Constrained Delaunay Triangulation Revisited

J-M. Moreau
Dept INFA,
E.M.S.E.
158, Cours Fauriel
F Saint-Étienne, 42023 Cedex
E-mail: moreau@emse.fr

P. Volino*
Centre Universitaire d'Informatique
Université de Genève
24, rue du Général Dufour
CH 1211 Genève
E-mail: pascal@cui.unige.ch

Abstract

Paul Chew ([3], and [4]) has given an elegant and optimal $O(n \log n)$ algorithm to construct a Delaunay triangulation for the embedding of a planar graph. His method is an extension of Lee and Schachter's algorithm ([13]), designed to work on point sets. Chew's algorithm makes strong assumptions on the distribution of data, and creates virtual vertices to enable the "Divide-and-Conquer" paradigm. It is shown in this paper that such assumptions may be abandonned, and that the resulting optimal algorithm is much closer to a practical implementation.

1 Introduction

Large-scale applications make extensive use of Delaunay triangulations for representing complex data. One example of such applications is a solver for fluid mechanics, for which the results are optimal when computations are performed on a Delaunay mesh. Another example is a flight simulator where optimum performance is achieved when terrains are represented by means of Delaunay triangulations. In both cases, there is a strong need for representing the complexity of the data by means of a planar graph $\mathcal{G}$ in which the vertices are original data points, and segments are *constraints*: Typically, segment chains will represent mountain ridges, river banks, or contour lines.

The resulting Delaunay triangulation is then said to be *constrained* by the graph in the sense that it contains two types of edges:

1. **prescribed** $\mathcal{G}$-edges, which are exactly the edges of the constraint graph, and
2. **Delaunay** (unprescribed) edges, which are the edges needed to complete the triangulation, and may at most have an endpoint in common with any $\mathcal{G}$-edge.

Several optimal solutions have been published for constructing the constrained Delaunay triangulation of a planar graph, and are referenced below. Our paper suggests to withdraw the very limiting conditions imposed on one such published solution ([3]), thus transforming it into a new, independent and practical algorithm, even more akin than it already was to the original divide-and-conquer method for constructing Delaunay triangulations of point sets it was derived from ([13]). We also claim that this enhanced version may be generalized to situations when only a localized triangulation is required among a global hierarchy of polygons. However, this aspect of the problem will later be presented in a companion paper.

Section 2 gives definitions, notations and a brief historical introduction. Section 3 describes the original algorithm our work stems from. Section 4 presents the revisited version of the algorithm. Section 5 concludes on further research.

2 Historical background

Delaunay triangulation is a well-known subject with a vast literature. Refer to [1] for a complete list of references on this structure and its dual – the Voronoi diagram – and to [14] for detailed presentations. The chronology below does not claim to be exhaustive; it is merely intended to place the present work in the context of all previous research.

Let $V_n = \{v_1, v_2, \ldots, v_n\}$ be a set of n distinct points – hereafter referred to as *sites* – in the Euclidian plane E. The *Voronoi diagram* of V_n (noted $VD(V_n)$) is the planar subdivision of E in which each site $v_i \in V_n$ is assigned one unique convex region containing all points closer to v_i than to any other site. If no three sites are aligned, and no more than three sites are on the same circle, the sites are said to be in *general position*, and the straight-line dual of the Voronoi diagram is a unique maximal planar graph $DT(V_n)$, called the *Delaunay triangulation* of the set.

Let v_i, v_j, v_k be three sites determining one Delaunay triangle $\triangle_{ijk}$ in $DT(V_n)$. The circle (Γ_{ijk}), circumscribed to $\triangle_{ijk}$, is such that it contains no site in its (relative) interior. This property will be referred to as the *circle criterion* in the sequel.

