University of Nottingham
UK | CHINA | MALAYSIA

EEE Department of Electrical and Electronic Engineering

EEEE3001 / EEEE3021 / EEEE4008

(delete whichever of these is inappropriate, and this text)

Final Year Individual Project Thesis

Path Planning for Mobile Robot Navigation

AUTHOR:

ID NUMBER:

SUPERVISOR:

MODERATOR:

DATE:

Third *(Fourth if applicable)* year project report is submitted in part fulfilment of the requirements of the degree of Bachelor *(Master if applicable)* of Engineering.

Abstract

This thesis develops the software design for shortest path planning for wheeled robots to minimize the energy consumption in robot applications. This path planning algorithm is expected to generate a collision-free path in an environment with obstacles and be optimized by adjusting different parameters.

The path planning problem becomes more essential in intelligent robot control as the increasing trend of demands of navigating robots. In this project, an optimal motion planning will be established by the combination of Delaunay Triangulation which the technique used for generating triangular mesh representations of an environment and Dijkstra's algorithm to find a shortest path between two points within that environment. Two different layout cases are created for test and triangulated into a set of meshes by Delaunay algorithm for Dijkstra's algorithm to obtain the near-shortest path between two vertices in a suitable amount of computational time. The experimental results demonstrate that this shortest path planning algorithm is capable of any static environment with fixed obstacles within the terrain.

This report also examined the comparison of the shortest path when using different variables to optimize the final performance. The parameters and errors are analyzed obtain the shortest path of high accuracy and consume the least computational effort (memory and time). Finally, the improvements of shortest path planning for future work are proposed and discussed.

Key Words: Shortest Path Planning, Navigating Robot, Automation

Table of Contents

Abstract……2
Chapter 1: Introduction and Background Review……4
1.1. Introduction and Summary of Literature Review……4
1.2. Aims and Objectives……9
1.3. Report Structures……10
Chapter 2: Methodologies and Example Test……11
2.1. Delaunay Triangulation……11
2.2. Shewchuck's Program……14
2.3. Mesh Generation……17
2.4. Dijkstra's Algorithm……18
2.5. A Basic Shortest Path Example Test……20
Chapter 3: Shortest Path Planning Program Design……21
3.1. Software Programming ……21
3.1.1 Program Structure……21
3.1.2 Bottlenecks……24
3.2. Optimization Analysis……26
3.2.1. STL Data Structures……26
3.2.2. Parallelization……27
3.2.3. Portability for MSYS2 or Linux……28
Chapter 4: Result Analysis……29
4.1. Layout of *concentric boxes*……29
4.1.1. Parameter Analysis……29
4.1.2. Shortest Path Layout……33
4.2. Layout of *'A' character*……36
4.2.1. Parameter Analysis……36
4.2.2. Shortest Path Layout……40
4.3. Comparison Between Two Layouts……43
Chapter 5: Discussions of Future Works and Review of Project Time Management……44
5.1. Discussions of Future Works……44
5.1.1. Program Structure……45
5.1.2. Bottlenecks……45
5.1.3. STL Data Structures……45
5.1.4. Parallelization……46
5.1.5. Portability for MSYS2 or Linux……48
5.2. Review of Time Management……48
Chapter 6: Conclusions……50
References……51
Appendix ……54

Chapter 1: Introductions and Background Review

At first, a brief introduction of the backgrounds and the objectives of this project is given in this section. This will include the summary of the literature review for the topic of shortest path planning for navigated mobile robot, the aims and objectives of this project and the explanation of the report structure.

1.1. Introduction and Summary of Literature Review

Due to the demands of reducing workforce, the area of automation expands in human daily life. Therefore, different kinds of robots are utilized in lots of working fields in the past few decades, for example, the self-driving cars [7], warehouse robots [8], and wheelchair robots [9]. The wheeled navigation robots are designed to be able to localize steadily, find the safe navigation path and reach the destination smoothly in an unstructured environment. [4] Human welfare is covered by robot applications more and more, accompanying with more demand for energy-efficient systems due to its increasing energy consumption. Energy cost minimization on robot applications has been deeply analyzed, some techniques are emerged to improve the efficiency of robot systems.

