ENS Lyon

Formalisation of the Delaunay Triangulation

Clément Sartori

09/06/2017

Advisor ..... Yves Bertot

Abstract

This report is the result of my internship at INRIA Sophia-Antipolis in the Marelle team, for the validation of the second year of my master's degree in Computer Science. During this internship I formalised Delaunay Triangulations in Coq. My code, although incomplete, can be found here : https://github.com/Nemeras/StageDelaunay.

Contents

Figure 1: This is a triangulation

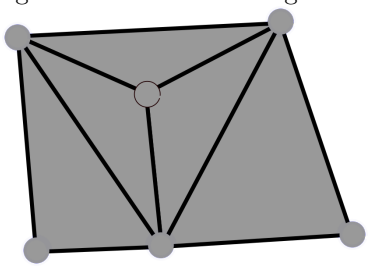
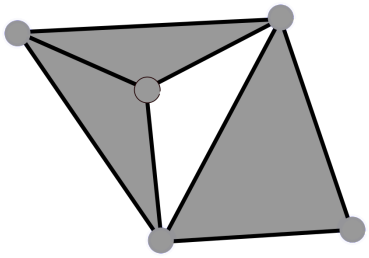


Figure 2: This is not a triangulation



1 Introduction

I did my second year of master's degree internship under the supervision of Yves Bertot at the INRIA Sophia-Antipolis in the Marelle team. The goal of this internship was to formalise Delaunay triangulations in the proof assistant Coq.

The goal of this internship was to formalise the Delaunay triangulations in the Coq proof assistant, that is, to try and find a minimal number of geometrical properties that had to be satisfied in order to get Delaunay triangulations. One of the goal of the internship was to stay as close as possible to Knuth's paradigm in [Knu92] with the CC system, by describing the biggest number of geometrical properties using only combinatorial properties such as the orientation predicate "is left of".

1.1 Triangulations and Applications

In geometry, a triangulation T of an object X of $\mathbb{R}^d$ is the partition of this object into simplices such that :

- the intersection of two simplices is either empty or a face of those simplices
- every bounded set of $\mathbb{R}^d$ cuts a finite number of simplices of T
- the union of the simplices of T is X itself.

In the plane (where we stayed during this internship), the triangulation of a set of points is the partition of this set of points into triangles.

More particularly, during this internship, we were interested in Delaunay triangulations : in the plane, a triangulation T of a set P of points is said to be a Delaunay triangulation if no point of P is in the circumcircle of a triangle of T.

Figure 4 for instance, is not a Delaunay triangulation.

An example of domain where Delaunay triangulations are very useful is Robotics. Indeed, one of the interesting byproducts of Delaunay triangulations is the Voronoi diagram (the dual graph of the Delaunay triangulation) that can be very useful, for instance, to modelise cellular coverage maps [PA08], or for path planning [GMB06].

Figure 3: This is not a triangulation

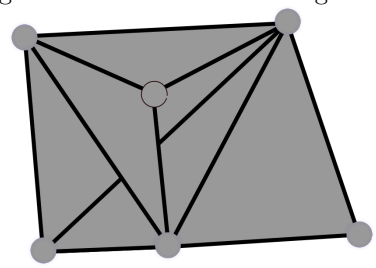
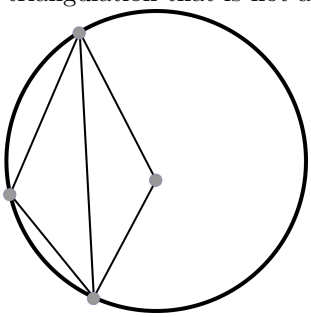


Figure 4: Instance of a triangulation that is not a Delaunay triangulation



1.2 Coq/The Mathematical Component library

Machine-checked proofs are a research domain which has more than forty years (see [dB70]). As such, the domain acquired a certain maturity; however, the main problem stays the difficulty of use of proof assistants.

I wrote my formalisation of the Delaunay triangulation in the proof assistant `Coq`, an environment and a language in which you can express and prove mathematical assertions and which automatically checks them. The advantage of `Coq` (and proof assistants in general) is that, as long as you trust `Coq`'s kernel, you can trust any `Coq` proof. The choice of `Coq` was very simple : `Coq` is the most accomplished proof assistant in my opinion, and has the enormous advantage of having access to the `Mathematical Component` library.

To make proofs simpler, I used the Mathematical Component library and the extension of Coq's tactic language, SSReflect. The goal of the Mathematical Component library project is to create a library of mathematical results, so that those result are easily reusable in Coq. Examples of successful application of the Mathematical Component library and SSReflect are the proof of the four-color theorem [Gon08] and of the odd order theorem [GAA+13].

