Fully dynamic Delaunay triangulation in logarithmic expected time per operation

Olivier Devillers, Stéphane Meiser, Monique Teillaud

Fully Dynamic Delaunay Triangulation in Logarithmic Expected Time per Operation*
Triangulation de Delaunay pleinement dynamique en temps moyen logarithmique par opération

Olivier Devillers[†] **Stefan Meiser**[‡] **Monique Teillaud**[†]

Once upon a time was a Delaunay tree
that wanted to get rid of some site p
or murders in the Delaunay tree.

Programme 4 : Robotique, Image et Vision.
Rapport INRIA 1349, Décembre 1990, révisé en Septembre 1991. Résumé paru à *WADS Août 91 à Ottawa (LNCS 519)*, a paraître dans *Computational Geometry Theory and Applications.*

Keywords : Delaunay triangulation, Dynamic algorithms, Randomized algorithms.

[†]INRIA, BP 93, 06902 Sophia-Antipolis cedex (France),
Phone : +33 93 65 77 62, E-mail : Firstname.Name@alcor.inria.fr
[‡]Max Planck Institut für Informatik, D-6600 Saarbrücken, Phone : +49 681 302-5356, E-mail : meiser@cs.uni-sb.de. This work was done while this author was visiting INRIA
This work has been supported in part by the ESPRIT Basic Research Action Nr. 3075 (ALCOM).

Abstract

The Delaunay Tree is a hierarchical data structure that has been introduced in [6] and analyzed in [7,4]. For a given set of sites $\mathcal{S}$ in the plane and an order of insertion for these sites, the Delaunay Tree stores all the successive Delaunay triangulations. As proved before, the Delaunay Tree allows the insertion of a site in logarithmic expected time and linear expected space, when the insertion sequence is randomized.

In this paper, we describe an algorithm maintaining the Delaunay Tree under insertions and deletions of sites. This can be done in $O(\log n)$ expected time for an insertion and $O(\log \log n)$ expected time for a deletion, where n is the number of sites currently present in the strucuture. For deletions, by expected time, we mean averaging over all already inserted sites for the choice of the deleted sites. The algorithm has been effectively coded and experimental results are given.

Résumé

L'arbre de Delaunay est une structure hiérarchique introduite dans [6] et analysée dans [7,4]. Pour un ensemble $\mathcal{S}$ de sites du plan et un ordre d'insertion de ces sites, l'arbre de Delaunay stocke toute les triangulations de Delaunay successives. L'arbre de Delaunay permet l'insertion d'un site en temps moyen logarithmique et un coût linéaire en moyenne pourvu que les insertions soient randomisées.

Dans ce papier, nous décrivons un algorithme permettant de supprimer un site de l'arbre de Delaunay. Cette opération est menée en temps moyen $O(\log n)$ pour une insertion et $O(\log \log n)$ pour une suppression, où n est le nombre de sites actuellement présent dans la structure. En ce qui concerne les suppressions, par moyen nous entendons en moyenne sur tous les sites déjà insérés pour le choix du site à supprimer. L'algorithme a été effectivement programmé et des résultats expérimentaux sont fournis.

1 Introduction

The Delaunay triangulation and its dual, the Voronoï diagram, are subjects of major interest in Computational Geometry. A lot of algorithms compute it in optimal $\Omega(n \log n)$ time [23,14,13,10,20]. But these algorithms are rather complicated and difficult to implement effectively, so the sub-optimal algorithm [11] is often preferred. Furthermore, this algorithm is on-line and does not impose to compute again the whole triangulation at each insertion.

In the last few years some simpler algorithms have been proposed, non optimal in the worst case but with a good randomized complexity. Some of these algorithms [9,15] use a conflict graph and so are static. The others are on-line ; a first idea of on-line algorithms was presented in [6] and a randomized analysis can be found in [12,4]. Recently, a very simple kind of analysis has been proposed for randomized geometric algorithms [22].

Incremental randomized algorithms have also been used for constructing higher order Voronoï diagrams [16,5,3].

None of the above algorithms allows deletion. Using the algorithm of [1] a site can be removed from a Delaunay triangulation in time sensitive to the modification; their algorithm can however not be combined with an algorithm to insert sites with a good complexity.

In this paper we propose an extension of the Delaunay Tree [6,7] to allow insertion, deletion or location of a site in the Delaunay triangulation with expected logarithmic complexities. The bounds are randomized, i.e. all possible orders for already inserted sites are supposed to be equally likely, and when a site is deleted, it may be any site with the same probability.

The Delaunay tree[6] stores all the successive versions of the Delaunay triangulations during the insertion process. The principle of the deletion algorithm is to reconstruct the past for the triangulation as if the deleted site had never been inserted.

Very recently, some authors took interest in the possibility of deletions in randomized structures. Clarkson, Mehlhorn and Seidel [8] solve the problem for the convex hull in any dimension using the same principle (reconstruction of a new past). Schwarzkopf [21] constructs a larger history of the structure storing not only the insertions but also the deletions. Mulmuley [17,18] uses a radically different approach, and succesfully avoids the storage of the history of the construction.

Section 2 explains the principle of the Delaunay Tree, for insertion only, Section 3 defines the problem of deleting a site, Sections 3.2 and 3.3 describe the algorithm, and Section 4 gives the complexity analysis. Finally in Appendix A we present some practical results for three sets of sites, one is evenly distributed and the others are fairly degenerated.

2 The Delaunay Tree

The Delaunay Tree was introduced in [6], studied in a randomized context [7] and also extended to higher order Voronoï diagrams [5] and to various problems (convex hulls,

arrangements, Voronoï diagrams of line segments...) [4]. We first recall some basic ideas of this structure, before we take interest in the deletion algorithm. More details can be found in [7].

Let $\mathcal{E}$ be the euclidean plane, and $\mathcal{S}$ a set of n sites such that no four sites are cocircular. The Delaunay triangulation of $\mathcal{S}$ is the unique triangulation, with the sites of $\mathcal{S}$ as vertices, such that the circumscribing disk to each triangle does not contain any other site of $\mathcal{S}$. If a site lies inside the circumscribing disk to a triangle, we say that the site is *in conflict* with the triangle. In these terms, the Delaunay triangulation is the set of triangles without conflict. The algorithm described in [6,7] is an on-line algorithm to construct the Delaunay triangulation of $\mathcal{S}$ by adding sites one by one.

The Delaunay Tree is a hierarchical structure based on the incremental procedure of [11]. During the incremental algorithm, the sites are introduced one after another and the triangulation is updated after each insertion. Let p be a site to be introduced in the triangulation. All the triangles in conflict with p can no longer be triangles of the triangulation (and are eliminated in the incremental algorithm). The union of these triangles is a star-shaped polygon $R(p)$ with respect to p. Let $F(p)$ denote the set of edges on the boundary of $R(p)$. The new triangles are obtained by linking p to the edges of $F(p)$.

The Delaunay Tree is constructed in a similar way. But, instead of eliminating triangles during the different steps of the construction, we store all the triangles which have been constructed as nodes of the Delaunay Tree, and at each step we define relationships between triangles of the successive Delaunay triangulations. The aim of this structure is to find $R(p)$ efficiently.

For the initialization step we take the first three sites. They generate one finite triangle and three half-planes (infinite triangles). These 4 triangles will be the sons of the root of the tree.

2.1 Structure of the Delaunay Tree

After the insertion of site p, the triangles in conflict with p are called *dead* and p is their *killer*. Observe that not every triangle must be incident to an edge belonging to $F(p)$ and thus gives rise to new triangles.

Let T be one of the triangles in conflict with p that has an edge E belonging to $F(p)$.

