Final Report

D.T. Luttik J. R. Klarenbeek I. Lisenkov Y. Li T. Honcoop

Abstract

This report aims to demonstrate our approaches to the DBL algorithms project. The project is targeting to develop applications for the variants of the curve- and network reconstruction. The application will be evaluated by running it on a set of tests. The goal is to reconstruct the curves or a network that seems a possible solution (to a human) for the given instance. After studying several relevant papers we developed our application, tested it with real cases, and gathered the results and analyses. The results of our application were to a certain extent successful, depending on the complexity of the input.

1 Introduction

In the modern digital age, 3D-printing is becoming more important every day. With more materials that can be printed and more complex figures that can be made by 3D printers. As the technology progresses, more users will have the possibility to use 3D-printing. There is still one flaw that the 3D-printer has compared to a regular 2D-printer, a 3D-printer does not have the ability to scan 3D objects. A nice application of having a 3D-scanner at home is to scan your body and print clothes with your exact size. 3D-scanning can also be used for medical applications, such as creating a prosthetic that fits perfectly [1].

Some other source for point clouds are GPS coordinates, these can be gathered from road users and can later be used to construct road networks. Which is useful for navigating and updating road maps. Approaches to this problem have to deal with different parallel point paths, created by different GPS users that do not drive the same. Other methods of mapping roads come closer to 3D-scanning, in the research of Masuda and He [2], they use a car with a laser scanner that scanned the road, their research can be considered a hybrid between 3D scanning and digital road construction.

The problems that we have to solve are simpler versions of both reconstruction problems. The input of our algorithm is a set of points $\mathcal{P}$, these points have coordinates in the $\mathbb{R}^2$ space. The output of our algorithm is a set of lines $\mathcal{L}$, these lines will have to form either a curve(s) or a network with no lines intersecting. In the curve reconstruction problem the points need to be connected to form a 2D-figure. The input to our algorithm also informs us what kind of problem the algorithm has to deal with, curve or network and multiple or single curves.

What we know about the points, concerning curve reconstruction, is that every point can have at most two outgoing edges and that all the points need to be used. What we also know is that there are two variants to the curve figure. The points given can form one figure or multiple figures. A figure can be closed, then every point has exactly two outgoing edges, the simplest example of a closed curve is a circle. A curve can also be open, which means that some points should have either one or two outgoing edges.

The network reconstruction problem of this project is easier compared to the real-life network problem because there are no multiple paths that need to be processed, but multiple

points. Points in the network reconstruction can have more outgoing edges since a point may be at the intersection of lines (roads), because there is a slim chance that points are exactly on an intersection, it is allowed to add extra intersection points. In figure 3 an example of a correct network reconstruction can be seen, the red points are added intersection points.

Research on the 2D curve reconstruction problem has resulted in a few methods to solve the problem. A common approach is to use Delaunay triangulation, explained in section 2.1, after the Delaunay triangulation different researches take different approaches. T. Dey [5] followed up the triangulation with a crust algorithm which leaves the shortest edges between points and which have a big angle with another edge. Problem with crust is that it does not perform well when a figure contains many sharp edges. Other research from T. Dey [6] resulted in an algorithm known as the GathanG algorithm, this algorithm is better in handling sharp edges in a curve. The algorithm is still not perfect for single curves, because it can result in multiple curves where only one curve is present. Research in network reconstruction has focused on different complexities of solutions. The research of D. Chen[3], has resulted in a method that uses as input a set of paths in $\mathbb{R}^2$ space. These paths need to be processed into a single path, before they can be used for navigation. Their method processes a collection of paths and finds shared structures, resulting in a single path. For our problem is this method not applicable since we only have one path of points. The method that will be used is going to be a modification on the curve reconstruction.

Our algorithm is not using a very special or new approach, it first triangulates a set of points $\mathcal{P}$, using the Delaunay triangulation. The resulting graph will be turned into a Gabriel graph, explained in section 2.2, erasing edges that are not between points that are Gabriel neighbors. In the final step are lines evaluated on their length, angles to neighboring lines and angles to lines that extend them the best. Lines are assigned a value, based on that value lines are removed. Better explanation of our algorithm theory can be seen in section 2. Our approach follows previously work to a certain extent, Boyer et al.[4] use a similar approach with Gabriel graphs and Delaunay triangulation. The main difference is our final step, which is simple yet effective. Theoretical guarantees which our algorithm has is from the Delaunay triangulation and the Gabriel graph, these guarantees are that no lines intersect and every point is connected to one or more other points with an edge(s). Our last step does have technical guarantees, implementing it results in a good curve, multiple and single, closed and open on point sets with a regular point density. It does however still have a problem with network reconstruction, with processing crossroads. Results can be seen in section 3. A more detailed conclusion is given in section 4.

2 The algorithms

In this section we will describe the algorithms we use for the problems. The two main algorithms that we use in our algorithm, which are Delaunay Triangulation and Gabriel Graph, will be discussed. Furthermore the final step, that happens after the Gabriel Graph is constructed, will be discussed.

2.1 Delaunay Triangulation

Delaunay triangulation is aimed to triangulate a set of points $\mathcal{P}$ in a plane, such that there are no any other points in set $\mathcal{P}$ inside the circle of any triangle in the triangulation. Implementing the Delaunay triangulation results in all the points in set $\mathcal{P}$ being connected in a

undirected graph, with no lines crossing each other.