The Voronoi diagram and Delaunay triangulation of V_n may both be constructed in $\Theta(n \log n)$ time and $\Theta(n)$ space ([14]). The first optimal algorithm for constructing the Voronoi diagram of a set of points was suggested by Shamos and Hoey ([16]), who observed that the divide-

*Was working at EMSE when this work was done.

and-conquer paradigm ideally applied to proximity problems. [10] then generalized this technique to the merging of any two Voronoi diagrams on points, and even line segments. The first optimal construction of the Delaunay triangulation of a planar point set was given in [13]. ([9] later gave a similar, 'divide-and-conquer' solution, with treatment of co-circularity. See [6] for a discussion.)

S. Fortune published an optimal *sweep-line* algorithm for constructing the Voronoi diagram of a set of points ([7], [8]), which may be applied to line segments, but is then more involved. Finally, researchers from INRIA ([2], [5]) have suggested powerful randomized techniques for dynamic insertions and deletions in the Delaunay triangulation of a set of points, in $\log(n)$ expected-time per operation.

The pioneering work on the extension of divide and conquer techniques to Voronoi diagrams of segments (in various metrics) is [11]. Lee's work triggered all subsequent research on the so-called "constrained" case, which we shall now define more precisely.

Let E be a set of segments in the plane, with endpoints in V_n, and $\mathcal{G} = (V_n, E)$ be the embedding in $\mathbb{E}$ of a planar straight-line graph, with vertices in V_n and edges in E. Let v_i be a site in V_n. We shall say that site $v_j \neq v_i$ is *visible* from v_j (with respect to E) if and only if the line segment $[v_i, v_j]$ intersects no segment in E, except possibly in a common endpoint.

The *constrained* Delaunay triangulation of $\mathcal{G}$ is a maximal planar straight-line graph $CDT(\mathcal{G}) = (V_n, E')$ where $E \subseteq E'$, and such that the circle circumscribed to any triangular face $\Delta_{v_i v_j v_k}$ contains in its interior no site that would be visible from v_i, v_j *and* v_k at the same time.

Clearly, constrained Delaunay triangles are Delaunay triangles when $E \equiv \emptyset$. Also note that the dual of the constrained Delaunay triangulation of a planar graph is not well-defined (see [4], [15] and [1] for counterexamples).

A planar graph with n vertices has at most $3n-6$ edges ([14]), so its size is $O(n)$. It may be shown that the lower bound for constructing both structures on a graph of size n is $\Omega(n \log n)$ ([1]). The first breakthrough in the study of constrained Delaunay triangulation of a planar graph was made by Lee and Lin, in [12]. However, their algorithm is only optimal when applied to a simple plane polygon (with empty interior).

Two independent and different $O(n \log n)$ constructions of the *constrained* Delaunay triangulation were published at the *Third ACM Symposium on Computational Geometry (1987)*. Wang and Schubert ([17]) give a non-trivial method to construct the so-called "bounded" Voronoi diagram of a set of line segments, and they also mention how their solution may be used to build the pseudo-dual, but give no solution for a direct construction. [3] gives an algorithm for directly constructing the constrained Delaunay triangulation of a graph, but no hint on the direct construction of the pseudo-dual. Seidel ([15]) will settle the whole matter by showing how constrained and bounded Voronoi diagrams (and their duals) are related, and how these structures may be constructed using a variant of Fortune's sweep-line algorithm.

Chew's algorithm is one of the simplest solutions, but the strong assumptions it imposes greatly impair implementation. However, the adaptation of the divide-and-conquer paradigm to such a complex problem has great theoretical and practical implications.

3 Paul Chew's algorithm

If necessary, the graph $\mathcal{G}$ to be triangulated will be augmented with two virtual infinite horizontal lines, bounding it above and below; suppose further that it is possible to subdivide the plane into n vertical strips, in such a way that every vertex in the graph belongs to exactly one strip; and finally that a pre-processing phase has allowed to determine, for each vertex, which (possibly virtual) edge of $\mathcal{G}$ is immediately below, and which is immediately above it. Thus, elementary strips contain exactly one such *active* "piece" (between two $\mathcal{G}$-edges). While each $\mathcal{G}$-edge is active in at least one strip, many edges may cross a given strip without begin active in it. This is a good way to only consider the two edges closest to one given vertex. Let us also ask that the pre-processing phase effectively clips each active portion of $\mathcal{G}$-edge to the vertical boundaries of the strip it is active in. This creates $O(n)$ virtual vertices.

