ELSEVIER

Computational Geometry 18 (2001) 95–123

Computational Geometry
Theory and Applications

www.elsevier.nl/locate/comgeo

VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments

Martin Held [1]

Universität Salzburg, Institut für Computerwissenschaften, A-5020 Salzburg, Austria

Communicated by K. Mehlhorn; received 11 October 2000; received in revised form 30 November 2000; accepted 22 December 2000

Abstract

We discuss the design and implementation of a topology-oriented algorithm for the computation of Voronoi diagrams of points and line segments in the two-dimensional Euclidean space. The main focus of our work was on designing and engineering an algorithm that is completely reliable and fast in practice. The algorithm was implemented in ANSI C, using standard floating-point arithmetic. In addition to Sugihara and Iri's topology-oriented approach, it is based on a very careful implementation of the numerical computations required, an automatic relaxation of epsilon thresholds, and a multi-level recovery process combined with "desperate mode". The resulting code, named vroni, was tested extensively on real-world data and turned out to be reliable. CPU-time statistics document that it is always faster than other popular Voronoi codes. In our computing environment, vroni needs about $0.01n \log_2 n$ milliseconds to compute the Voronoi diagram of n line segments, and this formula holds for a wide variety of synthetic and real-world data. In particular, its CPU-time consumption is hardly affected by the actual distribution of the input data. Vroni also features a function for computing offset curves, and it has been successfully tested within and integrated into several industrial software packages. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Voronoi diagram; Topology-oriented algorithm; Reliability; Robustness; Experimental analysis

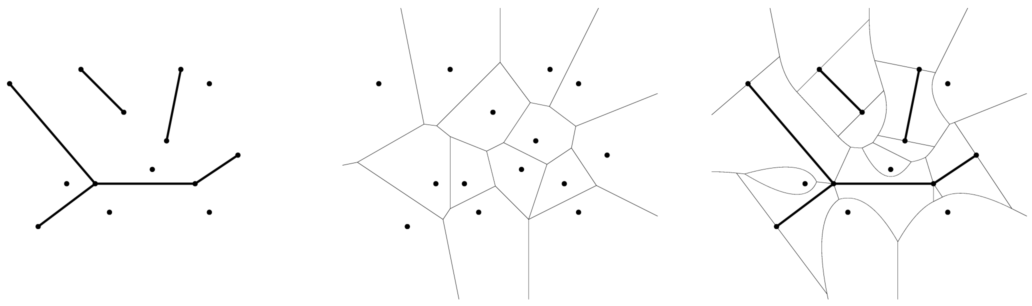
1. Introduction

1.1. Basic definitions

Consider a set $\mathcal{S}$ of disjoint points and straight-line segments in the two dimensional Euclidean space $\mathbb{E}^2$. For technical reasons, we follow the convention established by Kirkpatrick [33] and regard

E-mail address: held@cosy.sbg.ac.at (M. Held).

[1] Partially supported by grants and hardware donations from Sun Microsystems, and by NSF Grant CCR-9732220. Part of this work was carried out while the author visited SUNY Stony Brook.



(a) Input sites (b) Point Voronoi diagram (c) Final Voronoi diagram

Fig. 1. Sample input sites, and the corresponding Voronoi diagrams.

a line segment as the union of three objects: an open line segment (i.e., the segment minus its endpoints) and the two endpoints. (We will always assume that the two endpoints of a line segment are among the input points.) Points and open line segments are called *sites*.

For a point $p \in \mathbb{E}^2$ and a site s, the (Euclidean) distance from p to s is the minimum of $d(p,q)$ for q out of the closure of s. The *clearance* $d(p,\mathcal{S})$ of p with respect to $\mathcal{S}$ is defined as the minimum distance from p to any site $s \in S$. The *clearance disk* at p is the disk with radius $d(p,\mathcal{S})$ centered at p.

Obviously, every clearance disk touches at least one site of $\mathcal{S}$. The *Voronoi diagram* $\mathcal{VD}(\mathcal{S})$ is the set of points of $\mathbb{E}^2$ whose clearance disks touch at least two disjoint sites of $\mathcal{S}$. See Fig. 1 for a sample set $\mathcal{S}$, and (portions of) the Voronoi diagram of the points of $\mathcal{S}$, and (portions of) $\mathcal{VD}(\mathcal{S})$.

The Voronoi diagram $\mathcal{VD}(\mathcal{S})$ partitions $\mathbb{E}^2$ into mutually disjoint sub-areas, where every sub-area is associated with exactly one site of $\mathcal{S}$. Essentially, such a sub-area is the nearest neighborhood of its defining site. By definition, the common boundary of two adjacent sub-areas is the loci of points equidistant from the areas' defining sites and is called a *bisector*. All bisectors of $\mathcal{VD}(\mathcal{S})$ are portions of straight lines or parabolas. Points shared by three or more bisectors are commonly called *nodes* of the Voronoi diagram.

1.2. Motivation

Voronoi diagrams (and related concepts such as constrained or conforming Delaunay triangulations or medial axis) are a basic building block for many applications with a geometric flavor. Well-known applications comprise, for instance, shape representation, conversion and reconstruction [3,37,44,54], mesh generation [23,50,51] and NC machining [24,28].

Computing the Voronoi diagram of a set of sites also is a fundamental operation in computational geometry, and it has received widespread interest over the last two decades. See the surveys by Aurenhammer [5], Okabe et al. [43] and Fortune [18].

Virtually all published Voronoi algorithms put stringent restrictions on the input sites. In particular, it is common practice to assume that the sites do not intersect except at common endpoints. If the sites form polygons then the polygons are typically required to be *simple*. (In a simple polygon, the vertices of the polygon are the only points of the plane that belong to two edges, and no point of the plane belongs to more than two edges.) Some authors also ban data that contains four or more sites which are touched by the same clearance disk, or multiple collinear sites. Sites that are disjoint (and form simple polygons in the case of polygonal environments) are called "clean", while the latter assumptions on the input data are summarized by the term "general position assumed".

Obviously, the requirement to be simple prevents a polygon from having self-intersections. However, it also excludes so-called "degeneracies", such as an edge passing through another vertex, zero-length edges, and edges that partially overlap.

While avoiding the hassles involved with handling "self-intersecting" or "degenerate" polygons, the restriction to clean data constitutes a major road-block for the practical application of Voronoi algorithms. Unfortunately, real-world polygonal environments cannot be assumed to be simple polygons that are in general position. Rather, real-world polygonal environments tend to exhibit all types of deficiencies, such as self-intersections, grazing contacts between polygons, and polygons that intersect.

We note the practical handling of data which is not clean is an issue even if exact arithmetic is to be used for the computation of the Voronoi diagram. For a CAD code that computes a tool path based on Voronoi diagrams and sends it to an NC machine it does not matter whether a polygon is classified as "simple", "degenerate" or "self-intersecting". In any case, the code must not crash or loop, and a reasonable Voronoi diagram has to be computed. And, of course, this Voronoi diagram has to be computed without human intervention.

In a recent editorial, Fortune [19] wrote that "it is notoriously difficult to obtain a practical implementation of an abstractly described geometric algorithm". According to the author's personal experience this remark is particularly true for the implementation of Voronoi diagrams of line segments. This paper discusses the design and implementation of a reliable and efficient Voronoi code, named vroni, that is able to cope with any form of polygonal input data, be it clean or not. Specifically,

- we *describe a topology-oriented implementation* based on the approach by Sugihara, Iri and Imai [31, 52,53];
- we *reveal missing algorithmic details* of the description given by Imai [31];
- we present a *detailed discussion of reliability issues* related to the handling of data that is not clean;
- we explain how a *reliable and efficient* code that uses *conventional floating-point arithmetic* has been engineered; and
- we report on a *thorough experimental comparison* with other publicly available Voronoi codes.

To our knowledge, vroni is the first code for computing Voronoi diagrams of points and line segments, and for Voronoi-based offsetting, which is both reliable and fast in practice. Vroni has been successfully tested within industrial CAD/CAM and GIS software.

1.3. Survey of vroni

The success of our triangulation code FIST [25,27] motivated us to apply the same design principles to the computation of Voronoi diagrams of line segments. The basic idea that drives FIST is to triangulate a polygon by performing a series of incremental steps which are simple enough such that

(1) the correctness of every step can be monitored easily,
(2) the correctness of the overall algorithm is a simple consequence of the correctness of every step,
(3) the "successful" completion of every step can be enforced in case of algorithmic or numerical problems.

Thus, we decided to compute Voronoi diagrams in an incremental manner, based on the topology-oriented approach by Sugihara et al. [31,53]. Starting with an initially empty set of processed sites (and a corresponding trivial Voronoi diagram), the final Voronoi diagram is obtained by incrementally adding one new site to the set of processed sites and updating the Voronoi diagram accordingly. Every update of the Voronoi diagram is performed by deleting old Voronoi nodes (and inserting new Voronoi nodes) according to topological properties that the Voronoi graph has to fulfill. One important topological property to fulfill is that the set of Voronoi nodes which are deleted during one update forms a tree. Numerical predicates are used to select those nodes that are the best candidates for deletion.

Provided that every incremental update is performed successfully, we are bound to obtain a final Voronoi diagram which also conforms to the set of topological properties. The obvious crux is to guarantee that every incremental update is performed successfully. This task splits into three completely different subtasks:

(1) handling all purely algorithmic problems in order to preserve the topological properties when the algorithm is applied to clean data;
(2) extending the basic algorithm to data that need not be clean; and
(3) handling numerical inaccuracies due to the use of (inexact) floating-point arithmetic.

Subtask 1 merely means filling in the details missing in Imai's paper [31]. In particular, care has to be taken in order to prevent the removal of loops while deleting Voronoi nodes during an incremental update.

Subtask 2 involves the identification of data which is not clean, on-the-fly local cleaning of the data by inserting points of intersection and removing any duplicate (portions of) sites, and the amendments necessary in order to handle the resulting data during the incremental update. In particular, this requires the handling of endpoints that are shared by more than two line segments.

Subtask 3 is the only subtask which is specific to our goal of using conventional floating-point arithmetic. (This goal is motivated by the need to achieve processing times which are acceptable in practice, and by our desire to avoid any potential problems associated with porting (and licensing) somebody else's arithmetic library. Also, the re-conversion from exact reals to standard floating-point output is sensitive.) This subtask is handled by a careful implementation of the numerical computations required, and by an automatic selection of a suitable precision threshold, dubbed "relaxation of epsilon thresholds".

Note that the original topology-oriented approach by Sugihara et al. [31,53] does not rely on any precision thresholds: topological consistency is guaranteed without the use of "epsilons". However, we feel that topological consistency in the absence of numerical fidelity is of little practical use. Experience tells us that the use of "epsilons" helps to achieve numerical fidelity. The combination of a topology-oriented approach with our "relaxation of epsilon thresholds" seems to yield superior results, indeed; see Section 5.

Common to all three subtasks is the need to ensure that every computational step will always yield a result which is an acceptable input for the subsequent step. Typically, problems (such as arithmetic operations which cannot be performed, like divisions by zero) will be due to numerical inaccuracies or due to data which is not clean. We employ a multi-level recovery process which starts with back-up procedures that attempt to compute the data sought in a different way, gradually relaxes the (local) numerical correctness of the Voronoi diagram under computation, and finally applies "desperate mode". (Desperate mode is a concept which we have successfully used in the triangulation code FIST , cf. [25, 27].)

