will be the site to be inserted.

3.1. Weakly intersecting sites

The insertion procedure consists of four steps:

1. Find the first conflict of the site S with the Voronoi skeleton $\mathcal{V}_1(\mathcal{S})$.

2. Find the entire conflict region of S with $\mathcal{V}_1(\mathcal{S})$.

3. Construct the Voronoi diagram of $\mathcal{V}(\mathcal{S} \cup \{S\})$.

4. Update the location data structure.

We will postpone the discussion on nearest neighbor queries and the location data structure updates until the next section. We will first describe the insertion procedure if the site S is a point p. The first step of the insertion starts by finding the nearest neighbor $N_{\mathcal{S}}(p)$ of p among the sites in $\mathcal{S}$ (if p is at equal distance to more than one site, any of these sites can be chosen arbitrarily). Once $N_{\mathcal{S}}(p)$ has been determined, one of two things can happen. If $N_{\mathcal{S}}(p)$ is a point and p is the same point as its nearest neighbor, there is nothing to be done: the point has already been inserted in the diagram. Otherwise, p has to be in conflict with at least one of the edges on the boundary of $V(N_{\mathcal{S}}(p))$. The first step of the insertion then terminates by considering the Voronoi edges on the boundary of $V(N_{\mathcal{S}}(p))$, and returning one in conflict with p.

The second step of the insertion is based on the fact that the conflict region $\mathcal{R}_{\mathcal{S}}(p)$ of p is a simply connected subset of $\mathcal{V}_1(\mathcal{S})$ – possibly through the edges on the boundary of the cell of the site at infinity (cf. [KMM93, KY03]). There are two possibilities as to the type of conflict of p with the Voronoi edge e that was found at the end of the first step. Either p is in conflict with at least one of the endpoints of e or it is in conflict with a simply connected subset of the interior of e. In the latter case we have found the entire conflict region of p with respect to $\mathcal{V}(\mathcal{S})$. In the former case though, i.e., when one of the endpoints of e is in conflict with p, we need to find the rest of the conflict region. This is done by performing a DFS on the Voronoi skeleton, until all Voronoi edges in conflict with p are found.

At the end of the second step we have discovered the entire conflict region $\mathcal{R}_{\mathcal{S}}(p)$, and in particular, we have discovered its boundary $\partial\mathcal{R}_{\mathcal{S}}(p)$. The third step simply consists of using this boundary to construct the Voronoi cell of S.

If the site S is a closed segment t, we first insert the endpoints p_t' and p_t'' of t in the diagram $\mathcal{V}(\mathcal{S})$ using the procedure described above. Since p_t' and p_t'' are necessarily neighbors of t° in the Voronoi diagram $\mathcal{V}(\mathcal{S} \cup \{p_t', p_t'', t^\circ\})$, t° has to be in conflict with at least one of the Voronoi edges on the boundary of the Voronoi cell of p_t' or p_t''. Thus, the first conflict of t° with $\mathcal{V}(\mathcal{S} \cup \{p_t', p_t''\})$, can be found by examining the Voronoi edges on the boundary of $V(p_t')$ or $V(p_t'')$ in $\mathcal{V}(\mathcal{S} \cup \{p_t', p_t''\})$ (i.e., we can omit the nearest neighbor query step). The remaining steps of the insertion are performed as in the case of points. Note that if a segment has already been inserted, this can be detected during the location of the first conflict, in which case the insertion stops.

3.2. The location data structure

The location data structure we use here is similar to the location data structure used in [KY03] to compute the Voronoi diagram for a set of disjoint smooth convex objects in the plane. In order to comply with the requirement that segments are treated as three different objects, the two endpoints and the interior, the way the hierarchy is created in [KY03] has to be modified. We describe how this is done below.

The Voronoi diagram hierarchy, denoted by $\mathcal{H}(\mathcal{S})$, is a hierarchical data structure similar to the Delaunay hierarchy introduced by Devillers [Dev02]. It consists of a set of Voronoi diagrams $\mathcal{V}(\mathcal{S}_\ell)$, $\ell = 0, 1, \ldots, L$, where the sets $\mathcal{S}_\ell$, form a hierarchy of subsets of $\mathcal{S}$, i.e., $\mathcal{S} = \mathcal{S}_0 \supseteq \mathcal{S}_1 \supseteq \ldots \supseteq \mathcal{S}_L$. The hierarchy $\mathcal{H}(\mathcal{S})$ is built together with the Voronoi diagram $\mathcal{V}(\mathcal{S})$ according to three rules:

1. Every site in $\mathcal{S}_{\mathcal{I}}$ is inserted in $\mathcal{V}(\mathcal{S}_0) = \mathcal{V}(\mathcal{S})$.

2. A site S inserted in $\mathcal{V}(\mathcal{S}_\ell)$, for some $\ell \geq 0$, is inserted in $\mathcal{V}(\mathcal{S}_{\ell+1})$ with probability β, $0 < \beta < 1$.