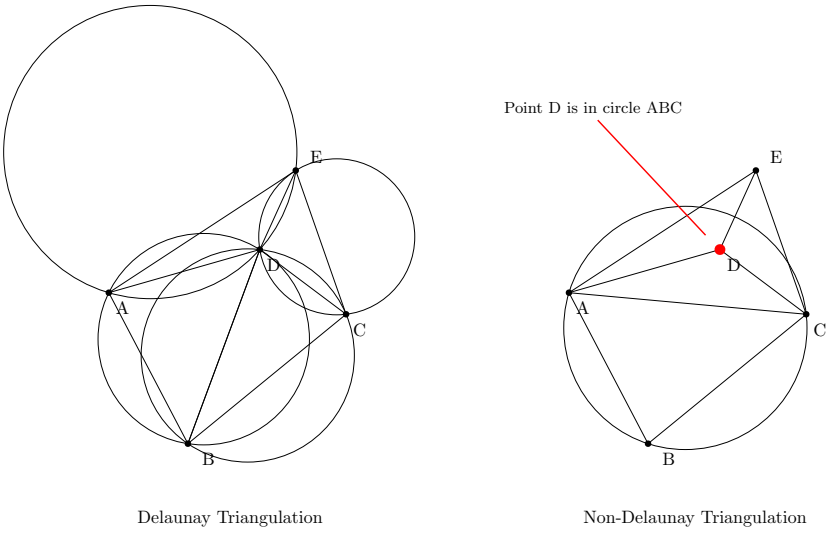

Figure 1: Examples of Delaunay Triangulation

An example in figure 1 of a correct and a non-correct Delaunay triangulation can be seen. In the left figure of the example, the circles do not contain a fourth point from $\mathcal{P}$, the Delaunay triangle tries to avoid a "skinny" triangle by maximizing the minimum angle of all the angles of the triangles in the triangulation.

During the triangulation process, once we have an existing triangle set and a point which is going to be added to the set. The whole process is initialized by generating a *supertriangle*. A *supertriangle* is an triangle containing all the points in the point set. At the end of the triangulation process, all the triangles which share an edge with the *supertriangle* will be deleted from the triangle list.

For all triangles whose *circumcircle* (The *circumcircle* of a triangle is a circle which passes through all the *vertices* of the triangle) contains the point to be added are identified, the outside edges of those triangles (which would be saved in a list called $edgeBuffer$) form an enclosing polygon. The triangles in the enclosing polygon will be deleted, and new triangles will be formed with the point to be added an outside edges of the enclosing polygon.

Algorithm 1 Delaunay Construction

```
1: Initialize the triangle list
2: Add supertriangle vertices to the end of the vertex list
3: Add the supertriangle to the triangle list
4: for Each sample point in pointlist do
5:     Initialize the edgeBuffer
6:     for Each triangle in trianglelist do
7:         Calculate the triangle circumcircle center and radius
8:         if Point is inside of triangle circumcircle center and radius then
9:             Add the three triangle edges to the edge buffer
10:            Remove the triangle from the triangle list
11:    Initialize the polygon list
12:    for Each edge in edgeBuffer do
13:        if Edge is not in the edgeBuffer then
14:            Add this edge in the polygon
15:    for Each edge in polygonlist do
16:        Add to the triangle list all triangles formed between the point and the edges
17:        of the enclosing polygon
18: for Each triangle in trianglelist do
19:    if A triangle uses the supertriangle vertices then
20:        Remove it from the trianglelist
21: Remove the supertriangle vertices from the vertex list
22: return trianglelist
```

The pseudo code describes the procedure to complete the triangulation. However, the triangulation loop will be extremely time-consuming in this algorithm because of the nested looping structure. To finish this loop it will take $O(n^2)$ time. Compare with the runtime of the loop for removing the insufficient triangle, the runtime of triangulation loop will dominate the overall runtime. So in order to finish this procedure it will take $O(n^2)$. The algorithm does not require a large amount of memory, since we will not get a massive amount of inputs and all the arrays used are logical arrays.

2.1.1 Proof of Correctness

Loop invariant: Suppose the precondition holds. After first iteration the triangle-list contain the triangle which does not use the *supertriangle vertices* that have been checked.
Initialization: Before the loop iterates the first time, the triangle-list is initialized and all the points in the vertex list are inside the *supertraingle*, and this *supertriangle* is added to the triangle-list.
Maintenance: After each iteration, the triangle which shares an edge with the *supertriangle* will be removed from the triangle-list. For the triangles whose *circumcircle* contains the point to be added is marked, the outside edges of those types triangles form an enclosing polygon. The triangles in the enclosing polygon will be deleted, and new triangle will be formed with the point to be added an outside edges of the enclosing polygon.
Termination: After all edges have been traversed, and the triangle-list only contains the triangles which are not sharing any edges with the *supertriangle*. The graph only contains

Delaunay edges that constructed from the points of the set $\mathcal{P}$.

2.2 Gabriel Graph

After the calculation of the Delaunay triangles we turn our set of lines into a Gabriel graph, which is a graph with a vertex set $\mathcal{P}$ which any distinct points i and j in $\mathcal{P}$ are adjacent if the circle with segment PQ as diameter contains no any other elements from $\mathcal{P}$. The Gabriel graph is used in our algorithm to thin out the edges we added after the Delaunay triangulation. The method is sensitive to the space between three points, if our set of edges would not become a Gabriel graph, then the final step would not result in a smooth curve.

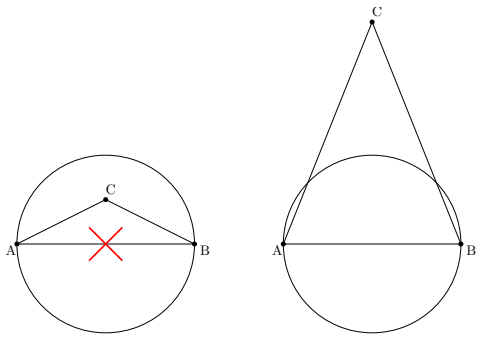

Figure 2: Examples of Gabriel Graph

The figure above shows a correct example and an incorrect example of the Gabriel graph. In the incorrect example can it be seen that point c is in the circle between a and b. In the correct example the point is outside this circle. All edges, in the Gabriel graph, are a diameter for a specific circle in Delaunay Triangulation, thus it could be considered as a sub-graph of Delaunay Triangulation.

To implement the Gabriel graph, we made use of the result generated from the Delaunay Triangulation since the Gabriel graph is a *subgraph* of the Delaunay Triangulation[7]. So we created a new point set $closestNeighbors$ to store all the nearest adjacent points of a *samplepoint* which found by Delaunay triangulation process from the point list. Then inside the $closestNeighbors$, the Gabriel edges can be identified by calculating the distance l between a circle (formed by point a, b with a diameter r of length of segment ab) with a center c and a point x by using the *Euclidean distance* formula. If the distance l is bigger than the radius $d/2$ then it is not inside circle, and the segment ab is a Gabriel edge and it will be added to the graph.