Summarizing, the design and implementation of vroni is based on the following *four corner stones*:

(1) Sugihara and Iri's *topology-oriented approach* [31,52,53];
(2) a very *careful implementation* of the numerical computations required;

(3) an automatic *relaxation of epsilon thresholds*; and
(4) a *multi-level recovery process* combined with *desperate mode*.

Vroni was implemented in ANSI C. It has been tested extensively, and we present statistics on a series of test runs. Using synthetic test data generated by means of RPG [4], we compare the CPU-time consumption of vroni to all those other Voronoi codes which we could obtain for testing:

- Imai's Fortran code plvor [31],
- Seel's C++ code avd [48],
- Sethia's C++ code pvd [49].

As witnessed by our experiments, vroni outperformed the other codes in all tests. See Section 5 for details.

This paper is accompanied by several color plates available on the WWW. Point your browser to the WWW home-page [29] of vroni.

The remainder of this paper is organized as follows. We discuss prior and related work in the following subsection. Section 2 contains a presentation of the basic Voronoi algorithm. The next two sections, Section 3 and Section 4, discuss enhancements of the basic algorithm for improving its reliability. Experimental results are given in Section 5.

1.4. Prior and related work

Work on Voronoi diagrams of a simple polygon started with the wave-front propagation algorithms of Preparata [46] and Persson [45]. Their algorithms compute the Voronoi diagram in the interior of a simple n-gon in time $O(nm)$, where $m < n$ is shape-dependent. Held [26] demonstrated that a tuned version of their algorithm is very fast in practice, since $m \ll n$ can be expected for most practical applications. A similar approach was also used by Yao and Rokne [55].

Lee and Drysdale [35] showed how to compute the Voronoi diagram of a set of n line segments in $O(n \log^2 n)$ time. Using a divide-and-conquer algorithm, Lee [34] and Yap [56] reduced this bound to $O(n \log n)$. The same worst-case bound is achieved by an elegant sweep-line algorithm due to Fortune [16]. Srinivasan and Nackman's extension of Lee's algorithm handles polygons with h holes in time $O(n \log n + nh)$.

For convex polygons, a linear-time Voronoi algorithm has been known for a few years, cf. [1]. In 1991, Devillers [13] devised a randomized $O(n \log^\star n)$ algorithm, based on "influence graphs", for computing the Voronoi diagram of a general simple polygon. Recently, Chin et al. [12] achieved a linear-time deterministic solution to the same problem. It has yet to be seen, though, whether their algorithm is practical.

Several authors [2,22,31] have suggested to construct the Voronoi diagram of a set of n line segments as a two-phase process: In Phase I of the algorithm, the Voronoi diagram of a discrete set of $O(n)$ many points is constructed, where the particular choice for this set of points depends on the algorithm. Then, in Phase II the actual Voronoi diagram is derived by "replacing" those points by their corresponding line segments. Alt and Schwarzkopf [2] first select one point out of the relative interior of each segment [2], and then insert the segments in a randomized order, obtaining an expected running time of $O(n \log n)$. Gold et al. [22] pick one endpoint of each segment, compute the (point) Voronoi diagram, and then obtain the actual Voronoi diagram by "moving" these points along the corresponding segments. This approach yields an $O((n + t) \log n)$ algorithm, where t is the number of topological events encountered during the

[2] At least in theory, their algorithm can also be applied to general curved segments.

moves of the points. Imai [31] uses the endpoints of the line segments as initial points, and then inserts the actual line segments one after the other.

Recent research has focused on providing the theoretical basis for a reliable implementation of Voronoi diagrams. Fortune [17] calls an algorithm robust if it never fails, and if there exists a perturbation of the input such that the algorithm's output is correct with respect to this perturbed input. If small perturbations suffice then he calls the algorithm stable. Several tools for achieving robustness have been proposed recently, e.g., symbolic perturbation [14], "smart" floating-point evaluation [6] and determinant computation [8], lazy evaluation [7], floating-point filters and "smart" integer arithmetic [20,21], and effective geometric rounding [41].

Imai's algorithm is based on the topology-oriented approach of Sugihara and Iri [52,53]. Mayya and Rajan [37] claimed that the iterative tracing of bisectors yields a robust algorithm for the computation of Voronoi diagrams of line segments. Their algorithm is based on work by Milenkovic [40]. It is not clear, though, whether their algorithm is practical since both its worst-case and average-case complexities seem to be $O(n^2)$. Hoff et al. [30] have recently advocated that interpolation-based graphics hardware can be used to speed up this process. Etzion and Rappoport [15] report on a boundary sampling algorithm that computes the topological structure of a Voronoi diagram.

The major roadblock on the road to robustness of Voronoi algorithms is given by the fact that the computations required during the construction of the Voronoi diagram of polygonal data go well beyond the determination of signs of determinants, and also well beyond the domain of rational arithmetic. In [9,11], Burnikel has shown how to evaluate the geometric tests required for the computation of Voronoi diagrams of line segments in a reliable way. A Voronoi code which uses Burnikel's exact arithmetic, based on LEDA [38], has been implemented by Seel [48].

The issues of exact computation versus floating-point arithmetic are discussed in detail in survey papers by Yap and Dubé [57,58]. Exact arithmetic is offered by the geometric software packages LEDA [38] and CGAL [47]. See also the LEDA number type `leda_real` [10] and the CORE library [32]. The main problem with any Voronoi algorithm that is based on exact arithmetic is the high "degree" of the problem, which makes exact arithmetic very expensive. (See [36] for the "degree" of a problem.)

There seem to be far more theoretical studies of Voronoi algorithms than published reports on actual implementations. A few authors – e.g., Etzion and Rappoport [15], Gold et al. [22], Imai [31], Meyer [39], Seel [48], Yao and Rokne [55] – hint at the existence of implementations. However, only the codes published by Imai and Seel (and Sethia's code) could actually be obtained for testing.

2. Basic incremental algorithm

2.1. *Voronoi diagram of points*

We use a randomized incremental construction. Four dummy points (outside of the bounding box of the point sites) define the initial Voronoi diagram. Those four points are given by the corners of an appropriately scaled bounding box of the sites. (In our implementation, the user can choose the factor used for scaling the bounding box.) Then the point sites are inserted incrementally in random order.

The incremental updates of the Voronoi diagram are carried out according to Sugihara and Iri's topology-oriented approach [52]. They postulated the following minimum set of topological requirements for a graph to form the Voronoi diagram (node/edge graph) of a set of point sites.

- Every site has its own Voronoi region.
- Every Voronoi region is connected.
- Two Voronoi regions share at most one edge (Voronoi bisector).

In order to update the Voronoi diagram, the algorithm first chooses a seed node from the Voronoi region of the nearest neighbor of the new site. Any node whose clearance disk contains the new site qualifies as a seed node; we pick the node whose clearance disk is violated the most. (Let ρ_ν be the radius of the clearance disk of node ν, and δ_ν be the distance between the new site and node ν. Then we choose ν such that $\rho_\nu - \delta_\nu$ is maximized over all nodes ν of the Voronoi region of the nearest neighbor of the new site.) Then nodes adjacent to this seed are deleted if their clearance disks contain the new site. Note that a node may only be deleted if the structure formed by the deleted nodes and their interconnecting edges is a tree. This deletion process recursively grows the tree of deleted nodes to its maximum size. That is, the deletion process stops once no more nodes adjacent to deleted nodes exist whose clearance disks contain the new site and which can be added to the tree without creating a cycle. It remains to compute new Voronoi nodes on those edges which have one deleted node. (These new nodes form the corners of the Voronoi region of the new site.)

The computations needed for the site-in-disk check and the generation of new Voronoi nodes are carried out using conventional floating-point arithmetic. Note that the numerical (in)accuracy of those computations has no impact on the validity of the topological requirements postulated. As argued in [52], these topological properties form a loop invariant for every update since only a tree of nodes is removed from the Voronoi diagram when inserting a new site.

Sugihara et al. [53] emphasize, though, that these topological properties only constitute necessary conditions for a graph to form the node/edge graph of a Voronoi diagram of a set of point sites. Thus, the preservation of these topological properties does not imply that there actually exists a set of point sites whose Voronoi diagram is given by the structure computed. This so-called "topological consistency" would be very time-consuming to achieve for Voronoi diagrams of point sites, while no necessary and sufficient conditions are known for Voronoi diagrams of line segments.

The algorithm described lends itself to a simple and quick implementation. In particular, there is no need for exceptional branches in the algorithm to handle "special cases" (such as four or more points being cocircular).

2.2. *Voronoi diagram of points and segments*

After the Voronoi diagram of the point sites has been completed the (open) line segments are inserted incrementally. The basics of a topology-oriented approach to the incremental construction of Voronoi diagrams of line segments were proposed by Imai [31] and restated in [53]. Our approach is inspired by Imai's work. Nevertheless, our algorithm might differ substantially from his algorithm. (Any potential difference is difficult to assess as [31] seems to lack a few details that are essential in order to make the algorithm work.)

We require the Voronoi diagram under computation to fulfill the following topological conditions.

- Every site has its own Voronoi region.
- Every Voronoi region is connected.
- The Voronoi region of an open line segment is adjacent to the Voronoi regions of its endpoints.

The initial Voronoi diagram is given by the Voronoi diagram of all point sites. (Recall that the two endpoints of a line segment are among the point sites.)

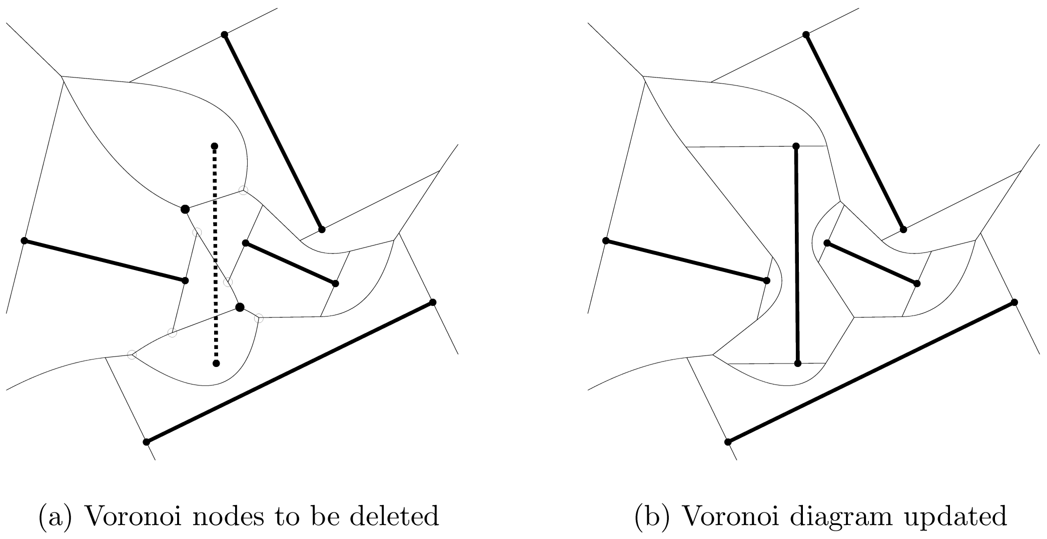


Fig. 2. Incremental update of the Voronoi diagram in order to insert the dashed line segment. The two seed nodes are marked by filled disks; all other Voronoi nodes to be deleted are marked by circles.

