Implementation of Delaunay Triangulation

Hussein Houdrouge
houdrouge.hussein@gmail.com

April 2022

1 Introduction

In this project, we aim at implementing Delaunay triangulation. This type of triangulation plays a central role in Computational Geometry, Computer Graphics, Computational Physics, and Geographic Information System. For instance, Delaunay triangulation is used in height interpolation to model terrains. Furthermore, its dual, Voronoi Diagram, can be deduced from the triangulation which has several applications.

In the following sections, we will start by defining Delaunay triangulation. Then, we will briefly describe the algorithm we used in the implementation, randomized incremental construction. More precisely, our implementation is based on the algorithm described in [dBCvKO08]. After, we will describe the Data Structure used to maintain the triangulation. Finally, we will show some of the results.

2 Preliminaries

To define Delaunay triangulation, we begin by introducing the following definitions.

Definition 2.1 (Planar Graph)**.** A graph $G = (V, E)$ is said to be planar if it can be drawn in the plane, $\mathbb{R}^2$, such that the edges intersects only at their endpoints.

Definition 2.2 (Maximal Planar Graph)**.** A maximal planar graph is a planar graph that adding an extra edge breaks its planarity i.e. it will introduce an intersection.

Definition 2.3 (point set triangulation)**.** Given a set of points $P \subset \mathbb{R}^2$, a triangulation of P is a maximal planar graph where the vertices are the points of P and edges are segments whose end points are in P.

Definition 2.4 (Delaunay Triangulation)**.** Given a set of point $P \subset \mathbb{R}^2$. The Delaunay triangulation of P is a triangulation of P such that the circumcircle of any triangle in the triangulation does not contains a point of P in its interior.

Based on the definition of Delaunay triangulation, the algorithm will be designed. The details are discussed in the next section. Furthermore, in the spirit of this definition, we can talk about legal or illegal edge as Lemma 9.4 in [dBCvKO08] states.

Lemma 2.1. *Given two triangles $s = p_ip_jp_k$ and $t = p_ip_jp_l$ that share the edge p_ip_j. Let C be the circumcircle of s. We say the edge p_ip_j is illegal if and only if the point p_l lies in the interior of C. Furthermore, if the points p_i , p_j , p_k , p_l form a convex quadrilateral and do not lie on a common circle, then exactly one of p_ip_j and p_kp_l is an illegal edge.*

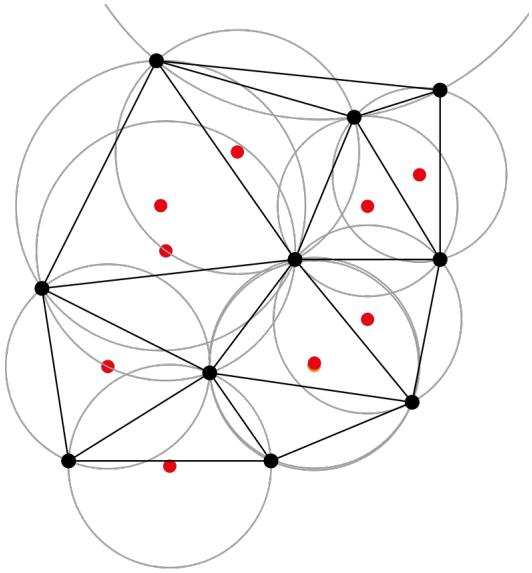


Figure 1: Delaunay triangulation where every circumcircle of a triangle t does not contain a point other than those of t. The red points are the center of the circles. (source: wikipedia)

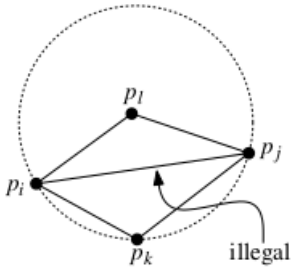


Figure 2: An example of Illegal Edge [dBCvKO08].

3 Randomised Incremental Construction

In this section, we describe briefly the algorithm Randomized Incremental Construction. For complete details and analysis, the reader can consult [dBCvKO08].