We construct the new triangle S as having vertex p and edge E. Let N be the triangle sharing edge E with T. Because the triangulation is a Delaunay one, the circumscribing disk of S is included in the union of the two disks circumscribing T and N (see Figure 1).

The newly created triangle S will be called : *son of* T and *stepson of* N through edge E. Notice that T is killed by p and is no longer a triangle of the Delaunay triangulation. We call p the *creator* of S.

If we now insert a new site p' in conflict with S but not with N, S will be killed by p' in turn, and its son S' having vertex p' and edge E will be another stepson of N. Thus a node has at most one son and one list of stepsons through each edge, that is at

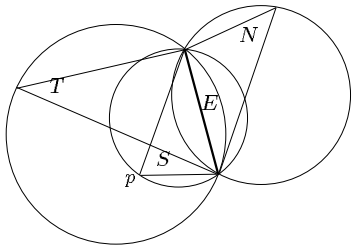

Figure 1: Inserting a new site

most three sons and at most three lists of stepsons.

We also maintain adjacency relationships between the triangles of the current triangulation.

This hierarchical structure is called a *Delaunay Tree* for short, but it is more exactly a rooted directed acyclic graph. This graph contains a tree : the tree whose links are the links between fathers and sons.

We will call a *leaf* of the Delaunay Tree a node associated with a triangle of the final triangulation. Such a triangle is not dead, and so a leaf has no son, but possibly stepsons. The other nodes will be called *internal nodes* (an internal node may have no son but the associated triangle is dead).

2.2 Inserting a new site p

Let p be a site to be introduced in the triangulation. If p is in conflict with a triangle T, we know that it is in conflict with the father of T or with its stepfather. So we will be able to find all the triangles which are killed by p by exploring the Delaunay Tree (in fact, it is also possible to find only one triangle of $R(p)$ by searching the Delaunay Tree, and to deduce the others using neighborhood relationships). For each leaf T in conflict with p, we create some sons if necessary : we look at each neighbor N of T ; if N is not in conflict with p, we create a triangle, son of T and stepson of N. Then we create the adjacency relationships between the triangles created by p.

Insertion(p,T)
    if T has not been visited yet and p is in conflict with T
        for each stepson S of T **Insertion(p,S)** ;
        for each son S of T **Insertion(p,S)** ;
        if T is a leaf
            mark T killed by p ;
            for each neighbor N of T if p is not in conflict with N
                create S son of T stepson of N ;
                update the adjacency relation between S and N
            endfor
        endif

endif

In order to create the adjacency relationships between the new triangles, we proceed as follows. For each new triangle, we know one of its neighbors, namely its stepfather, and we need to find the two others which are also new triangles.

From one of the triangles killed by p, we traverse $R(p)$ by turning around its vertices (using the adjacency relations between dead triangles) and for each pair of consecutive edges on $F(p)$ we create a neighborood relation between the corresponding new triangles.

Given a leaf of the Delaunay tree, we can use the adjacency relations to draw the Delaunay triangulation or the Voronoï diagram. If we want to only locate a new site in the Voronoï diagram (and not insert it) we can find all leaves in conflict as in procedure **Insertion**. This is useful for example for nearest neighbor queries, because the nearest neighbor is a vertex of one of these triangles.

Procedure **Insertion** is described in detail in [6]. A more comprehensive analysis than in Section 4 can be found in [7].

3 Deletion of a site in the Delaunay Tree

Let $\mathcal{S}$ be a set of n sites. We assume that the Delaunay Tree has been constructed for the set $\mathcal{S}$, by using the incremental randomized algorithm. We now want to remove a site p of $\mathcal{S}$. All the triangles incident to p must be removed from the Delaunay Tree : some of them are triangles of the Delaunay triangulation of $\mathcal{S}$ (so they are leaves of the Delaunay Tree), but other ones already died ; they correspond to internal nodes of the Delaunay Tree, and must be removed, too. Moreover, we must restore the Delaunay Tree in the same state it would be in if p had never been inserted, and if the other sites had been inserted in the same order. That way, we preserve the randomized hypothesis on the sequence of sites, and the conditions for further insertions or deletions are fulfilled.

We must thus reconstruct a past for the final triangulation in which p takes no part. The deletion of p creates a "hole" in each successive triangulation after the insertion of p, which the tree keeps a trace of. The idea of our algorithm is to fill each hole with the right Delaunay triangulation.

Let us describe the structure of a node of the Delaunay Tree (some fields of a node have not been used yet and will be defined in the following) :
- the triangle : creator vertex, two other vertices, circumscribed circle
- a mark **dead**
- pointers to the at most three sons and the list of stepsons
- pointers to the father and the stepfather
- the three current neighbors if the triangle is not dead, the three neighbors at the death otherwise
- the three neighbors at the time of the creation
- two special neighbors (defined in Section 3.3)

- a pointer **killer** to the site that killed the triangle
- a mark **to be removed**
- three pointers **star** to elements of structure **Star** (defined in Section 3.3)

3.1 Different kinds of modified nodes

Let us describe how the deletion of p affects the nodes in the Delaunay Tree.

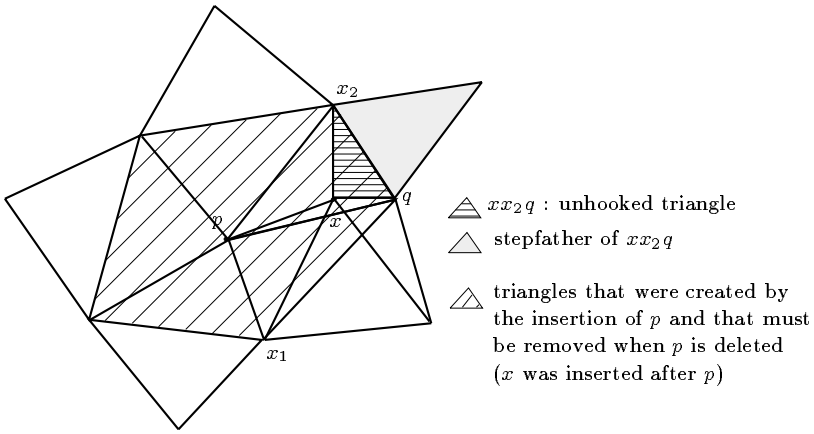

Figure 2: The two kinds of modified nodes : xx_1p and xx_2p where created by the insertion of x and must be removed when p is deleted, xx_2q is unhooked because its father px_2q is removed

Some nodes must be removed : they correspond to triangles having p as a vertex. Depending on its time of creation there are two cases for such a node : either it has been created by the insertion of p, or it has been created by the insertion of some site afterwards ; the latter occurs *iff* its father and stepfather both have p as a vertex and thus both the parents must be removed, too. During the construction of the Delaunay Tree, some sites did not create any triangle to be removed now, but if a site x created such a triangle, it created in fact two triangles to be removed : the two triangles created by x sharing edge px, see Figure 2.

A node to be deleted xx_2p may have a son (or a stepson) xx_2q that does not have p as vertex and thus must remain in the Delaunay tree, see Figure 2. Such a node loses just one of its two parents and is therefore called *unhooked*. We must find a new parent in replacement of the lacking one.

The sketch of the method is the following :
Search step : Find all nodes of the Delaunay Tree that have to be removed, and all unhooked nodes
Reinsertion step : Locally reinsert the sites that are creators of the triangles found during the Search step, and update the triangulation

3.2 The Search step

By the discussion above, the set of nodes to be removed can be found by searching the Delaunay Tree starting from the nodes that were created by p. At each node marked to be removed we visit all its sons and stepsons recursively. If one of them has p as a vertex, it will be marked to be removed as well. Otherwise it is an unhooked node. The creator of both these types of triangles must be reinserted, in order to replace the removed triangles by other triangles, and to hang up unhooked triangles again.