In order to insert a new line segment, we first find a seed node on the boundary of the Voronoi region of each of its two endpoints. As in the case of point sites, we pick the node whose clearance disk is violated the most by the insertion of the new segment. Starting at one of the two seed nodes, adjacent Voronoi nodes are deleted recursively if their clearance disks intersect the line segment. Again, a node may only be deleted if this would not create a cycle in the structure to be deleted. Eventually, since the Voronoi region of a line segment is connected [53], the tree grown from the one seed node will reach the other seed node, and we get one tree of Voronoi nodes and edges to be deleted that contains both seed nodes. It remains to compute the corners (new Voronoi nodes) of the Voronoi region of the new line segment. See Fig. 2 for an illustration of this update process.

2.3. Geometric hashing for faster neighbor finding

The basic incremental Voronoi algorithm needs to find the nearest neighbor for every new point site. Obviously, a brute-force search for the nearest neighbors would require $O(n^2)$ time, which would render the algorithm infeasible for anything but small inputs. Ohya et al. [42] discuss various schemes for establishing an insertion order of the points such that a near-constant amount of time can be expected to suffice for finding a nearest neighbor. A more formal method for reducing the time complexity of the nearest neighbor search was presented by Devillers [13].

We have resorted to a scheme that is simpler to implement: we store the points that have already been inserted into the Voronoi diagram in a regular grid. Then a nearest neighbor search translates to locating the cell of the grid which contains the new point, and scanning this cell and adjacent cells until the nearest neighbor is found.

Predicting a suitable resolution of the grid is not exactly easy as it may depend on the distribution of the point sites. Clearly, we cannot afford to waste memory on a very fine grid. Thus, we decided that the grid should have roughly $w\sqrt{n} \times h\sqrt{n}$ cells for n sites, where w and h depend on the aspect ratio of the bounding box of the sites. After a series of experiments we decided to choose w and h such that $w \cdot h = 2$. Thus, our grid consists of roughly $2n$ cells, and the additional memory consumption caused by the grid is negligible on a standard workstation. As documented in Section 5, using a grid for finding nearest neighbors is simple but very effective in avoiding an $\mathrm{O}(n^2)$ complexity even if the input points are distributed very irregularly.

3. Details of the incremental update

Besides any potential numerical problems, one will quickly realize that the basic algorithm sketched in the previous section will not suffice to insert the line segments incrementally. What shall we do if all nodes of a Voronoi region are to be deleted? Is it correct to stop the deletion of Voronoi nodes once a cycle would occur?

Experience tells us that things will not always work the way they are supposed to work when dealing with real-world data on a standard floating-point arithmetic. In this and the subsequent section we provide answers to the following questions.

- What happens if we should delete a cycle?
- What happens if we cannot find two seed nodes?
- What happens if we cannot find a tree that contains both seed nodes?
- What happens if we cannot compute a new Voronoi node?
- How can we make the numerical computations reliable?
- What is a good precision threshold (“epsilon”)?

As we shall see, the first question truly is an issue even if the algorithm were to be executed for clean data, using exact arithmetic. And all but the last two questions have to be answered if real-world data (that is not required to be clean) is to be handled, even when using exact arithmetic.

3.1. Preserving Voronoi regions

Note that the insertion of the dashed line segment in Fig. 2(a) causes all Voronoi nodes of the Voronoi region of the lower endpoint of this segment to be deleted. This would effectively create a cycle in the structure of nodes and edges that are to be deleted. However, some portion of its bottom-most parabolic bisector has to be preserved.

As a first and simple remedy we routinely split a parabolic bisector at its apex if this apex is part of the bisector. (The apex is the point on the parabolic arc which is closest to the two defining sites.) The split is carried out by inserting a degree-two Voronoi node at the appropriate position. Similarly, we split a bisector defined by two point sites at their mid-point if this point belongs to the bisector.

Fig. 3 depicts all degree-two nodes (as filled disks) that are part of the Voronoi diagram of Fig. 2(a). Note that this includes degree-two nodes at those point sites which have already one line segment incident at them. In the case of Fig. 2(a), this suffices to prevent the removal of the Voronoi regions of the lower endpoint of the segment that is to be inserted. (We explain below what to do if a Voronoi region still would be deleted completely.) As a nice side effect, we get that the clearance always either monotonically increases or monotonically decreases as one moves from one node of a bisector to its other node.

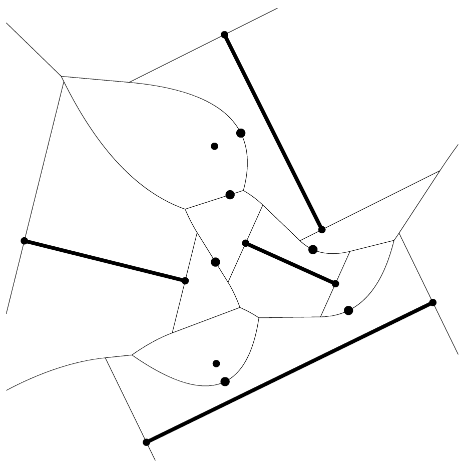


Fig. 3. The degree-two Voronoi nodes are marked by filled disks.

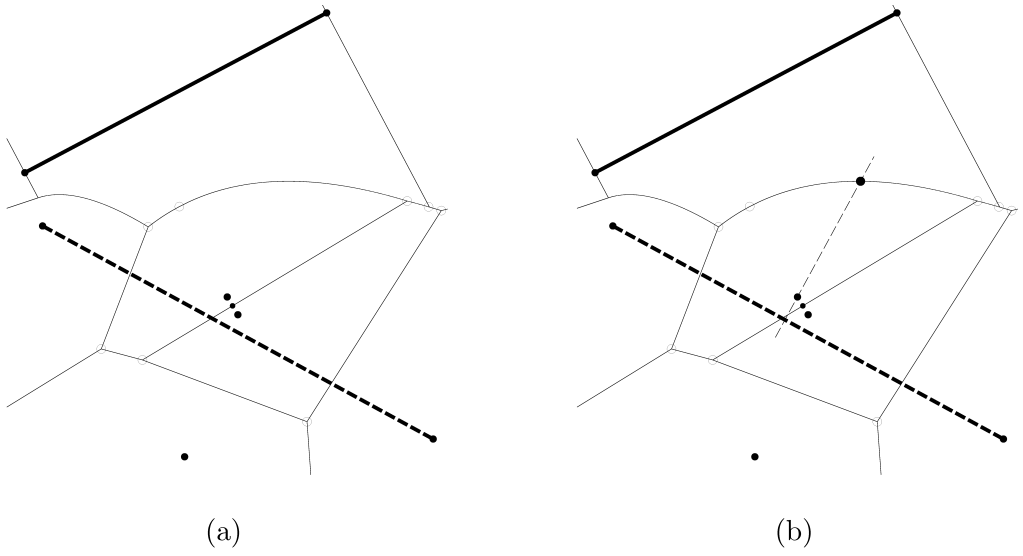


Fig. 4. Inserting the dashed segment would cause a cycle of nodes (marked by circles) to be deleted. We break up the loop by inserting one degree-two node (marked by a filled disk).

3.2. Breaking up cycles

Can it ever happen that we would have to delete Voronoi nodes that form a cycle? Fig. 4(a) shows a simple scenario in which the insertion of the dashed line segment would (rightfully) cause the deletion of a set of Voronoi nodes and edges which form a cycle. Note that no Voronoi region would disappear entirely as the degree-two node between the two point sites that are fairly close would be kept.

The obvious problem is that this cycle contains one bisector which should be preserved partly. In order to solve this problem we scan all edges (bisectors) of a cycle and check whether they contain a point that is guaranteed to be closer to its defining sites than to the new line segment. (In Fig. 4(b), such a point is marked by a filled disk.) Note that only one such bisector can exist per cycle. (Otherwise, the structure to be deleted would break up into disjoint components.)

4. Reliability issues

4.1. Coping with data deficiencies

Real-world data tends to exhibit all types of deficiencies, and a Voronoi code that is supposed to handle real-world data needs special care that goes well beyond a flawless implementation of the basic algorithm. Real-world polygonal areas may have degeneracies – such as zero-length edges, vertices that lie on other edges of a polygon, edges that overlap partially – and they may also have more serious deficiencies, such as (self-)intersections.

Whereas some degeneracies may be intentional (such as a grazing contact between a vertex and a line segment), most deficiencies are the result of numerical or algorithmic problems of the software that generated the data. For instance, many solid modelers still seem to have troubles with generating consistent and clean polyhedral approximations of curved objects. Simplifications of complex polygonal environments (e.g., in GIS applications) are also prone to cause deficiencies. In any case, be the deficiency intentional or by mistake, our algorithm has to cope with it.

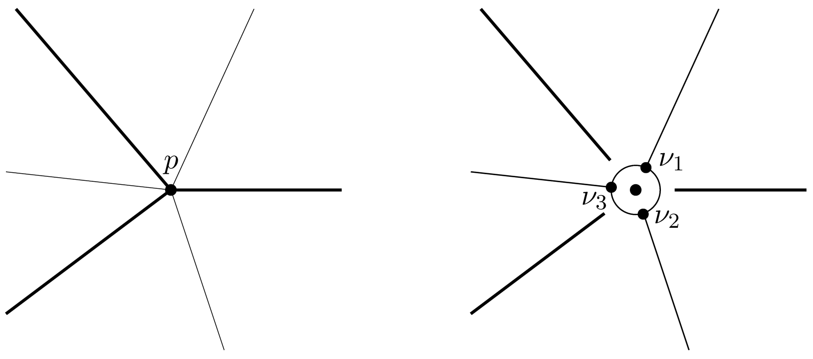
We emphasize that handling polygonal deficiencies cannot be accomplished by resorting to using exact arithmetic instead of the conventional floating-point arithmetic. Vertices may lie on other edges, and interior angles may indeed be 0° or 360°. Also, the use of symbolic perturbation [3] does not solve the problem as it may transform a degenerate polygon into a self-intersecting polygon, thus aggravating the situation. The only remedy is to design an algorithm that is able to cope with any form of polygonal input.

4.1.1. On-the-fly remedy and restart

Whenever vroni encounters a status that should not occur for clean data – e.g., if the two seed nodes determined during the incremental insertion of a new line segment do not end up in one tree of nodes that are to be deleted – it locally checks the data. (We use the word “local” to indicate that the complexity of this check is linear in the number of sites handled so far, as opposed to a full all-pairs test among all sites.) This check of the input data is carried out by checking (a) whether the new segment is duplicate, (b) whether the new segment intersects any already inserted segment, and (c) whether any point site lies on the segment. If such a deficiency is found, the segment(s) are split appropriately and the generation of the Voronoi diagram is restarted from scratch.

Let us point out that the inclusion of local checks of the input data does not guarantee that vroni will handle any corrupted data correctly. After all, any local check of the input data is only carried out if the code encounters a status that should not occur. There is no reason to believe that the insertion of an illegal site necessarily will cause the code to encounter such a status right when it deals with this site. Rather, algorithmic problems might occur later on, and then a local check of the data might fail to reveal the

[3] See Edelsbrunner and Mücke [14].



