© Proceedings of the *6th Canadian Conference on Computational Geometry*
Saskatoon (Saskatchewan, Canada), August 2-6, 1994, pp. 165-170.

Hierarchical Delaunay triangulation

J-M. Moreau
LISSE*

Extended abstract

Abstract *This paper presents the generalization of a formerly published divide-and-conquer algorithm for constructing the Delaunay Triangulation of a planar graph. This generalization allows to triangulate portions of a graph, considered as logical and separate entities, as for instance the polygonal faces in a hierarchically organized graph. This is an improvement over all previous constrained Delaunay triangulation algorithms, in the sense that the triangulation of vast databases may be split into hierarchical subtriangulations in an autonomous way. The algorithm is optimal and may be applied to various other structures.*

1 Introduction

Several optimal algorithms have been published for constructing the *Constrained Delaunay triangulation of a planar graph* G (refer to the survey by F. Aurenhammer [1]).

In many applications (*e.g.* flight simulators, *GIS* systems, etc.), such planar graphs may be thought of as representing terrains, in which faces (fields, roads, rivers, cities, ...) are frequently arranged in nested sequences. In such contexts, it is often important to construct the constrained Delaunay triangulation of faces independently (*e.g.* need for local refinements, distributed tasks, etc.).

This paper presents an optimal divide-and-conquer algorithm for performing such local tasks, which will hereafter be called "hierarchical (Delaunay) triangulations". It is organized in the following way: Section 2 gives a formal definition of the hierarchical Delaunay triangulation, and presents previous research of interest. Sections 3 presents the structures and the various mechanisms required by the algorithm. Section 4 presents the divide-and-conquer algorithm for constructing the Delaunay triangulation of a hierarchical graph, and Section 5 concludes on future work.

2 Hierarchical triangulations, and previous work

Let $G = (E, S)$ be a planar graph with S a set of N vertices (also called sites). Since the number of edges in E is at most $3N - 6$, it makes sense to say that the graph is of size $O(N)$. Two sites in G are *mutually visible iff* the line segment between them intersects no edge in the graph, except possibly at endpoints.

The Delaunay triangulation of G is a maximal planar graph, in which each (triangular) face Δ_{pqr} has the "constrained Delaunay circle property": The circle through p, q, and r contains no other site of S visible from all three. Such a structure is unique *iff* no more than three sites in S lie on the same circle of empty interior. If the set of edges in the graph is empty, this property coincides with the standard "Delaunay (or empty) circle criterion": The corresponding circumscribed circle has no other site of S in its interior. The edges of the standard Delaunay triangulation of a set of sites are called Delaunay edges, as opposed to the "graph edges" that are to be found in the constrained version of the problem.

The construction of the constrained Delaunay triangulation of a graph takes time proportional to $N \log N$. In fact, this problem has the same asymptotic complexity as the construction of the (standard) Delaunay triangulation of a set of points in the plane, which was first solved in optimal $\Theta(N \log N)$ time by Lee and Schachter ([10]). See also the article by Guibas and Stolfi ([6]).

Various papers have been written on the important subject of constrained Delaunay triangulations. The first optimal $\Theta(N \log N)$ algorithm was discovered by Paul Chew ([2]) in 1987. Steven Fortune then published an optimal sweep-line algorithm for constructing the standard Voronoi diagram of a set of points in the plane ([5]) which may be applied to the construction of the bounded Voronoi diagram. Raimund Seidel then unified the notions of bounded and constrained Voronoi diagrams in a seminal paper ([13]). In 1993, the author and Pascal Volino suggested in [11] a new version of Chew's algorithm that ridded the latter of all the rather restricting and cumbersome hypotheses needed for its justification.

2.1 Hierarchical graphs

Both divide-and-conquer algorithms in [2] and [11] consider the triangulation of the graph relatively to its natural implicit external face: The boundary of its convex hull. If such a graph is presented as data to the new algorithm, it will perform the same task as described above.