In order to be able to perform the Reinsertion step, we must store the list of sites to be reinserted :

> We need an auxiliary structure, **Reinsert**, which is a balanced binary tree consisting of the set of sites which created the nodes to be removed and the unhooked nodes ; the sites are sorted by order of insertion. This will allow us to reconstruct the triangles which will fill the holes in the successive triangulations, and to hang up again the unhooked nodes.
>
> An element of **Reinsert** contains :
> • the site x to be reinserted
> • pointers to the two triangles xx_1p and xx_2p that were created by the insertion of x, if they exist (see Section 3.1). xx_1p is turning clockwise
> • the list of unhooked triangles that were created by the insertion of x

The search is initialized by the set C of nodes created by p.

> To this aim, we must maintain an auxiliary array, **Created**, containing, for each site s of $\mathcal{S}$, a pointer to one of the nodes created by s.

From this node, we can then compute the set C using the neighborhood relations at the time of creation and examining the creator of the triangles (Figure 3).

```
for each element S of C
    for each son or stepson U of S
        examine(U)
    endfor ;
    remove links between S and its father and stepfather ;
    put S in "garbage collector"
endfor
```

We then simply recursively traverse the subgraph consisting of removed and unhooked nodes. Each son or stepson of a removed node is removed if it has p as vertex and unhooked otherwise. All these nodes are added in the element of **Reinsert** associated to their creator. This process is detailed above.

```
examine(T)
if T is marked to be removed
    { T has already been visited }
```

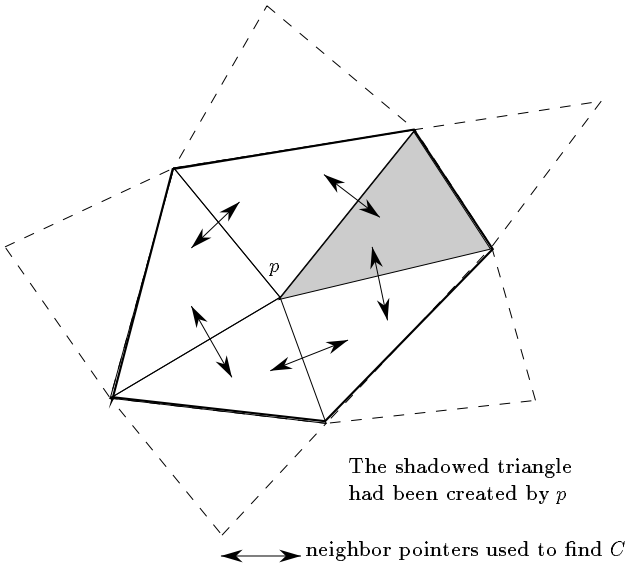

Figure 3: The search step : Initialization

```
    nothing
else
if p is a vertex of T
{ T = xsp}
    mark T to be removed ;
    for each son or stepson S of T
        examine(S)
    endfor ;
    locate the creator x of T in Reinsert ;
        { if the location fails, a new element is created }
    if xsp turns clockwise
        store T in xx_1p
    else
        store T in xx_2p
    endif
else
    { T is either son or stepson of a removed node }
    locate the creator x of T in Reinsert ;
    add T to the list of unhooked nodes associated to x ;
    set the pointer to the removed parent of T as null
endif
```

Observe that in the search step the triangles are only visited. They are removed from the Delaunay Tree later during the reinsertion step.

3.3 The Reinsertion step

The sites contained in **Reinsert** must be reinserted in the Delaunay Tree in order to construct the successive triangulations without site p. The scheme of the reinsertion of a site x is the same as the usual scheme of insertion, except that everything happens locally : the location of a site x to be reinserted in the whole Delaunay Tree is unnecessary and would be too expensive.

The location in a generally small set A (for active) of triangles is sufficient. At the beginning of the reinsertion set A is initialized with all triangles killed by the insertion of p. They can be computed by looking at the fathers of the triangles in C and following their neighbor pointers at their death.

Then, during the Reinsertion step, A is maintained so that it is the set of triangles in conflict with p in the Delaunay triangulation at the time just preceeding the reinsertion of x. In each step, A is modified as follows : all the triangles of A in conflict with x are killed by x and thus disappear from the Delaunay triangulation and from A. The triangles created by the reinsertion of x appear in A (Figure 4), because they are in conflict with p (otherwise they would have existed in the triangulation containing p). The triangles of A not in conflict with x still remain in A. The triangles outside A are not modified by a reinsertion since they are not in conflict with p ; only their neighborhood or stepson relations involving removed nodes must be updated.

More precisely, the set A of triangles must be organized so that location of conflicts is efficient. We can notice that the triangles in A form a star-shaped polygon with respect to p, since they are in conflict with p, cf. the discussion in Section 2.

The edges and vertices (sites) of this polygon are stored in counterclockwise order in a circular list called **Star**. Furthermore we maintain some pointers :

- Each edge of **Star** points to the adjacent triangle in A.
- Each triangle in A points to the adjacent edges of **Star** (at most three, pointers **star** in the description of a node).
- Each site on **Star** points to the edge of **Star** following the site.

Some elements of A are not represented in **Star**, but the whole set A can nevertheless be traversed using **Star** and pointers to neighbors.

Star can be initialized by first choosing a vertex of a triangle in A. We then follow the boundary of the star-shaped polygon using the neighborhood relations, and the pointers **star**.

We know the current neighbors of each triangle of A. Each edge e of **Star** is an edge of a triangle U of A and of a triangle V that does not belong to A.

> The current neighbor of U through e is V, but the reciprocal relation does not always exist ; the neighbor pointer of V through e may reach another triangle W created a long time later.
>
> If W must not be removed, this pointer must remain after the deletion of p. So we do not want to systematically modify the current neighbors

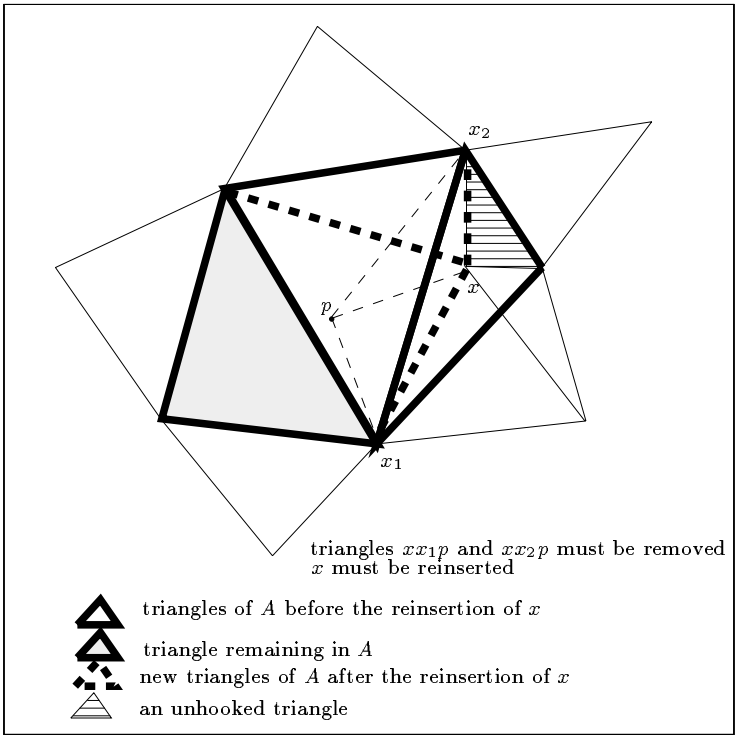

Figure 4: A reinsertion

of triangles not belonging to A. We need to put a special neighbor pointer from V to U. Thus each triangle outside A, having an edge on the boundary of A, has a special neighbor pointer to the adjacent triangle in A. It is easy to see that two special neighbor pointers are enough : at most two edges of a given triangle lie on the boundary of a star-shaped polygon, if it is exterior to it. The special neighbor pointers store intermediate relations between triangles.