(a) Degree-3 spike (b) Local Voronoi topology

Fig. 5. Handling spikes.

illegal site. However, experimental tests seem to indicate that it is highly unlikely that vroni terminates successfully while missing a data deficiency. (That is, we are yet to observe such an instance.)

Also, we emphasize that this way of handling multiple deficiencies one after the other is rather time-consuming. However, for the sake of implementational simplicity, we have opted against implementing an update of the already computed Voronoi diagram as segments are split. The only goal of this soundness check is to help vroni process real-world data that might have a few deficiencies. If a user decides to apply vroni to data that resembles a bail of hay then vroni can be expected to survive, but it may require substantial amounts of CPU time. (Basically, its CPU-time consumption will be k times its normal CPU-time consumption for data that exhibits k deficiencies.) If highly corrupted data is suspected then it may be best to let vroni clean the data by running a global soundness check on the data prior to the actual computation of the Voronoi diagram. (Although time consuming, vroni provides a (brute-force) global consistency check as a convenience utility.)

4.1.2. Handling spikes

Since we do not require the input data to be clean, vroni has to handle the case of multiple line segments being incident at the same endpoint. (E.g., Fig. 1(a) contains three segments that are incident at the same endpoint.) Note that the insertion of intersection points, as explained above, also creates endpoints shared by three or more line segments.

Fig. 5(a) shows a zoomed view of the degree-three spike of Fig. 1(a). The local topology of the Voronoi diagram around p is depicted in Fig. 5(b). Obviously, the boundary of the Voronoi region of p contains three Voronoi nodes ν_1, ν_2, ν_3, which coincide with p. We call such Voronoi nodes *degenerate* Voronoi nodes.

Suppose that we insert another line segment, shown dashed in Fig. 6, into the scenario of Fig. 1(c). Note that this line segment is incident at p, too. Thus, the distance of ν_1 to this line segment is zero. Same for ν_2 and ν_3. Deleting all three degenerate nodes would cause the Voronoi region of p to disappear, while deleting none of them would not be consistent with choosing one of them as a seed node. (Recall that we are to choose one seed node each on the Voronoi boundaries of the endpoints of a new line segment.)

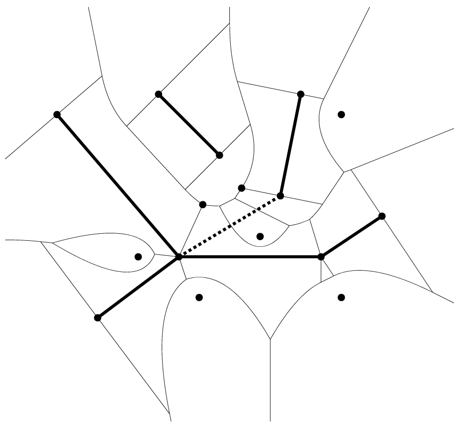


Fig. 6. Modified seed nodes for handling spikes.

Let $\mathcal{B}(p)$ denote the boundary of the Voronoi region of a point site p. To solve the problem, we modify the basic algorithm for finding a seed node as follows. If no segment incident at p has been inserted so far then we pick a seed node from $\mathcal{B}(p)$. Otherwise, $\mathcal{B}(p)$ contains at least one degenerate node. (Recall Section 3.1.) In this case we start a recursive search for a seed node at one of those degenerate nodes. A node is selected as seed node if it is not degenerate and if its clearance disk intersects the new line segment. In Fig. 6, the seed nodes selected for the dashed segment are marked by filled disks. Note that degenerate seed nodes never will be deleted during an incremental update. (Thus, it is guaranteed that in the final Voronoi diagram every point site will have a Voronoi region associated with it.)

4.2. Numerical considerations

4.2.1. Ensuring consistency

Before the incremental insertion starts, we sort all point sites lexicographically, first according to x-coordinates, and second according to y-coordinates. A simple scan through the array of sorted points reveals all true duplicates [4], which are discarded. Then, every input point site is assigned its index in the sorted array of points. As a result, points that have identical coordinates have the same index, and it is trivial to discard duplicate sites. Also, zero-length line segments can be discarded easily. In the sequel, whenever we speak of the index of a point site we will refer to its index in this sorted array.

Care is taken that the floating-point computations do not depend on the actual order of the sites in the list of parameters passed to a subroutine. For instance, we always compute determinants such that the index of the first point is less than the index of the second point, which in turn is less than the index of the third point. If this re-ordering of the vertices causes a change of their cyclic order then the sign of the resulting determinant is inverted. (Trivially, a determinant is zero if any two indices of the points are identical.)

Whenever possible, these indices are also used for deducing incidence and distance relations. For instance, the point of intersection of two line segments is trivially known if the segments share a common endpoint.

[4] I.e., points which have identical coordinates.

4.2.2. Numerical sanity checks

The basic principle of a topology-oriented algorithm is to value topological correctness and consistency higher than numerical correctness. Typically, the "success" of a topology-oriented algorithm is measured in terms of whether or not the output generated meets some topological criteria. It is obvious, though, that the quality of the output of any algorithm depends substantially on the fidelity of the numerical computations used.

Voronoi diagrams consist of both topological data (which describes the structure of the underlying graph) and numerical data (which describes the position of its nodes). Whenever a new Voronoi node is computed, we subject its coordinates to a series of sanity checks that shall help to detect any numerical garbage. (We explain in Section 4.3 how to proceed if one of the following tests fails.)

During the incremental update, a new Voronoi node is computed for every bisector (Voronoi edge) that has one deleted node and one undeleted node. Thus, we can check whether the new node actually lies on the old bisector. (For a line bisector, this test translates to testing whether the new node lies on the line segment defined by the two old nodes.) Similarly, the clearance distance of a new node has to be greater than the minimum clearance of the old nodes, and less than the maximum clearance of the old nodes.

The position of a Voronoi node with respect to a line segment is subject to a constraint, too: since the normal projection of the Voronoi region of a line segment onto the supporting line of the line segment equals (the closure of) the line segment, the normal projection of a new Voronoi node onto its defining line segments has to lie within (the closures of) those line segments. Similarly, a new node has to lie on the same side of its defining line segment as the node that it replaces, except for the current line segment that is inserted into the Voronoi diagram.

4.3. Multi-level recovery process

It does not come as a surprise that the incremental insertion algorithm may encounter a status that should not occur when it is applied to data which is not clean. However, implementations of abstract geometric algorithms are also prone to errors that are beyond the control of the user. Most notably, numerical problems caused by inaccuracies of the floating point arithmetic may manifest themselves later on during the execution of the algorithm. And, last but not least, subtle bugs in the code (or even flaws in the algorithm) may remain undetected for a long time. Still, the code is supposed to finish with some acceptable output, rather than to crash or end up in an infinite loop.

In order to cope with those basic problems that plague most (geometric) algorithms, we have built a multi-level recovery strategy into vroni. This recovery process selects the more aggressive means for making progress towards the final Voronoi diagram the more obvious it gets that the data is corrupted.

As a general design principle, we cast abstract geometric algorithms into actual code such that the code will always yield a result. In an ideal setting, this result will be correct with respect to all the topological and numerical criteria that it has to fulfill. For example, recall that we select a seed node ν by maximizing $\rho_\nu - \delta_\nu$ for all nodes on the boundary of some Voronoi region, cf. Section 2.1. Of course, this approach will yield a valid seed node in an ideal setting. However, it will also yield a "best-possible" seed node if the new site should not violate the clearance disk of any node (due to whatever reason).

4.3.1. Back-up numerical primitives

The main numerical task a Voronoi code needs to handle is the computation of Voronoi nodes. Typically, this involves finding the roots of second-degree polynomials, and solving systems of two linear equations in two unknowns. We refrain from going into any implementational details as any good textbook on numerical mathematics contains advice on how to implement those numerical primitives reliably.

The simple but important key observation is that the numerical reliability of a geometric algorithm/code cannot only be achieved at the lowest level of the standard numerical primitives. That is, once we are to solve a system of two linear equations we can only hope that this system is not ill-conditioned. If it is ill-conditioned then any standard algorithm that attempts to solve it on a floating-point arithmetic will be prone to suffer from numerical instability. Thus, it would be much better to avoid solving this system if it is ill-conditioned.

Rather than relying entirely on the reliability of the standard numerical primitives, we try to avoid any numerically sensitive calculation by using an alternative algorithm if the standard algorithm should be expected to be numerically instable for the data given. For instance, we can obtain a point on the bisector of two lines by computing the point of intersection of the two lines. Obviously, this standard algorithm will be unstable if the two lines are some distance apart and (nearly) parallel. In the latter case, an approach that is much more reliable is to intersect both lines with a third line which is perpendicular to one of the two lines, and to obtain the point sought as the intersection point of the pairwise bisectors of the three lines. Note that this modified approach requires to intersect three pairs of lines rather than only one pair. However, the lines involved in those three tests are roughly perpendicular to each other, and the computation can be expected to be numerically stable.

We implemented at least one alternative back-up routine for every numerical computation required by the Voronoi algorithm. In case that it is difficult to judge which numerical algorithm should be best used we apply both (or all) algorithms, and accept the numerical result which best meets the topological conditions and our numerical sanity checks. (Recall Section 4.2.2.)

4.3.2. Relaxation of epsilon thresholds

Since vroni relies on the standard floating-point arithmetic, it is obvious that it also relies on an appropriately chosen precision threshold (“epsilon”) in order to decide whether or not some numerical value is equal to zero. Just what is an appropriate epsilon?

In the author’s opinion asking the user of a geometric code to choose an appropriate epsilon comes close to asking for witch-craft. Shall the user opt for a high-precision output and specify a small epsilon, or shall the user hope that a larger epsilon might cause the code to work better? To make the situation worse, our experience tells us that one fixed precision threshold will rarely suffice to handle all different input data. That is, different input data may require different precision thresholds.

We decided to implement a different approach to choosing precision thresholds, based on the following observation: A natural lower bound for an epsilon is given by the floating-point precision of the machine used. And a user typically has little problem to specify the maximum distance that two points (out of the unit square) may be apart in order to allow the code to treat them as one point. Thus, we have natural lower and upper bounds for any admissible epsilon.

These bounds can be used to let vroni choose a “good” epsilon automatically. That is, prior to the computation of some numerical data, we set epsilon to the lower bound. If the computation should fail, as indicated by the evaluation of the topological conditions and our numerical sanity checks, we increase epsilon and make another attempt to compute the numerical data. Repeated attempts are made until the data sought has been computed successfully or until epsilon reaches the upper bound. Since the current epsilon is used for both the computation and the evaluation of the conditions for success, increasing epsilon can indeed be expected to cause vroni to use different branches of the algorithm.

If epsilon reaches the upper bound then we run a local soundness check on the input data. (Recall Section 4.1.1.) If local evidence for corrupted data is found then the data is cleaned and the computation of the Voronoi diagram is restarted from scratch. Otherwise, the code enters “desperate mode” (described below).