Given a set of points P of size n, the algorithm starts by computing a random permutation of P, and a special point p_0 that is lexicographically larger than any other point i.e the rightmost point with highest y-coordinates. In addition to the point p_0, the algorithm will have to extra points p_{-1} and p_{-2} that are treated symbolically. The algorithm has an access to a Data Structure D that maintains the triangulation and the operation performed on this triangulation.

After initiating all the parameters, the algorithm passes over P, and adds the point p_i in D one by one. When a point id added into D, the first step will be detecting the triangle $t = abc$ where it lies. Then, three edges is added from p_i to a, b, and c. Now, the Delaunay Algorithm has to check if the edges ab, bc, and ca remain legal. Otherwise, the edge will be flipped and testing for the legality of new edges in the new triangles will be performed. The pseudocode of the algorithm can be found in [dBCvKO08].

In the next section, we will give a detailed description of the data structure D that is described above and how it handles most of the necessary operations used to compute the Delaunay triangulation.

4 Triangulation Data Structure

The crux of the Delaunay Algorithm, described above, lies in the implementation of this data structure. Basically, this data structure is a collection of triangles that stores some parent-child relationships in addition to the adjacency relationships. The building blog of the data structure is the triangle that has the following specifications.

```
Triangle() {
 vertices = {v_0, v_ccw(0), v_cw(0)};
 opposites = {t_0, t_ccw(0), t_cw(0)};
 children = {};
}
```

The triangle structure has an array of vertices such that if a vertex v_i is stored at index i, the counter clockwise vertex to v_i is stored at index $ccw(i)$ and a clockwise vertex is stored at index $cw(i)$. The ccw and cw are functions of addition by 1 and 2 modulo 3. In addition, for the opposite triangle to a vertex v_i, the pointer $opposites[i]$ points to it. Finally, the children of a triangles are the triangles created after adding a points in its interior or the triangle resulted from flipping one of its edges. Figure 3 shows the structure of a triangle.

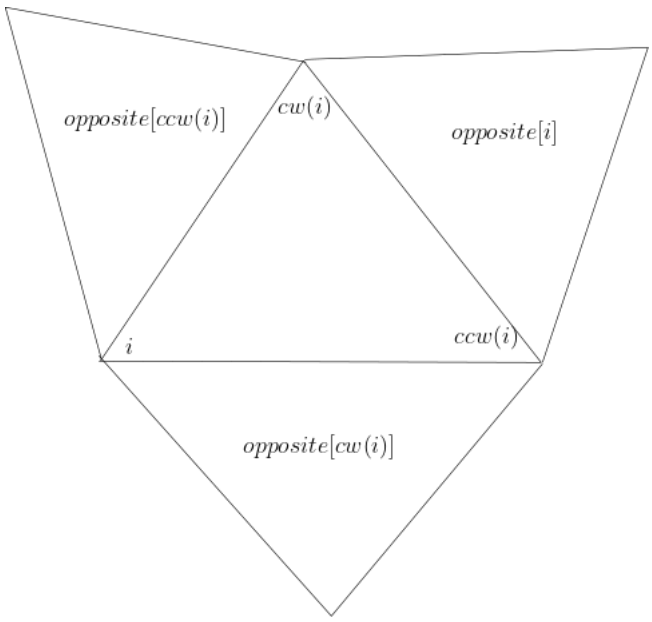


Figure 3: The structure of a triangle.

The triangulation data structure contains a root of type triangle. It will be the source of a directed acyclic graph. The main operations of this data structure are locate a point, add a point, and compute Delaunay. In sum, it looks as follow (By the * we mean a pointer as in $c++$).

```
Triangulation {
```

```
 Triangle* root;
 points = {};
 p_0;

 void init_root();
 Boolean is_inside(p, Triangle * t);
 Triangle* locate_point(p, root);
 void add_point(p);
 void legalize(base index, triangle *s, triangle * t);
 Boolean is_legal(Triangle *t, Triangle * s);
 void Compute_Delaunay();
}
```

Notice that the signature of these function and structure in the implementation might be different. But in this document, we are trying to keep the description as simple as possible.