Nevertheless, if W must be removed, then a new neighbor must be found for V, and the current neighbor must be maintained identical to the special one. When we update a special neighbor U of V, if the neighbor at creation of V is removed, then U is also the neighbor at creation of V. Similarly, if the current neighbor of V is removed, or if it belongs to A, then it must be updated to be U.

Everything is now set up to start the reinsertion. Each site in Structure **Reinsert** is reinserted in the right order (the order used for first insertion).

Processing the unhooked triangles
Each element of **Reinsert** contains a site x to be reinserted, and the list of corresponding unhooked triangles. To hang up such a triangle T again, we only have to go to the remaining parent of it, which must have an edge in **Star**, and then hang T up to the appropriate special neighbor of this parent. There may also exist some removed triangles created by x (Figure 5). Notice that this is not always true (Figure 6). If there is no removed triangle, the unhooked triangle necessarily needs a stepfather, which is also the neighbor at creation and the special neighbor, all these three triangles are set to the special neighbor of the father.

Replacing the triangles to be removed by new ones
For each element n of **Reinsert**, we check if triangle xx_1p (and xx_2p) exists. If xx_1p and xx_2p do not exist in the triangulation, then nothing has to be done. Otherwise we have to fill the gap of triangles incident at x between edges xx_1 and xx_2. We must look at **Star** in order to find the triangles that have to be created by the reinsertion of x. There are two cases : there may exist no triangle of A in conflict with x, or several such triangles.

First note that x_1 and x_2 both belong to **Star**. Let U be the triangle of A adjacent to the edge following x_1 on **Star** (remember that **Star** is oriented counterclockwise and xx_1p clockwise). U serves to distinguish between the two cases above.

After the reinsertion of x, the edges xx_1 and xx_2 will be on the boundary of the new set A of triangles in conflict with p. So, if there are some vertices on the current boundary of A, between x_1 and x_2, the triangles adjacent to the edges of this chain of vertices must be in conflict with x. U is such a particular triangle. Thus, if U is not in conflict with x, x_1x_2 is an edge on the boundary of A, and consequently of U, and the first case occurs, otherwise the second case occurs.

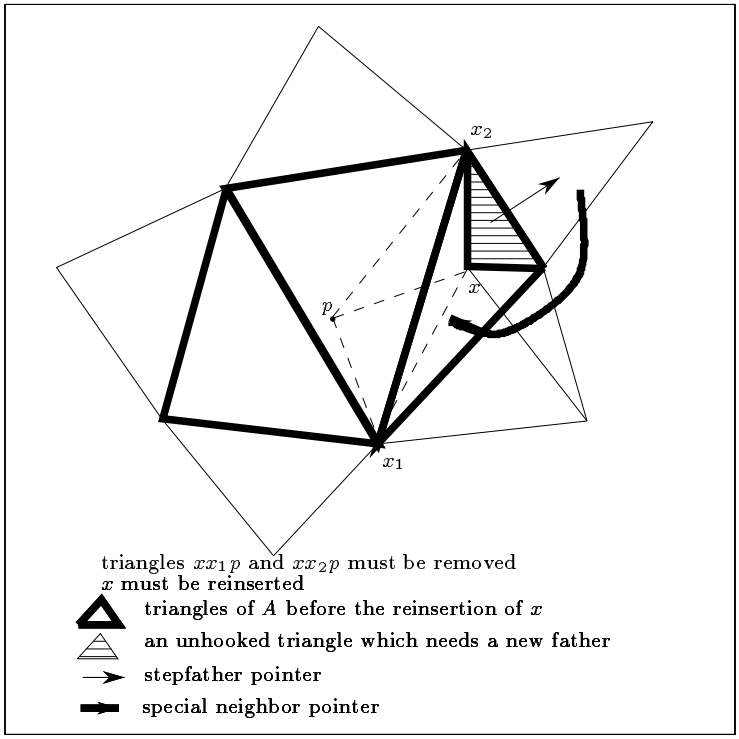

Figure 5: An unhooked triangle with some removed triangles

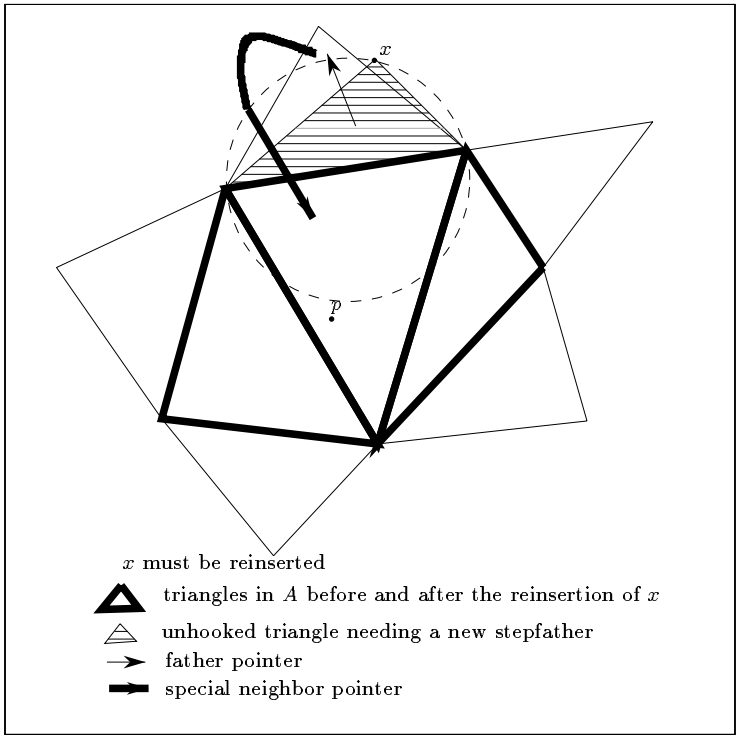

Figure 6: An unhooked triangle in the case that there is no removed triangle

First case, see Figure 7 :
In this case, the only way to fill the gap is to replace the removed triangles xx_1p and xx_2p around x by only one new triangle xx_1x_2. The new triangle xx_1x_2 has U as stepfather and the neighbor of U, which does not belong to A, as father. Details concerning other neighbor pointers can be found in the following pseudo code procedure :