3. If an open segment is inserted in $\mathcal{V}(\mathcal{S}_\ell)$, both its endpoints are inserted in $\mathcal{V}(\mathcal{S}_\ell)$ as well.

It is easy to show that the expected size of $\mathcal{H}(\mathcal{S})$ is $O(\frac{1}{1-\beta}n)$ and that its expected height is $O(\log_{1/\beta} n)$, where n is the cardinality of $\mathcal{S}_{\mathcal{I}}$. Note that if a site has been inserted at some level ℓ, it has also been inserted at all levels ℓ' with $\ell' < \ell$. The *height* $h(S)$ of a site is the maximum level in which it has been inserted. $h(S)$ is computed before S is actually inserted in the diagram, and by definition $h(S) = H$ with probability β^H. In this context, if the site to be inserted is a segment t, rule (3) above requires that $h(p_t') = h(p_t'') = h(t^\circ) = h(t)$.

In contrast to the construction of the hierarchy, which is done in a bottom-up fashion, nearest neighbor queries are performed in a top-down manner. Let $x \in \mathbb{E}^2$ be a query point for which we want to find its nearest neighbor in $\mathcal{V}(\mathcal{S})$. We start by finding the nearest neighbor of x in the topmost diagram $\mathcal{V}(\mathcal{S}_L)$ by means of a *simple walk*. The simple walk starts from any site $S_i \in \mathcal{S}_L$, and compares the distance $\delta(x, S_i)$ with the distances $\delta(x, S)$ to the neighbors S of S_i in $\mathcal{V}(\mathcal{S}_L)$. If some neighbor S_j of S_i is found to be closer to x than S_i, the walk proceeds to S_j. If all neighbors of S_i are further from x than S_i, then S_i is the nearest neighbor of x in $\mathcal{V}(\mathcal{S}_L)$. At the remaining levels of the hierarchy the same simple walk is performed, but now the starting site is the nearest neighbor of x at the previous level. The expected number of sites visited at each level of $\mathcal{H}(\mathcal{S})$ when performing the simple walk is $O(1/\beta)$ (cf. [KY03]).

The nearest neighbor query described above takes $O(n)$ time. As we will see in Section 4, we can improve this time to $O(\log^2 n)$ by making use of the cell trees. The cell trees are associated with the Voronoi cells of each Voronoi diagram in $\mathcal{H}(\mathcal{S})$. In particular, the cell tree of the Voronoi cell $V_\ell(S_i)$ of S_i in $\mathcal{V}(\mathcal{S}_\ell)$ contains, for each Voronoi vertex $v \in V_\ell(S_i)$, the direction $d_i(v)$ of the ray whose apex is the point of S_i closest to v, and which passes through v. The directions $d_i(v)$ are sorted according to their angle with the x-axis. When S_i is visited during the simple walk at level ℓ, we need to locate the direction $d_i(x)$ in the cell tree. Suppose that $d_i(x)$ is located between the directions $d_i(v_1)$ and $d_i(v_2)$. In order to find a new candidate nearest neighbor for x it suffices to look at the neighbor S_j of S_i in $\mathcal{V}(\mathcal{S}_\ell)$ sharing the vertices v_1 and v_2.

When an insertion is performed, we also need to update the cell trees. This can be done, at each level of $\mathcal{H}(\mathcal{S})$, by removing from the existing cell trees the directions corresponding to vertices in the diagram that will be destroyed, and adding the directions of the new vertices in the diagram. Finally, we need to create the cell tree for the newly inserted site.

3.3. Strongly intersecting sites

In this section we describe how to further modify the algorithm and the location data structure in the case of strongly intersecting sites. The modifications presented here, as well as the complexity analysis in the following section, constitute the major non-trivial extensions with respect to the approach in [KY03].