In addition to different energy conservation techniques in some industries, path planning for wheeled robots has been proved to have some positive effects on tracking the problem of power consumption minimization for mobile robot navigation. Motion planning is demonstrated to reduce energy consumption for the robot system for navigation robots and has been experimentally demonstrated to save up to 42% of the energy cost [1] compared with the traditional utility-based method. Shortest path computation plays an essential role in wheeled robot navigation system due to its sensible and time savings decisions and the path planning based on this has been further studied. [2]

Both the path between the particular node pair and the one among all the nodes are included in the analyze of shortest path algorithm [3], this paper will focus on the shortest path from the starting point to one destination node. The challenge addressed in this project is to make the path planning for navigated mobile robots to travel between two points in an actual route according to the computational effort requirements.

The analyze on autonomous robot navigation involves the environments not only flat with regular obstacles also the uneven and cluttered ones. Figure 1 demonstrates the robot navigation for a wheeled robot in the environment with narrow slope areas and cluttered space. [4] This robot is designed to travel stably to a higher platform which is the prerequisite for the service robot like the intelligent wheelchair robot. [4] The vehicle with onboard navigation system is capable to enable the robot localize itself robustly [1], realizing the slope areas from the staircases and mapping out a suitable path for itself.

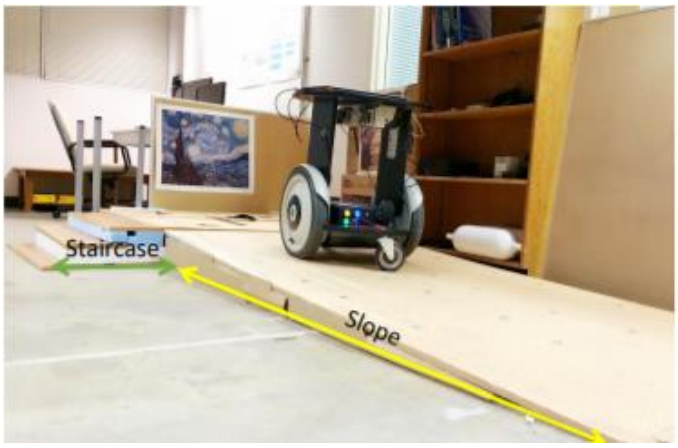


Figure 1: An illustration of a wheel robot navigating in a challenging environment [4]

Although the robot navigation in plane with narrow slope and cluttered space is utilized mostly in the actual environments, the analyze for 3D environments is more complex to perform, considering the velocity and slope detection. Therefore, this thesis will only analyze the navigation on a 2D plane and the shortest path planning is also considered to be performed in an even environment.

As shown in Figure 2, the environment of a 2D route model consists of a flat table plane including some obstacles, such as bottles, cans and building blocks. The path planning model is implemented as the planform of this environment, and the obstacles are defined as the impassable blocks in the route. The mobile robot is designed to travel from one point to another safely without hitting any obstacles and mostly the process is expected to consume less energy and be more efficient.

Figure 2: An example for the robot navigation example in an 2D plane [12]

Path planning aims at finding the shortest, most feasible and first-rank path for a navigation

robot in order to implement the move from one point to another. [5] Shortest path issue plays an essential role in both indoor and outdoor environments. For indoor applications, food delivery robots in the restaurants [10] and transport robots in aerospace [11] are analyzed deeply due to the huge demand markets. This shortest path algorithm also applied widely in many outdoor environments, for example the handling city emergency way and guiding driver system in traffic network. [2] This computation of roadwork is required to be solved quickly because the condition of traffic in a city is changing periodically and there are larger amounts of drivers requesting this service. [2] The auto-driving vehicles also requires this shortest path planning to obtain the safest and most efficient path for itself. Considering the computation efforts for a laptop, this experiment only focuses on the routes creating a few meshes and that means the layouts of the routes are in relatively small sizes which is the indoor navigation robot examples.

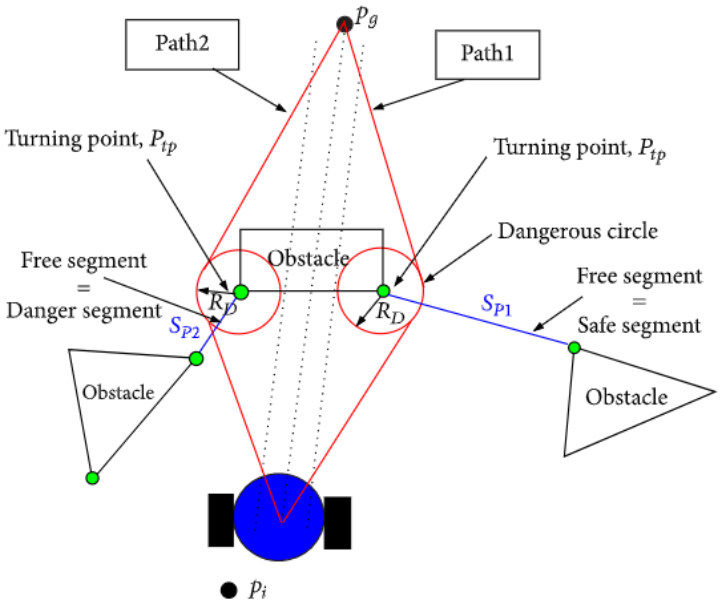


Figure 3: Framework of navigation [13]