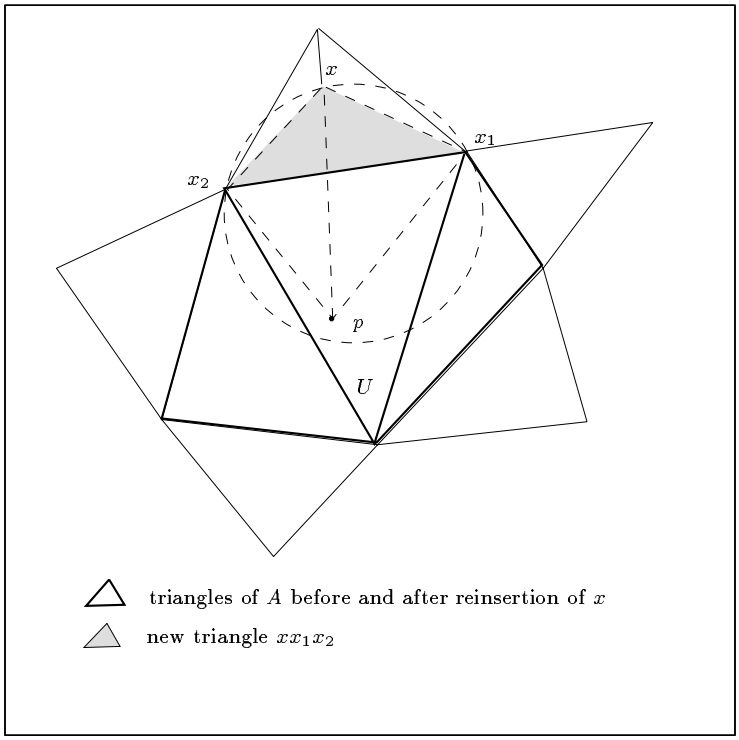

Figure 7: Reinsertion - removed triangles - First case

Reinsertion - removed triangles - First case
create triangle xx_1x_2,
with U as stepfather and the neighbor of U through x_1x_2 as father ;
update the neighbor through x_1x_2 of U to be xx_1x_2 ;
find the neighbors at the creation of xx_1x_2, by looking at those of xx_1p and xx_2p ;
update the special neighbor pointers of the neighbors of xx_1x_2
and their neighbor at creation, and current neighbor, if necessary ;
throw xx_1p and xx_2p away in "garbage collector" ;
add edges xx_1 and xx_2 in **Star** ;
update **Star** by letting xx_1 and xx_2 be incident to xx_1x_2 ;
put two **star** pointers from xx_1x_2 to the elements xx_1 and xx_2 of **Star** ;
in **Created**, xx_1x_2 is a triangle created by x

Second case, see Figures 8 and 9 :
We know that U is in conflict with x. We must find all the triangles in A in conflict with x. Those triangles may be fathers for the nodes that will be created by x. They form a connected subset of A, so they will be found owing to neighbor pointers in the following way :

Starting with U we visit the triangles in A incident at x_1 in counterclockwise order until we reach a triangle not in conflict with x. Let V denote the last such triangle in conflict with x and let e denote the edge of V at which the visit stops. Let $e = x_1x'$. We create triangle $W' = xx_1x'$ and start this process again at vertex x' with V as starting triangle. When vertex x_2 is reached, all the new triangles have been created. The iteration continues until vertex x_1 is reached in order to mark the triangles killed by x.

During the traversal, when a new triangle is created, we update the neighbor and **star** pointers of its neighbors. Once this is achieved, it remains to compute all kinds of neighborhood relations involving edges xx_1 and xx_2. Particularly, as the current neighbor of the just created triangle having edge xx_1, we take the neighbor of the now removed triangle xx_1p at its creation. The same holds for edge xx_2. The pseudo code procedure below formalizes these operations.

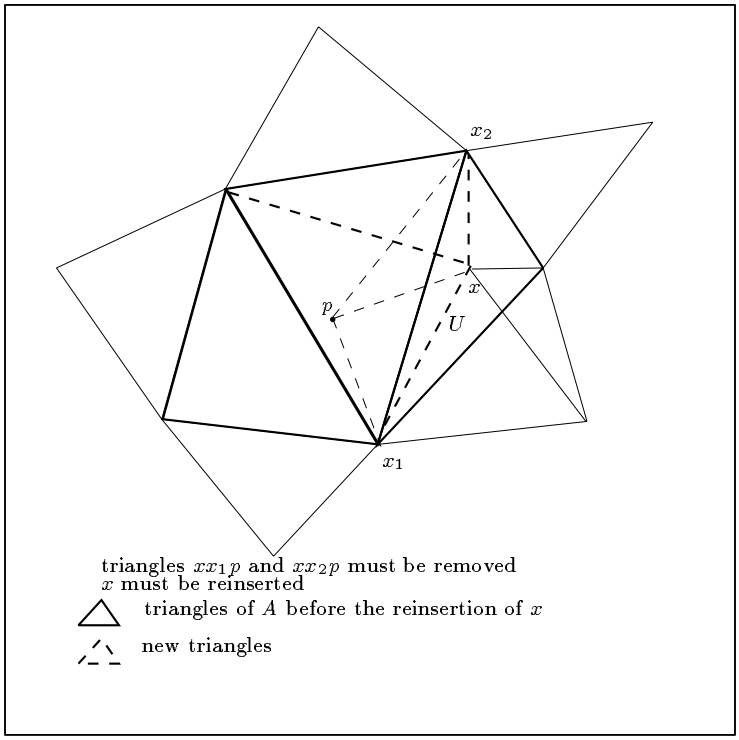

Figure 8: Reinsertion - removed triangles - Second case

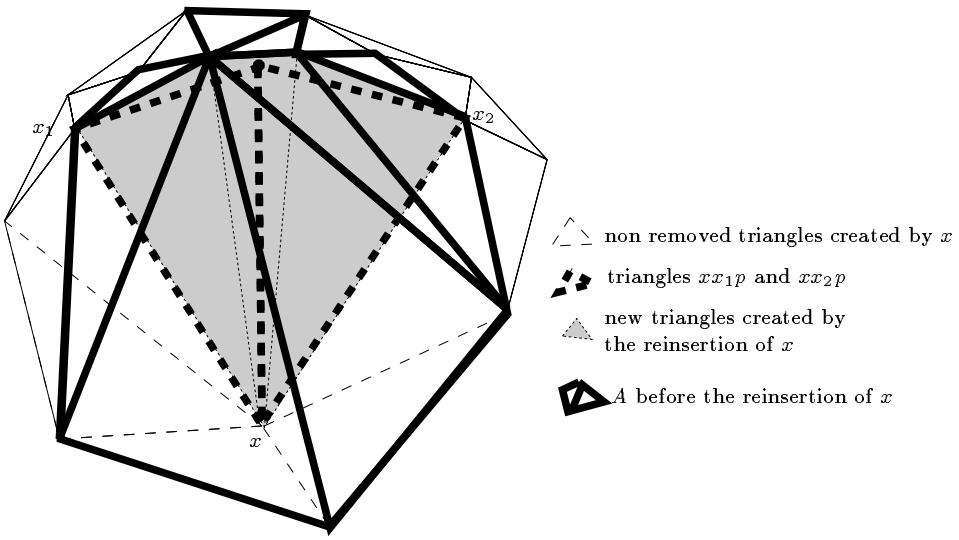

Figure 9: A non trivial example for reinsertion

Reinsertion - removed triangles - Second case
$V \leftarrow U$
$s \leftarrow x_1$
repeat
while the neighbor of V sharing vertex s is in conflict with x
{ we turn around s counterclockwise }
$V \leftarrow$ this neighbor ;
the **killer** of V is x
endwhile ;
$e \leftarrow$ edge of V through which we stopped finding conflict ;
create W' with edge e and vertex x ;
W' is the son of V, and the stepson of the neighbor of V through e ;
the neighbor at creation of W' through e is its stepfather ;
if e is an edge of **Star**
update the element of e in **Star** by replacing V by W' ;
let the **star** pointer of W' reach e ;
update the special neighbor of the neighbor of V through e to be W'
and the ordinary neighbor if necessary
else
update the neighbor of the neighbor of V through e to be W'
endif ;
if $s \neq x_1$
W' and W are neighbors and neighbors at creation through edge xs
endif ;

```
    if s = x_1
        W_1 ← W' ;
        W ← neighbor at creation of x x_1 p through x x_1 ;
        W' and W are neighbors at creation through edge x x_1 ;
        the ordinary neighbor through x x_1 of W_1 is W ;
        the special neighbor through x x_1 of W is W_1 ;
        the ordinary neighbor through x x_1 of W is W_1 if necessary
    endif ;
    s ← the other vertex of e ;
    if s = x_2
        W_2 ← W' ;
        W ← neighbor at creation of x x_2 p through x x_2 ;
        W' and W are neighbors at creation through edge x x_2 ;
        the ordinary neighbor through x x_2 of W_2 is W ;
        the special neighbor through x x_2 of W is W_2 ;
        the ordinary neighbor through x x_2 of W is W_2 if necessary
    endif ;
    W ← W'
    { W must be stored for future neighborhood relations }
until s = x_2 ;
store for example W_1 as created by x in Created ;
throw x x_1 p and x x_2 p away in "garbage collector" ;
repeat
    while the neighbor of V sharing vertex s is in conflict with p
        { we turn around s counterclockwise }
        V ← this neighbor ;
        the killer of V is x
    endwhile ;
    s ← the third vertex of V
until s = x_1 ;
replace the polygonal chain of Star between x_1 and x_2
    by the edges x_1 x and x x_2, associated with W_1 and W_2 ;
the star pointers of W_1 and W_2 reach respectively x x_1 and x x_2 ;
```

4 Analysis

4.1 Randomized analysis of the insertion algorithm

This subsection aims at providing a randomized analysis of the space and time required to build the Delaunay Tree. Randomization here only concerns the order in which the inserted sites are introduced into the structure. Thus, if the current set of sites is a set S of cardinality n, our results are expected values that correspond to averaging over the $n!$ possible permutations of the inserted objects, each equally likely to occur.