Now, we will move to describe briefly how each of these functions works. The function

```
void init_root();
```

initiates the root as follow.

```
root->vertices = {-2, -1, p_0};
root->opposites = {nullptr, nullptr, nullptr};
```

It basically creates the triangle in Figure 4.

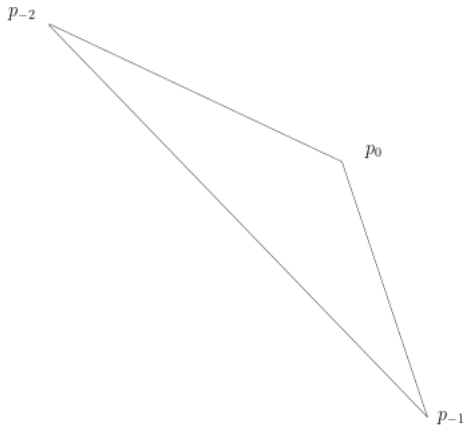


Figure 4: The initiation of the root, where the space outside the triangle is the *nullptr* and the inside is where the points will be inserted. (p_{-1} and p_{-2} are symbolic points.)

To implement *Locate_point*, you want to detect if a point is inside a triangle or not. Given a triangle $t = abc$ and a point p, we can easily detect if p inside or outside t by doing simple orientation tests. If the $det(\overrightarrow{ab}, \overrightarrow{ap}) > 0$ the p is on the left of $\overrightarrow{ab}$. Thus, for a point p to be inside a triangle t as we represent, it must satisfy the followings.

$$det(\overrightarrow{iccw(i)}, p) > 0 \text{ and } det(\overrightarrow{ccw(i)cw(i)}, p) > 0 \text{ and } det(\overrightarrow{cw(i)i}, \overrightarrow{p}) > 0$$

However, it remains to determine how to deal with triangles that contain p_{-2} and/or p_{-1}. If a triangle contains both p_{-2} and p_{-1} then we only check if p is lexicographically smaller than the third point. If the

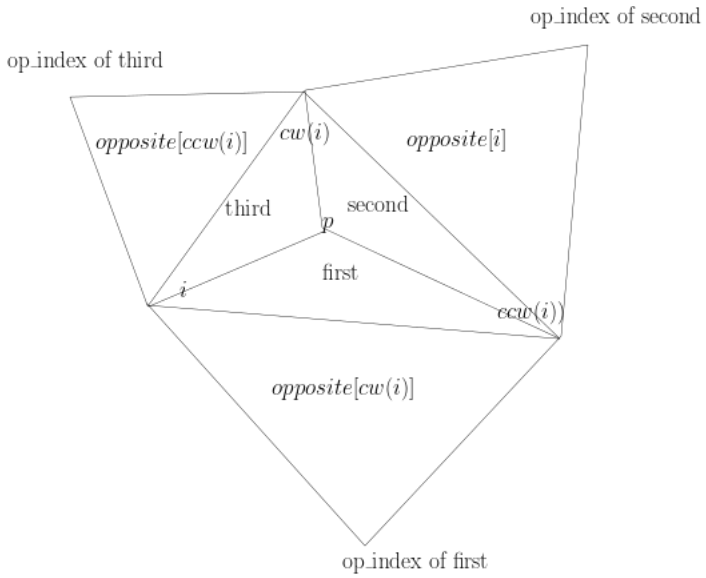


Figure 5: Adding the point p.

triangle $t = abc = ap_{-1}c$ then p belongs to t if and only if $det(\overrightarrow{ac}) > 0$ and c is lexicographically larger than p and p is lexicographically larger than a. In other words, the points of P around p_{-1} must be sorted according to their lexicographical order in the clockwise order. Lastly, if $t = p_{-2}bc$, then p is inside t if and only if $det(\overrightarrow{bc}) < 0$ and the point p is lexicographically smaller than c and larger than b. In other words, the points around p_{-2} is sorted lexicographically according to the counter clockwise order [dBCvKO08].