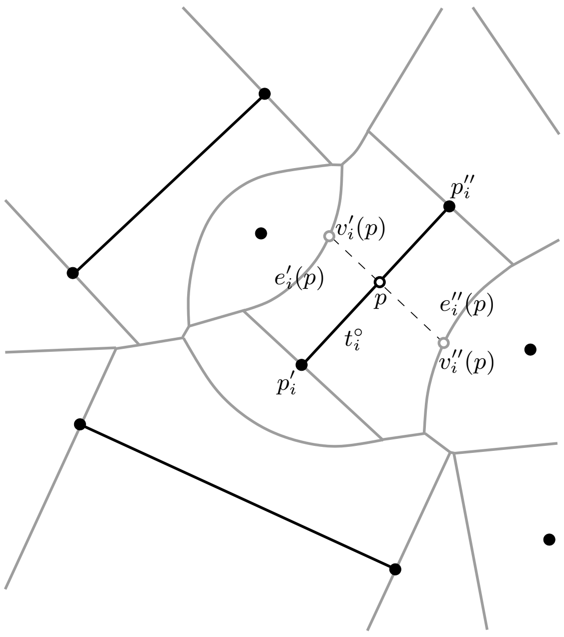

Figure 2. The insertion procedure when the new point p lies on the interior of a segment t_i°: t_i° is replaced in $\mathcal{S}$ by the open segments $p_i'p$ and pp_i'', and the edges $v_i'(p)p$ and $v_i''(p)p$ are added in the Voronoi diagram $\mathcal{V}(\mathcal{S})$.

Suppose that we have a point site p that strongly intersects with a segment t_i, i.e., $p \in t_i^\circ$. Clearly, t_i° must also be the nearest neighbor of p in $\mathcal{S}$. Let $e_i'(p)$ and $e_i''(p)$ be the two edges on the boundary of $V(t_i^\circ)$ intersected by the two rays perpendicular to t_i°, with apex p (cf. Fig. 2). Let $v_i'(p)$ and $v_i''(p)$ be the points of intersection of these rays with $e_i'(p)$ and $e_i''(p)$, respectively. We are now ready to describe the insertion of a point p, that possibly intersects with a segment already inserted in the diagram. We start by finding the nearest neighbor $N_\mathcal{S}(p)$ of p. If $N_\mathcal{S}(p)$ is a point and $N_\mathcal{S}(p) \equiv p$, there is nothing to be done. If $N_\mathcal{S}(p)$ is either a point different from p or an open segment that p does not intersect with, the insertion procedure continues as described in Subsection 3.1. Otherwise, let $N_\mathcal{S}(p)$ be the open segment t_i° (cf. Fig. 2). We first add p to $\mathcal{S}$, and replace t_i° in $\mathcal{S}$ by the open segments $p_i'p$ and pp_i'', where p_i' and p_i'' are the endpoints of t_i. We then search for the edges $e_i'(p)$ and $e_i''(p)$, split them at

$v_i'(p)$ and $v_i''(p)$, and add the segments $v_i'(p)p$ and $v_i''(p)p$ in the Voronoi diagram.

If the site to be inserted is a (closed) segment t, we first insert its two endpoints p_t' and p_t'', as described above. Then, using p_t' (or p_t'') as a starting point, we search for the first conflict of t° with the current Voronoi diagram and subsequently search for its entire conflict region. During this search, before testing a Voronoi edge e of the existing diagram for a potential conflict, we test whether one of the sites that define the supporting bisector of e intersects with t°. If such a site S_i is found the search is stopped. In the case that S_i is a point p_i, we recursively insert the open segments $p_t'p_i$ and p_ip_t''. If S_i is an open segment t_i°, we first test if t° and t_i° coincide. If this is indeed the case, t° has already been inserted, and there is nothing more to be done. Otherwise, we compute the point of intersection p_+ of t° with t_i°, insert it in the Voronoi diagram and then recursively insert the open segments $p_t'p_+$ and p_+p_t''.

We finally need to discuss how the location data structure is affected by allowing strongly intersecting sites. The main idea here is to modify the Voronoi diagram at all levels of the location data structure in such a way so as to preserve the three rules according to which the location data structure is built. When we allow intersecting sites, a segment site that has already been inserted in the diagram may be split in two or more subsegments. Essentially, what we do is to propagate these splits at all levels of the hierarchy.

More specifically, if the site to be inserted is a point p, and p intersects with an open segment $t_i^\circ \in \mathcal{S}$, we recompute the height of p, with the new height being $h_{new}(p) = \max\{h(p), h(t_i^\circ)\}$. We then insert p in all diagrams $\mathcal{V}(\mathcal{S}_\ell)$ with height $\ell \leq h_{new}(p)$.