We first prove some probabilistic results that are purely combinatorial.

Probabilistic Lemma

We first introduce some additional notation and definitions. $\mathcal{F}(\mathcal{S})$ is the set of all the triangles having sites of $\mathcal{S}$ as vertices. We define the width of a triangle of $\mathcal{F}(\mathcal{S})$ to be the number of sites of $\mathcal{S}$ in conflict with it. $\mathcal{F}_j(\mathcal{S})$ is the set of triangles of width j and $\mathcal{F}_{\leq j}(\mathcal{S})$ is the set of triangles of width at most j. $\mathcal{F}_0(\mathcal{S})$ is the Delaunay triangulation of $\mathcal{S}$.

We also define a bicycle as a pair of triangles sharing an edge. A site is said to be in conflict with the bicycle, if it is in conflict with one of the triangles (but it is not one of their vertices). We denote by $\mathcal{G}(\mathcal{S})$ the set of bicycles, and we derive the notations $\mathcal{G}_j(\mathcal{S})$and $\mathcal{G}_{\leq j}(\mathcal{S})$ accordingly.

The first Lemma, due to Clarkson and Shor [9], bounds the numbers $|\mathcal{F}_{\leq j}(\mathcal{S})|$ and $|\mathcal{G}_{\leq j}(\mathcal{S})|$ of triangles and bicycles with width at most j defined by $\mathcal{S}$. The proof of this lemma uses the random sampling technique [9], (proofs are also given in [7,4]).

Lemma 4.1 [Clarkson, Shor]

$$\begin{aligned} |\mathcal{F}_{\leq j}(\mathcal{S})| &= O\left(n(j+1)^2\right) \\ |\mathcal{G}_{\leq j}(\mathcal{S})| &= O\left(n(j+1)^3\right) \end{aligned}$$

Expected storage

Lemma 4.2 *If $\mathcal{S}$ has cardinality n, the expected size of the Delaunay Tree of $\mathcal{S}$ is $O(n)$.*

Proof : The expected number of nodes $\eta(\mathcal{S})$, in the Delaunay Tree of $\mathcal{S}$ can be obtained by summing, for all the triangles T of $\mathcal{F}(\mathcal{S})$the probability that T occurs as a node in the Delaunay Tree. If j denotes the width of T, then this probability is $\frac{3!j!}{(3+j)!}$ (the 3 vertices of T must be inserted before the j sites in conflict with T and the order of the other sites is not relevant). By Lemma 4.1,

$$\begin{aligned} \eta(\mathcal{S}) &= \sum_{j=0}^{n-3} |\mathcal{F}_j(\mathcal{S})| \frac{3!j!}{(3+j)!} \\ &= \left(|\mathcal{F}_0(\mathcal{S})| + \sum_{j=1}^{n-3} \left(|\mathcal{F}_{\leq j}(\mathcal{S})| - |\mathcal{F}_{\leq (j-1)}(\mathcal{S})| \right) \frac{3!j!}{(3+j)!} \right) \\ &= \left(\sum_{j=0}^{n-4} |\mathcal{F}_{\leq j}(\mathcal{S})| 3 \frac{3!j!}{(4+j)!} + |\mathcal{F}_{\leq (n-3)}(\mathcal{S})| \frac{3!(n-3)!}{n!} \right) \\ &\leq O\left(\sum_{j=1}^{n} \frac{n}{j^2} \right) = O(n) \end{aligned}$$

As each node has exactly two parents, the number of edges in the Delaunay tree is bounded by the same quantity.

□

Expected time

Lemma 4.3 *If $\mathcal{S}$ has cardinality n, the expected time for inserting the last site in the Delaunay Tree is $O(\log n)$.*

Proof : The main cost of insertion is the cost of locating the new site. The computing time spent to locate the triangles in conflict with the last inserted site p is proportional to the total number of bicycles in conflict with p. Actually, if a node S is visited, at least one of its two parents, say T, is in conflict with p so the bicycle ST is in conflict with p.

Let B be a bicycle of $\mathcal{G}_j(\mathcal{S})$. B is in conflict with p if p is one of the j sites in conflict with B and if the 4 objects defining B have been inserted before the j objects in conflict with B. This occurs with the probability : $\frac{j}{n}\frac{4!(j-1)!}{(3+j)!}$. The expected number $\theta(\mathcal{S})$ of nodes visited during the last insertion is then obtained by summing, for all the bicycles B of $\mathcal{G}(\mathcal{S})$, the above probability. Using Lemma 4.1, this yields :

$$\theta(\mathcal{S}) = \sum_{j=1}^{n-4} |\mathcal{G}_j(\mathcal{S})| \frac{4!j!}{n(3+j)!} = O\left(\frac{1}{n}\sum_{j=1}^{n}\frac{n}{j}\right) = O(\log n),$$

from a calculation similar to the proof of Lemma 4.2 □

It is important to notice that this bound applies also if the point is only located in the Voronoï diagram and not really inserted.

4.2 Randomized analysis of deleting a point

We assume that p is a random site in $\mathcal{S}$, i.e. p is any of the preceedingly inserted sites, with the same probability and independently from the insertion order. More precisely, an event is now one of the $n!$ permutations and one of the n sites. Each event occurs with the same probability $\frac{1}{n.n!}$.

Lemma 4.4 *The expected number of removed nodes is constant.*

Proof : Since p is chosen independently from the insertion order, the expected number of removed nodes is

$$\begin{aligned} & \sum_{T \text{ triangle}} Prob(p \text{ vertex of } T) Prob(T \text{ exists in the Delaunay Tree}) \\ = \; & \frac{3}{n} \times \text{ expected number of vertices of the Delaunay Tree} \\ = \; & O(1) \end{aligned}$$

□

Lemma 4.5 *The expected number of unhooked nodes is constant.*

Proof :
In fact the number of unhooked nodes is bounded by the number of edges disappearing in the Delaunay tree, which is :

$$\begin{aligned} & \sum_{T,S \text{ adjacent triangles}} Prob(p \text{ vertex of } T \text{ or } S) \\ & \qquad \times Prob(TS \text{ is an edge of the Delaunay Tree}) \\ = \; & \frac{4}{n} \times \text{ expected number of edges of the Delaunay Tree} \\ = \; & O(1) \end{aligned}$$

□

Lemma 4.6 *The expected number of nodes created by the deletion of p is constant.*

Proof : The number of created triangles during the deletion of p is

$$\sum_{T \text{ triangle}} Prob(T \text{ appears during the deletion of } p)$$

A triangle T of width j will appear *iff* p is one of the j sites in conflict with T, and p and the 3 vertices of T are introduced before the $j-1$ other sites in conflict with T, and p is not inserted after the 3 vertices of T. So the probability that T appears is :

$$\frac{j}{n} 3 \frac{3!(j-1)!}{(j+3)!} = \frac{3}{n} \frac{3!j!}{(j+3)!}$$

Hence by virtue of the proof of Lemma 4.2, the expected number of triangles created by the deletion of p is :

$$\frac{3}{n} \sum_{T} Prob(T \text{ apears during the insertion phase}) = O(1)$$

□

Lemma 4.7 *The expected cost of a deletion is $O(\log\log n)$.*

Proof : The expected number of triangles killed by p is constant using Lemma 4.4 (which also implies that the initialization of **Star** is achieved in constant time), and the traversal that is done during the Search step visits a constant number of nodes by Lemma 4.5. For each node, we must locate the creator of the node in **Reinsert**, which can be done in $O(\log\log n)$ worst case deterministic time, by using a bounded ordered dictionary [24]. The universe for this dictionary is the insertion age of the points, or in other words the number of the sites. The required finiteness of the universe can be circumvented using standard dynamization techniques, see for example [19, section5.2].