However, the natural input for the algorithm in this paper is a *hierarchical graph*, that is: A planar graph represented by a finite union of disjoint connected components of the form (P_i, V_i, E_i, H_i), where each P_i is the polygonal

*Address: ENSM.SE, 158 cours Fauriel, F 42 023 St-Etienne, France. Email: moreau@emse.fr.

boundary of the i-th connected component, V_i is a set of vertices, E_i a set of edges, and H_i is a (possibly empty) hierarchical graph, all of which constituting a planar subgraph entirely contained in the interior of region P_i. The V_i's and E_i's (H_i's) will also be referred to as the "elements" ("holes") of the P_i's.

The elements are either isolated vertices (*e.g.* altimetric information), isolated edges or chains of edges (*e.g.* planimetric information). Such edges will also be called "constraints" in the sequel. Holes are recursively defined as hierarchical subgraphs. They may, for instance, be thought of as representing imprints of houses, buildings or complex objects, the faces of which are to be obtained by means of a specialized external 3-D modeller. Note that holes may be empty regions or not, depending on the application. The algorithm successfully handles both cases, naturally.

For the sake of conciseness, we shall only consider here the case of a hierarchical graph $G = (P, V, E, H)$, with one single connected component, P, and empty holes. All that will be said applies directly to the more general definition.

A first step towards the Delaunay triangulation of a hierarchical graph was made by Lee and Lin ([9]) who presented, in 1986, a (non-optimal) divide-and-conquer method for constructing what they called a "Generalized Delaunay Triangulation" (*i.e.* the constrained Delaunay triangulation of a graph). Lee and Lin also detailed a specialized version of their algorithm that applies to the Delaunay triangulation of a simple polygon, and that works in $O(N \log N)$ time. This algorithm – which is rather involved and requires computing visibility polygons in the merge phase – does not generalize to the hierarchical case described above. Namely, the presence of holes, graph edges, chains or vertices cannot be taken into account in their algorithm without increasing the overall $O(N \log N)$ asymptotic running time.

Finally, let us note that all the general algorithms for the constrained Delaunay triangulation of a graph mentioned in the preamble of this section could be applied to the problem at hand: It would be sufficient to triangulate the whole graph (P, its elements and holes) relatively to its convex hull and then to delete from the list of faces those that would not be relevant. However, this technique is not very well suited here, since a lot of work would be necessary to create irrelevant simplices and then to destroy them! It would be even more detrimental to resort to this solution if the number of holes were important (relative to the size of the external polygon), a rather frequent situation in the applications mentioned in the introduction.

3 Structures and mechanisms

3.1 Pseudo polar angles

Consider the directed edge $\overrightarrow{AB}$ of Figure 1, and define $d_x = x_B - x_A, d_y = y_B - y_A$. It is possible to assign to any such *half-edge* a unique real number in $]0, 4]$ – called its *pseudo polar angle* (*ppa*) – that behaves like the true polar angle (*cf.* [12], page 353). For instance, for all endpoints B in the first (North-East) open quadrant of Figure 1, the

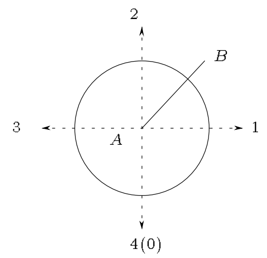

Figure 1: Pseudo polar angles for half-edges.

pseudo polar angle of $\overrightarrow{AB}$ is a real number in $]1, 2[$, given by $1 + \frac{dy}{dx+dy}$. Similar expressions may be found for edges in the other three quadrants, and the special cases of vertical and horizontal directions are dealt with by straightforward inspection of dx and dy. Note that the origin for pseudo polar angles (0) is actually represented by 4, for practical reasons.

3.2 Planar Maps

A number of solutions have been suggested for representing planar graphs, among which the *planar map* ([3], [8]), which allows all the operations required by the algorithm at minimal cost, although other well-known structures would be eligible. Suppose that the hierarchical graph G has N_P vertices on its external boundary, contains N_{VE} vertices (including isolated vertices and the endpoints of constraint edges in its subgraph, excluding holes), and h (empty) holes, the boundary of which are composed of N_H vertices. Then, the total number of edges in any triangulation of G – and hence, in its constrained Delaunay triangulation – is exactly: $N_E = 2N_P + 3N_{VE} + 2N_H + 3h - 3$.