Let us consider now the situation where the site to be inserted is a segment t. We denote by $\mathcal{T}_t$ the set of open subsegments of t that appear in the Voronoi diagram after the insertion of t. An endpoint p_τ of a segment $\tau^\circ \in \mathcal{T}_t$ will be inserted as described above (the initial height of p_τ is defined to be $h(p_\tau) = h(t)$). Finally, an open segment $\tau^\circ \in \mathcal{T}_t$ will be inserted in all Voronoi diagrams $\mathcal{V}(\mathcal{S}_\ell)$ with $\ell \leq h(t)$, i.e., $h(\tau^\circ) = h(t)$. It is easy to show that the expected height of the hierarchy remains $O(\log_{1/\beta} n)$. The expected size of $\mathcal{V}(\mathcal{S}_\ell)$ is $O(\beta^\ell(n+m))$, $\ell = 0, 1, \ldots, L$, and thus the expected size of the entire hierarchy becomes $O(\frac{1}{1-\beta}(n+m))$. Finally, the number of sites visited by the simple walk is $O(\frac{1}{\beta}(1+\frac{m}{n}))$ per level.

4. Complexity analysis

Let n be the cardinality of the set $\mathcal{S}_\mathcal{I}$ of input sites, consisting of points or closed segments. Let also m be the number of pairs of strongly intersecting sites in $\mathcal{S}_\mathcal{I}$. The cost of inserting a site can be decomposed into the following subcosts:

1. Cost of finding the first conflicts.
2. Cost of finding the conflict regions and updating the Voronoi diagrams.
3. Cost of inserting points of intersection at levels higher than their original height.
4. Cost of updating the cell trees.

Finding the nearest neighbors of a site, at all levels of the hierarchy, takes $O(\frac{1}{\beta}(1+\frac{m}{n})\log n_\ell)$ time per level and $O((1+\frac{m}{n})\log^2 n)$ overall, where $n_\ell = O(\beta^\ell(n+m))$ is the cardinality of $\mathcal{S}_\ell$. Once the nearest neighbor at level ℓ is found, we can find the first conflict at this level in time $O(\log n_\ell)$. Hence, the expected total cost of finding the first conflicts at all levels of $\mathcal{H}(\mathcal{S})$ is $O((1+\frac{m}{n})\log^2 n)$. Let k_ℓ be the number of changes (number of destroyed and created edges) in the Voronoi diagram at level ℓ because of sites inserted at that level in the diagram. The sites inserted at level ℓ are of two types: (i) sites whose height is ℓ and (ii) intersection points whose original height was smaller than ℓ. The cost of finding the conflict region and updating the Voronoi diagram for sites of the first type is $O(k_\ell)$. The number of sites of the second type is $O(k_\ell)$ and inserting each one of them takes $O(\log(n_\ell + k_\ell))$ time. Finally, the cost of updating the cell trees at level ℓ of the hierarchy takes $O(k_\ell \log(n_\ell + k_\ell))$ time. Summing up the various costs, we conclude that the cost per insertion is $O((1+\frac{m}{n})\log^2 n) + \sum_{\ell=0}^{L} O(k_\ell \log(n+k_\ell))$. By applying a randomized analysis similar to that in [BY98, Chapter 5], we can show that $k_\ell = O(\beta^\ell(1+m/n))$. Hence the expected cost per insertion is $O((1+\frac{m}{n})\log^2 n)$, yielding a total of $O((n+m)\log^2 n)$ for inserting n sites. The expectation in the analysis above refers to the order of insertion. Note that the hierarchy introduces another kind of randomization, that has also been taken into account in our analysis.

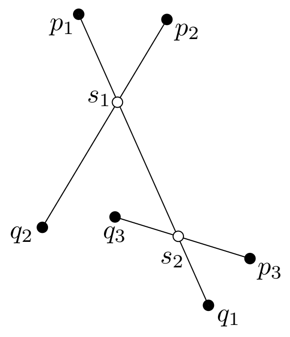

Figure 3. Site representation. The point s_1 is represented by the four points p_1, q_1, p_2 and q_2. The segment p_1s_1 is represented by the points p_1, q_1, p_2, q_2 and a boolean which is set to *true* to indicate that the first endpoint in not a point of intersection. The segment s_1s_2 is represented by the six points: p_1, q_1, p_2, q_2, p_3 and q_3. The remaining (non-input) points and segments in the figure are represented similarly.

5. Representation and filtering

In the case where we have no strongly intersecting segments, segment sites are represented by their endpoints. In the case of strongly intersecting segments, point sites, that are points of intersection, are represented by four points, namely by the endpoints of the segments that define them (cf Fig. 3). Segment sites that have endpoints that are points of intersection are represented either by four points and a boolean or by six points depending on whether only one or both endpoints are points of intersection. The representation allows us to achieve two goals that are very important for the efficiency of our implementation:

- We are able to avoid an exponential blow up on the bit complexity of the coordinates of the points of intersection. Such an exponential blow up could easily occur if we had chosen to represent points of intersection by their coordinates. Moreover, our representation allows us to guarantee that the algebraic degrees of the predicates in our algorithm are bounded by a constant, independently of the size of the input.