To preserve the simplicity of the auxiliary data structures we can use a simple balanced binary search tree [2]. In this way we achieve a complexity of $O(\log n)$ time. During the reinsertion phase **Star** can be updated in time proportional to the number of removed nodes.

The total cost of the work on unhooked triangles is constant, since we only have to reach the neighbor of the parent of each of them, and by Lemma 4.5.

For the triangles deleted by the reinsertion of x, the cost is linear in the number of triangles in conflict with both p and x, which is linear in the number of triangles created by the reinsertion of x. By Lemma 4.6, this expected cost is thus constant.

The expected whole cost is then less than $O(\log\log n)$. □

It is important to notice that the randomized hypothesis is preserved by a deletion. Namely, consider now the permutation σ of $\mathcal{S} \setminus \{p\}$ obtained by removing p from the insertion order ; there are n permutations of $\mathcal{S}$ which give the same σ, so the probability that σ occurs is $n \times \frac{1}{n!}$ and σ is really a random permutation of $\mathcal{S} \setminus \{p\}$. The randomization of the $n-1$ currently present sites is actual and the deletion of p does not affect the analysis of further insertions or deletions. Thus Lemmas 4.2, 4.3 and 4.7 yield the main theorem of this paper :

Theorem 4.8 *The Delaunay triangulation (or the Voronoï diagram) of a set $\mathcal{S}$ of n sites in the plane can be dynamically maintained in $O(\log n)$ expected time to insert or locate a point and $O(\log\log n)$ expected time to delete a point. This result holds provided that, at any time, the order of insertion on the sites remaining in $\mathcal{S}$ may be each order with the same probability, and when a site is deleted, it may be any site with the same probability.*

It is possible to avoid the hypothesis that the random deleted site and the random insertion permutation are independent. It is clear that the deletion of the first inserted site is more expensive that the deletion of the last one, but we show in the sequel that even the deletion of the first inserted site can be done with a good complexity. For this other kind of analysis of deletions, the probability space is only the set of permutations for the insertion, each equally likely to occur.

Lemma 4.9 *The expected number of removed, unhooked and created nodes during the deletion of the first inserted site is $O(\log n)$.*

Proof : We here only give the proof for the removed nodes, the other quantities can be obtained in the same way. A triangle T of width j exists in the Delaunay tree and is removed during the deletion of the first inserted site if the first site is a vertex of T and if the two other vertices of T are inserted before the j sites in conflict with T. This happens with probability $\frac{3}{n}\frac{2!j!}{(j+2)!}$. By summing for all triangles T, the number of nodes removed in the Delaunay Tree is :

$$\sum_{j=0}^{n-3} |\mathcal{F}_j(\mathcal{S})| \frac{3}{n} \frac{2!j!}{(2+j)!} = O(\log n)$$

□

The same result holds for the k^{th}site : if we do not consider the first site but the k^{th}site, the probability that a triangle is removed during the deletion of the k^{th}site is clearly less than $\frac{3}{n}\frac{2!j!}{(j+2)!}$.

Therefore, the proof of Lemma 4.7 can be modified in a straightforward manner to obtain the following result :

Theorem 4.10 *The expected cost of deleting the k^{th}site is $O(\log n \log\log n)$.*

Thus, for any deletion sequence, the whole set of sites can be deleted in expected time $O(n \log n \log\log n)$, where the expectation is only on the insertion sequence.

5 Conclusion

We have shown that the Delaunay triangulation can be maintained in $O(\log n)$ expected time per insertion and $O(\log\log n)$ per deletion and $O(n)$ space (where n is the number of sites at the time of the operation).

The analysis is randomized, i.e. the result holds provided that at any time the order of insertion of the sites in the triangulation at that moment may be any possible order with the same probability. And when a site is deleted, it may be any site in the triangulation with the same probability. An important point is that our hypotheses are on the insertion order only; there are no assumptions on the distribution of the sites.

This algorithm is practical and has been effectively coded (see Appendix A), the numerical computations involved are simple; the only numerical calculous is to test if a site lies inside or outside a circle. The data structures involved by the algorithm are not too much complicated; besides the Delaunay Tree itself, we only need a balanced binary search tree and a split and find structure.

Further investigations are to be done. In the same way as the insertion aspect of the Delaunay Tree [7] was generalized to various geometric problems [4], the deletion aspect can probably be generalized to other problems. The major difficulty seems to be to find the analogous of structure **Star** to locate efficiently an element in the set A. Even for the straightforward generalization to the three dimensional case, the problem is unsolved.

Acknowledgements

The authors would like to thank Jean-Daniel Boissonnat, who participated in the first work on the Delaunay tree, Mariette Yvinec, for a careful reading of the paper and Jean-Pierre Merlet for supplying us with his interactive drawing preparation system JPdraw .

A Practical Results

The algorithm described in this paper has been effectively coded. This section presents practical results. Figures 10, 12 and 14 illustrate the triangulations of a set of random sites in a square, on an ellipse and on a parabola respectively (100 sites only to preserve the possibility of visualisation). The sites are first inserted in a random order, and afterwards they are all deleted in another random order. Figures 11, 13 and 15 show the size of the Delaunay Tree in bold line, the size of the Delaunay triangulation in dashed line, and in thin line, a measure of the complexity of the operation. For insertions, it is the number of visited nodes, for deletions it is the number of unhooked triangles plus the numbers of triangles created during this deletion plus the cost of each location in structure **Reinsert**. The cost of deleting a site has a higher variance than the cost of inserting a site ; it may be important if the site had been inserted at the beginning of the construction, but this happens with a low probability.

Even in the case of sites lying on a parabola, which gives a bad behaviour for most of the existing algorithms, our algorithm has a good behaviour in practice.

The Delaunay triangulation of 15000 sites has been computed in 35 seconds on a Sun 4/75 and the deletion phase has been computed in 50 seconds.