In the context of planar maps, the basic structure for half-edge $\overrightarrow{AB}$ (going from A to B) is a pointer to vertex B, a pointer to the next CW half-edge around A, and one to the next CCW half-edge around A. If we allocate an array of $M = 2N_E$ cells for such edges, and if edge $\overrightarrow{AB}$ is located in this array at index i, its "dual" edge $\overrightarrow{BA}$ is implicitly located at index $\Phi_M(i) = M - i$. This involution allows to locate dual edges implicitly (without storing an explicit pointer for them in the basic data structure). The total space required for storing the constrained triangulation of the hierarchical graph G with N_E edges, is therefore $2N_E \times 3 \times 4 = 24N_E$ bytes on 32-bit machines.

3.3 Synchronized plane sweep

A balanced-tree structure (AVL), already present in ([11]), will be used in the new algorithm, in co-ordination with the recursive divide-and-conquer process. Suppose all the vertices in the planar graph have been sorted in increasing xy-order. The basic idea behind using an AVL is that the leaves of the divide-and-conquer recursion tree are the vertices themselves, and are visited in their increasing lexicographical order. Hence, the updation of the planar map, which will eventually capture the structure of the final triangulation, may be done using a balanced tree structure for sweeping the plane, to be updated at the bottom of every recursive call in the following fashion.

3.4 A scheduling mechanism for edges

As a preliminary step, all the graph edges are inserted, one at a time, in a priority queue, PQ^+, built for instance on the *leftist tree* model ([7], [4]), and allowing constant-time extremum search, and optimal logarithmic time insertion or extremum extraction.

Think of this priority queue as dedicated to inserting half-edges "originating from their lexicographical leftmost endpoint". Clearly, all such half-edges have a pseudo polar angle in $]0, 2]$, exclusively. (A directed edge $\overrightarrow{AB}$ with *ppa* 4 *is* half-edge $\overrightarrow{BA}$ with *ppa* 2.) Since the graph is assumed to be planar, it is impossible for two distinct half-edges with identical origin to have the same pseudo polar angle.

Consider two edges $e_1 = [A_1, B_1], e_2 = [A_2, B_2]$, with $A_1 <_L B_1$, and $A_2 <_L B_2$ ($<_L$ = lexicographical order). The order relation ruling $PQ+$ is designed to let half-edges with smaller lexicographical leftmost endpoints go first, and in case of ties, those with smaller pseudo polar angles, as summarized in the following routine:

PQ^+`Order`(e_1, e_2)
  if $((x_{A_1} < x_{A_2}))$ return "e_1 comes before e_2";
  if $((x_{A_1} > x_{A_2}))$ return "e_2 comes before e_1";
  if $((y_{A_1} < y_{A_2}))$ return "e_1 comes before e_2";
  if $((y_{A_1} > y_{A_2}))$ return "e_2 comes before e_1";
  if $(ppa(e_1) < ppa\,(e_2))$ return "e_1 comes before e_2";
  if $(ppa(e_1) > ppa\,(e_2))$ return "e_2 comes before e_1";
  return "$e_1 == e_2$";

After insertion of all G-edges in the "origin" priority queue PQ^+ using this comparison procedure, the root of the associated leftist tree will at any time contain the less slanted half-edge originating from the lexicographically leftmost left-endpoint in the graph that has not been processed yet.

If we use another similar priority queue PQ^- for storing the half-edges considered as ending at their lexicographically rightmost endpoint, and if we use PQ^-`Order`, a "dual" version of the previous comparison routine applied to the rightmost endpoint of such edges, it is possible, each time an edge is being extracted from PQ^+, to insert its dual in PQ^-, the root of which will hold, at any time, the most slanted half-edge ending at the lexicographically leftmost, yet unprocessed, right-endpoint in the graph.