- Our representation is coupled very naturally, with what we call *geometric filtering*, which is discussed below.

5.1. Site representation

Burnikel [Bur96] showed that in the case of weakly intersecting sites represented in homogeneous coordinates of bit size b, the maximum bit size of the algebraic expressions involved in the predicates is $40b + O(1)$. In the case of strongly intersecting sites, the maximum degree of the expressions involved in the predicates depends on how we represent the points of intersection and the subsegments of input segments. For example, if we choose to represent the points of intersection by their coordinates and the subsegments by their endpoints, the algebraic degree of the predicates involved can increase arbitrarily: consider two strongly intersecting segments t_i and t_j, whose endpoints have homogeneous coordinates of size b. Their intersection point will have homogeneous coordinates of bit size $6b+O(1)$. This effect can be cascaded, which implies that after inserting k (input) segments we can arrive at having points of intersection whose bit sizes are exponential with respect to k, i.e., their homogeneous coordinates will have bit size $\Omega(2^k b)$. Not only the points of intersection, but also the adjacent subsegments will be represented by quantities of arbitrarily high bit size, and as a result we would not be able to give a bound on the bit sizes of the quantities involved in the evaluation of the predicates.

Such a behavior is undesirable. For robustness, efficiency, and scalability purposes, it is critical that the bit size of the algebraic expressions in the predicates does not depend on the input size. For this reason, as well as for others to be discussed below, we decided to represent sites in a implicit manner, which somehow encodes the history of their construction. In particular, we exploit the fact that points of intersection always lie on two input segments, and that segments that are not part of the input are always supported by input segments.

For example, let us consider the configuration in Fig. 3. We assume that the segments $t_i = p_iq_i$, $i = 1, 2, 3$, are inserted in that order. Upon the insertion of t_2, our algorithm will split the segment t_1 into the subsegments p_1s_1 and s_1q_1, then add s_1, and finally insert the subsegments p_2s_1 and s_1q_2. How do we represent the five new sites? s_1 will be represented by its two defining segments t_1 and t_2. The segment p_1s_1 will be represented by two segments, a point, and a boolean. The first segment is t_1, which is always the segment with the same support as the newly created segment. The second segment is t_2 and the point is p_1. The boolean indicates whether the first endpoint of p_1s_1 is an input point; in this case the boolean is equal to `true`. The segment s_1q_1 will also be represented by two segments, a point, and a boolean, namely, t_1 (the supporting segment of s_1q_1), t_2 and `false` (it is the second endpoint of s_1q_1 that is an input point). Subsegments p_2s_2 and s_2q_2 are represented analogously. Consider now what happens when we insert t_3. The point s_2 will again be represented by two segments, but not s_1q_1 and t_3. In fact, it will be represented by t_1 (the supporting segment of s_1q_1) and t_3. s_2q_1 will be represented by two segments, a point, and a boolean (t_1, t_3 and `false`), and similarly for p_3s_2 and s_2q_3. On the other hand, both endpoints of s_1s_2 are non-input points. In such a case we represent the segment by three input segments. More precisely, s_1s_2 is represented by the segments t_1 (the supporting segment of s_1q_1), t_2 (it defines s_1 along with t_1) and t_3 (it defines s_2 along with t_1).

The five different presentations, two for points (coordinates; two input segments) and three for segments (two input points; two input segments, an input point and a boolean; three input segments), form a closed set of representations and thus represent any point of intersection or subsegment regardless of the number of input segments. Moreover, every point (input or intersection) has homogeneous coordinates of bit size at most $3b + O(1)$. The supporting lines of the segments (they are needed in some of the predicates) have coefficients which are always of bit size $2b + O(1)$. As a result, the bit size of the expressions involved in our predicates will always be $O(b)$, independently of the size of the input.

5.2. Geometric filtering

As we have already mentioned our representation is coupled very naturally, with what we call *geometric filtering*. The technique amounts to performing simple geometric tests exploiting the representation of our data, as well as the geometric structure inherent in our problem, in order to evaluate predicates in seemingly degenerate configurations. Geometric filtering can be seen as a preprocessing step before performing arithmetic filtering. Roughly speaking, by arithmetic filtering we mean that we first try to evaluate the predicates using a fixed-precision floating-point number type (such as `double`), and at the same time keep error bounds on the numerical errors of the computations we perform. If the numerical errors are too big and do not permit us to evaluate the predicate, we switch to an exact number type, and repeat the evaluation of the predicate. Geometric filtering can help by eliminating situations in which the arithmetic filter will fail, thus decreasing the number of times we need to evaluate a predicate using exact arithmetic.