2 The Algorithm

The context of the algorithm studied is, given a set of points in a plane, how to compute the Delaunay triangulation of this set of points. The algorithm studied is an incremental algorithm in two parts : given an existing triangulation and a point, the algorithm consists in

1. Adding the point to the existing triangulation
2. Flipping the illegal edges to obtain a Delaunay triangulation.

2.1 Adding points

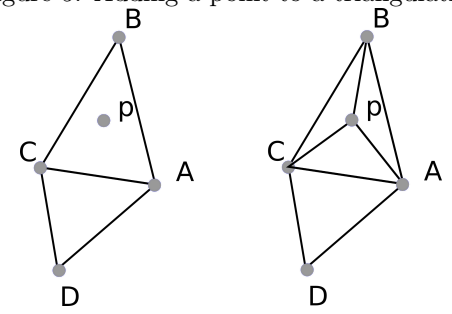
In itself, adding the point to the existing triangulation is not a trivial problem, depending on whether the new point is in the convex hull of the existing points or not. In the litterature ([SD97]), for incremental algorithm, it is generally considered that the sites studied are inclosed within large triangles, which

basically means that every new point will be in the convex hull of the previous ones. This was considered to be the case here. If this is not the case, then there is the need of an incremental algorithm to create convex hulls. The formalisation of such an algorithm has been done by David Pichardie in [PB01].

Thus, adding a new point P to a triangulation T, when the new point is in the convex hull of the triangulation, is a two step process :

1. The first step is to find the triangle t of the triangulation in which P is
2. The second step is to replace t by three new triangles.

Figure 5: Adding a point to a triangulation

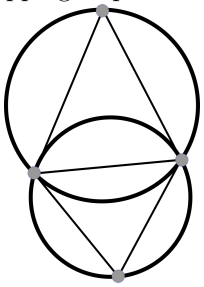


2.2 Flipping edges

It is possible , from any triangulation of a set of point, to obtain a Delaunay triangulation by repeating the same step : for each pair of triangles that do not meet the Delaunay condition, flip the common edge of the triangles.

For instance, while Figure 4 was not a Delaunay triangulation, the result of an edge-flipping step, Figure 6, however, is a Delaunay triangulation

Figure 6: Result of an edge-flipping step : example of Delaunay triangulation



This makes it possible to create a "flipping" algorithm : for any set of points, it is possible to obtain the Delaunay triangulation of this set of point by creating a random triangulation then applying this flipping step as many times as necessary.

3 Formalisation of the Delaunay triangulation

The goal of the formalisation is to stay as much as possible at a level where geometrical properties, predicates, etc, speak about mathematical concepts (for instance, points) as concepts (for instance, a point being a point versus a point being a pair of coordinates). Basically, the formalisation describes what properties every implementation of the concept should satisfy.

Here we describe a formalisation of the Delaunay triangulation. Any implementation should contain a type P for the points, a type T for the triangles, a type E for the edges, a type $\mathbb{R}$ for the coordinates of the

points (a real integral domain where all elements are positive or negative). Then , any implementation should contain basic functions :

- A function `vertices_to_edge` to create an edge from two points
- A function `vertices_to_triangles` to create a triangle from three points
- A function `vertex` that returns the first, second or third vertex of a triangle
- A function `xCoord` that returns the X coordinate of a point
- A function `yCoord` that returns the Y coordinate of a point

Here, a choice has been made so that proofs are simpler : the order of the vertices of a triangle is important. This means that the triangle *abc* won't be the same triangle than the triangle *bca*.

3.1 Geometrical predicates

Then, any implementation should also contain some geometrical predicates :

1. `is_left_of` and `is_left_or_on_line` *a b c* which test if the point *c* is (strictly) left of the points *a*,*b* e.g. if *a*,*b*,*c* form a (strictly) well-oriented (counter-clockwise) triangle.
2. `in_circle` *p a b c* which tests if a point *p* is in the circumcircle of the triangle described by *a*,*b*,*c*.
3. `in_triangle` *t p* and `in_triangle_w_edges` *t p* which test if p is (strictly) in t.
4. `is_on_line` which tests is 3 points are aligned.

Those predicates easily come from another predicate : `oriented_surface` *a b c* which computes the oriented surface described by the three points *a*,*b*,*c* :

- `is_left_of a b c` = (`oriented_surface a b c > 0`)
- `is_left_or_on_line a b c` = (`oriented_surface a b c` $\geq$ `0`)
- `is_on_line a b c` = `oriented_surface a b c = 0`
- `in_triangle t p` = `is_left_of (vertex1 t) (vertex2 t) p` $\wedge$ `is_left_of p (vertex2 t) (vertex3 t)` $\wedge$ `is_left_of (vertex1 t) p (vertex3 t)`
- `in_triangle_w_edges t p` = `is_left_or_on_line (vertex1 t) (vertex2 t) p` $\wedge$ `is_left_or_on_line p (vertex2 t) (vertex3 t)` $\wedge$ `is_left_or_on_line (vertex1 t) p (vertex3 t)`

.

The `is_left_of` predicate is inspired by Knuth's idea of CC systems [Knu92]. However, unlike him, we don't considere that our points are necessarily in general position. Thus, the opposite of `is_left_of a b c` is not `is_left_of b a c` but `is_left_or_on_line b a c`. A broader explanation of what happens to Knuth's axioms when degenerate cases (aligned points) can happen can be found in [PB01].

3.2 Hypotheses

We put assumptions (that have to be proven true for any implementation) on the implementations of the predicates/functions. Some are easy :

- `Hypothesis oriented_surface_xx :` $\forall$ `x y, oriented_surface x x y` = 0.
- `Hypothesis oriented_surface_circular` $\forall$ `a b c, oriented_surface a b c = oriented_surface c a b`.
- `Hypothesis oriented_surface_change :`
  $\forall$ `a b c, oriented_surface a b c = - oriented_surface b a c`.