Thus, for each successive vertex v encountered at a leaf of recursion in the algorithm:

1. All half-edges ending in v (if any) are extracted (in decreasing polar angle) from PQ^-, removed from the AVL, inserted in the planar map representing the triangulation, and the space they previously occupied in the queue is returned to the system.
2. Then all the half-edges originating from v (if any) are extracted (in increasing polar angle) from PQ^+, inserted in the plamar map, and their duals are inserted in the AVL and in PQ^-.

3.5 Marking angular sectors

Imagine an observer moving from $-\infty$ towards $+\infty$ on a virtual vertical line separating two strips being merged in the algorithm. To find out whether this observer is currently standing inside one internal face of the hierarchical graph or not, it suffices to count the number of non-vertical boundary edges below him or her. (Vertical edges at the exact vertical of the observer and below may be omitted, as they convey no information *w.r.t.* inclusion.)

For readers not familiar with the algorithm in [11], let us say that it selects a privileged graph-edge for each vertex in the graph. Such an edge is called a "floor edge" for v, in the sense that it is the first edge in the graph hit by a vertical line drawn from the vertex towards $-\infty$. The algorithm then constructs alternating sequences of such edges and subtriangulations (pieces), called "strips".

The virtue of only considering floor edges for successive pieces in a strip is to remove from the current merge any crossing graph edge that stands no chance of being made captive (at either end) in either subtriangulation during the merge phase. The main disadvantage of this is to remove from the process potentially many such crossing graph edges, and therefore to stop the algorithm from knowing exactly how many boundary edges the merge process has already passed, coming from $y = -\infty$.

Fortunately, the AVL structure carries enough information to solve this problem. Let us attach to every node in the AVL a boolean field indicating the parity of the number of boundary edges contained in the nodes in its subtrees, and including the possible boundary edge it represents (explicitly excluding non-boundary edges). Vertical edges below one given vertex v are no problem, since they have all already been removed from the AVL as recursion reaches v.

Maintaining such a boolean field in the case of a standard binary tree is straightforward: Suppose we are adding one boundary edge to either subtree of node x. Then its new parity field is simply negated, and the process is recursively applied to the traversed subtree. Of course, if the newly inserted graph edge is not part of a boundary, no parity field updation should be made.

In the case of an AVL, things are a little bit more difficult. We shall limit ourselves to one case, since all other cases are symmetrical, and double rotations are compositions of two single ones. Thus, referring to Figure 2 where the figures indicate balance status, suppose x is the last "unbalanced" node (with balance -1, meaning the AVL has a heavier right subtree at x) on the insertion path of a new boundary edge; suppose further that this edge has just been inserted in the right subtree (with root z) of the right subtree (with root y) of node x. Re-balancing the AVL will terminate with a single left rotation around x. Denote by u^b the parity field of any non-empty AVL node u, by ^ the XOR operator, and by $(B(u)?)$ the test yielding true *iff* the edge contained in node u is a boundary edge in G.

After the rotation, the parity fields of the black subtrees on Figure 2 remain unchanged, those of z and its right

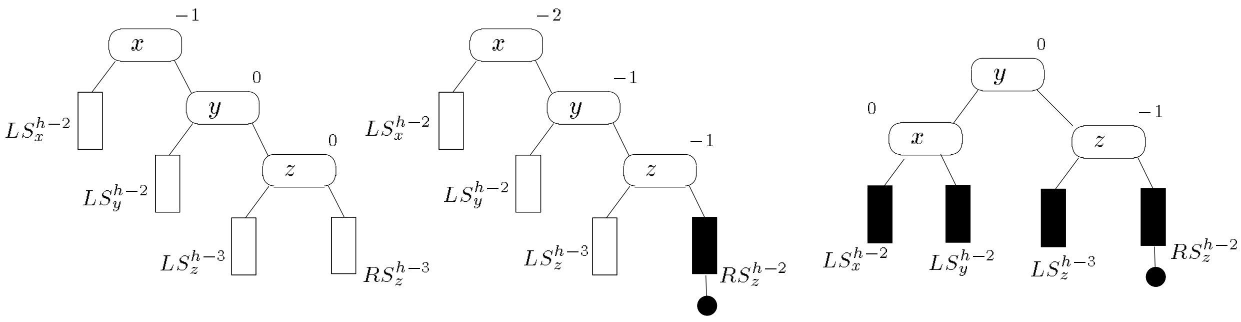

Figure 2: Parity field updation during a single left rotation in the AVL structure.