Supposing V_n non empty and sorted by increasing x-coordinates, Chew's algorithm may be summarized as follows:

```
Chew($V_{l..r}$: VertexArray): Strip; {
    if ($r - l \leq 2$) return TrivialCDT($V_{l..r}$);
    return Merge(Chew($V_{l..\lfloor\frac{l+r}{2}\rfloor}$), Chew($V_{\lceil\frac{l+r}{2}\rceil..r}$));
}
```

Comments: When subsets with at most 3 vertices are to be treated, the specialized TrivialCDT function is called, the details of which will be omitted.

Sets with at least four vertices are divided into two equally sized subsets, such that all elements in the first subset have strictly smaller abscissæ than those in the other. This is ensured by the fact that the initial vertices may be placed in n different vertical strips – which proscribes vertically aligned vertices – and have initially been sorted by x-coordinates.

The algorithm recursively builds the constrained Delaunay triangulations of the two subsets, and *merges* the two sub-triangulations into one.

Of course, the only difficult part is the implementation of the merge process. In the original algorithm, this task is divided into two subtasks: Stitching two active pieces, and then merging their contents. Initially, each active "piece" consist of *one* graph vertex and its two immediate neighbouring edges. Merging two initial active pieces yields a composite active piece, whose contents is a sub-triangulation, and so forth...

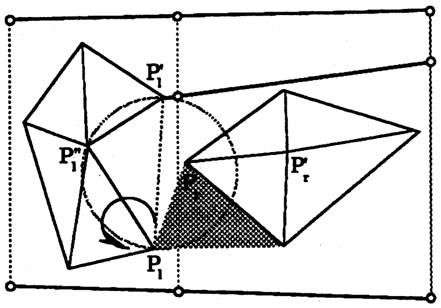

Figure 1: Basic merging techniques.

3.1 Stitching two pieces

Stitching pieces consists in scanning them in the direction of increasing y's, and doing the following operations:

1. Remove all virtual vertices on the common boundary of the two pieces, and all attached Delaunay edges.
2. If an active $\mathcal{G}$-edge in the left piece is not present in the right one, extend it up to the new virtual vertex at the intersection with the right boundary. Proceed to the next element in both pieces. (A similar action is to be performed for the symmetrical situation.)

   Else if a $\mathcal{G}$-edge in the left piece exactly matches a $\mathcal{G}$-edge in the right piece, unify them in a single one.

In view of this operation, the pre-processing phase should also arrange all the pieces in one strip as an ordered linked list, provided no two vertices lie on the same vertical line.

3.2 Merging two sub-triangulations

As soon as two pieces have been stitched together, their contents are merged into one sub-triangulation, using a variant of the technique in [13], which we both shall now review. Refer to Figure 1 for help.

Lee and Schachter's merge

Find the two supporting lines for the convex hulls of the two sub-triangulations, as in the merging phase of two linearly separated convex hulls (cf. [14] for a detailed account). Start from the lower one, which connects, say vertices P_l and P_r on the hulls. Following P_l on the left hull in counter-clockwise order is vertex P_l', and following $[P_l, P_l']$ in counter-clockwise order around P_l is $[P_l, P_l'']$. P_r' and P_r'' may be defined symmetrically in the right sub-triangulation. While the circle through P_l, P_l', P_l'' contains P_r in its interior, remove edge $[P_l, P_l']$ from the left sub-triangulation, replace P_l' with P_l'' and P_l'' with its own counter-clockwise successor around P_l. Continue until the circle criterion is no longer violated, or P_l'' falls into the lower half-plane defined by line P_lP_r. Now, do the same for vertices around P_r. When this is finished, use the circle criterion in the quadrilateral (P_l, P_r, P_r', P_l') to either replace P_l with P_l' or P_r with P_r', accordingly. The process may now be started over again, until $[P_l, P_r]$ coincides with the upper supporting line of the two hulls.