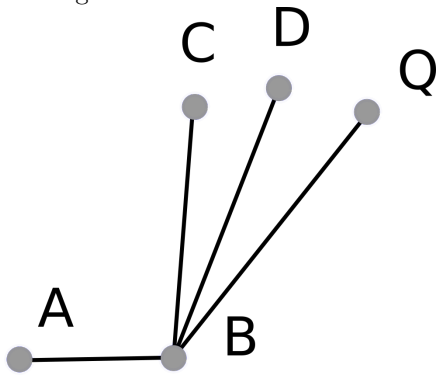
- `Hypothesis vertices_to_edge_sym : ∀ a b, vertices_to_edge a b = vertices_to_edge b a.`
- `Hypothesis vertices_to_triangle_oriented :`
  `∀ a b c, oriented_triangle (vertices_to_triangle a b c).`
  `oriented_triangle t` meaning that the triangle t is either well-oriented or flat.
- `Hypothesis is_on_line_trans :`
  `∀ a b c d, a != b → is_on_line a b c → is_on_line a b d → is_on_line a c d.`
- `Hypothesis vertices_to_triangle_correct2 : ∀ p1 p2 p3, ∀ t, (t = vertices_to_triangle p1 p2 p3) -> ((p1 ∈ vertex_set t) ∧ (p2 ∈ vertex_set t) ∧ (p3 ∈ vertex_set t)).`

There are also assumptions on the functions that are made to link functions between each other : for instance, to explain what is the edge set of a strictly well-oriented triangle created thanks to vertices-to-triangle :
`Hypothesis edges_set_vertices_to_triangle:`
`∀ a b c, is_left_of a b c → edges_set (vertices_to_triangle a b c) =`
`[ fset (vertices_to_edge a b); (vertices_to_edge a c); (vertices_to_edge b c) ].`

Some assumptions are harder, because they are generalizations of Knuth's axioms or just come from harder geometrical facts. For instance:

- `Hypothesis is_left_of_trans : ∀ a b c d q,`
  `is_left_of a b c →`
  `is_left_of a b d →`
  `is_left_of a b q →`
  `is_left_of q b d →`
  `is_left_of d b c →`
  `is_left_of q b c`
  .

Figure 7: Knuth's 5^{th} Axiom



The idea to understand this hypothesis is that we're sorting `c d` and `q` in a semi-plane defined by `a` and `b`.

- `Hypothesis Axiom4 : ∀ a b c d, is_left_of a b d →`
  `is_left_of b c d →`
  `is_left_of c a d →`
  `is_left_of a b c →`
  .
- `Hypothesis on_line_on_edge` (Figure 9) :
  `∀ a b c, is_left_of a b c → ∀ q, is_on_line a c q → is_left_of a b q → is_left_of b c q → on_edge (vertices_to_edge a c) q.`
- `Hypothesis on_edge_on_line` (Figure 9):
  `∀ a b c, is_left_of a b c → ∀ q, on_edge (vertices_to_edge a c) q → is_on_line a c q ∧ is_left_of a b q ∧ is_left_of b c q.`

Figure 8: Knuth's 4^{th} Axiom

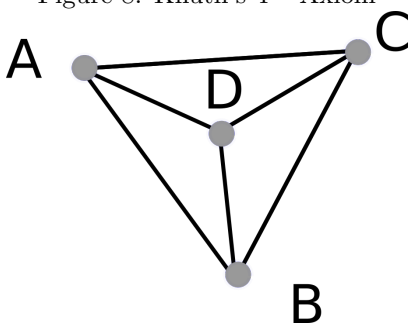
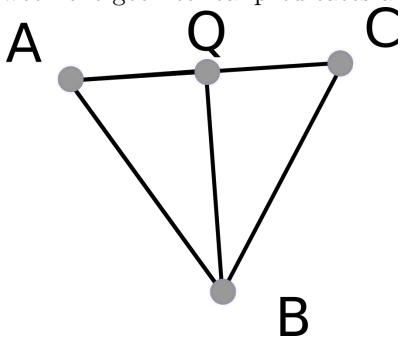
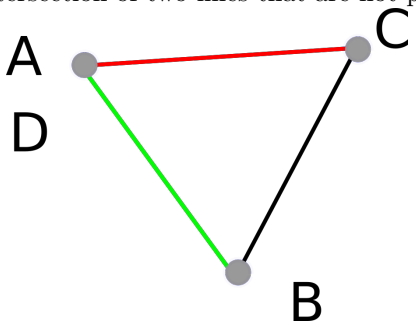


Figure 9: Link between the geometrical predicates and being on an edge



- `Hypothesis intersection_of_lines` (Figure 10) : $\forall$ `a b c d, is_left_of a b c -> is_on_line a b d -> is_on_line a c d -> d = a.`

Figure 10: the intersection of two lines that are not parallel is a point !



3.3 Geometrical Properties

Then, we define a triangulation Tr on a data set D as a finite set of triangles with points of D as vertices which satisfies some properties :

1. `covers_hull Tr D` : the whole convex hull of D is covered by the triangles of Tr
2. `covers_vertices Tr D` : the points of D are exactly the vertices of the triangle of Tr.
3. `no_cover_intersection` : there is no point which is (strictly) in two different triangles of the triangulation. This is written by saying that if two triangles of the triangulation have a common point, then the two triangles are actually the same.