Algorithm 2 Gabriel graph implementation

```
1: Initialize graph
2: for Each samplepoint a in pointlist do
3:     Initialize closestNeighbors
4:     Fill closestNeighbors with neighbors found in Delaunay Triangulation
5:     for Two points b and c in closestNeighbors do
6:         Create circle between a and b
7:         if point c in circle ab then
8:             Remove edge ab to the graph
9:         Create circle between a and c
10:        if point b in circle ac then
11:            Remove edge ac to the graph
12: return graph
```

Implementing the pseudo-code above results in a program with worst-case running time $O(n)$. The Delaunay triangulation already calculates the neighbors of a point, so no $O(n)$ is needed for that part. For each point in set $\mathcal{P}$ do the neighbors need to be checked, which takes $O(n)$. The checking of the Gabriel properties takes constant time.

2.2.1 Proof of correctness

Loop invariant: After each iteration does the graph contain the edges between the Gabriel neighbors of the points in set $\mathcal{P}$ that have been checked since time i.
Initialization: No points have been checked so no Gabriel edges could have been found to add to the graph. So the loop invariant holds.
Maintenance: After each iteration i the neighbors of a point a have been checked on being Gabriel neighbors. The algorithm takes two close neighboring points, and creates circles between the a and the other point respectively. The points are Gabriel neighbors, if the other point is not in the circle. When two points are Gabriel neighbors an edge will be added to graph. To ensure that the graph contains all the Gabriel edges found since time i.
Termination After all the points have been checked does the graph contain all the Gabriel edges that could be possibly constructed from the points in set $\mathcal{P}$. So the loop invariant holds.

2.3 Final step

The algorithm has a final step to get from the Gabriel graph to the desired curve or network. This step contains of 2 sub-steps; first we remove the 'worst' lines, then we try to fix invalid results (e.g. a unconnected network), keep the opening curves from "closing", and some other minor details.

2.3.1 Evaluation function

The foundation of the final step is the evaluation function. The final graph can only be as good as the knowledge on the value of each line in its context. That is why we took a lot of time experimenting and evaluating every value in this function, resulting in a slightly strange looking but heavily tailored function.

1. The length of the line. ($GlobalLengthValue, LocalLengthValue$)
   We use the length in 2 ways. Take line $x = a \leftrightarrow b$. x will have a $GlobalLengthValue$ equal to its length divided by the mean length of all the lines in the graph. x will also have a $LocalLengthValue$, this is equal to its length divided by the mean length of all the lines in the graph that are connected to a or b.

2. The angles between the lines that extend the corresponding line best. ($AngleToExtensionValue$)
   This means if you look at the lines as vectors, the lines on both sides with the smallest angle to the corresponding line. For instance like in figure 6, the algorithm would take angle a and angle c and sum them together. To work with intuitive values the AngleToExtensionValue is also divided by 4π giving it a range of $[0, 1]$ (in practice even smaller due to Gabriel) before the multiplier is applied.

3. The angles between all neighbors. ($AngleToNeighborsValue$)
   This means the angles between the corresponding line and the neighboring lines summed up. An example of this can be seen in figure 7. This is meant to find lines that are inside a polygon, in this case line c. Since in this case a bigger value is better it is multiplied with a negative constant, such that a bigger value for this factor results in a smaller evaluation value. For every angle x the value $y = \sqrt{\frac{x}{2\pi}}$ is used, because a range of $[0, 1]$ is nice for the evaluation process and roots gave the best results during evaluation.

With these factors the evaluation value is calculated as follows:
$EvaluationValue = GlobalLengthValue^2 \cdot 3 + LocalLengthValue^2 \cdot 0.5 + AngleToExtensionValue \cdot 6 + AngleToNeighborsValue \cdot -0.8$.

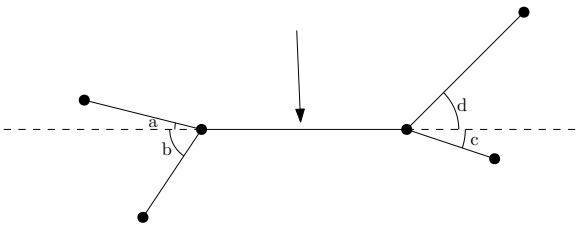

Figure 3: Angles to all neighbors

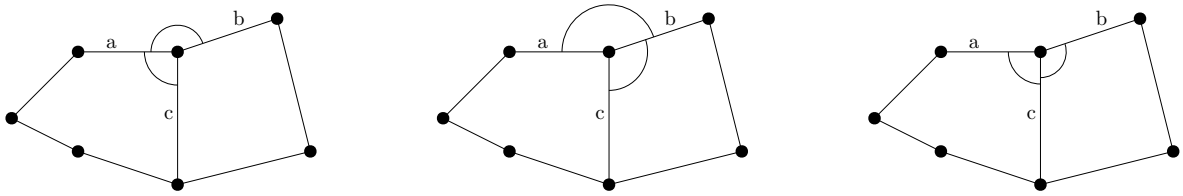

Figure 4: Angles to all neighbors

2.3.2 Structure

The final step consists of many small sub-steps.

Algorithm 3 Final Step

```
1: if Case ≡ single ∨ Case ≡ multiple then
2:     MoreThan2Fix(graph)
3:     Lessthan2Fix(graph)
4:     if Case = multiple then
5:         NetworkLineFix(graph, VarianceMultiplier = 3)
6:     SingleCurveFix(graph)
7: else
8:     NetworkLineFix(graph, VarianceMultiplier = 1.7)
9:     ProbableCornerFix(graph)
```

The final step uses an unconnected graph as data structure (where every node contains a list of all its lines) so points are vertices, and have the corresponding properties (like degree).