Chew's merge

In the case of a graph, things are slightly different. Every active piece is surrounded by two $\mathcal{G}$-edges, themselves bounded by virtual vertices which have been precisely located, but have no relationship with the final triangulation. These vertices are now going to behave like infinite vertices in the triangulation process. As such, they will have little influence on the circle criterion: the limiting circle through three points, one or two of which are vitual vertices, is an infinite line, and asking whether one vertex is inside this "circle" boils down to finding out whether it is above or below the line.

There will be four types of virtual vertices: $(-\infty, -\infty), (-\infty, +\infty), (+\infty, -\infty)$, and $(+\infty, +\infty)$. If a virtual vertex is situated on the left (right) boundary of a strip, its abscissa will be $-\infty(+\infty)$. If a virtual vertex lies on a bottom (top) bounding $\mathcal{G}$-edge for the current piece, its ordinate will be $-\infty(+\infty)$.

Finally (refer to Figure 1), if a $\mathcal{G}$-edge originating from one piece is directed towards the other piece but does not terminate (at a vertex) in it, it will hide some section of the latter to vertices in its own, and this will be another cause for interrupting the merging's elimination phase.

3.3 Analysis

The object returned by the algorithm is a strip, as defined earlier, containing one unique triangulation (regardless of the distribution of vertices). This strip is given as a list of edges from which it is possible to extract the sought triangular faces in linear time.

The asymptotic performance of Chew's algorithm for a graph of size n is easy to determine:

1. Pre-processing phase: A sort and a simple sweep-line algorithm may do the job, which yields an $O(n \log n)$ running time.
2. The dividing phase is a constant time process.
3. The stitch-and-merge phase takes time proportional to the number of elements in both strips, which is $O(n)$.
4. The overall equation is $T(n) = 2T(n/2) + O(n)$, which yields an $O(n \log n)$ running time.

4 A less restrictive version

We have already stated the qualities of Chew's algorithm. Let us now mention its major drawbacks:

1. It forbids vertically aligned vertices. Chew suggests to rotate the data to avoid this problem, but such an operation is very hazardous when precision is a crucial issue, to say the least.

2. It keeps creating and then destroying virtual vertices, which have nothing to do with the final triangulation. The presence of such vertices is not only detrimental from the point of view of efficiency (memory management, and computational cost for updations), but also from that of precision again.
3. It may not be used directly to triangulate a simple polygon (one must destroy the faces of the polygon's complement in the convex hull). We shall not go into this problem in this paper, although the new version we are about to detail may be made to allow this operation.

The main objective of the improved algorithm is to accept all kinds of inputs. This will, in turn, allow to lift off the obligation to divide the plane into n strips, and rid the algorithm of virtual vertices.

Let us redefine an *elementary strip* as an entity containing either one isolated vertex, or else all vertices with the same abscissa, linked in increasing y-order. If two successive vertices on the same vertical line are mutually visible, there is no harm in connecting them by a Delaunay edge (which may possibly be later questioned, and then deleted in the merge process, as seen above).

This simple observation hardly modifies the original algorithm, but its consequences will be considerable:

```
Chew+($V_{l..r}$: VertexArray): Strip; {
    if ($r = l$) return TrivialCDT+($V_{l..r}$);
    return Merge+(Chew+($V_{l..\lfloor\frac{l+r}{2}\rfloor}$), Chew+($V_{\lceil\frac{l+r}{2}\rceil..r}$));
}
```

When the subset of vertices has at least two elements, Chew+ is recursively called on equal sized subsets, and the resulting strips are then merged. Of course the new merging procedure is somewhat more involved than it was, as we shall shortly see.

On the other hand, if the subset V_n only has one element, a specialized function is called, whose operation will now be detailed.