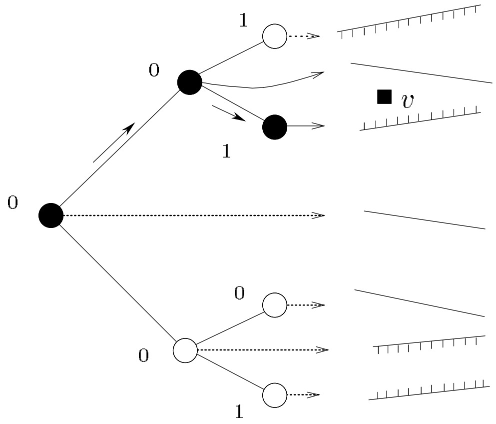

Figure 3: Inside/Outside testing using the AVL.

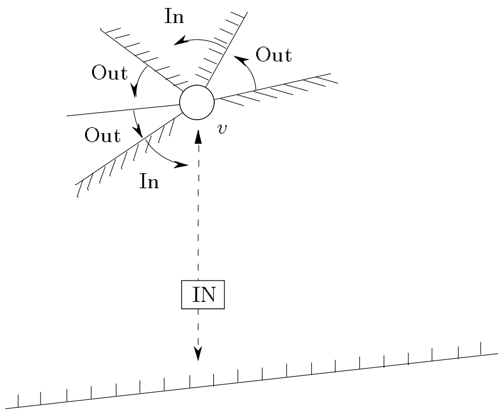

Figure 4: Marking the angular sectors around v.

subtree (both shaded) must be negated, and then those for nodes x and y may be computed in constant time, in the following order and fashion:

$$x^b = LS_x^b \,\hat{}\, LS_y^b \,\hat{}\, (B(x)?), \quad y^b = x^b \,\hat{}\, z^b \,\hat{}\, (B(y)?).$$

It is straightforward to infer from the parity fields of its subtrees and from its own, how to "interpret" the graph edge in any given node. For instance, the node at the root of the AVL in Figure 3 has an even parity field, and so have its two subtrees: It must hold a plain graph edge, and lie in the external face of the hierarchical graph. When the merge process reaches vertex v, it is possible to find out, in logarithmic time, whether it lies inside or outside the hierarchical graph, excluding the possible boundaries attached to it: Locate this vertex in the AVL, and infer the status of v from the parity field information collected on the search path (bold on Figure 3) from the root to the nearest floor edge below it.

Knowing where v lies in the graph, all the angular sectors from the planar map around v may now easily be assigned a unique boolean `InOut` label: Referring to Figure 4, start from the vertical direction (known to be "In" on the figure) in the so far empty adjacency list of v in the planar map. Using a boolean variable, `InOut`, initialized to the `InOut` value found for v, one edge ending at v must first be removed from the AVL. Hence, a directed edge must be created in the planar map, as first element in the adjacency list around v. This edge is labelled `In`, meaning an observer crossing this edge in the CCW direction will move into one face of the hierarchical graph. Each time a boundary edge is to be removed, `InOut` is negated, and the corresponding directed edge in the planar map is given this new `InOut` value.

Next, edges originating from v must be treated in a similar fashion, after winding `InOut` back to its initial value (`In` in our case), after which both (possibly empty) semi adjacency lists may be chained into one, in constant time.