4. `no_point_on_segment` : no point can be a vertex of a triangle, and on the edge of another triangle (remark : the vertex of a triangle is considered to not be on the edge of this triangle). This is written as: if a vertex of a triangle of the triangulation is in another triangle of the triangulation, then it is a vertex of this triangle.

5. `triangle_3_vertices Tr` : every triangle of the triangulation has 3 vertices.

6. `triangle_nempty Tr` : no triangle of Tr is empty (no flat triangle).

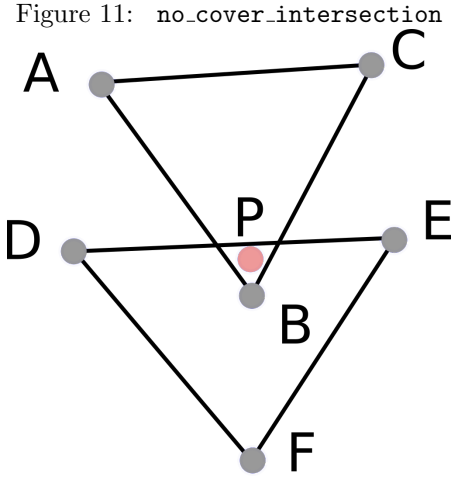
7. `oriented_triangle_triangulation Tr` : all triangle of the triangulation are well-oriented.

   Remark : by hypothesis, we force `vertices_to_triangle` to return a well-oriented triangle so this hypothesis is extremely simple to prove in general.

The two first properties are here to express that the finite set of triangles Tr is indeed a triangulation of the whole data set D, and covers its convex hull with triangles. Then, the two next properties are here to avoid problems with how the triangles are positioned :

1. `no_cover_intersection` is here to avoid problems like in Figure 11 where two triangles `ABC` and `DEF` aren't well positioned; there exists a point `P` that is in both `ABC` and `DEF`.

Figure 11: `no_cover_intersection`



2. `no_point_on_segment` is here to avoid problems like in Figure 12, where the vertice `B` of the triangle `ABC` is on the edge `DE` of the triangle `DEF`.

In both case, the properties are here to assure the fact that the first part of the definition of a triangulation is respected : the intersection of two simplices should either be empty or a face of those simplices.

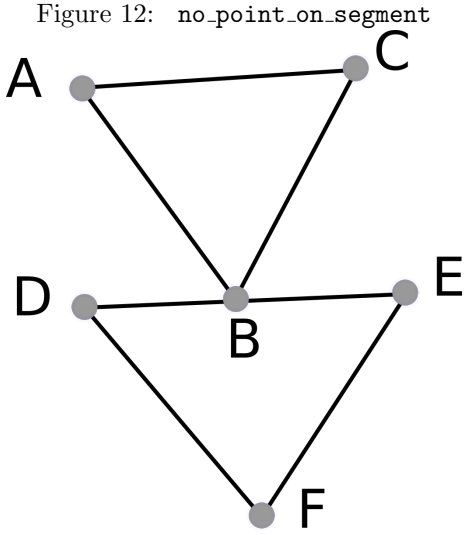
Finally, the two last properties are here to avoid degenerate cases with the triangles of the triangulation (triangulation shouldn't contain degenerate triangles). It is to be noted that the second property implies the first.

3.4 Convex Hulls

The definition of what is a triangulation in dimension 2 requires first to define what is a convex hull of a point set D. Two different approachs exist :

1. The first possibility, used by Knuth, is to define the convex hull as a cyclic sequence S : `(S1; S2; ...; Sn; S1)` of elements of D such that every point of D is "encompassed" within S, with `P` being "encompassed" within S iff `is_left_or_on_line Si Si+1 P` is true. It's a more geometrical definition.

Figure 12: `no_point_on_segment`



2. The second possibility is via the notion of barycenter : it is the smallest subset `H` of `D` such that every point of `D` is a barycenter of the points of `H`.

It's the second approach that was used here. It has two main benefits :

1. First, it is independant of the dimension, whereas the first possibility only works in dimension 2.
2. It is independant to the definition of the geometrical predicates, and those can change (for instance between Knuth's case where points are in general positions and where they are not).

Also, it is very nicely linked with `oriented_surface` in dimension 2. Carathéodory's theorem for Convex hulls says :

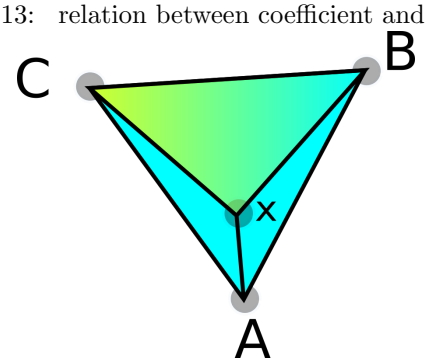
Theorem 1. *If a point of* $\mathbb{R}^d$ *lies in the convex hull of* `D`*, then it is a barycenter of* $d+1$ *points of* `D`*.*

In dimension 2, this means that every point in the convex hull of `D` is a barycenter of three points of `D` (ie, in a triangle with vertices some points of `D`). Moreover, there is a nice thing about the coefficients : if x is in ABC then we have :

$$x = x_A A + x_B B + x_C C$$

with : $x_A = \frac{\texttt{oriented\_surface B C x}}{\texttt{oriented\_surface A B C}}$, (...), which gives a nice relation between the coefficients and the surface of the triangles (Figure 13).