4.1 TrivialCDT revisited

Recall that this specialized function is called for every vertex in the graph, and that the vertices of the graph have been once and for all sorted in lexicographical order. Let ν be the current vertex. Basically, all we need is a process – to be resumed at each call – indicating which $\mathcal{G}$-edges come nearest to ν above and below.

The simplest idea is to use an AVL tree that will contain all "active" vertices and $\mathcal{G}$-edges at ν's abscissa. Using y-order at the logarithmic-time primitives on AVL's to perform the following complex operations, each time the function is called:

1. Insert ν in the AVL.
2. Delete from the AVL all $\mathcal{G}$-edges whose right endpoint coincide with ν, or insert all the $\mathcal{G}$-edges whose left endpoint coincide with ν.
3. Detect the $\mathcal{G}$-edges immediately above and below ν.

Now, recall that a divide-and-conquer process may be modelled by a recursion tree traversed in symmetric order. Since the leaves of this recursion tree represent the vertices of the graph, the recursion process will reach vertices in lexicographical order. Thus, if two vertices have the same abscissa, when the one with larger ordinate is reached, it will be possible to check whether the former is visible from the latter (*i.e.* no $\mathcal{G}$-edge lies between them), and to connect them accordingly, as announced.

4.2 Merging revisited

The merging process is somewhat different now, although still linear in the number of vertices in the subsets.

4.2.1 Subgraphs and pieces

Actually, *all* virtual elements (horizontal bounding lines, vertical boundaries, vertical strips, virtual vertices) are no longer necessary. To understand why, let us investigate the nature of the pieces the algorithm will generate. A piece will generally consist of a sub-graph (not necessarily a sub-triangulation) contained between one "floor" and a "ceiling" (both possibly nil or a $\mathcal{G}$-edge). To be more precise, such sub-graphs will consist of a bounded and triangulated convex hull, from which "infinite" rays will emerge. Of course, these rays are not infinite: there are merely $\mathcal{G}$-edges with only one known endpoint, the second being at some other location, not yet *connected* to the current piece. (Note that the same thing happened previously, but was masked by the constant updation of the virtual vertices as strips were stitched.) Such pieces will be said to be *real pieces*, as opposed to *virtual pieces* which will only consist of one $\mathcal{G}$-edge.

We shall say that a $\mathcal{G}$-edge completely immerged in a subgraph is totally captive, a $\mathcal{G}$-edge emerging from a real piece is half-captive, and that the $\mathcal{G}$-edge in a virtual piece is totally free.

4.2.2 Stitching virtual and real pieces

The stitching operation should now be easier to understand. The series of pieces returned by the algorithm is a *strip*, *i.e.* a linked list, alternating with virtual and real pieces. Stitching two such strips is made by scanning them from the bottommost element and trying to match their corresponding $\mathcal{G}$-edges. Several cases may occur:

Virtual *vs* virtual: The scanning process encounters two virtual pieces. Proceed as follows.

1. If the pieces are the same $\mathcal{G}$-edge, unify them into a single one and append it as a virtual piece in the new strip being constructed. Then move on to the next piece in both strips.
2. If the virtual piece to the left is below the other (remember that the graph is planar, and hence if the $\mathcal{G}$-edges are different, they may not intersect "in the

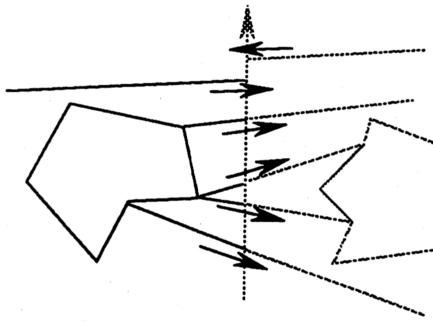

Figure 2: Stitching two pieces.