Also since this algorithm is mainly responsible for the correctness in the eyes of humans, rather than maintaining mathematical properties many properties (e.g. connected graph for single curve) are not ensured, because we could not get the same finesse in our result while ensuring these properties without getting a extremely long run-time.

2.3.3 MoreThan2Fix

We start by creating a set of removable lines $R = \{x = a \leftrightarrow b \quad | \quad \#a \geq 3 \wedge \#b \geq 3\}$ this is done by iterating over the lines in the graph once. Then we select the worst line x, so the line for which the evaluation value is higher then for all other lines, and remove it from the graph, for the nodes a, b that connect x we check $\#a \geq 3$ and $\#b \geq 3$ and remove the lines connected by these nodes from R if this condition is violated, we also update the evaluation value for all lines that connect to a or b.

2.3.4 LessThan2Fix

The LessThan2Fix Handles a very specific but also very common case. At the end of a sharp line, the corners can get too sharp for Gabriël and then a vital line is removed from the graph, this method is a step towards counteracting that effect.

It creates a set of lines $S = \{x = a \leftrightarrow b \quad | \quad \#a = 1 \wedge \#b \geq 2\}$. Then for every x in S it finds the neighboring lines y_1, y_2 to x and connects a to a node $c \neq b$ where c is a vertex of y_1 or y_2, the choice between y_1 and y_2 is based on the $AngleOfExtention$ (as explained in the evaluation function).

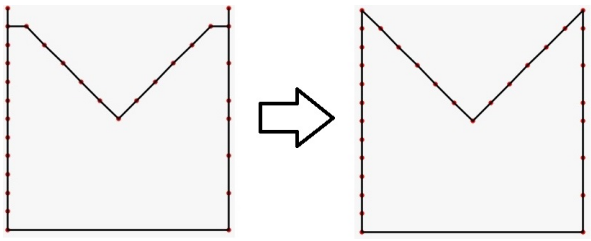

Figure 5: An example of LessThan2Fix

2.3.5 NetworkLineFix

Just like $MoreThan2Fix$ $NetworkLineFix$ removes lines based on their $EvaluationValue$ and although it does look at the degree of nodes it works applies it in a different way. First the evaluation function is ran for all the lines, then from the evaluation values a NormalDistribution n is created. All lines for which the evaluation value is larger than $mean(n) + var(n) \cdot VarianceMultiplier$ (where $VarianceMultiplier$ is 3 for multiple and 1.7 for network) are removed. If lines are removed n will be recalculated at the end of the process and the lines will be compared to $mean(n) + var(n) \cdot VarianceMultiplier$ again.

2.3.6 SingleCurveFix

SingleCurveFix is a set of very small functionalities. First if the curve has no vertices with degree 3 or higher it will find the longest line and remove it if it is longer than 3 times the average length (opening up cases like the M).

If a curve has vertices with degree 3 or higher (and therefore is a graph) an algorithm similar to *MoreThan2Fix* is applied with the difference that $R = \{x = a \leftrightarrow b \mid \#a \geq 3 \wedge \#b \geq 2\}$. Also lines longer than 5 times the average length are removed.

2.3.7 ProbableCornerFix

ProbableCornerFix is a relatively complex (almost 300 lines of code) algorithm. It looks at vertices with a degree of 1, points with an exceptionally high angle and lines with an exceptionally high angle (compared to its neighbors) 1.7π and 2.3π respectively, call the set of these corners end endings X. And evaluates weather or not they should be attached to a nearby part of the graph. First creates a list of points Y that are within a certain distance (based on the average distance of lines) and a certain angle compared to the direction the sharp end of the corner is pointing to. Then for all the possible lines L from the points in X to the corresponding points in Y an evaluation (with the previously explained evaluation function) again a *Normal Distribution* n is made off all the existing lines in the graph. If the line $l \in L$ with the lowest evaluation value has an evaluation value smaller than $mean(n) + standradDeviation(n) \cdot 3$ the line is added to the graph.

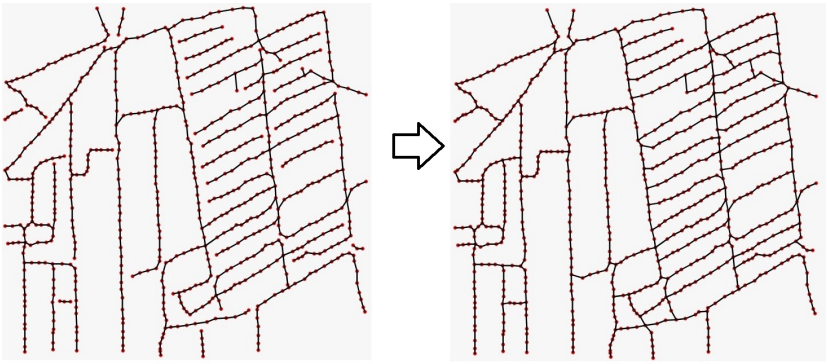

Figure 6: An example of ProbableCornerFix

3 Experimental evaluation

In this section we will illustrate the experimental phase of this project. We conducted a series of experiments to verify and validate our algorithm. Including an algorithm runtime analysis and an algorithm accuracy verification.

3.1 Runtime test

In order to see the performance of our algorithm, we decided to make several test cases which contained more than 1000 random points. We wanted to perceive under these conditions how fast our algorithm could give its result.

We used the 1,000 points as a base, and increased the number of points with steps of a 1000 to build the test cases for extending the scales of tests. The algorithm will be tested from 1,000 points up to 15,000 points to fully analyze the algorithm running time. We ran all the iterations 15 times and took the average time of this.

The testing process contains two parts. First, we test the Delaunay runtime separately since it contributed the most to the total runtime. Then we tested the Gabriel runtime (it contains the Delaunay runtime since Gabriel graph is a sub-graph of the Delaunay triangulation) to ensure the total runtime did not exceed the given time (5 minutes).

From the results we concluded that our algorithm satisfied our intention respect to the time. A curve was created within 5 minutes. Moreover, we also verified the proof of correctness of the algorithm. By plotting the runtime and point-count, the relation will be shown and it could be used to verified our proof.