Figure 13: relation between coefficient and surface

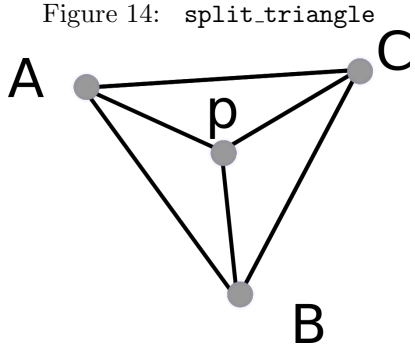


4 Theorems

4.1 Adding points

Following the algorithm described in section 2 the first step was writing a way to add points to a triangulation : given a triangle $t = ABC$, a point p, a triangulation tr, `split_triangle` tr t p returns $tr \setminus t \cup \{ABp; pBC; ApC\}$.

Figure 14: `split_triangle`



The theorem proven was :

```
Theorem triangulation_split_triangle:
∀ tr, t, p, d, d != fset0 -> p ∉ d → triangulation tr d → t ∈ tr ->
in_triangle t p → triangulation (split_triangle tr t p) (p |‘ d).
```

In english, this would be written :

Theorem 2. *Forall triangulation tr of a nonempty data set d, triangle of this triangulation t, point strictly in this triangle p , if p is not in d, then* `split_triangle tr t p` *is a triangulation for the data set* $d \cup \{p\}$.

This is proven by proving the preservation of the six properties described in the definition of a triangulation (Section 3.3) :

4.1.1 For `covers_hull`

For the proof for `covers_hull`, we prove that q is in the convex hull of d by using the fact that q is in t which is in the triangulation. Then, it comes down to proving three lemmas.

The first lemma is that if we add to a data set a point that was already in the convex hull of the data set, then the convex hull of this data set doesn't change. Even if looks quite intuitive, the proof of this lemma was actually quite hard to write in `Coq` because of complicated problems of typings within the BigOps part of `MathComp`.

The second lemma is that a point in a triangle of the triangulation is in the convex hull of the data set of the triangulation.

The third lemma is that if a point q was in the triangle deleted from the triangulation, then it's in one of the new three triangles. The proof of this lemma is actually the longest proof (for this theorem). It's a big case-based reasoning on where q is in the big triangle compared to the segments Ap, Bp and Cp.

This proof could have been quite smaller, I think, if I used more the tactic `easygeo` I'll speak about in Section 6.2. However, I wrote this tactic later and then didn't have the time to redo my proofs using it.

4.1.2 For `covers_vertices`

The proof for `covers_vertices` looks also quite simple. However, it is actually quite long to do with Coq : for instance, when we say that q is the vertex of a triangle t, we write that q is in `edges_set t` and then we have to do a case-based reasoning (is q the first vertex of t or the second or the third...).

4.1.3 For `no_cover_intersection`

For the proof for `no_cover_intersection`, it was needed to prove two lemmas.Then it's a big case-based reasoning using the two lemmas, again.

The first one is stating that if a triangle is (strictly) in a new triangle, then it was in t. Its proof is quite simple and mainly uses Knuth's 5^{th} axiom, the "transitivity" (see Section 3.2).

The second one is stating that a point is in (large definition) one of the new three triangles if and only if it was in (large definition) the triangle deleted from the triangulation. The proof of this lemma was actually very hard and long. It's a case-based reasoning on the position of the point in the big triangle. A trick was used to do the proof : I was certain that some hypotheses were not consistent (and thus I could do a proof by contradiction in some cases), but I couldn't find how to use Knuth's axioms to prove it. After a discussion with Yves, we used an automated tool made by some previous students that tested the consistency of a list of hypotheses (of the type `is_left_of A B C` ...) using Knuth's axioms, and finally found how to finish my proof from analysing the results.

The proof of the lemma itself was much simpler than the (already proven) reverse and mainly used Knuth's 5^{th} axiom, the "transitivity" (see Section 3.2).

4.1.4 For `no_point_on_segment`

The proof for `no_point_on_segment` also comes from a lemma, which is actually the same as another before (being in one of the three new triangles means being in t) , but using the large definition of being in a triangle, instead of the strict one.
The lemma is proven using an hypothesis (provable, but I didn't have the time to do it) : if a point is on the edge of one of the three new triangles, then it's either in t on an edge of t.

4.1.5 For `triangles_3vertices`

The proof for `triangles_3vertices` can look trivial, but, again (like for `covers_vertices`), it is not. The "difficulty" comes from the number of overlapping case-based reasonning on the vertices of the triangles. For this proof, I tried to do as much "automation" I could by treating cases together in order to avoid having very long simple proofs.

4.1.6 For `triangles_nempty`

The proof for `triangles_nempty` is extremely simple. We know that the triangles we have have all a nonzero oriented surface by hypothesis (definition of `in_triangle` t p), and we deduce from this that there exist at least a point inside them.