4.3.3. Desperate mode

If the logic of the algorithm requires the set of nodes to be deleted during an incremental update to form a tree, then supplying the subsequently executed portions of the code with anything but a tree will likely cause the code to loop or crash. Similarly, if a new node is to be inserted into the Voronoi diagram, then the code may subsequently need to access the coordinates of this node.

If the relaxation of the epsilon threshold and the heuristics of the multi-level recovery process have not helped to compute data that meets the topological conditions and numerical sanity checks then the code finally enters “*desperate mode*”. Basically, at this stage we replace “correct” or “optimum” by “best possible”.

If numerical data is sought which could not be computed (e.g., since the computation would have involved a division by zero), the code invokes a routine which computes some approximation of the data, disregarding any numerical sanity checks. Care is taken that the approximation is bullet-proof and does not contain any operation (such as a division or the computation of a square root) which is not defined for all floating-point numbers.

Similarly, if some numerical data could be computed but it failed the numerical sanity checks then vroni accepts the data whichever meets most of the sanity checks. Finally, any violation of a topological condition is “cured” by forcing its validity. For instance, if the code cannot break up a cycle of nodes during the incremental update then desperate mode will force a break-up by inserting a dummy degree-two node randomly on one of the edges of the cycle.

It is important to point out that vroni always first attempts to perform a computation as it would normally perform it, irrelevant of whether or not it had already entered desperate mode once before. Indeed, after intentionally setting the lower bounds on the precision thresholds to odd values, we have already witnessed sample polygons for which vroni proceeded with computing a meaningful Voronoi diagram after it had used desperate mode in order to overcome some violation of the topological conditions or the numerical sanity checks. Anyway, even if the numerical sanity checks should fail continuously – e.g., due to an odd precision threshold – vroni will not get stuck, crash or loop. Rather, due to the topology-oriented approach it will still output a structure that meets the topological conditions specified. (We admit that its output may nevertheless be of little practical use in the event of such a worst-case scenario.)

The typical structure of the computations carried out by vroni is summarized in Algorithm *Typical computational unit of vroni*. As explained, vroni resorts to desperate mode only if all other means have failed to make progress towards a meaningful Voronoi diagram.

We emphasize that reliability does not come for free. The repeated attempts of the algorithm to avoid desperate mode may lead to an increase of the CPU-time consumption. The relaxation of the epsilon threshold effectively may increase the CPU-time consumption by a constant factor (compared to the case where no relaxation is necessary). Checking for local soundness of the input data once accounts for at least $\mathrm{O}(n)$ additional time, where n is the number of sites of $\mathcal{S}$. This decrease of efficiency will hardly be

Algorithm *Typical computational unit of vroni.*

```
 1. $\varepsilon$ = lower_bound;                       (* set epsilon to maximum precision *)
 2. repeat
 3.    $x$ = ComputeData($\varepsilon$);                      (* compute some data *)
 4.    success = CheckConditions($x, \varepsilon$);    (* check topological/numerical conditions *)
 5.    if (not success) then
 6.       $\varepsilon = 10 \cdot \varepsilon$;                       (* relaxation of epsilon threshold *)
 7.       reset data structures appropriately;
 8. until (success OR $\varepsilon >$ upper_bound);
 9. $\varepsilon$ = lower_bound;                                (* make sure to reset epsilon *)
10. if (not success) then
11.    illegal = CheckInput();                  (* check locally for soundness of input *)
12.    if (illegal) then
13.       clean data locally;                       (* fix the problem in the input data *)
14.       restart computation from scratch;
15.    else
16.       $x$ = DesperateMode();                              (* time to hope for the best *)
```

noticeable for small inputs. However, performing multiple restarts in the case of corrupted input data may very well gobble up significant CPU resources for complex polygonal input data. (The actual execution of desperate mode is computationally cheap, though.)

5. Experimental results

5.1. Set-up of experiments

A major goal of this work was to provide experimental evidence that this algorithm is both reliable and efficient. Thus, it seems to be natural to test the code on data arising in practice, and to compare it to other Voronoi codes. Unfortunately, practical data usually does not scale well. That is, due to idiosyncrasies of the test data it is somewhat problematic to predict the behavior of an algorithm unless it can be run on a very large amount of test data.

We tested the reliability of vroni on about 2,450 real-world polygonal data sets which we obtained from companies, colleagues and the web. Our data sets include polygonal cross-sections of human organs, GIS maps[5] of roads and river networks, polygonal outlines of fonts, and boundaries of work-pieces for NC machining or stereo-lithography. Using vroni 's built-in offsetting function, we also generated offset patterns of some data sets, and re-applied vroni to polygonal approximations of those offset patterns. The complexity of our real-world test data ranged from a few tens of segments to more than 524,000 segments, with most data sets having a few thousand segments. See the WWW home-page [29] of vroni for color images of some of the data sets.

[5] All longitude/latitude coordinates were regarded as standard x, y-coordinates.

In order to evaluate the performance of vroni we opted for a different approach, based on synthetic test data. We generated 10 polygons each for sets of 2^i vertices, where $3 \leqslant i \leqslant 15$. (Thus, the numbers of segments ranged between 8 and 32,768.) This procedure was repeated for four different classes of test polygons, which we call "RPG" polygons. In addition, polygons with special characteristics were created, such as polygonal approximations of an ellipse. In total, we tested vroni on about 600 synthetic data sets.

All polygons of the first three classes were generated by means of RANDOMPOLYGONGENERATOR (RPG), cf. [4], which is a tool designed for the machine generation of pseudo-random polygonal test data. All polygons of the first class of test polygons, which was dubbed "random", were generated by using RPG's heuristic 2-opt Moves, applied to points uniformly distributed within the unit sphere. As discussed in [4], this heuristic generates highly complex pseudo-random polygons. Our second class of test polygons, dubbed "smoother", was generated by smoothing the output of 2-opt Moves four times by means of RPG's algorithm Smooth. (Smooth produces a new simple polygon with twice the number of vertices by replacing vertex v_i of the polygon by the two new vertices $(v_{i-1} + 3v_i)/4$ and $(v_{i+1} + 3v_i)/4$. The result of a repeated application of Smooth is a fairly smooth polygon which resembles a straight-line approximation of a free-form curve.) Our third class of test polygons, called "smooth", was generated by applying Smooth only twice to the output of 2-opt Moves. The fourth class, called "thinned" polygons, was obtained by randomly clipping three quarters of the ears of the "random" polygons by means of our triangulation code FIST [27].

Fig. 7 depicts four polygons with 64 vertices, one for each of our four "RPG" classes. The difference in the characteristics of the polygons (and of the corresponding Voronoi diagrams) is clearly visible. The "thinned" polygons tend to cover much less space than the "random" polygons do. Both the "thinned" and the "smoother" polygons have vertices that are (highly) non-uniformly distributed. Also, the "smoother" polygons tend to have very short edges and internal angles close to 180°. Of course, the complexity of the polygons within each class increases as the number of vertices increases.

5.2. Tuning vroni

All tests reported here were carried out on a Sun Ultra 30, running Solaris 2.6 on a 296 MHz processor. (Our test machine had 384 MB of main memory, but memory clearly was no issue for any of the codes.) The CPU-time consumption of our code, and of the other codes, was obtained by using the C system function "getrusage()". We report both the system and the user time. Of course, any file I/O and similar preprocessing is not included in the timings reported. All CPU times are given in milliseconds.

We started with experimentally determining the optimal resolution of the grid used for the nearest neighbor search. (Recall Section 2.3.) The chart shown in Fig. 8 plots the average CPU times (per point) versus the scale factor $w \cdot h$ of the grid, averaged over data sets with 400–2,000 points, 2,000–10,000 points, 10,000–50,000 points and more than 50,000 points. No specific distribution of points was tested. Rather, we ran vroni on all our data sets and timed only the generation of the point Voronoi diagram. As it could be expected, the less regularly spaced the points are the higher the resolution of the grid should be. This is particularly noticeable for the data sets with 50,000 or more points. (Virtually all these complex data sets are GIS data which model irregularly distributed objects such as contour lines, coast lines or road networks.) We decided to set $w \cdot h$ to 2, which seems to be a fair choice for most of the point sets, for a wide range of cardinalities. Thus, the remaining tests are based on a grid that had roughly $2n$ cells.

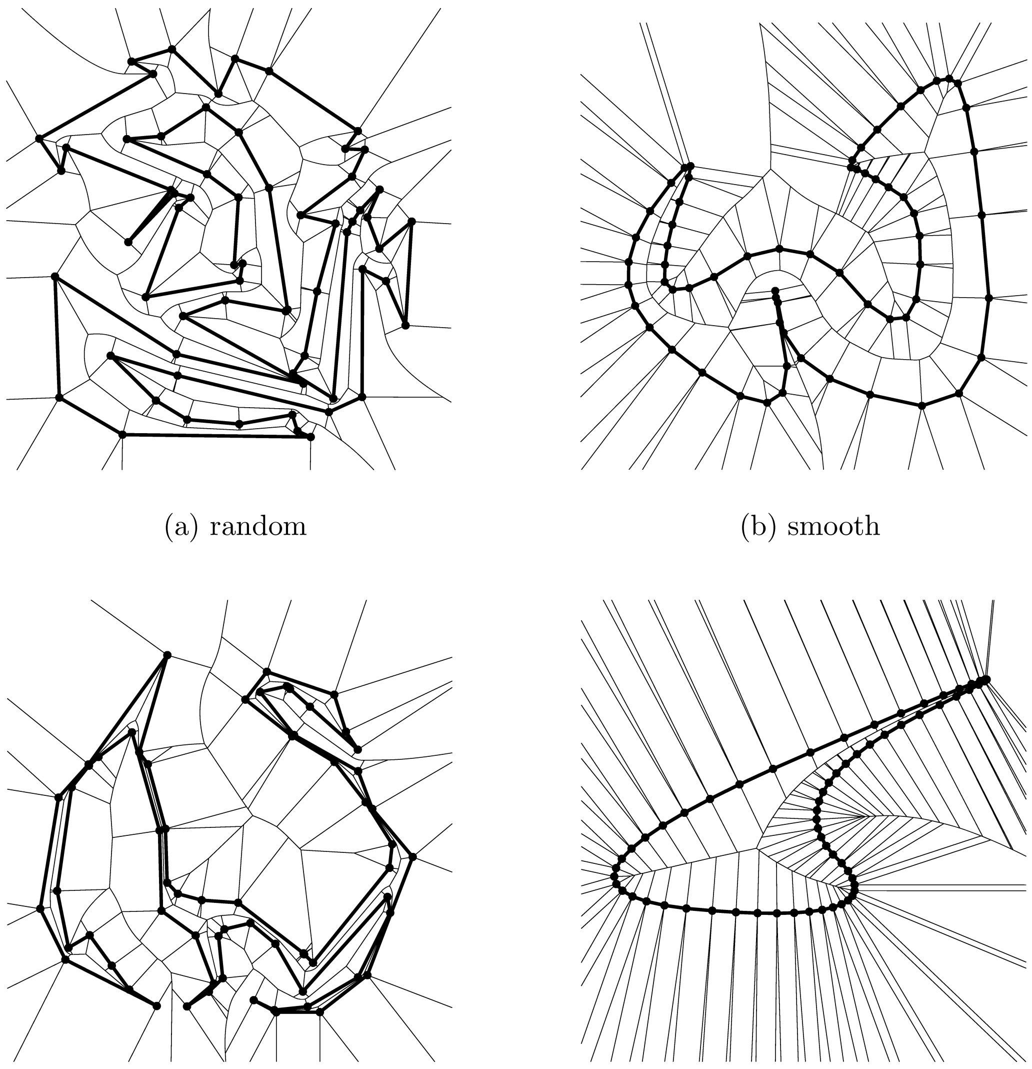


Fig. 7. Sample 64-gons for the four classes of “RPG” polygons.