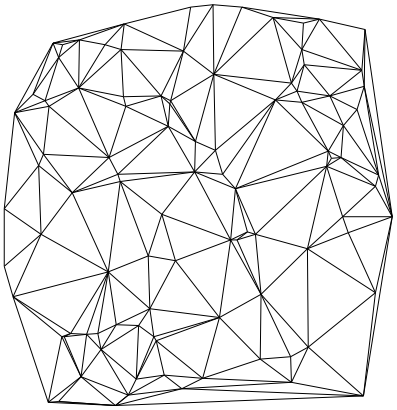

Figure 10: Delaunay triangulation of 100 random sites in a square

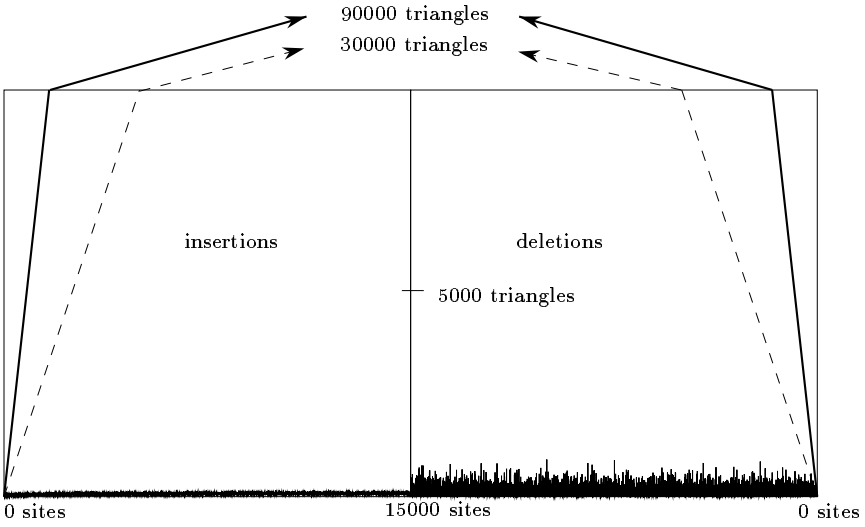

Figure 11: Statistics on 15000 random sites in a square

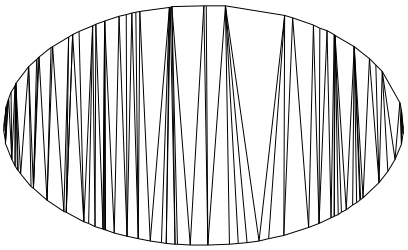

Figure 12: Delaunay triangulation of 100 random sites on an ellipse

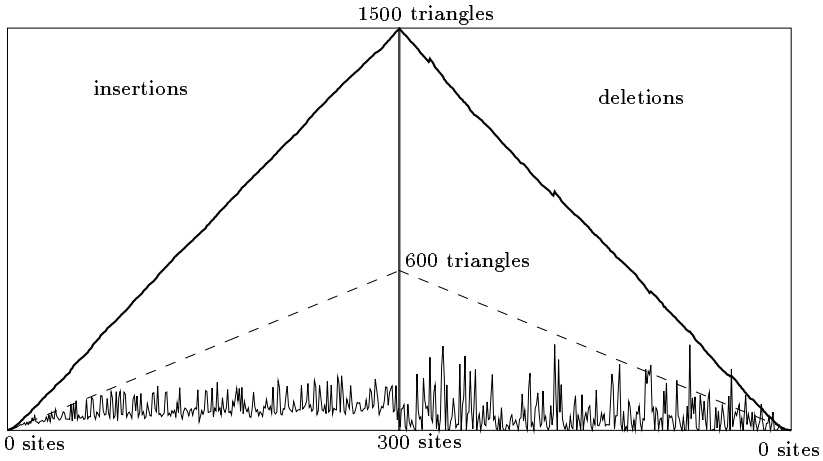

Figure 13: Statistics on 300 random sites on an ellipse

Figure 14: Delaunay triangulation of 100 random sites on a parabola

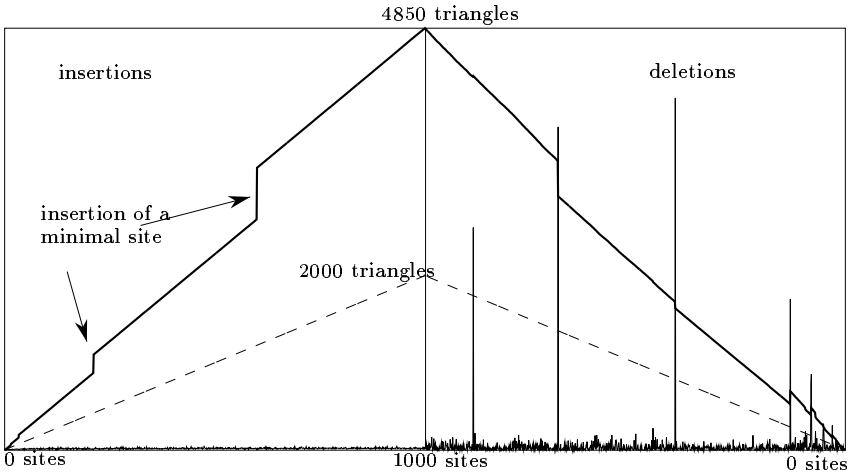

Figure 15: Statistics on 1000 random sites on a parabola

References

[1] A. Aggarwal, L.J. Guibas, J. Saxe, and P.W. Shor. A linear time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The design and ananlysis of computer algorithms. Computer science and information processing*, Addison Wesley, 1983.

[3] F. Aurenhammer and O. Schwarzkopf. A simple on-line randomized incremental algorithm for computing higher order Voronoi diagrams. In *7th ACM Symposium on Computational Geometry in North Conway*, pages 142–151, June 1991. Full paper available as Technical Report B 91-02 Universität Berlin, to appear in IJCGA.

[4] J-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete and Computational Geometry*. To appear. Available as Technical Report INRIA 1285. Abstract published in IMACS 91 in Dublin.

[5] J-D. Boissonnat, O. Devillers, and M. Teillaud. A semi-dynamic construction of higher order Voronoi diagrams and its randomized analysis. *Algorithmica*. To appear. Available as Technical Report INRIA 1207. Abstract published in Second Canadian Conference on Computational Geometry 1990 in Ottawa.

[6] J-D. Boissonnat and M. Teillaud. A hierarchical representation of objects: the Delaunay Tree. In *Second ACM Symposium on Computational Geometry in Yorktown Heights*, June 1986.

[7] J-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. *Theoretical Computer Science*. To appear. Available as Technical Report INRIA 1140.

[8] K.L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *LNCS 577 (STACS 92)*, Springer-Verlag, 1992.

[9] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4(5), 1989.

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

[11] P.J. Green and R. Sibson. Computing Dirichlet tesselations in the plane. *The Computer Journal*, 21, 1978.

[12] L.J. Guibas, D.E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[13] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[14] D.T. Lee and B.J. Schachter. Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer and Information Sciences*, 9(3), 1980.

[15] K. Mehlhorn, S. Meiser, and C. Ó'Dúnlaing. On the construction of abstract Voronoi diagrams. *Discrete and Computational Geometry*, 6:211–224, 1991.

[16] K. Mulmuley. On levels in arrangements and Voronoi diagrams. *Discrete and Computational Geometry*, 6:307–338, 1991.

[17] K. Mulmuley. Randomized multidimensional search trees : dynamic sampling. In *7th ACM Symposium on Computational Geometry in North Conway*, pages 121–131, 1991.

[18] K. Mulmuley and S. Sen. Dynamic point location in arrangements of hyperplanes. In *7th ACM Symposium on Computational Geometry in North Conway*, pages 132–142, 1991.

[19] M.H. Overmars. *The design of dynamic data structures. LNCS 156*, Springer-Verlag, 1983.

[20] F.P. Preparata and M.I. Shamos. *Computational Geometry : an Introduction.* Springer-Verlag, 1985.

[21] O. Schwarzkopf. Dynamic maintenance of geometric structure made easy. In *IEEE Symposium on Foundations of Computer Science*, October 1991. Full paper available as Technical Report B 91-05 Universität Berlin.

[22] R. Seidel. Backwards analysis of randomized geometric algorithms. June 1991. Manuscript.

[23] M.I. Shamos. *Computational Geometry.* PhD thesis, Department of Computer Science, Yale University, (USA), 1978.

[24] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Sysyems Theory*, 10:99–127, 1977.