4.1.7 For `oriented_triangle_triangulation`

Finally, the proof for `oriented_triangle_triangulation` is extremely simple because of the fact we force, by hypothesis, any implementation of `vertices_to_triangle` to return a well-oriented triangle. Because of this, the proof is only a very simple case-based reasonning.

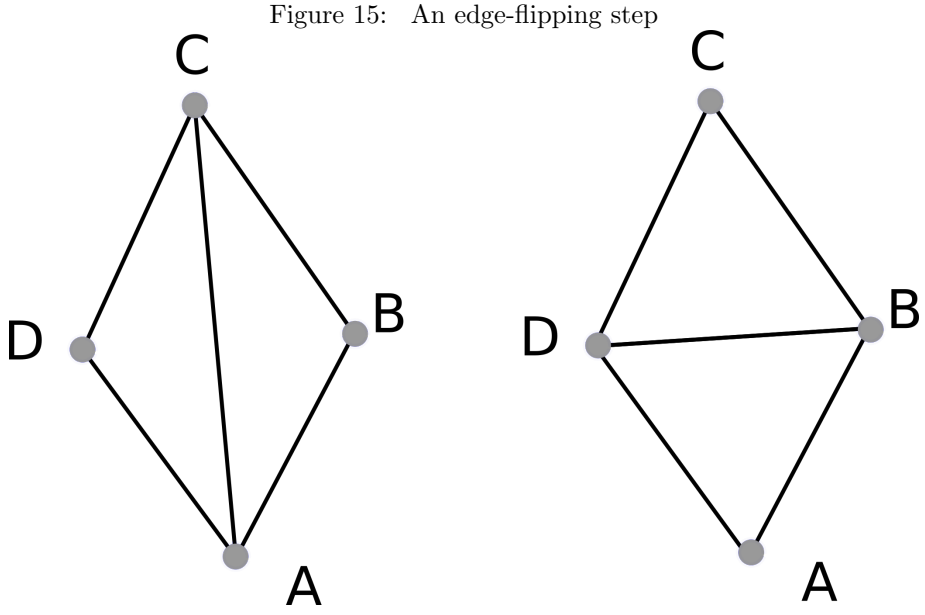
4.2 Flipping edges

The next big step was writing the edge-flipping step `flip_edge` tr $t1$ $t2$ a b c d. It returns $tr \setminus t1 \setminus t2 \cup \{abd; bcd\}$. It's meant to be called with $t1 = abc$ and $t2 = acd$.

I won't write here the theorem I proved in this report, but, basically, the idea is the same as before (prove that a triangulation in transformed into another triangulation, but with more conditions) :

- t_1 and t_2 should be triangles of a triangulation tr on a data set d.
- a,b and c should be points of the data set, vertices of t_1 and form a well oriented triangle (we have `is_left_of` a b c)
- a,c and d should be points of the data set, vertices of t_2 and form a well oriented triangle (we have `is_left_of` a c d)

Figure 15: An edge-flipping step



The way it was written helps with the proofs. It avoid writing things like $t1$ =`vertices_to_triangle` a b c which would be a problem because of the fact that the order of the vertices of a triangle is important. Again, this is proven by proving the preservation of the six properties.

4.2.1 For `covers_hull`

For `covers_hull`, the idea of the proof is quite the same as in Section 4.1.1, but much simpler. It relies on a lemma : if a point was strictly in one of the previous two triangles, then it is in one of the two new triangles.

The proof of this lemma is also a big case-based reasoning, but it is better automated this time because of the experience I gained between proving all the lemmas necessary for the first theorem.

4.2.2 For `covers_vertices`

The idea of the proof for `covers_vertices` is the same than in Section 4.1.2. It's a big case-based reasoning that didn't need the proof of another lemma.

4.2.3 For `no_cover_intersection`

The proof for `no_cover_intersection` needed the proof of a two new lemmas.

The first one is : if q was (strictly) in a deleted triangle then it is either in one of the new triangles or on the common edge of those triangles.

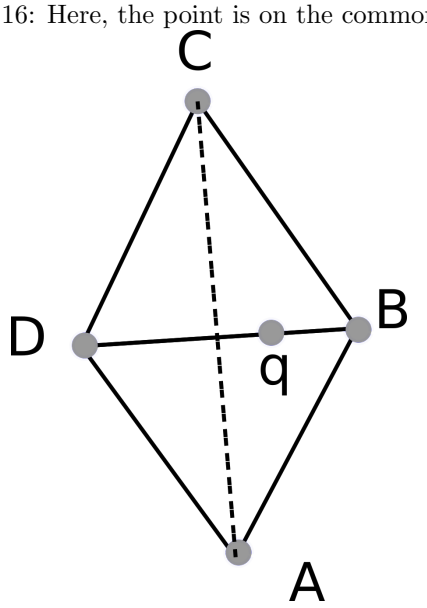
The second lemma proven is : if a, b and c are in the vertex set of a triangle t, if t is well oriented then forall p, p is in `vertices_to_triangle` a b c if and only if p is in t. This looks simple, but actually made me realize a theorem was missing in Cyril Cohen's finmap library , which was quickly added.

The rest of the proof is, as always, just a case-based reasoning.