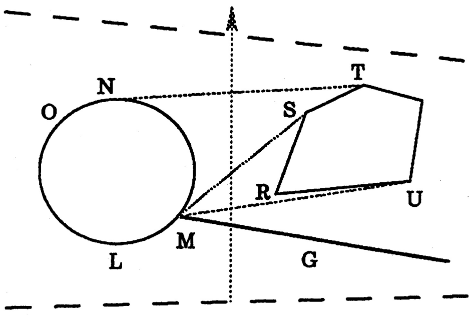

Figure 3: Stitching and merging two real pieces.

middle of" two strips), append the left virtual piece to the new strip, and move on to the next piece in the left strip.

3. Else, append the right virtual piece to the new strip and proceed to the next piece in the right strip.

Virtual *vs* real: Suppose the virtual piece is the left-hand side one. Scan the boundary vertices of the real piece clockwise to try and find a half-captive edge that matches the virtual piece. If such an edge is found, unify it with the virtual piece, which now becomes part of the real piece. Move on to the next piece in the left strip.

Else, move up on the right piece until the scan reaches a portion of the real piece beyond visibility for the known extremity of the virtual piece. The virtual piece is necessarily above the real piece, so add the latter at the end of the new strip, and move on to the next piece of the right strip.

Real *vs* real: Both pieces are to be scanned in parallel, and in (counter-) clockwise order for the (left) right piece. When reaching a vertex, its oriented incident edge list (as built by `TrivialCDT+` using the *AVL*) gives the successive outward pointed G-edges originating from it. If a match between two half-captive edges is found, simply unify them and resume both scans on the other side of this common bridge. As before, the scan may be stopped as soon as visibility is no more possible from either end.

The scanning of any piece is only completed when visibility is lost, or the upper supporting is reached. Also note that more than one $\mathcal{G}$-edge may emerge from a single vertex in either piece, which means that the scanning process may be synchronized using the slopes of such edges. Some examples of stitching are given in Figure 2. The operations involved in the stitching procedure are either pointer comparisons, differences of vertical intercepts between *initial* edges of the graph, or straightforward visibility tests.

4.2.3 Merging

Since all virtual vertices have been removed, Lee and Schachter's merging technique must be adapted. Great care must be taken as to which element should be elected for starting the scan phase around a piece. Some conventions may be taken to always priviledge the lowest outward pointed $\mathcal{G}$-edge, or if there is none, the lowest vertex on the hull, as usual. Here is a sketch of the merge phase procedure:

Search for lower supporting line: The information gathered during the stitching operation helps determine all cases. The only troublesome case is "Real *vs* real".

Obviously, a supporting line is only valid if and only if its endpoints are mutually visible. This is ensured both by the synchronization of the stitching scan and by the usual scan of the convex hulls merging algorithm. In other words, the stitching operation ensures that only mutually visible sections of hulls are undergoing triangulation.

Partial triangulation: Fill the space between visible sections of both *real* pieces, using Lee and Schachter's algorithm, until either reaching the upper supporting line or a $\mathcal{G}$-edge attached to both pieces. In the latter case, proceed to the other side of this bridge, and repeat this item. If the partial triangulation is over, resume stitching.

Let us take the following example to illustrate how the stitching and triangulation processes must co-operate. Refer to Figure 3.

The left (right) piece begins in L (R). Scanning the left piece first (for instance), the half-captive $\mathcal{G}$-edge G is encountered, so the scanning process now begins on the right piece, up to T, where M is no longer visible. Thus, no matching $\mathcal{G}$-edge was found on the right piece. There is no hope of completing the triangulation below G, so the action now takes place *above* it. M and R are mutually visible and no $\mathcal{G}$-edge was found up to S. Thus, the supporting edge between both sections is found to be MU, and Lee & Schachter's algorithm is used to triangulate the space between both pieces up to NT since, at this stage, the concurrent stitching and merging processes infer from reaching the upper supporting line that the merging of the two pieces is over.

4.2.4 Analysis

`TrivialCDT+` is called n times, and performs as many *AVL* insertions/deletions/look-up as the number of elements in the graph. Hence, its contribution to the running time is $O(n \log n)$.