Point count	Delaunay	Gabriel	Final steps	Total time
1000	23.7	2.7	102.9	129.2
2000	45.7	3.3	161.4	210.4
3000	100.2	1.8	313.6	415.6
4000	179.3	2.8	454.1	636.2
5000	284.3	3.3	687.1	974.8
6000	418.9	4.7	965.1	1388.7
7000	590.7	6.3	1304.1	1901.1
8000	796.9	7.0	1678.3	2482.2
9000	1099.8	8.4	2141.7	3249.9
10000	1466.8	9.1	2608.3	4084.2
11000	1927.2	11.8	3155.0	5094.0
12000	2279.7	12.2	3699.0	5990.9
13000	2639.2	16.6	4536.3	7192.1
14000	3128.7	13.7	5468.8	8611.1
15000	3772.2	16.6	6650.1	10438.9

Table 1: Delaunay and Gabriel runtime tests

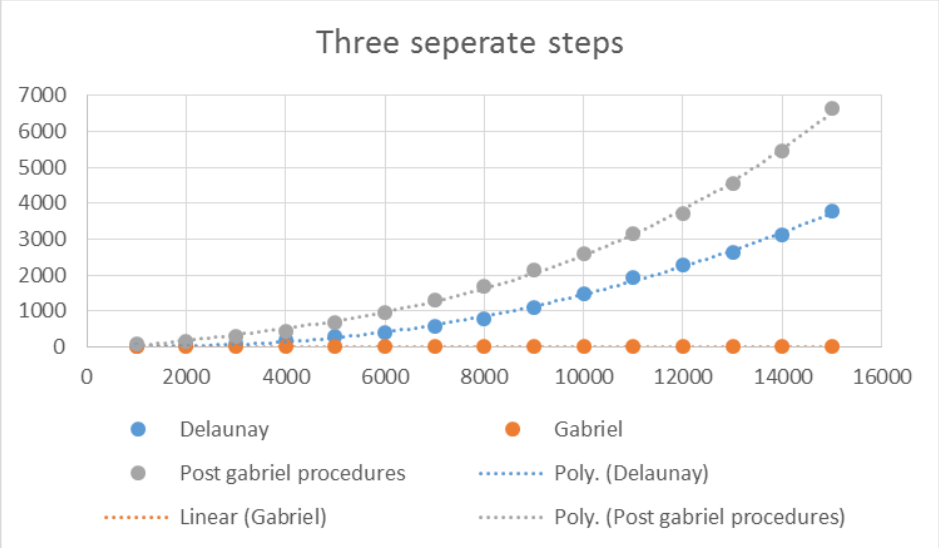

Figure 7: Runtime Plotting 1

Figure 5 shows the relation between the runtime of the algorithm and the point-count. First we can clearly see that constructing a Gabriel graph takes linear time, which justify the proof of correctness in Gabriel runtime analysis. Then we see that the Delaunay runtime approximately fits a polynomial line, which implies that Delaunay runtime is polynomial. The post Gabriel procedures shows a polynomial tendency as well, which satisfied our proof as well.

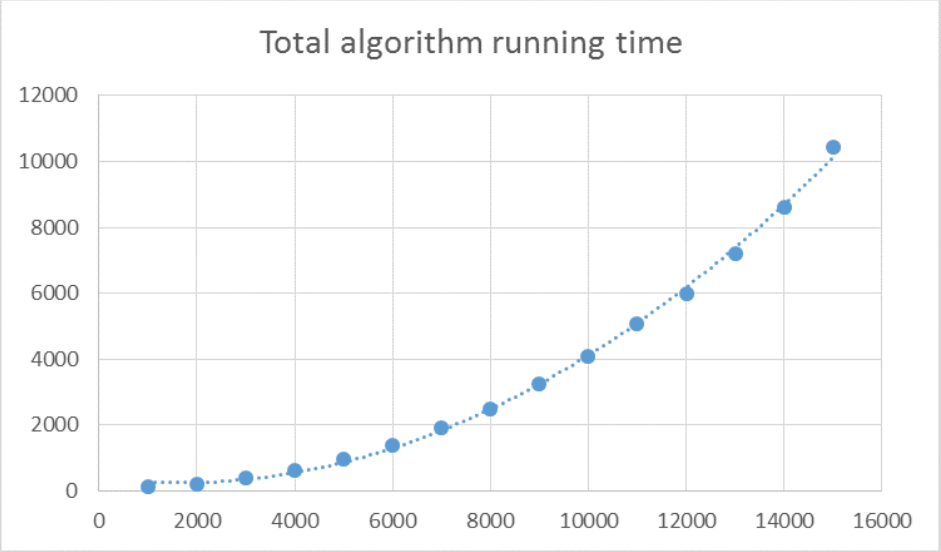

Figure 8: Runtime Plotting 2

Figure 6 is the plotting result of runtime of entire algorithm. Because the Delaunay triangulation and post Gabriel procedures contributed the most to the runtime, thus the total runtime is likely to become a polynomial line, which indicates the algorithm will generate the result in polynomial time asymptotically.

With the respect to the test results, we concluded that the algorithm was able to process 10,000 points within the given 5 minutes, which meant the algortithm satisfies the requirements of runtime.

3.2 Case test

Whilst testing the runtime of the algorithm, several test cases were built to test the accuracy of the algorithm. Those test cases were mainly created in 3 different intentions which were single-curve, multiple-curve and the road-network.

The way to construct the cases was basically plotting points in IPE using existing material found in the Internet, photos and drawings. Coordinates of the points could be extracted from the .ipe file, using notepad++. These coordinates would then be changed into the input .txt format. We tried to make a diverse amount of cases to cover as much possibilities as possible. In this section, we will discuss some identical cases including the case that our algorithm provided perfect solutions, as well as the failures. We tried to make difficult test cases for our algorithm, test cases with sharp edges and with the distances between points highly irregular.

Random data for our runtime tests was generated in our program, the amount of random points could be chosen and the points would be placed on random places between 0 and 1.