By means of compile-time switches we can select different orders for inserting the line segments: sequential (in the order they are given in the input file), random, sorted (in increasing order of their length), inversely sorted (in decreasing order of their length). Our tests clearly showed that there is nothing to gain by inserting the line segments not in random order. All other insertion orders performed only slightly better than random insertion on a few data sets, but significantly worse on several other data sets. Thus, in all tests reported below vroni inserted the points and line segments into the Voronoi diagram in random order.

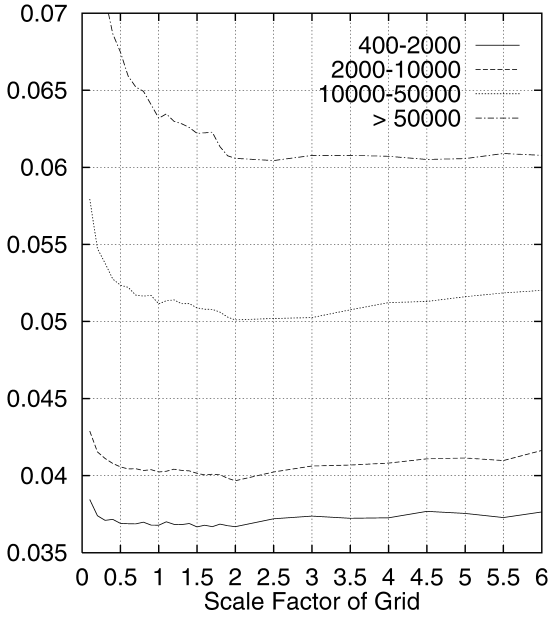


Fig. 8. CPU time (per point) versus scale factor $w \cdot h$ of the grid, for computing the Voronoi diagram of sets of points of four different classes of cardinalities.

5.3. Reliability of the Voronoi codes tested

Prior to testing the CPU-time consumption, we focused on the reliability of vroni and of Voronoi codes designed and implemented by others. Besides our own code we tested the following codes:

- Imai's Fortran code plvor [31],
- Seel's C++ code avd [48],
- Sethia's C++ code pvd [49].

Testing these other Voronoi codes turned out to be a difficult task as they can not handle all our test data. Here we summarize our impressions of the reliability of the other Voronoi codes.

Imai's plvor (version 0.9) repeatedly generated cryptic warning messages and ran into floating-point exceptions for collections of line segments that did not intersect except at their endpoints. Also, we had to manually terminate several test runs when plvor consumed extreme amounts of CPU time (and, likely, was looping). Unfortunately, as plvor is not documented, the meaning of its warning messages remained unclear. However, a visual inspection of PostScript output generated by plvor clearly revealed that those warning messages were an indication of serious problems that plvor had run into: if plvor had output multiple warning messages then the numerical properties of the Voronoi diagram generated were seriously flawed, with Voronoi edges scattered around in no apparent relation to the input data. We gained the impression that plvor finished prematurely for some of the larger input files. Also, contrary to the claims stated in [31], plvor seems to be incapable of handling line segments that meet at common endpoints if the line segments are input as individual segments (rather than as a polygonal chain).

We can only speculate why plvor performed less than perfect for a good deal of our simple polygons. One reason might be that its input file is not free-format ASCII but rather requires a rigid input format, with only a few digits of precision allowed per numerical value. (Also, preparing an input file for plvor is a multi-step process that requires several other tools provided with plvor .) Potentially, this restricted precision in the input data may cause points to coincide and line segments to intersect that all the other Voronoi codes regard as disjoint. In any case, plvor is only applicable to clean data. Summarizing, plvor could not handle most of our (real-world) test data.

Seel's avd (version 1.1, LEDA 4.1) performed correctly on all data sets that we applied it to. As it relies on LEDA's exact arithmetic, it should not run into numerical problems. To be more precise, it is the only code among the Voronoi codes tested that would be guaranteed to generate correct Voronoi diagrams if its implementation could be proved to be correct. Indeed, the current version of avd (and LEDA) seems to handle any arrangement of point sites and line segments successfully as long as the data is not corrupted. (It dies gracefully when it encounters two line segments that intersect in an interior point.) However, as documented below, it is too slow to be applied to large data sets, and we ran it only on data sets with 2,048 or less points/segments. Thus, our tests of avd are not conclusive in the sense that algorithmic problems could occur for more complex input data. However, it seems that a user is more likely to loose patience than to see avd fail.

Sethia's pvd is restricted to clean polygonal data that forms the boundary of a multiply-connected area. That is, it can only handle simple polygons which do not intersect pairwise and where one polygon contains all the other polygons in its interior. In particular, it cannot deal with open polygonal chains or individual line segments, which precluded it from handling most of our real-world test data. We saw it crash a few times but, in general, it seems to process clean data fairly reliably.

Summarizing, vroni is the only code among the Voronoi codes tested that can handle any polygonal input of arbitrary complexity. Thus, when running tests for the CPU-time comparison of the code, we made sure that all test polygons were simple polygons. (Of course, this means that vroni did not need to spend time on cleaning the data.)

5.4. *Experimental CPU-time consumption*

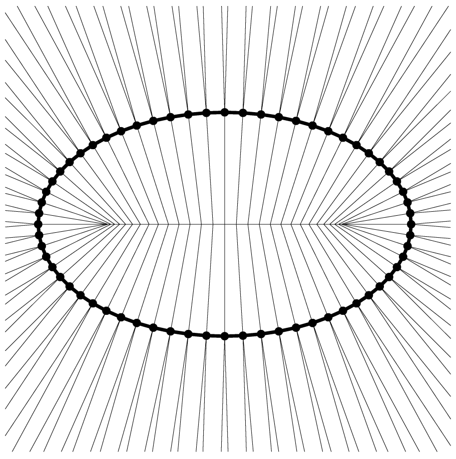
Finally, we turned our attention to a comparison of vroni with the other Voronoi codes. Imai's plvor, Sethia's pvd and vroni were compiled with the SunPro cc/CC/f77 (4.2) compiler, using the optimization level "-xO4". Seel's avd was compiled with GNU's g++ (2.95.1), using the optimization level "-O". (Our attempts to compile it with the SunPro CC failed.)

Table 1 lists the average CPU-time consumptions (per segment, in milliseconds) for the "RPG" polygons. (Recall Fig. 7.) The individual entries of this table were generated by averaging the CPU times over each group of polygons with the same number of vertices. The entry "n/a" means that a code consumed excessive CPU time (and was stopped before it finished), that it generated a strikingly incorrect Voronoi diagram, or that it crashed. (In any case, the timings for plvor are to be taken with a grain of salt, as plvor might have finished prematurely for some of the larger data sets without us taking notice of the problem.)

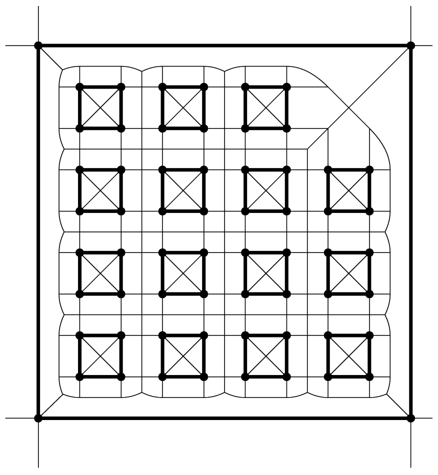
All three codes which are based on a conventional floating-point arithmetic – plvor, pvd and vroni – processed the "RPG" data sets significantly faster than the LEDA-based avd. We note that pvd computes only "one half of the Voronoi diagram", as it computes the Voronoi diagram only for the interior of a polygon, while avd, plvor, and vroni compute the Voronoi diagram both in the interior and the exterior of a polygon. (Recall that these three codes do not require the input to form a closed polygon.)

Table 1
CPU time (per segment) for the different Voronoi codes applied to the "RPG" and "holes" polygons.

	RPG				Holes			
Size	avd	plvor	pvd	vroni	avd	plvor	pvd	vroni
64	50.55	1.359	0.164	0.109	108.6	1.250	0.469	0.156
128	75.26	0.922	0.160	0.107	119.4	0.859	0.625	0.078
256	119.8	0.657	0.162	0.108	302.2	0.625	1.289	0.078
512	209.6	0.565	0.158	0.115	324.8	0.508	3.477	0.117
1024	404.4	0.517	0.167	0.114	1007.0	0.449	15.53	0.098
2048	758.9	0.511	0.167	0.117	900.0	0.444	99.23	0.098
4096	n/a	0.561	0.175	0.126	4296.0	0.532	732.2	0.105
8192	n/a	0.643	0.185	0.136	n/a	0.621	5671.0	0.117
16384	n/a	0.898	0.209	0.148	n/a	0.982	n/a	0.132
32768	n/a	1.293	0.234	0.156	n/a	1.281	n/a	0.142



(a) ellipse



(b) holes

Fig. 9. Sample 64-gons for "ellipse" and "holes" test data.

A closer inspection of the timings for the "RPG" polygons let us suspect that all other codes are more vulnerable than vroni to irregularly distributed input data. Thus, we ran the Voronoi codes on straight-line approximations of an "ellipse", see Fig. 9(a) and the entries in Table 2. While vroni handles the "ellipse" polygons even slightly more efficiently than the "RPG" polygons, all other three Voronoi codes

Table 2
CPU time (per segment) for the different Voronoi codes applied to the “ellipse” and “circle” polygons.

	Ellipse				Circle			
Size	avd	plvor	pvd	vroni	avd	plvor	pvd	vroni
64	82.66	1.406	0.156	0.156	215.3	1.250	0.625	0.156
128	133.4	1.016	0.312	0.156	400.0	0.938	0.938	0.156
256	262.9	1.211	0.469	0.117	400.4	0.625	1.562	0.117
512	674.2	3.105	0.664	0.117	500.1	0.586	2.832	0.137
1024	1892.0	11.08	1.182	0.098	566.4	0.703	5.352	0.127
2048	6059.0	43.38	2.222	0.107	987.7	1.133	10.23	0.146
4096	n/a	49.69	4.187	0.117	2003.0	2.473	22.57	0.164
8192	n/a	52.34	8.179	0.121	4997.0	4.980	48.78	0.190
16384	n/a	647.4	18.61	0.135	n/a	6.246	99.21	0.219
32768	n/a	n/a	38.56	0.145	n/a	10.92	203.2	0.244

need significantly more CPU time. This increase in the CPU-time consumption is particularly apparent for avd and plvor. It is interesting to learn that avd and plvor perform much better when being applied to increasingly finer polygonal approximations of a circle, see Table 2; “circle” polygons cause pvd to become rather inefficient for larger input sizes. Both pvd and avd also struggled to process multiple square holes within one square boundary polygon (cf. Fig. 9(b)): see the entries for the “holes” data in Table 1.