4.2.4 For `no_point_on_segment`

The proof for `no_point_on_segment` is, as usual, a case-based reasoning which needed a new lemma: if a point was in (large definition) a triangle that was deleted from the triangulation, then it's in (large definition) one of the two new triangles.

Figure 16: Here, the point is on the common edge



4.2.5 For `triangles_3vertices`, `triangles_nempty` and `oriented_triangle_triangulation`

The proof for `triangles_3vertices` is a simple case-based reasoning that I succeed in automating quite a lot, which made it short.

The proof for `triangles_nempty` was also very short since it was direct from the hypotheses (we know that the oriented surface of the new triangles are positive by hypothesis so, directly, we know that there exist some points in them).

Finally, the proof for `oriented_triangle_triangulation` is very direct for the same reason as in Section 4.1.7

Proving the conservation of these properties was the main part of the internship. While they looked very simple to prove by hand, proving them in `Coq` was a way more difficult task.

Indeed, `Coq` stops us from using a big part of the mental shortcuts we use to shorten proofs in our head (and it's the good aspect about `Coq`, because it's one of the things that can induce false proofs on paper). One of the most important thing to remember, when proving things with `Coq` and the `Mathematical Component` library, is that it's usually best to try and make a plan for the proof, on paper, and then try to follow this plan. If something is wrong during the proof and it's not possible to follow the plan, it's that either there is a problem with the proof, or that there is something missing in the library. It's actually very easy to get trapped by `Coq` and try to do the proof by following `Coq` instead of a proof plan!

5 Instantiation of the model

The next step of the work was to do an instantiation of the model. For the types :

- A point is a vector of size 2 of reals : `Definition P :=` $'rV[R]_2$`.`
- A triangle is a vector of size 3 of points : `Definition T :=` $'rV[P]_3$`.`
- An edge is a vector of size 2 of points : `Definition E :=` $'rV[P]_2$`.`

For the implementation of the functions, with ordnx being the $x+1th$ element of the Ordinal n (a well ordered set with n elements)::

- `Definition xCoord (p : P) := p ord10 ord20` (in `MathComp`, a vector of elements of $\mathbb{R}$ of size 2 is a function of the Ordinal 1 in the Ordinal 2 in $\mathbb{R}$).

- `Definition yCoord (p : P) := p ord10 ord21.`
- `Definition vertex (t : T) := t ord10` which gives a function of the Ordinal 3 in P.
- `oriented_surface a b c` can be defined as the determinant of the matrix :

$$\begin{matrix} 1 & \texttt{xCoord a} & \texttt{yCoord a} \\ 1 & \texttt{xCoord b} & \texttt{yCoord b} \\ 1 & \texttt{xCoord c} & \texttt{yCoord c} \end{matrix}$$

  This makes it so that most proofs on `oriented_surface` are very nice to do : one only needs to expand the determinant and then use automatic tactics on rings. The only difficulty was to add the ring on which we worked ($\mathbb{R}$ with the usual operations) so that the automatic tactic `ring` worked.

- The predicate `in_circle p a b c`, testing if p is in the circumcircle of the triangle `abc` can also be defined as testing if the determinant of this matrix is negative :

$$\begin{matrix} 1 & \texttt{xCoord p} & \texttt{yCoord p} & \texttt{(xCoord p)}^2 \texttt{ + (yCoord p)}^2 \\ 1 & \texttt{xCoord a} & \texttt{yCoord a} & \texttt{(xCoord a)}^2 \texttt{ + (yCoord a)}^2 \\ 1 & \texttt{xCoord b} & \texttt{yCoord b} & \texttt{(xCoord b)}^2 \texttt{ + (yCoord b)}^2 \\ 1 & \texttt{xCoord c} & \texttt{yCoord c} & \texttt{(xCoord c)}^2 \texttt{ + (yCoord c)}^2 \end{matrix}$$

- The implementations of `vertices_to_triangle` and `vertices_to_edge` were not trivial to do.

  First, for `vertices_to_triangle a b c`, since we require in the formalisation that any triangle obtained by `vertices_to_triangle` should be well-oriented, we have to check if *abc* form a well-oriented triangle. If yes, we return *abc*, else we return *bac* which will be oriented by definition of the `is_left_of` predicate.

  Then, for `vertices_to_edge`, we require, for every `a` and `b`, that `vertices_to_edge a b` = `vertices_to_edge b a`. To do that, we define a lexicographic order on the points. Then, if `a` $\leq$ `b` we return `ab`, else we return `ba`.

The proofs can be grouped in several types :

1. The proofs on the geometrical predicates (for instance, the fact that `oriented_surface a b c` = `oriented_surface b c a` = - `oriented_surface b a c`) were quite easy because of the fact that proofs on determinants only require to expand the determinants and then do some calculus, which is automatic with the tactic `ring`. I reused some work done by a previous intern, Wassim Haffaf, in this part (the way he expanded 3 x 3 determinants for instance).

2. The proofs requiring more geometrical facts (a good example of this is the proof of the hypothesis `on_edge_on_line`, see Section 3.2).

   Those were way harder and I didn't have the time to finish them all. `on_edge_on_line` and `on_line_on_edge` are two examples of proofs I tried to do but couldn't finish because I was running out of time and the proofs were very difficult.