3.2.1 Single curve reconstruction

The algorithm is able to give a correct result for the most of test cases. However, because of the density difference, some biases are still existing and cannot be avoided because the output of our algorithm is totally dependent on the quality of the input. Likewise, the figure 9 shows the defect which is caused by the density difference.

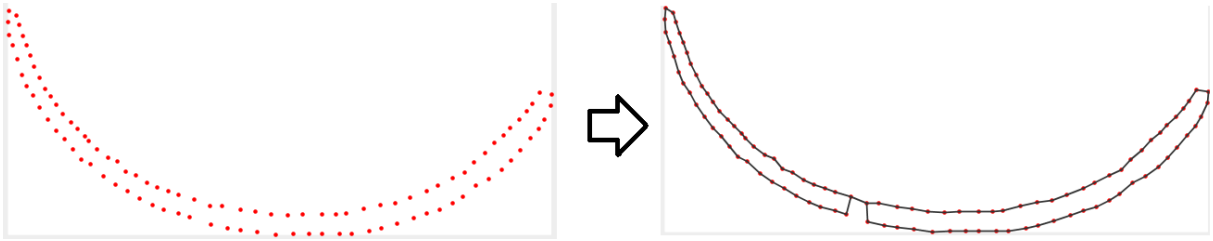

Figure 9: A failed case

The algorithm uses the density of the points to determine if the curve should be opened or closed. So in figure 7 we could see that apart from the "gap" part, the rest part of points are clustered in a same average density. The distances among points in the clusters do not differ from each other significantly except the "gap" part. At this point the algorithm choose to remain a reasonably small scale connection and erase the relative "large" scale segments, thus the gap is created by this mechanism. In order to verify it, we tested on a proper distributed density point set, which is the "M" case.

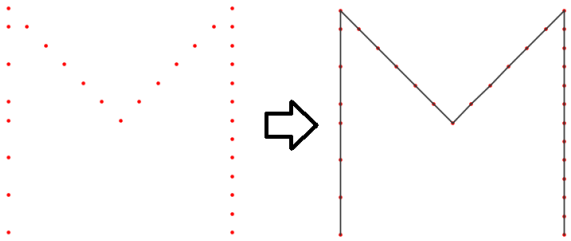

Figure 10: A connected M

In this case, the algorithm provided a really accurate connections. Since the points density of the connection part is evenly distributed, so we do not have any biases in the connecting procedure until the "gap" in the bottom and the top corners of M. The "gap" is identifiable since the point density is too low and it will be recognized as a open curve by the evaluation. However, those two corners are too sharp that it does not remained after Gabriel process (see figure 11).

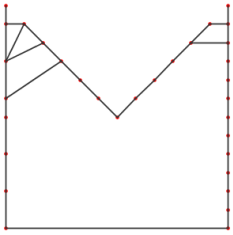

Figure 11: The M generated by Gabriel Graphing

To solve the sharp corner problem, we did the further process on the Gabriel graph, which is the *final step* discussed in the previous section. The final output of the "M" is shown in figure 10. On the other hand, we need to verify if the algorithm still provides a sufficient result under a complex condition which involved more than one factor might cause biases on the output. So we used the elephant case since it did not have a distributed point density and it had several "sharp corners". The output did verify the sufficiency of the algorithm as is shown in figure 12.

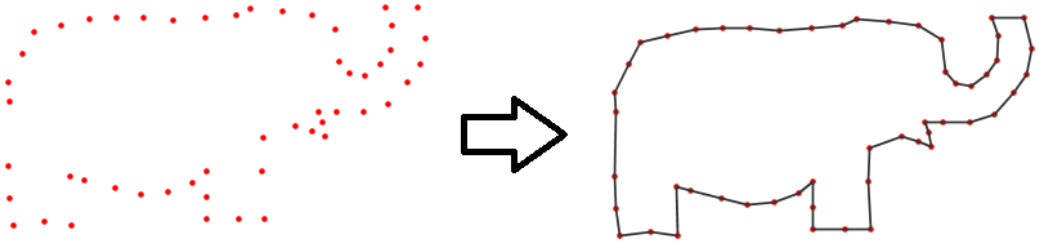

Figure 12: Connected Elephant curve

We wanted to check the performance of the algorithm, thus we decided to compare the results between our algorithm and the curst algorithm from the literature. We used the elephant case as the tester for both algorithms. From the result we could see that crust algorithm did not perform well when the point density changes slightly, and it did not provide a good solution for the sharp corner problem. On the contrary our algorithm solved both problem and provided an accurate output.(see figure 13)

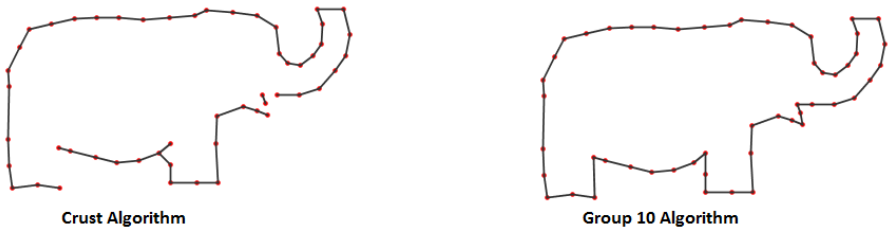

Figure 13: Curst Algorithm vs Group 10 Algorithm

What's more, we also tried to make use of some cases from the literature and compared our result to it to see the performance of our algorithm. As an example we chose the dragon[8] example. After running the algorithm, our output is shown in figure 14.

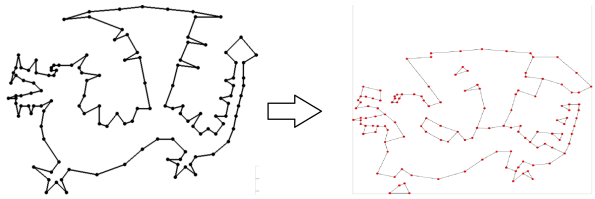

Figure 14: Dragon example

Clearly our algorithm failed in this case. As we could derive from the picture, the algorithm did not process it as a single-curve, that explains why we have a front leg which is self-connected but does no connect to the body. The problem with this figure is that it has a lot of sharp corners and the distances between points are irregular, unlike the other figures we often have to solve. We considered that this case would need a more specialized algorithm, so we did not bother ourselves too much with solving this problem.