Let us consider a simple, yet very effective, application of geometric filtering. Suppose we want to determine if two non-input points are identical (we assume here that the input sites are represented by `double`s). In order to do that we need to compute their coordinates and compare them. If the two points are identical, the answer to our question using `double` arithmetic may be wrong (due to numerical errors), in which case we will have to reside to the more expensive exact computation. Instead, before testing the coordinates for equality, we can use the representation of the points to potentially answer the question. More specifically, and this is the geometric filtering part of the computation, we can first test if the defining segments of the two points are the same. If they are not, then we proceed to comparing their coordinates as usual. Testing the defining segments for equality does not involve any arithmetic operations on the input, but rather only comparisons on `double`s. By performing this very simple test we avoid a numerically difficult computation, which could be performed thousands of times during the computation of a Voronoi diagram.

6. Implementation and experimental results

In this section we present our implementation, and discuss its performance in three series of experiments, designed to test different aspects of our algorithm and implementation.

Our code has been written in C++ and it follows the design of CGAL [CGA03]. The Voronoi diagram is represented by its dual and we use CGAL's triangulation data structure to do that. The version of CGAL used is 3.0. We have not implemented the cell trees discussed in Subsec-

Results for the RandomSegments data set						
n	m	N	Operations	Number type	$T(n,N)$	$R(n,N)$
50	*317*	*1101*	Field + Sqrt	filter + `real`	0.83	2.478
			Field	filter + `Gmpq`	0.65	1.941
250	*7634*	*23652*	Field + Sqrt	filter + `real`	15.63	1.511
			Field	filter + `Gmpq`	13.26	1.941
500	*29148*	*88944*	Field + Sqrt	filter + `real`	54.38	1.235
			Field	filter + `Gmpq`	46.22	1.050
2500	*697630*	*2100390*	Field + Sqrt	filter + `real`	1242.14	0.935
			Field	filter + `Gmpq`	1047.22	0.789

Table 1. Results for the RandomSegments and RoadNetwork series of experiments. $T(n,N)$ is the total time (in seconds) for computing the Voronoi diagram of n sites. m is the number of strongly intersecting pairs of input segments and N is the number of Voronoi cells in the diagram. $R(n,N)$ is the (dimensionless) quantity $10^4 T(n,N)/(N\log_{10} N)$. The column labeled "Operations" indicates the kind of arithmetic operations used in the evaluation of the predicates.

Results for the AlignedSquares data set							
n	N	Operations	Number type	$T(n,N)$	$R(n,N)$	$T(n,N)$	$R(n,N)$
			Random shuffling?	Yes		No	
500	*1000*	Field + Sqrt	filter + `real`	0.43	1.433	0.58	1.933
		Ring	filter + `Gmpq`	0.4	1.333	0.59	1.967
5000	*10000*	Field + Sqrt	filter + `real`	5.23	1.308	8.43	2.108
		Ring	filter + `Gmpq`	4.89	1.222	8.5	2.125
50000	*100000*	Field + Sqrt	filter + `real`	56.22	1.124	239.12	4.782
		Ring	filter + `Gmpq`	53.24	1.065	239.43	4.789
500000	*1000000*	Field + Sqrt	filter + `real`	625.96	1.043	4461.54	7.436
		Ring	filter + `Gmpq`	588.59	0.981	4467.83	7.446

Table 2. Results for the RandomSegments and RoadNetwork series of experiments. $T(n,N)$ is the total time (in seconds) for computing the Voronoi diagram of n sites. N is the number of Voronoi cells in the diagram. $R(n,N)$ is the (dimensionless) quantity $10^4 T(n,N)/(N\log_{10} N)$. The column labeled "Operations" indicates the kind of arithmetic operations used in the evaluation of the predicates. There are no strongly intersecting pairs of segments in this data set.

tion 3.2; we use the simple walk at each level of the Voronoi hierarchy. Our code has two modes of operation, depending on whether we assume that strongly intersecting sites exist in the input data or not. The user has the ability to indicate which mode of operation is to be used. By default our algorithm checks for strongly intersecting sites. In the case of strongly intersecting sites our implementation of the hierarchy differs from the one described in Subsections 3.2 and 3.3. In particular, at the higher levels of the hierarchy, i.e., for all $\mathcal{V}(\mathcal{S}_\ell)$, $\ell > 0$, we only insert point sites and segment endpoints, rather than the segments themselves. As a result, all Voronoi diagrams $\mathcal{V}(\mathcal{S}_\ell)$, $\ell > 0$, are actually Voronoi diagrams of points. We are currently in the process of implementing the hierarchy as discussed in Subsections 3.2 and 3.3. Moreover, we will provide the user with the choice of whether or not to include segments at the upper levels of the hierarchy. The experimental results shown below, as well as results on other data sets not presented in this paper, indicate that such a choice, although without any theoretical guarantees, seems to work very well.