3. The proofs on the functions (for instance, the proofs on `edge_set` and `vertices_to_triangle`). Those were generally simpler than the previous type of proofs (the difficulty coming more from the use of `Coq` and how to do case-based reasoning in an intelligent/automatic way, more than from geometrical problems). I still couldn't finish them all , but I managed to do a good part of them.

4. The proofs of Knuth's axioms.

   I didn't have time to begin proving these at all. However, some (most ?) of them could probably be recovered from previous work done by other students (for instance, the work on convex hulls done by David Pichardie (see [PB01]).

The instanciation of the model was a very interesting part of the internship. When I tried to do it, I realised a lot of problems about the first version of my formalisation of the Delaunay triangulation : some hypothesis were not precise enough, some were even false (for instance, a problem I had was that it's sometimes hard to realise that triangles can be flat when thinking about hypotheses), and I had to rethink a lot of hypotheses and proofs.

6 Conclusion

6.1 Possible improvements

There are some clear possible paths of improvement for my work. First, all the proofs aren't finished. Then, the next step for me would have been to implement, in `Coq`, from the different steps. The biggest difficulty would have been to prove the termination of the algorithm which transform a random triangulation into a Delaunay triangulation: the way it is done is by saying that, during the algorithm, the lifted image of the triangulation on the unit paraboloid is a quantity that is decreasing. A more detailed explanation of this fact can be found here : https://www.ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter 6.pdf.

Then, when this is finished, there are some other possibilities to extend my work :

1. First, a good way to extend this work could be to generalize the formalisation of Delaunay Triangulations to algorithm that work in higher dimension. In fact, the method I formalised, the incremental construction using flips, can be generalized in dimension 3 or highed ([BCKO08]). However, the complexity can be exponential in the dimension (as shown in [ES92])

2. Another idea could be to try and formalise other Delaunay Triangulation algorithm. It is shown in [SD95] that the better method to compute Delaunay Triangulation is a Divide and Conquer algorithm presented in [GS85]. An implementation of this algorithm is presented here : `http://www.geom.uiuc.edu/~samuelp/del_project.html`

6.2 Perspectives

There are also some things that could be changed about the work I've done.

First, my proficiency with `Coq` and, especially, `SSReflect` changed during the internship. I already followed, before it, a one week course on `SSReflect` and I've done an internship in IRISA,Rennes where I've used `Coq` to do proofs on programs. However, this was not comparable to how I had to use `Coq` during this internship. As such, at the beginning of the internship, I've taken lots of time to prove things that would have been very easy for me at the end of the internship.

Moreover, the proofs I've done at the end of the internship are "cleaner" than the one I've done at the beginning at the internship, taking advantage from the use of the syntax of `SSReflect` and of tricks like the use of the tactic `try`.

Also, near the end of the internship, I wrote a little (very simple) tactic, which I named `easygeo` . I realised, maybe a little too late, that I spent too much time thinking on how to use my hypotheses. Indeed, there were lots of cases where I wanted to use a theorem, with a pack of hypotheses. But, instead of having, for instance, `is_left_of b c a`, I had, as an hypothesis, `is_left_of a b c` (which we know is equivalent). Thus, before applying the theorem I wanted to apply, I had to spend time rewriting my hypothesis with lemmas. The goal of this tactis, `easygeo`, was to basically use brute force to solve some cases: for instance, if I wanted to prove `is_left_of a b c` and I had, `is_left_of b c a`, the tactic would solve instantly the goal. Since I wrote this tactic very late in the internship, this tactic isn't (much) used in most of the proofs.

7 Acknowledgements

I would like to thank a lot my supervisor, Yves Bertot for all the time he spent helping me during this internship, and all the good references he gave to help me. I would also like to acknowledge Cyril Cohen, who helped me a lot during the internship and Laurence Rideau, who helped me find this internship. I also want to thank all the other members of the MARELLE team at INRIA Sophia-Antipolis for their friendly welcome and help.

I would also like to acknowledge Pierre Alliez, from the TITANE team at INRIA Sophia-Antipolis for the nice and helpful discussion he granted us which conforted us in the direction we were taking in this research.

Bibliography

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[dB70] N. G. de Bruijn. *The mathematical language AUTOMATH, its usage, and some of its extensions*, pages 29–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 1970.

[ES92] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, SCG '92, pages 43–52, New York, NY, USA, 1992. ACM.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[GMB06] S. Garrido, L. Moreno, and D. Blanco. Voronoi diagram and fast marching applied to path planning. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 3049–3054, May 2006.

[Gon08] Georges Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, December 2008.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, April 1985.

[Knu92] Donald E. Knuth. *Axioms and Hulls*, volume 606 of *Lecture Notes in Computer Science*. Springer, 1992.

[PA08] Jose N Portela and Marcelo S Alencar. Cellular coverage map as a voronoi diagram. *Journal of Communication and Information Systems*, 23, 2008.

[PB01] David Pichardie and Yves Bertot. Formalizing Convex Hulls Algorithms. In *Proc. of 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, number 2152 in Lecture Notes in Computer Science, pages 346–361. Springer-Verlag, 2001.

[SD95] Peter Su and Robert L. Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, SCG '95, pages 61–70, New York, NY, USA, 1995. ACM.

[SD97] Peter Su and Robert L. Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. *Computational Geometry*, 7(5):361 – 385, 1997.