3.2.2 Multiple curve reconstruction

The multiple curve test cases have a correct solution of multiple curves that can be open or closed. That is the only thing known about this test case. For testing we used the following test cases.

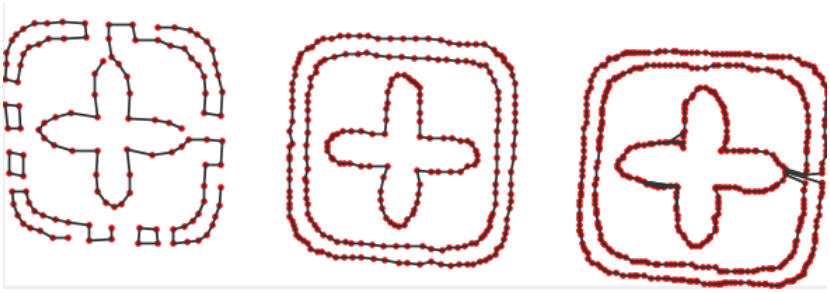

Figure 15: Three shapes case

The example case in figure 15 is a perfect example of the point density problem, in the most left figure the points are loosely clustered than in the other two figures. Because all distances would be evaluated by algorithm, which caused the algorithm removing the edges that did not seem to be a sufficient connection in the left figure whereas should have been preserved. The right-most case in figure 15 also did not seem to be correct due to its high point density. The triangulation generated too many undesired lines and it was regraded as sufficient by Gabriel process. We were not able to solve it because we do not have any other ideas if not using Gabriel graph.

We tried to motivated the reliability of our algorithm on handling multiple curves that it can solve figure containing multiple open curves and closed curves. Thus We used the painting 'Starry night' from Vincent Van Gogh. We made this case in IPE, adding points on top of the painting that trace the contours of the painting.

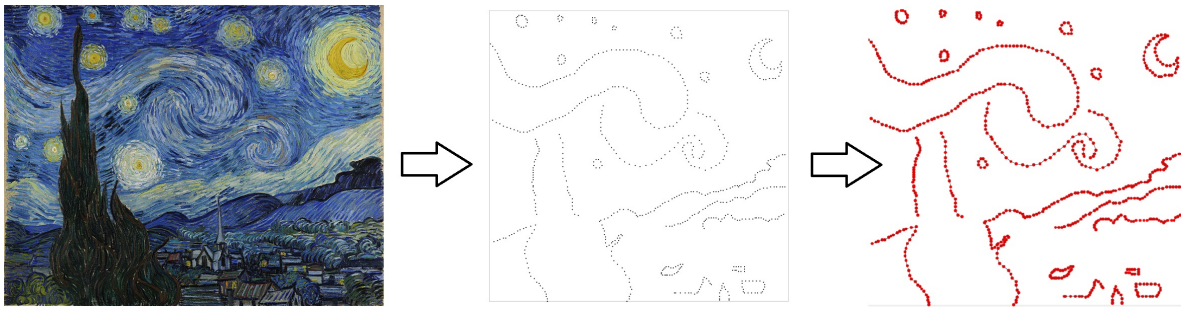

Figure 16: Starry night

The reconstruction of the case was almost successful with our algorithm, the only problem occurred at the sharp edges of the moon, shown in figure 17 in detail, this defect is similar to the problem in the previous section. The rest part of the painting was correctly processed and the open and closed curves were connected correctly. Overall it could be said that our algorithm provides a reasonable solution for multiple curves.

Figure 17: closeup of the moon

All the test cases above are relative complex. We wondered if the algorithm will generate a sufficient output in some relative simple cases. We made the sun case (see figure 18) and the multiple circles case (see figure 19)

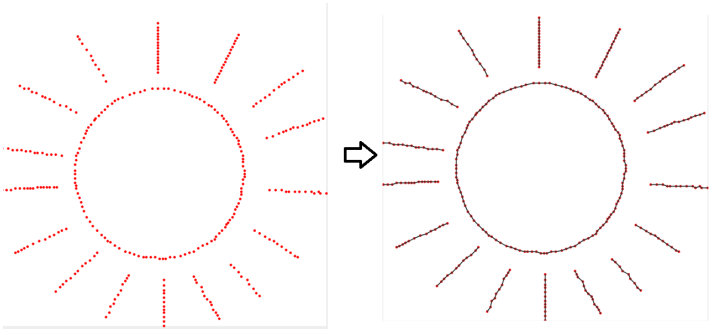

Figure 18: Sun example

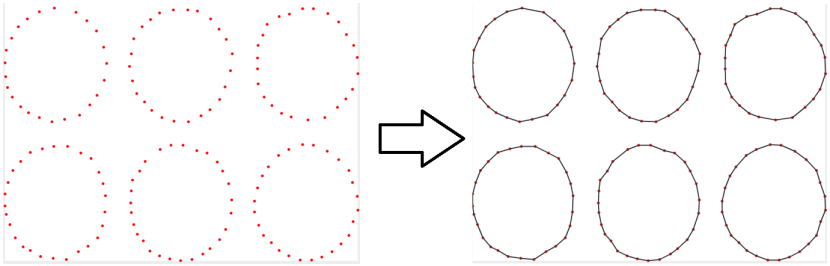

Figure 19: Multiple circles example

From the examples in figure 18 and 19 we could see that the algorithm performs well when the input is relatively simple and the point density is well-distributed. From the experiments with the multiple curves we concluded that not only the quality of the input would affect the performance of the algorithm, but also the complexity of the input will impact the output. This complexity is mainly the shape of a figure, does it have sharp edges or does it contain a spiral. An example of such a spiral can be seen in figure 20, which is small part of the Starry Night test case.