After specifying how we can locate a point inside a triangle, we move to describe how to locate a point.

```
locate_point(p, root);
```

Given a point and a triangle, initially the root, we check if the point is inside the triangle. If yes we recurse on every child of the triangle until reaching a triangle with no children.

Therefore, the Delaunay triangulation algorithm calls *locate_point*. Then, it subdivides the triangle t that contains p into three triangles and update the adjacency relationships. Figure 5 shows how p is added and the following pseudocode describes how the triangles are created and the adjacency relations are maintained.

```
first->vertices  = {t->vertices[0], t->vertices[ccw(i)] p};
second->vertices = {p, t->vertices[ccw(i)], t->vertices[cw(i)]};
third->vertices = {t->vertices[i], p, t->vertices[cw(i)]};

t->children = {first, second, third};

// now updated the adjacency relationship.
first->opposite[i] = second;
first->opposite[ccw(i)] = third;
first->opposite[cw(i)] = t->opposite[cw(0)];
if (t->opposite[cw(0)] != null) {
 t->opposite[cw(i)]->opposite[op_index of first] = first;
}
```

```
second->opposite[i] = t->opposite[i];
second->opposite[ccw(i)] = third;
second->opposite[cw(i)] = first;
if (t->opposite[i] != null) {
  t->opposite[i]->op[op_index of second] = second;
}

third->opposite[i] = second;
third->opposite[ccw(i)] = t->opposite[ccw(i)];
third->opposite[cw(i)] = first;
if (t->op[ccw(i)] != nullptr) {
  t->opposite[ccw(i)]->op[op_index of third] = third;
}
```

Now, after the adjacency relation is maintained, our algorithm will check for violated edges by calling $legalize(cw(i), first, t- > opposite[cw(i)])$, $legalize(i, second, t- > opposite[i])$, and $legalize(ccw(i), third, t- > opposite[ccw(i)])$.

Basically the function $legalize(p, s, t)$ checks if the opposite edge to the summit p in s and shared with the triangle t is legal or not. This check is done by checking if the quadrilateral formed by s and t is convex, and by checking if the circumcise of s contains the summit of t that is opposite to p. If the edge is legal the algorithm do nothing. Otherwise, it flips the edge and creates two new triangles that become the children of both s and t. Figure 6 shows the triangles that needs to be updated after the flip.

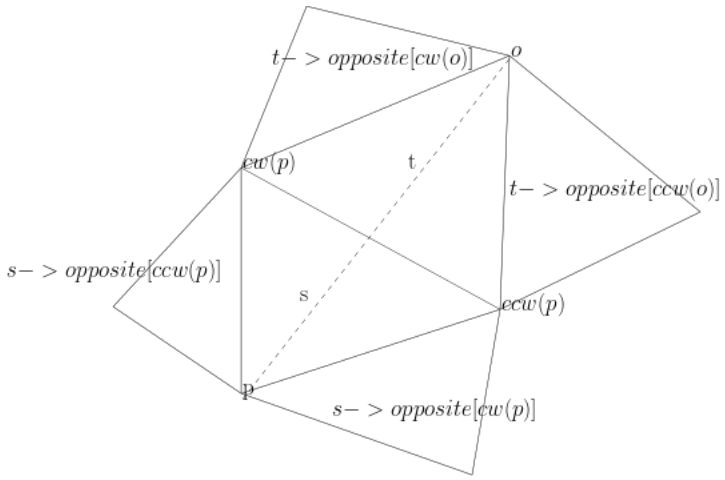


Figure 6: After flipping the edge $cw(p)ccw(p)$, one need to create two new triangles and update the adjacency relationship of the surrounding ones.

Finally, the algorithm passes over all the points in P and add them one by one to the data structure that performs the flips and the updates of the adjacency relationships.