The main issue that the path planning is technique utilized by the robot to detect when the robot is located between two obstacles in order to enable itself to travel to the target point without collision. [13] A classical algorithm is designed based on finding the turning point of a free segment for solving the issue of path planning. In Figure 3, the distance between two endpoints of two different obstacles is specified to be a free segment, and the robot is searching the endpoint of a safe segment to prevent itself from hitting the obstacles. [13] A dangerous circle R_d is determined centered on the turning point P_{tp} of the safe free segment around to enable the robot turn safely. [13] To ensure safety, the segment with the distance of S_{Pi} $(i = 1, \dots, n-1)$ is selected which is larger than the robot diameter D_r with a margin for security δ $(S_{Pi} \geq D_r + \delta)$. [13]

The basic principle of this path planning algorithm is illustrated as a flowchart in Figure 4.

When operating this algorithm, only the safe segments are taken into account ignoring danger segments and the point P_{tp} of the safest segment is determined to give the shortest path. [13] This algorithm aims at allowing the robot to reach the goal without hitting the obstacles, and the robot is programmed to search the next new turning point when it reserves the determined turning point. [13]

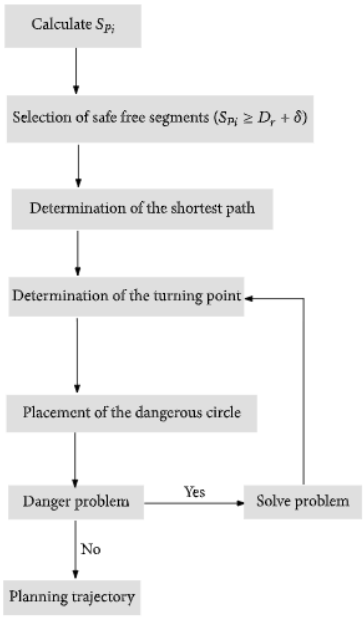


Figure 4: Strategy of path planning algorithm [13]

The challenge for the path planning is to make it computational in "real time" which is required to enable the navigated mobile robot response for the commands instantaneously. This difficulty maybe caused by the memory on small microcontrollers which would limits the maximum number of vertices it could process on. However, this problem is for testing a mobile robot in an actual environment which is a route with a few obstacles, and this project focuses on the software programming. Therefore, the construction of the shortest path planning algorithm and the result testing on the layouts representing the actual routes are the main tasks for this project.

The algorithm for the shortest path algorithm mentioned above takes much computation efforts to process on the free segments between each two obstacles, that makes it more difficult for a route with many obstacles. Therefore, another algorithm is applied to make it easier to operate, this algorithm creates a map with triangular meshes, including a starting point and a destination on the node. The environment for path planning can be either indoors or outdoors, and mostly the indoors are characterized by the presence of obstacles, such as surrounding walls, doorways, furniture or corridors. [5] Most indoor environment contains only fixed obstacles in the terrain, and this experiment will also mainly focus on the routes of a plane with obstacles in different locations. [5]

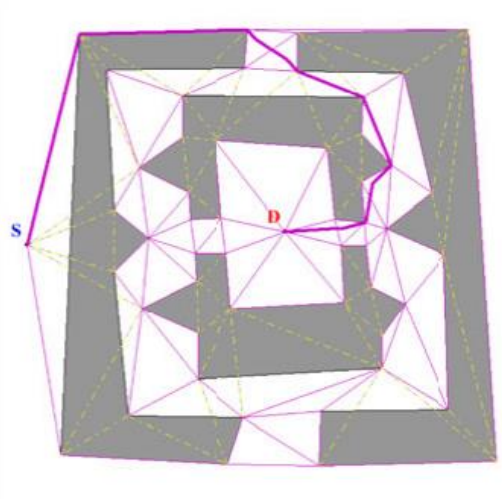


Figure 5: An example shortest path planning [6]

The triangles are the shapes used for the program to generate meshes for the following two main reasons. The first one is that it is an efficient and sensible way of representing the environment in the computer. A triangulated description is adopted because of its flexibility and the better approximations given to curves that rectangles. The other is that the routes are defined to follow edges of triangles in a piecewise linear manner, thus this it suitable for Dijkstra's algorithm which aims at finding the shortest edge-following path. The total number of triangles is no need to be more than necessary which means it is essential to construct an efficient and high-quality triangulation. There are many triangular meshes generating methods, but Delaunay triangulation in this project to provide an "optimal" [14] description and the source codes to generate that could be obtained online.