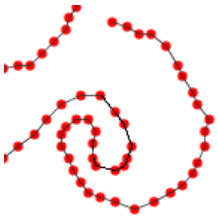

Figure 20: Spiral complexity example

3.2.3 Test case for network reconstruction

In order to have a good insight of our Network Algorithm. We need a road network which is complex enough with variety of testing factors such as cross road and roundabout for testing the sufficiency of the algorithm. We focused on a real-life map which has a massive amount of nonstructural roads. Therefore we chose Mumbai as a test case for the algorithm.

Figure 21: Map of Mumbai with algorithm result

In a brief view, the algorithm provides a reasonable result for this complex map as shown in the figure 21. Note that some points have different colors, because these points are closed to detected crossroads. This information could be used as a signal to add extra points at the center of the crossroads. The test case contains many roads that are close to each other, which increases the complexity of the test case. However, this was not the cause of the problems we had, the problems occurred when a roundabout or a junction showed up in the map.(see figure 22). We found out in this case we failed all crossroad reconstruction and also one roundabout reconstruction.

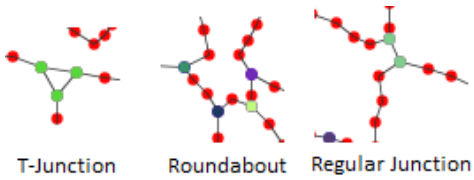

Figure 22: The different categories of junctions

For the junction the main reason of failing was because we did not implement the insert point tactic, thus the algorithm will still check and erase line at the crossroad. Moreover, we also found out that in some crossroads there would be some unconnected part (see figure 20), the T-junction was caused by not removing enough edges. The algorithm left a triangle on the place of the junction, this triangulation was caused by the relatively big spaces between the three points. All edges of this triangle are almost equal, thus the final step will not remove them. So in the regular junction case the algorithm leaves us with two points which have three connections rather than only one point with four connections. The algorithm does not have any preferences among four lines or three lines towards one vertex, since it will be only determined by distances and angles. However we did not insert points in the middle of the crossroad, so there is almost no points with four incoming lines except the existing ones. So we failed to reconstruct the crossroads.

As for a roundabout, the algorithm should have created a proper "circle". Points that could be part of a roundabout are often close to each other, so when the distance between two points is slightly larger then the other distances in the roundabout, the line between them will be removed by algorithm. It is same defect as we discuss in the single curve reconstruction

caused by the point density and the Gabriel function. So we think this bias is not caused by algorithm but a bad quality of the input. To verify this, we tried another input with a high quality(see figure 23). As our predication, the algorithm can give a correct output for roundabout by giving input with a good quality.

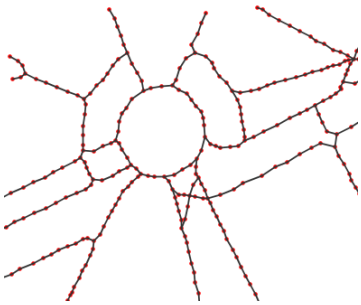

Figure 23: A good road network

4 Concluding remarks

The algorithm that we have developed in this project is useful in a few instances and less useful in other instances. The algorithm performs well on curves, single and multiple. Both with regular distances between the points, it can also handle sharp edges in a curve. The biggest fault in the algorithm is that it cannot handle a point set where the points have irregular distances between them, this also decreases the correctness of the construction of sharp edges. This problem might be solved by relying less on the distances between points and more on angles. But while that would work well for cases with high variation in line length it would work worse in cases with sharp corners or low variation in line length.

The network reconstruction is an instance for which our algorithm still has some clear flaws. The roads are correctly processed, as can be seen in the Mumbai example (figure 19), but there is a problem with processing crossroads. The algorithm does not add extra points where lines intersect and this results in strange shapes where roads collide. To fix this problem the algorithm would have to recognize where lines intersect and add a cross point. Lines would need to be removed and connected to the new point. Although this is doable we simply didn't have the time to finish this after the last major changes to the network algorithms, we believe that when we have found some good theoretical solutions, and that adding these cross points would give valuable information to reevaluate their context.

For future work, more specific implementations could be added to the curve and the network reconstruction in order to improve the accuracy, since we were not able to fix the case with a complex point clusters. A better solution would be to choose a different approach for our algorithm. We also believe that an effort toward getting our current output in line with the requirements where they are not would solve a lot of problems, we found that often the parts that look odd break strict rules that are given before hand (like 3 outgoing lines for curves), solving those would give more information about its context and therefore better results. Those improvements will definitely increase the accuracy. However the runtime will be affected by a different implementation and further tests will be needed to prove the accuracy and correctness of the new versions.

References

[1] Ciobanu O., Xu W., Ciobanu G.. The use of 3D scanning and rapid prototyping in medical engineering. In *Fiability & Durability*, pages 241–247, 2013.

[2] Masuda H., He J.. TIN generation and point-cloud compression for vehicle-based mobile mapping systems. In *Advanced Engineering Informatics*, vol. 29, no. 4, pages 841–850, 2015.

[3] Chen D., Guibas L. J., Hersberger J., Sun J.. Road Network Reconstruction for Organizing Paths. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1309–1320, 2010.

[4] Boyer E., Petitjean S.. Curve and Surface Reconstruction From Regular and Non-Regular Point Sets. In *IEEE Conference on Computer Vision & Pattern Recogni- tion*, vol. 2, pages 659–665, 2000.

[5] Dey T., Mehlhorn K., Ramos E.. Curve reconstruction: Connecting dots with good reason. In *Computational Geometry: Theory and Applications*, pages 229–244, 2000.

[6] Dey T., Wenger R.. Fast reconstruction of curves with sharp corners. In *International Journal of Computational Geometry & Applications* , vol. 12, no. 5, pages 353–400, 2002.

[7] Matula D. W., Sokal R. R.. Properties of Gabriel graphs relevant to geographic variation research and clustering of points in the plane. *Geographical analysis.*, vol. 12, no. 3, pages 205—222, 1980.

[8] S. Ohrhallinger, S. Mudur. An Efficient Algorithm for Determining an Aesthetic ShapeConnecting Unorganised 2D Points. *Computer Graphics Forum.*, Vol. 32, No. 8, pages 72–88, 2013.