It is important to note that the technique we have outlined globally creates non convex sub-graphs, with locally convex sections of boundary between two spikes (half-attached $\mathcal{G}$-edges). This is how it is possible to combine Lee and Schachter's original merging technique on the one hand, and Chew's adapted technique on the other. The trick is to carefully synchronize two scanning processes: the stitching operation between matching spikes, and the triangulation of mutually visible convex sections of triangulated sub-graphs.

The operations involved in the stitching-merging phase are conducted by means of a vertical scan from top to bottom. Of course, it may happen that an entire piece has to be scanned entirely for stitching purposes, and then again for the sake of its partial triangulation. But this is exactly what happens, in the worst case, when one tries to locate the supporting lines for two linearly separable convex hulls!

Some extra but straightforward visibility tests are required to ensure consistent supporting lines, but on the whole, the entire process requires a number of constant-time operations proportional to the size the strips to be merged. Therefore, the new version of Chew's algorithm has the same $O(n \log n)$ asymptotic running time.

5 Conclusion

We have shown how Chew's algorithm may be improved to accept all data configurations. This, in fact, allows to rid the original algorithm of all the virtual elements that impaired its implementation. The new algorithm may even be observed to be more closely related to [13] than [3] was.

However, the authors wish to acknowledge that both results are equally important to their eyes, and that their work was aimed at finding a way to implement Chew's algorithm in an efficient and robust way, without having to alter the true topology of data.

The algorithm presented in this paper has been implemented and run, and is currently being integrated into the two types of large-scale applications mentioned in the Introduction.

The contribution of the *AVL* structure is by no means limited to what has been said here: it is in fact possible to use it for generalizing the algorithm to the situation where one wishes to triangulate the interior of a complex polygon, *i.e.* a non self-intersecting polygon containing a whole sub-graph. This will be the subject of another publication.

Finally, our future goal is to adapt Chew's revisited algorithm to the Delaunay triangulation of a graph constrained to a surface.

References

[1] F. Aurenhammer. Voronoi diagrams – A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), September 1991.

[2] J-D. Boissonnat and M. Teillaud. A hierarchical representation of objects: The Delaunay tree. In *Proceedings of the 2nd ACM Symposium on Compututational Geometry*, 1986.

[3] P. Chew. Constrained Delaunay triangulations. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 215–222, 1987.

[4] P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[5] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic time per operation. Technical Report 1349, INRIA, 1991.

[6] M. Elbaz and J-Cl. Spehner. Construction of Voronoi diagrams in the plane by using maps. *Theoretical Computer Science*, 77:331–343, 1990.

[7] S. Fortune. A sweep-line algorithm for Voronoi diagrams. In *Proceedings of the 2nd ACM Symposium on Computational Geometry*, 1986.

[8] S. Fortune. A sweep-line algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[9] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphic*, 4:74–123, 1985.

[10] D.G. Kirkpatrick. Efficient computation of continuous skeletons. In *Proceedings of the 20th Annual IEEE Symposium on FOCS*, pages 18–27, 1988.

[11] D.T. Lee. *Proximity and reachability in the plane.* PhD thesis, Coordinated Science Lab., Univ. Illinois, Urbana, Ill., 1978.

[12] D.T. Lee and A.K. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete Comput. Geom.*, 1:201–217, 1986.

[13] D.T. Lee and B.J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computing Information Sciences*, 9:219–242, 1980.

[14] F.P. Preparata and M.I. Shamos. *Computational Geometry – An Introduction.* Springer-Verlag, New York, N.Y., 1985.

[15] R. Seidel. Constrained Delaunay triangulation and Voronoi diagram with obstacles. Technical Report 260, IIG-TU Graz, Austria, 1988.

[16] M.I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on FOCS*, pages 151–162, 1975.

[17] C.A. Wang and L. Schubert. An optimal algorithm for constructing the Delaunay triangulation of a set of line segments. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 223–232, 1987.