5 Implementation and Technical Details

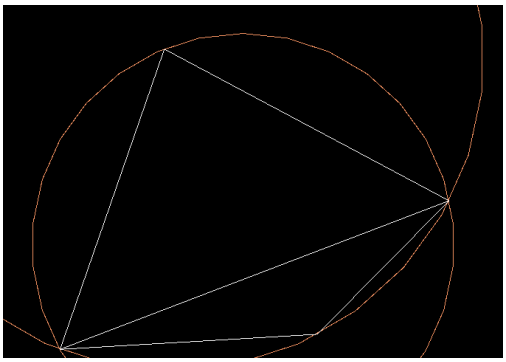
The source code of the project is available online on the following GitHub repository `https://github.com/CHoudrouge4/DelaunayAndVoronoi`. It is implemented and test on Ubuntu 20.04.4 LTS using $c++14$ and it requires $SFML$ library for visualisation. The library is found here `https://www.sfml-dev.org/download.php`. In addition, it requires cmake version >=3.10. To compile the project, open the terminal on the project folder and type the following.

```
cmake .
make
```

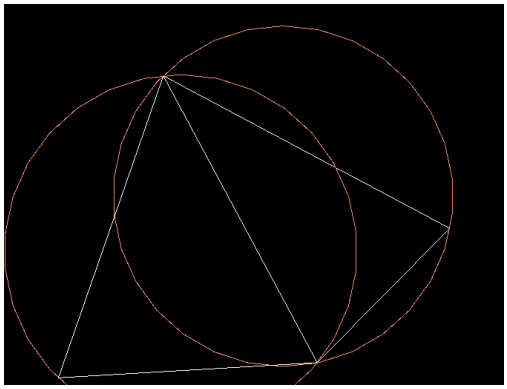
The reason behind using *c++14* instead of other version is to make use of the shared memory that avoid memory problems. However, we strongly believe that the algorithm can be re-implemented without using $c++$ pointers, only based on arrays. We suspect that the later approach will lead to a speed up in the performance. Thus, as a future objective is to test the above suggested techniques to achieve a better performance.

6 Results

In this section, we present some of the results of our implementation. Figure 7 represents a small instance of four points. It demonstrates how the algorithm performed the flip operation. Figure 8, shows the Delaunay triangulation of a slightly larger instance. We drew the circumcircle to show that all edges are legal. Unfortunately, showing larger results with the circles makes the pictures messy and cluttered. Lastly, Figure 9, shows the time performance as the number of points increases. The algorithm seems quite efficient as it is only took at most 0.5 seconds to compute the Delaunay triangulation of 40000.



(a) Non Delaunay Triangulation of Four points



(b) Delaunay Triangulation of Four points

Figure 7: Small instance of Four points.