4 A divide-and-conquer method for hierarchical Delaunay triangulations

Using all the structures and mechanisms in the previous section, and assuming the reader is familiar with the algorithms in [2] and [11], here is the outline of the new method:

1. Sort the vertices of the graph (including those of its elements, and holes boundaries) in lexicographical order, and arrange them in a "list" L;

2. Arrange all the input half-edges in a priority queue PQ^+. Initialize the AVL A, the second priority queue PQ^-, and allocate the planar map array PM, to organize the resulting triangulation.
3. Recursive step on L. Output is strip σ, a linked list of "pieces".
   (a) If L is reduced to a unique vertex v:
   Locate the nearest G-edge, say e_v, at the vertical of v (if any), and deduce `InOut` value for v.
   Extract from PQ^- (and A) all half-edges ending in v, insert them in PM, and set their `InOut` values on the fly.
   Extract from PQ^+ all half-edges originating from v, insert them in PM, and set their `InOut` values on the fly. Insert their duals in A, and in PQ^-.
   Return the linked list strip σ composed of piece e_v (if not empty) followed by piece v.

   (b) Else: Divide L into two equally-sized, linearly separable lists L_l, L_r; recursively "triangulate" L_l and L_r into σ_l and σ_r;
   Merge step: Let π_l and π_r be the current pieces in the left and right strips σ_l, σ_r, respectively.
   While (π_l and $\pi_r \neq \emptyset$):
      i. If (π_l is "In"):
         A. If (π_r is "In"):
         All resulting Delaunay edges and their duals are to be inserted in the planar map, with the appropriate `InOut` label.
         Match or/and triangulate π_l and π_r as in the merge phase of [11].
         Append all relevant resulting pieces to σ; Move on to next piece in σ_l and/or σ_r, accordingly.
         B. Else: Move on to next piece in σ_r.
      ii. Else: Move on to next piece in σ_l;

Comments

It is important to only merge-and-triangulate pieces that may see each other in the graph. See Figure 5 for an illustration of this, and also note that merging two pieces requires that they are "synchronous" (case 3.b.i.A), which may only be ensured by "passing" pieces that do not match. As we have seen, the planar map contains sufficient information to settle this matter. Thanks to the Jordan lemma, the `InOut` label for two oriented edges representing the pieces to be merged must match, and it takes $O(1)$ time to find out whether the merge should produce edges between both pieces or not. If either `InOut` label is found to have an "Out" value, the merge process should simply move on to the next piece in the same strip.

Otherwise, the merge process is identical to its homologue in the standard algorithm, except regarding the following important details:

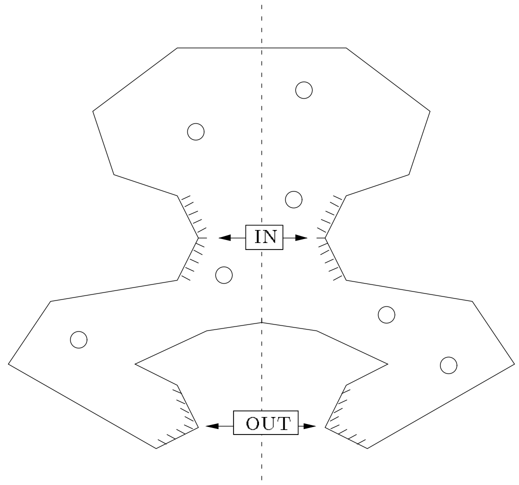

Figure 5: Merging two hierarchical pieces: *Fill the gap between both pieces in the topmost case, and move on to the next piece in each strip in the other.*

1. When a new Delaunay edge and its dual are created, their associated `InOut` label must also be given the appropriate values, so that, for instance, two internal, mutually visible, and graph-edge free pieces may be merged appropriately.
2. Elements (vertices or edges) that are found – during stage 3.a – to lie entirely outside the graph should be removed from the process, as it would be impossible for it to assemble such items into coherent pieces. This is akin to saying that mutually invisible edges such as those at the bottom of Figure 5 are each separately assembled into the planar map with the help of their respective internal colaterals.
3. When the interior of one isolated connected component of the graph has just been completely triangulated, its associated strip should simply be removed from the process, and appended to a specially dedicated list of triangulated connected components, for later retrieval.