Arithmetic filtering is supported by our implementation, and we use Cgal's filtering mechanism to this end (cf. [Pio99]). We have implemented two methods for computing the predicates per mode of operation, that depend on the type of operations supported exactly by the number type used. In both modes of operation, one of the two methods assumes that field operations (+, −, ×, /) and square roots ($\sqrt{\ }$) are supported exactly. The second method assumes that only ring operations (+, −, ×) are supported exactly, if our algorithm operates under the assumption that there no strongly intersecting sites. Otherwise, we require that field operations are supported exactly.

Finally, the sites in the segment Voronoi diagram are represented as discussed in Section 5, whereas geometric filtering is extensively used throughout the predicate evaluations. Our code will be available in the next public release of CGAL.

We have tested our algorithm on three series of experiments, code-named RANDOMSEGMENTS, ALIGNEDSQUARES and ROADNETWORK. All experiments were run on an Intel Xeon processor at 2GHz with 512K cache running Linux. The executables were created using the GNU `g++` compiler, version 3.3.2, with optimization flags `-O3 -march=pentium4 -mcpu=pentium4`. In our experiments we used CGAL's arithmetic filtering and the following exact number types: LEDA's `real` and CGAL's `Gmpq` (which is based on the GMP package [Gra]). The versions of LEDA and GMP used are 4.4.1 and 4.1.2, respectively. `real`s support field operations and square roots exactly, whereas `Gmpq` supports field operations exactly. The reason for testing different number types is to test the sensitivity of our implementation with respect to the number type used. The experiments here are not intended to be exhaustive with respect to the possible choices of number types, but rather provide some qualitative results.

The first series of experiments consists of n random segments in an axis-aligned square of edge length $2 \cdot 10^6$ centered at the origin, and $n \in \{50, 250, 500, 2500\}$ (see Table 1). Since in this case we have intersecting segments, we use the mode of operation that checks for intersections. The second series of experiments consists of axis-aligned squares of edge length and inter-distance 100, aligned on a w-by-h grid, where $(w, h) \in \{(25, 5), (50, 25), (250, 50), (500, 250)\}$ (see Table 2). The input to our algorithm consists of $n = 4wh$ segments. Notice that the data sets in this series are highly degenerate, and have been designed so as to test the robustness of our implementation under degeneracies. The third series are real world data of road networks of seven countries, obtained from the "Digital Chart of the World" database [DCW] (see Table 3). The results shown here are representative of the behavior of our algorithm/implementation on all data sets from this database.

As it can be seen from the last two series of experiments, performing random shuffling before inserting the data generally enhances the performance of our algorithm. Random shuffling spreads evenly the input sites, and as a result the shuffled input does not contain any patterns that may be present in the original input. As a result the point location is performed over a more evenly distributed set of sites, which conforms with our randomized analysis. This effect is very striking in the ALIGNEDSQUARES data set (we get speed-ups up to a factor of 7). In this case random shuffling also decreases drastically the number of degenerate or near-degenerate configurations encountered during the evaluation of the predicates, which results in fewer predicate evaluations using exact arithmetic. The time to perform random shuffling is included in our timings.

We mentioned in the introduction that the only other implementation of the segment Voronoi diagram that we know of, using the exact computation paradigm, is that by M. Seel [See]. We benchmarked that implementation against ours. Our implementation is about two orders of magnitude faster. For example, for the ALIGNEDSQUARES data set, and for $n = 500$, the implementation by M. Seel takes about 13 seconds if random shuffling is performed and about 76 seconds if random shuffling is not performed. For larger n the memory requirements of the M. Seel's implementation result in disk swapping or the system runs out of memory (our test platform had 1GB of memory).

In general, the time spent for computing the Voronoi diagram is $O((n + m) \log n)$, i.e., a $\log n$ factor better than the predicted running time (recall that we have not implemented the cell trees). The time spent is generally insensitive to the number type used. One final comment should be made with respect to the road network data tagged *"U.S.A."*. This consists of the combined road network data from [DCW] for the 48 contiguous United States. Unlike other road network data tested, there exists one point of intersection in this data set, that our algorithm handles successfully.