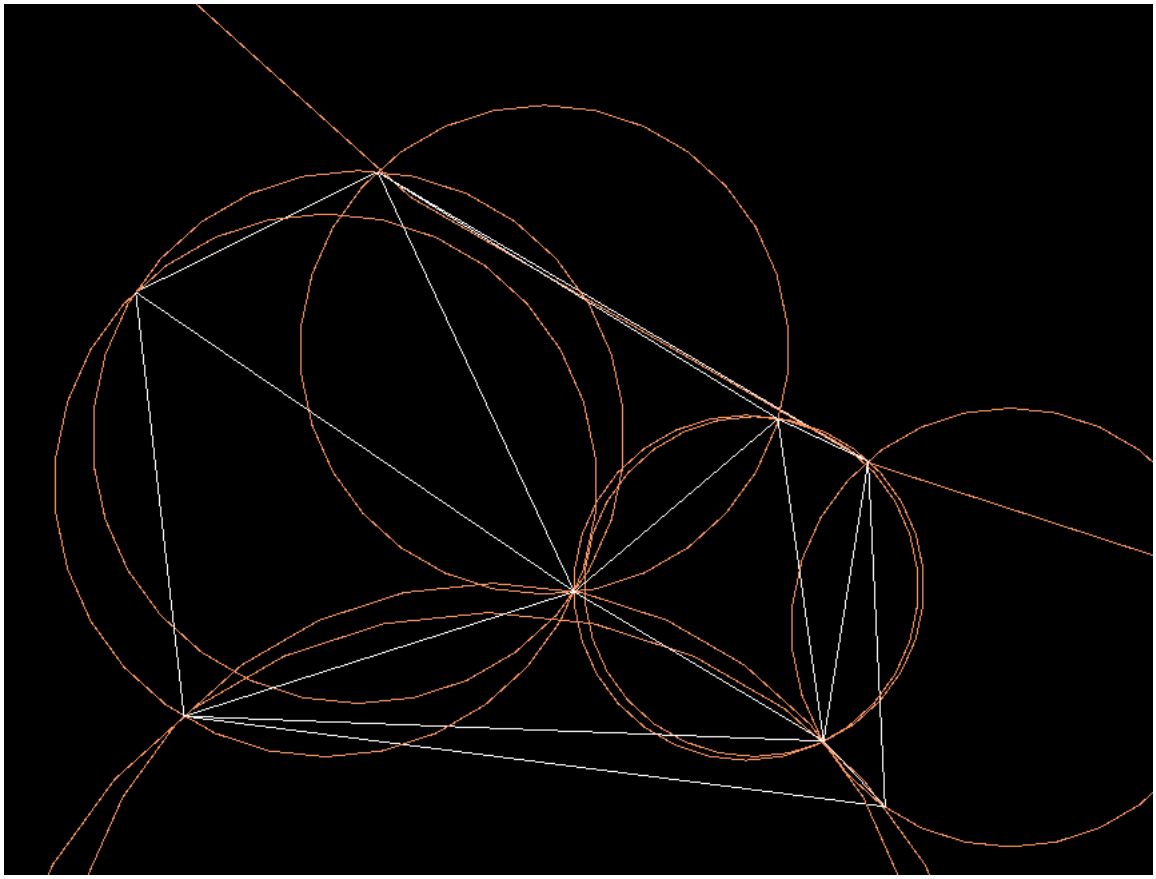


Figure 8: Example of Delaunay of eight points.

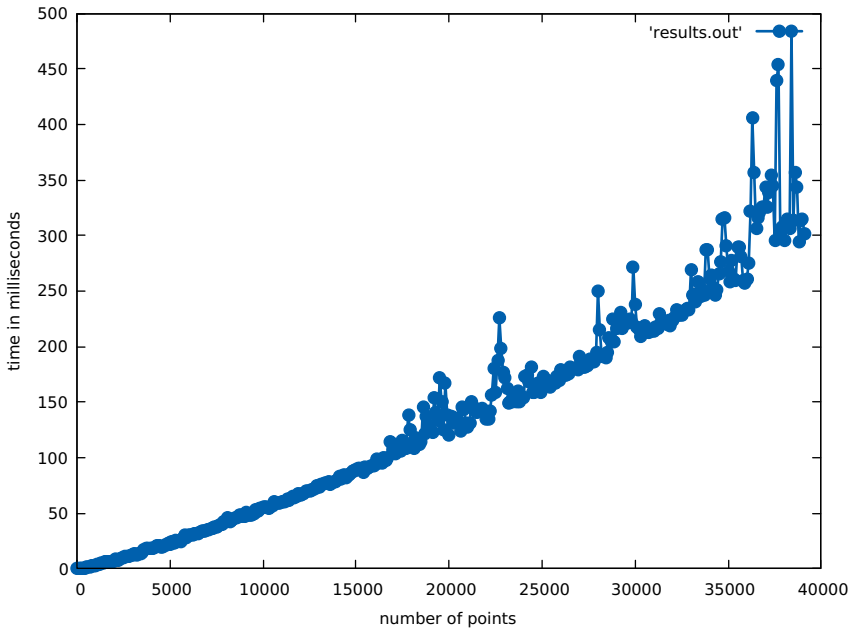


Figure 9: The performance of the algorithm in millisecond versus the number of points.

References

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer Berlin Heidelberg, 2008.