The overall running time of the new algorithm is the same as that of the original algorithm, $\Theta(N \log N)$, if N is the size of the graph. This is partly due to the fact that "mutually visible" sections to be merged are processed in a similar fashion, and that sections that fail to be visible are detected and then dealt with in constant time. Note that otherwise "synchronizing" two such sections could not be guaranteed to be a linear-time process! The scheduling mechanism for edges has the same complexity if one uses dynamic priority queues, and the manipulations related to the AVL have been shown to remain logarithmic. The subsequent insertion of each graph edge in the planar map is a constant time process.

Interestingly enough, the three data structures used in the algorithm may peacefully coexist in the planar map array described in Section 3.2: First, noting that there are

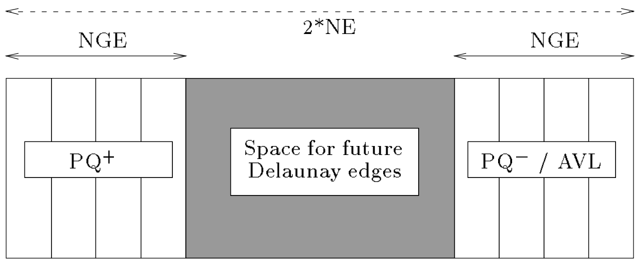

Figure 6: Fitting 3 data structures into a fourth one.

$N_{GE} < N_E$ graph edges in the final triangulation, PQ^- may simply be implemented by pre-sorting the "ending" half-edges in and arranging them in decreasing *xy-ppa* order, as described in 3.4, from the higher to lower array indices in the planar map array (see Figure 6). This imposes the position of their duals at the other end of the array, thanks to the implicit involution Φ_M. The successive minimums of PQ^- may thus be later retrieved by a simple scan of the higher section of the array.

Now consider the Edge structure, and let `Edge *links[2]` be its two self-referencing links. These may also be regarded as pointers to the left and right sons of a binary tree structure! Both the *AVL* and the leftist tree structures require an integral field which may also easily be provided for through some bit field in the generic Edge data structure.

Hence, it is possible – using this scheme – to maintain the first priority queue PQ^+ in the low section, and the *AVL* in the high section of the planar map array. This is possible because both structures leave elements in place, and only rearrange pointer links! Thus, the space required for the algorithm is exactly that of the allocated planar map array.

5 Conclusion

In this paper, an optimal divide-and-conquer algorithm for constructing the Delaunay triangulation of a hierarchical graph has been presented. It is based on a previous algorithm for the Delaunay triangulation of a standard graph. Clearly, there is no obstacle for applying the underlying paradigm to the divide-and-conquer construction of other important structures, such as the Constrained Voronoi Diagram or the Medial axis of polygons with holes (see [1]).

It is still an open problem to know whether such a paradigm may be applied to more difficult questions, such as the triangulation of graphs on complex surfaces in space. Research in these fields should have a great impact in *CAD* (Delaunay triangulation of unions of Bezier patches or parametric surfaces), or other geometric applications, such as *GIS*.

Another very important unsolved problem is the design of a divide-and-conquer $O((N+k)\log N)$-time algorithm for the ordered detection of all k intersections between a set of N planar segments. Clearly, such an algorithm would be most beneficial in the preparation of data for large scale applications, since it could be naturally coupled with the divide-and-conquer scheme described in this paper.

Acknowledgements The author wishes to thank P. Volino for invaluable discussions during his stay at *EMSE*, and for his first implementation of the constrained Delaunay triangulation algorithm that gave me the courage to get into this.

References

[1] F. Aurenhammer. Voronoi diagrams – A survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

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

[3] R. Cori. Un code pour les graphes planaires et ses applications. *Astérisque*, 27, 1975.

[4] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report CS-72-259, Dept of Computer Science, Stanford University, CA, 1972.

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

[6] 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.

[7] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, New York, 1993.

[8] A. Jacques. Constellations et graphes topologiques. In *Combinatorial Theory and Applications*, pages 657–673, Budapest, 1970.

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

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

[11] J-M. Moreau and P. Volino. Delaunay triangulation revisited. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 340–345, Waterloo, Canada, August 5-9, 1993.

[12] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, Mass., 1990.

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