Acknowledgments

The author wishes to thank Mariette Yvinec for helpful comments on preliminary versions of this paper, and Andreas Fabri and Sylvain Pion for discussions on the code design. This work started while the author was at INRIA, and at the time he was supported in part by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project IST-2000-26473 (ECG - Effective Compu-

Results for the RoadNetwork data set								
	n	N	Operations	Number type	$T(n,N)$	$R(n,N)$	$T(n,N)$	$R(n,N)$
				Random shuffling?	Yes		No	
France	*41765*	*81788*	Field + Sqrt	filter + `real`	48.97	1.219	62.53	1.556
			Ring	filter + `Gmpq`	56.4	1.404	79.68	1.983
Germany	*30674*	*59804*	Field + Sqrt	filter + `real`	36.54	1.279	46.37	1.623
			Ring	filter + `Gmpq`	44.41	1.555	62.97	2.204
Greece	*16816*	*33184*	Field + Sqrt	filter + `real`	17.28	1.152	24.05	1.603
			Ring	filter + `Gmpq`	21.11	1.407	29.58	1.971
Italy	*28534*	*56013*	Field + Sqrt	filter + `real`	34.12	1.283	42.8	1.609
			Ring	filter + `Gmpq`	39.91	1.501	56.81	2.136
Netherlands	*2949*	*5724*	Field + Sqrt	filter + `real`	3.19	1.483	3.82	1.776
			Ring	filter + `Gmpq`	3.1	1.441	5.06	2.353
Spain	*35162*	*69260*	Field + Sqrt	filter + `real`	40.65	1.213	53.02	1.582
			Ring	filter + `Gmpq`	47.12	1.406	65.15	1.943
U.S.A.	*335523*	*656873*	Field + Sqrt	filter + `real`	429.52	1.124	546.9	1.431
			Field	filter + `Gmpq`	554.55	1.451	844.57	2.210

Table 3. Results for the RandomSegments and RoadNetwork series of experiments. $T(n,N)$ is the total time (in seconds) for computing the Voronoi diagram of n sites. N is the number of Voronoi cells in the diagram. $R(n,N)$ is the (dimensionless) quantity $10^4 T(n,N)/(N \log_{10} N)$. The column labeled "Operations" indicates the kind of arithmetic operations used in the evaluation of the predicates. There are no strongly intersecting segments in this data set, except for the *U.S.A.* data set which has one pair of strongly intersecting segments.

tational Geometry for Curves and Surfaces).

References

[AS95] H. Alt and O. Schwarzkopf. The Voronoi diagram of curved objects. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 89–97, 1995.

[BDS$^+$92] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete Comput. Geom.*, 8:51–71, 1992.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Proc. 2nd European Sympos. Algorithms*, volume 855 of *LNCS*, pages 227–239, 1994.

[Bur96] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections.* Ph.D thesis, Universität des Saarlandes, March 1996.

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

[CGA03] *The CGAL Manual*, 2003. Release 3.0.

[DCW] Digital chart of the world. http://www.maproom.psu.edu/dcw/.

[Dev02] O. Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[DL78] R. L. Drysdale, III and D. T. Lee. Generalized Voronoi diagrams in the plane. In *Proc. 16th Allerton Conf. Commun. Control Comput.*, pages 833–842, 1978.

[DY93] K. Dobrindt and M. Yvinec. Remembering conflicts in history yields dynamic algorithms. In *Proc. 4th Annu. Internat. Sympos. Algorithms Comput.*, volume 762 of *LNCS*, pages 21–30. Springer-Verlag, 1993.

[For87] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[Gra] T. Granlund. GMP, the GNU multiple precision arithmetic library. http://www.swox.com/gmp/.

[Hel01] M. Held. VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Comput. Geom. Theory Appl.*, 18:95–123, 2001.

[Ima96] T. Imai. A topology oriented algorithm for the Voronoi diagram of polygons. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 107–112. Carleton University Press, Ottawa, Canada, 1996.

[Kir79] D. G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 18–27, 1979.

[KMM93] R. Klein, K. Mehlhorn, and S. Meiser. Randomized incremental construction of abstract Voronoi diagrams. *Comput. Geom.: Theory & Appl.*, 3(3):157–184, 1993.

[KY03] M. I. Karavelas and M. Yvinec. The Voronoi diagram of planar convex objects. In *Proc. 11th European Sympos. Algorithms*, volume 2832 of *LNCS*, pages 337–348, 2003.

[Lee82] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-4(4):363–369, 1982.

[Pio99] S. Pion. Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to systems and Control*, pages 99–110, 1999.

[See] M. Seel. The AVD LEP user manual. http://www.mpi-sb.mpg.de/LEDA/friends/avd.html.

[SIII00] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation - an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.

[Wei02] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. 10th European Sympos. Algorithms*, volume 2461 of *LNCS*, pages 884–895, 2002.

[Yap87] C. K. Yap. An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete Comput. Geom.*, 2:365–393, 1987.