We have already seen that vroni tends to run faster on arrangements of comparatively short line segments, even if the line segments are clustered or irregularly distributed. How would it handle a collection of rather long segments, such as the edges of star-shaped polygons? Fig. 10(a) depicts a sample star-shaped polygon (with vertices distributed uniformly within the unit sphere), and Table 3 shows that vroni does indeed consume more time than normal for star-shaped polygons. Likely, this increase in the CPU-time consumption is due to the fact that long line segments arranged among a set of uniformly distributed vertices cause lots of cycles during the process of deleting Voronoi nodes (cf. Section 3.2). Interestingly, star-shaped polygons do not alter the behavior of plvor (which is also based on an incremental insertion of the line segments), and avd even seems to favor this type of data. (It still is considerably slower than any of the other three codes, though.)

The worst-case input for vroni seems to be a set of line segments which all are incident upon one point. (See Fig. 10 for such a “spike”.) When applied to “spike” data, vroni exhibits a roughly quadratic CPU-time consumption. The LEDA-based avd handles a “spike” correctly but also shows a roughly quadratic CPU-time consumption. In particular, it remains drastically slower than vroni. Both plvor and pvd cannot process a “spike”. This worst-case behavior of vroni could be avoided (at least partially) by a smarter handling of multiple line segments that share a common endpoint. However, we did not sense any practical need that would justify additional implementational efforts to cure this problem.

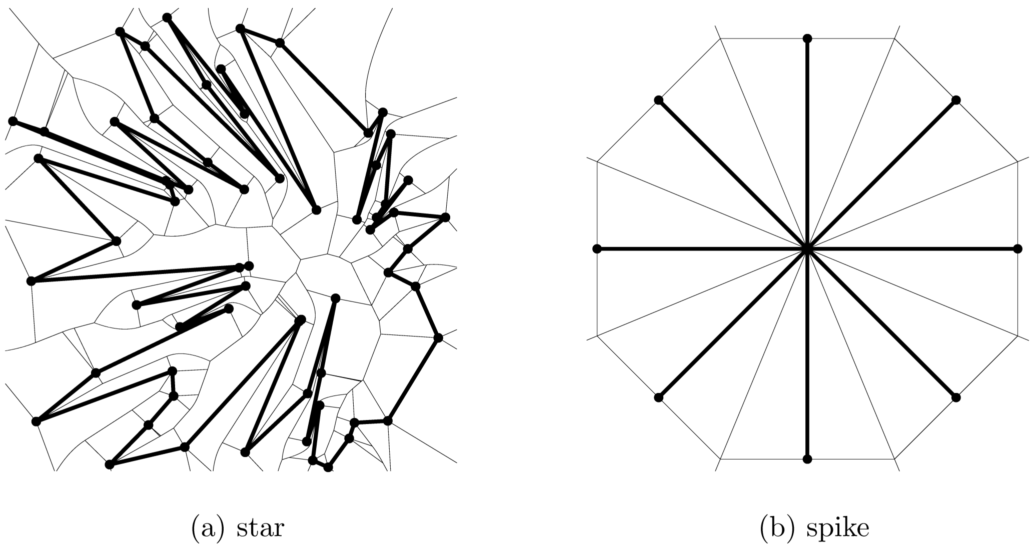


Fig. 10. Sample 64-gon for the "stars" data, and a degree-8 "spike".

Table 3
CPU time (per segment) for the different Voronoi codes applied to the "stars" and "spikes" data sets.

	Stars				Spikes			
Size	avd	plvor	pvd	vroni	avd	plvor	pvd	vroni
64	46.56	1.250	0.156	0.156	94.53	n/a	n/a	0.156
128	72.58	0.938	0.156	0.078	169.6	n/a	n/a	0.234
256	96.76	0.586	0.156	0.156	330.3	n/a	n/a	0.312
512	219.8	0.508	0.156	0.156	675.4	n/a	n/a	0.527
1024	319.9	0.449	0.175	0.156	1340.0	n/a	n/a	1.055
2048	755.5	0.444	0.209	0.181	2722.0	n/a	n/a	2.466
4096	n/a	0.454	0.241	0.205	5451.0	n/a	n/a	5.391
8192	n/a	0.546	0.288	0.239	n/a	n/a	n/a	13.95
16384	n/a	0.713	0.385	0.275	n/a	n/a	n/a	34.42
32768	n/a	1.362	0.585	0.317	n/a	n/a	n/a	73.18

It is fair to assume that a high-degree "spike" will hardly ever be part of any real-world input to a Voronoi algorithm. Can we assume that the CPU-time consumption of vroni applied to real-world data will be of the same order as the entries listed in Tables 1 and 2? The chart of Fig. 11 plots the ratio of the CPU times over $n \log_2 n$ for vroni applied to all our synthetic and real-world test data with $n \geqslant 2^9 = 512$

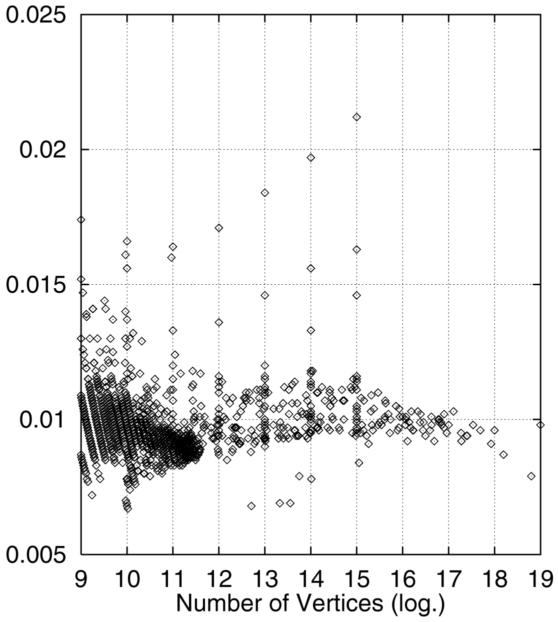


Fig. 11. CPU times divided by $n \log_2 n$ for vroni applied to all test data sets with $n \geqslant 2^9 = 512$ segments (except for the "spikes" data).

segments, except for the "spikes" data. (The data size is denoted by powers of two.) Note that all the CPU-time markers that do not lie on the vertical grid lines – i.e., which correspond to data sets whose cardinalities are no powers of two – represent timings for real-world data sets. Also, any marker whose "size"-coordinate is greater than 15 (for $2^{15} = 37{,}768$ segments) represents a real-world data set. The plot indicates that vroni tends to perform even better on real-world data than it does on our synthetic data.

The general pattern that can be observed in Fig. 11 can easily be cast into a formula. For all our data sets with $n \geqslant 512$ segments, again with the exception of the "spikes" data, vroni consumed $cn \log_2 n$ milliseconds, with $0.0067 \leqslant c \leqslant 0.0212$ and an average of $c = 0.0098$. Summarizing, vroni does not consume excessive amounts of CPU time when applied to real-world data but is as efficient as it could be expected based on our test results for synthetic data.

Finally, the reader is cautioned that any comparison of algorithms implemented by different programmers has to be taken with a grain of salt. Different programming languages require different compilers, and the optimizations carried out may differ even for the same optimization level within the same family of compilers. (Most notably, Seel's avd was not compiled with the SunPro family of compilers, and a weaker optimization level was used.) Also, minor differences in the CPU-time consumption may be caused by different programming styles, or different efforts placed on optimizing a code. However, since Imai's and Seel's codes are publicly available we feel that it is justified to assume that their codes also are optimized, at least to some extent. And our experience tells us that a different compiler/optimization set-up should indeed be expected to yield slightly different test results, but the overall picture will remain the same.

5.5. Robustness without desperate mode

Our Voronoi code vroni relies on a topology-oriented approach combined with a multi-level recovery process in order to prevent it from crashing. Thus, vroni cannot crash[6] and will always terminate with a Voronoi diagram which meets the topological conditions. The experience gained from applying vroni to real-world and synthetic data certainly concurred with its predicted reliability: we are yet to see it crash. Thus, the multi-level recovery process, which ultimately leads to desperate mode, seems to work in theory and in practice.

However, we note that the availability of such a recovery process can easily conceal bugs in the core of the Voronoi code. Obviously, vroni should not end up in desperate mode on a routine basis. In order to check its robustness we disabled desperate mode (but did keep all the other back-up heuristics of the recovery process, including the relaxation of epsilon thresholds). Then, we re-ran the tests for all our synthetic and real-world data sets: vroni finished successfully for all the data sets, with what seemed to be correct Voronoi diagrams.

Checking the correctness of a Voronoi diagram computed is not entirely trivial, though. We decided to check the topological consistency of a Voronoi diagram automatically, and used visual inspections to check for obvious numerical deficiencies. While it might not be particularly simple to notice any deficiency in the Voronoi diagram of a complex data set visually, a dense pattern of offset curves – computed by means of vroni 's offsetting function, based on the Voronoi diagram – is very likely to reveal any deficiency quickly. In particular, experience tells us that numerical deficiencies of the Voronoi diagram translate to spikes in the offset curves which are easily spotted visually, and which cause the offset curves to be not disjoint. The pairwise disjointness of the curves of an offset pattern was checked automatically by re-applying vroni to a tight polygonal approximation of the offset pattern; this also helped to give the code an additional work-out on more data. For small data sets we also checked the correctness of the positions of the Voronoi nodes (up to the floating-point precision used by the code) by means of brute-force distance computations.

Of course, even such extensive tests can only show the existence of a bug, but they do not help to prove that there is no bug lurking somewhere in the code. As with any other implementation of a non-trivial geometric algorithm, irrelevant of whether it is based on floating-point or exact arithmetic, confidence into its actual reliability or suitability for a specific purpose will only be established by satisfactory use. However, since vroni has already successfully handled real-world data within industrial applications, we are confident that its design principle will prove itself in practice.

6. Conclusion

We discussed an incremental algorithm for computing Voronoi diagrams of points and line segments in 2D. The algorithm is based on Sugihara and Iri's topology-oriented approach, a very careful implementation of the numerical computations required, an automatic relaxation of epsilon thresholds, and a multi-level recovery process combined with desperate mode. Extensive experimental tests with both synthetic and real-world test data have shown that the resulting code, vroni, is reliable indeed.

[6] Well, unless there were a bug in the actual implementation . . . .

An experimental comparison also demonstrated clearly that vroni is faster than Imai's plvor, Seel's avd and Sethia's pvd. In a computing environment similar to our 296 MHz Sun Ultra 30, vroni can be expected to consume $0.01n\log_2 n$ milliseconds in order to compute the Voronoi diagram of n line segments. In particular, it is orders of magnitude faster than avd, which is the only (publicly available) Voronoi code that could also handle all the data we applied it to.

Work on extending vroni to circular arcs has been started. Future work will include the practical coupling of vroni with a library for doing exact arithmetic. It would also be interesting to extend vroni to compute Voronoi diagrams on the surface of a sphere, which would enable it to handle GIS data that contains longitude/latitude coordinates without errors caused by a mapping to standard x, y-coordinates.

Acknowledgements

This work has benefited from discussions with Joe Mitchell and Saurabh Sethia (SUNY Stony Brook). Our experiments were run on a Sun Ultra 30 donated by Sun Microsystems, and involved Voronoi codes made publicly available by Michael Seel and Toshiyuki Imai. We could test our code on real-world data obtained from Mike Ady, Gill Barequet, Regina Estkowski, Jack Snoeyink, Markus Thoma, Baltic Drainage Basin Project, Bridgeport Machines, British Antarctic Survey, Digital Chart of the World, EOS GmbH, Minnesota River Basin Data Center, Taglio s.r.l., and U.S. Geological Survey.

References

[1] A. Aggarwal, L.J. Guibas, J. Saxe, P.W. Shor, A linear-time algorithm for computing the Voronoi diagram of a convex polygon, Discrete Comput. Geom. 4 (6) (1989) 591–604.

[2] H. Alt, O. Schwarzkopf, The Voronoi diagram of curved objects, in: Proc. 11th Annu. ACM Sympos. Comput. Geom., Vancouver, BC, Canada, 1995, pp. 89–97.

[3] N. Amenta, M. Bern, Surface reconstruction by Voronoi filtering, Discrete Comput. Geom. 22 (4) (1999) 481–504.

[4] T. Auer, M. Held, Heuristics for the generation of random polygons, in: Proc. 8th Canad. Conf. Comput. Geom., Ottawa, Canada, Carleton University Press, August 1996, pp. 38–44.

[5] F. Aurenhammer, Voronoi diagrams: A survey of a fundamental geometric data structure, ACM Comput. Surv. 23 (1991) 345–405.

[6] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, M. Yvinec, Evaluation of a new method to compute signs of determinants, in: Proc. 11th Annu. ACM Sympos. Comput. Geom., Vancouver, BC, Canada, 1995, pp. C16–C17.

[7] M. Benouamer, P. Jaillon, D. Michelucci, J.-M. Moreau, A Lazy solution to imprecision in computational geometry, in: Proc. 5th Canad. Conf. Comput. Geom., Waterloo, Ontario, Canada, 1993, pp. 73–78.

[8] H. Brönnimann, M. Yvinec, Efficient exact evaluation of signs of determinants, Algorithmica 27 (1) (2000) 21–56.

[9] C. Burnikel, Exact computation of Voronoi diagrams and line segment intersections, Ph.D. Thesis, MPI-Saarbrücken, Saarbrücken, Germany, 1996.

[10] C. Burnikel, R. Fleischer, K. Mehlhorn, S. Schirra, Efficient exact geometric computation made easy, in: Proc. 15th Annu. ACM Sympos. Comput. Geom., Miami Beach, FL, June 1999, pp. 341–350.

[11] C. Burnikel, K. Mehlhorn, S. Schirra, How to compute the Voronoi diagram of line segments: Theoretical and experimental results, in: J. van Leeuwen (Ed.), Algorithms Annu. European Symp. Algorithms'94, Utrecht, Netherlands, Lecture Notes in Computer Science, Vol. 855, 1994, pp. 227–237.
[12] F. Chin, J. Snoeyink, C.-A. Wang, Finding the medial axis of a simple polygon in linear time, in: Proc. 6th Annu. Internat. Sympos. Algorithms Comput., Lecture Notes in Computer Science, Vol. 1004, Springer, 1995, pp. 382–391.
[13] O. Devillers, Randomization yields simple $O(n \log^{\star} n)$ algorithms for difficult $\Omega(n)$ problems, Internat. J. Comput. Geom. Appl. 2 (1) (1992) 97–111.
[14] H. Edelsbrunner, E.P. Mücke, Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms, ACM Trans. Graph. 9 (1) (1990) 66–104.
[15] M. Etzion, A. Rappoport A boundary sampling algorithm for computing the Voronoi graph of a 2-D polygon, Technical report, Institute of Computer Science, The Hebrew University, Jerusalem, Israel, 1996.
[16] S. Fortune, A sweepline algorithm for Voronoi diagrams, Algorithmica 2 (2) (1987) 153–174.
[17] S. Fortune, Stable maintenance of point set triangulations in two dimensions, in: Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci., 1989, pp. 494–505.
[18] S. Fortune, Voronoi diagrams and Delaunay triangulations, in: D.-Z. Du, F. Hwang (Eds.), Computing in Euclidean Geometry, World Scientific, 1992.
[19] S. Fortune, Introduction, Algorithmica 27 (1) (2000) 1–4.
[20] S. Fortune, C.J. Van Wyk, Efficient exact arithmetic for computational geometry, in: Proc. 9th Annu. ACM Sympos. Comput. Geom., San Diego, CA, May 1993, pp. 163–172.
[21] S. Fortune, C.J. Van Wyk, Static analysis yields efficient exact integer arithmetic for computational geometry, ACM Trans. Graph. 15 (3) (1996) 223–248.
[22] C.M. Gold, P.M. Remmele, T. Roos, Voronoi diagrams of line segments made easy, in: Proc. 7th Canad. Conf. Comput. Geom., Québec City, Québec, Canada, August 1995, pp. 223–228.
[23] H.N. Gürsoy, N.M. Patrikalakis, An automatic coarse and fine surface mesh generation scheme based on medial axis transform: Part I Algorithm, Engrg. Comput. 8 (1992) 121–137.
[24] M. Held, On the computational geometry of pocket machining, Lecture Notes in Computer Science, Vol. 500, Springer, 1991.
[25] M. Held, Efficient and reliable triangulation of polygons, in: Proc. Comput. Graphics Internat. '98, Hannover, Germany, June 1998, pp. 633–643.
[26] M. Held, Voronoi diagrams and offset curves of curvilinear polygons, Comput. Aided Design 30 (4) (1998) 287–300.
[27] M. Held, FIST: Fast industrial-strength triangulation of polygons, Algorithmica (2001), to appear.
[28] M. Held, G. Lukács, L. Andor, Pocket machining based on contour-parallel tool paths generated by means of proximity maps, Comput. Aided Design 26 (3) (1994) 189–203.
[29] http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html.
[30] K. Hoff III, T. Culver, J. Keyser, M. Lin, D. Manocha, Fast computation of generalized Voronoi diagrams using graphics hardware, in: Comput. Graphics (SIGGRAPH '99 Proc.), Los Angeles, CA, August 1999, pp. 277–286.
[31] T. Imai, A Topology oriented algorithm for the Voronoi diagram of polygons, in: Proc. 8th Canad. Conf. Comput. Geom., Ottawa, Canada, Carleton University Press, August 1996, pp. 107–112.
[32] V. Karamcheti, C. Li, I. Pechtchanski, C. Yap, A core library for robust numeric and geometric computation, in: Proc. 15th Annu. ACM Sympos. Comput. Geom., Miami Beach, FL, June 1999, pp. 351–359.
[33] D.G. Kirkpatrick, Efficient computation of continuous skeletons, in: Proc. 20th Annu. IEEE Sympos. Found. Comput. Sci., 1979, pp. 18–27.
[34] D.T. Lee, Medial axis transformation of a planar shape, IEEE Trans. Pattern Anal. Machine Intell. PAMI-4 (4) (1982) 363–369.
[35] D.T. Lee, R.L. Drysdale, Generalization of Voronoi diagrams in the plane, SIAM J. Comput. 10 (1981) 73–87.

[36] G. Liotta, F.P. Preparata, R. Tamassia, Robust proximity queries: An illustration of degree-driven algorithm design, SIAM J. Comput. 28 (3) (1998) 864–889.
[37] N. Mayya, V.T. Rajan, An efficient shape-representation scheme using Voronoi skeletons, Pattern Recogn. Lett. 16 (2) (1995) 147–160.
[38] K. Mehlhorn, S. Näher, LEDA. A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1998.
[39] S. Meyer, A quick algorithm for the computation of medial axes of polygons and closed curves, Preprint 416/1994, Fachbereich 3, TU Berlin, Berlin, Germany, 1994.
[40] V.J. Milenkovic, Robust construction of the Voronoi diagram of a polyhedron, in: Proc. 5th Canad. Conf. Comput. Geom., Waterloo, Canada, 1993, pp. 473–478.
[41] V.J. Milenkovic, Shortest path geometric rounding, Algorithmica 27 (1) (2000) 57–86.
[42] T. Ohya, M. Iri, K. Murota, Improvements of the incremental method for the Voronoi diagram with computational comparison of various algorithms, J. Oper. Res. Soc. Japan 27 (4) (1984) 306–336.
[43] A. Okabe, B. Boots, K. Sugihara, Spatial Tesselations: Concepts and Applications of Voronoi Diagrams, Wiley, 1992.
[44] J.M. Oliva, M. Perrin, S. Coquillart, 3D reconstruction of complex polyhedral shapes from contours using a simplified generalized Voronoi diagram, Comput. Graph. Forum 15 (3) (1996) 397–408.
[45] H. Persson, NC machining of arbitrarily shaped pockets, Comput. Aided Design 10 (3) (1978) 169–174.
[46] F.P. Preparata, The medial axis of a simple polygon, in: Proc. 6th Math. Found. of Comput. Sci., Lecture Notes in Computer Science, Vol. 53, Springer, 1977, pp. 443–450.
[47] S. Schirra, R. Veltkamp, M. Yvinec, The CGAL Reference Manual, 1999, Release 2.0.
[48] M. Seel, An accurate arithmetic implementation of line segment AVDs, Technical report, MPI-Saarbrücken, Saarbrücken, Germany, September 1996.
[49] S. Sethia, M. Held, J.S.B. Mitchell, PVD: A stable implementation for computing Voronoi diagrams of polygonal pockets, in: Workshop Algorithm Engineering Experiments (ALENEX '01), Washington, DC, January 2001.
[50] J.R. Shewchuk, Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator, in: 1st ACM Workshop Appl. Comput. Geom., Philadelphia, PA, May 1996, pp. 124–133.
[51] V. Srinivasan, L.R. Nackman, J.-M. Tang, S.N. Meshkat, Automatic mesh generation using the symmetric axis transform of polygonal domains, Proc. of the IEEE 80 (9) (1992) 1485–1501.
[52] K. Sugihara, M. Iri, Construction of the Voronoi diagram for 'One Million' generators in single-precision arithmetic, Proc. of the IEEE 80 (9) (1992) 1471–1484.
[53] K. Sugihara, M. Iri, H. Inagaki, T. Imai, Topology-oriented implementation – an approach to robust geometric algorithms, Algorithmica 27 (1) (2000) 5–20.
[54] P.J. Vermeer, Medial axis transform to boundary representation conversion, Ph.D. Thesis, Computer Science Department, Purdue University, West Lafayette, IN, 1994.
[55] C. Yao, J. Rokne, A straightforward algorithm for computing the medial axis of a simple polygon, Internat. J. Comput. Math. 39 (1991) 51–60.
[56] C.K. Yap, An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments, Discrete Comput. Geom. 2 (4) (1987) 365–393.
[57] C.K. Yap, Robust geometric computation, in: J.E. Goodman, J. O'Rourke (Eds.), CRC Handbook of Discrete and Computational Geometry, CRC Press, 1997, pp. 653–668.
[58] C.K. Yap, T. Dubé, The exact computation paradigm, in: D.-Z. Du, F.K. Hwang (Eds.), Computing in Euclidean Geometry, World Scientific, 1995, pp. 452–492.