Figure 5 illustrates the shortest path planning from the starting point (S) to the destination (D) on a plane with obstacles, but note that the path in this figure travels through the triangle which wouldn't happen in this project because the design of edge-following. Delaunay triangulation is implemented on the plane generating the shortest path along the edges which is shown as the dark pine line in Figure 5. To obtain a reasonable result for the performance of shortest path planning, different shapes of routes are used to compare and the edges of the routes are representing obstacles.

This experiment implements the shortest path with the combination of Delaunay triangulation and Dijkstra's algorithm. The software package for Delaunay triangulation is expected to provide the triangular meshes with high quality, and it generates the set of vertices and edges which the wheeled navigation robots are able to travel through. According to the meshes created, the shortest path algorithm which is the Dijkstra algorithm used in this project, is performed along the edges in the meshes.

1.2. Aims and Objectives

The ultimate aim of this project is to design, implement and validate the software package to find the shortest path planning for mobile robot navigation in the most computationally efficient manner. The Delaunay triangulation is performed on an actual two-dimension layout of the actual route, with the vertices representing the points that the robot is able to locate and the edges considered as the possible paths. The shortest path algorithm performed in this project is Dijkstra's algorithm which aims at searching the shortest path through the edges in the meshes from the starting point to the destination. With the connection of Delaunay triangulation and Dijkstra's shortest path algorithm, the whole software package is expected to generate the shortest path between any two points in a 2D route efficiently. Considering the computational efforts for this shortest path planning algorithm, effective factors are tested and the errors of the results are improved to achieve more efficient results.

Stage 1: Delaunay Triangulation and Mesh Construction

Aim:

To construct 2D meshes for a layout of an actual route through Delaunay triangulation using the most suitable factors of triangles;

Objectives:

- Literature review for Delaunay triangulation and understand the working principles and characterize the "goodness" properties of meshes;
- Download the software package of Shewchuck's program of Delaunay triangulation;
- Create the files with necessary values, such as the *x,y* coordinates of each vertex (*.node*) and the vertices on each triangle (*.ele*);
- Generate meshes for a layout and test different values of the area and minimum angle to determine the best mesh factors using the created files;

Stage 2: Dijkstra's Shortest Path Planning Algorithm

Aim:

To find the shortest path between two vertices through the edges given by Delaunay triangulation using Dijkstra's algorithm;

Objective:

- Literature review about the strategy of Dijkstra's shortest path planning algorithm;
- Implement and test a basic example with the direct given values of points and distances;
- Design and develop the connection with Delaunay triangulation and using the values in files generated to perform Dijkstra's algorithm;
- Simulate the shortest path algorithm with various triangle factors to achieve the most accurate result with the least computational efforts consumption;

- Make sure the flexibility of the codes to the shape and triangulation density of the layouts, memory use and speed of evaluation;
- Evaluation of the performance of Dijkstra's algorithm by generating the certain files and plotted that on a graphic software to view the results;

Stage 3: Programming Optimization and Layout Test

Aim:

To optimize the whole program in order to achieve a computationally efficient code and test the results with various kinds of layout;

Objectives:

- Explore the use of STL data structures, for example vectors, to improve the code to be more computationally efficient;
- Two kinds of layouts are defined for the routes to test the universality and feasibility for this algorithm;
- Comparison between the time, memory cost and accuracy for different layouts to validate the whole shortest path program;
- Maintainability of the code by future developers;
- Visualization of the shortest path planning between different two points on various kind of layout to check the results;

Stage 4: Future Development of Shortest Path Planning

Aim:

Discussions and proposals of future shortest path planning for mobile robot;

Objectives:

- Several proposals for improving the shortest path planning for mobile robot navigation, such as O (n log n) near-shortest path algorithm;
- Raise some possible suggestions to save computational efforts when operating shortest path planning;
- Provide a sequence of suggested steps that the future coder could explore more on this shortest path algorithm;

1.3. Report Structures

This report is divided into 6 chapters. The first chapter provides a brief summary of background reviews and introductions of the project. The next chapter gives the working principles of methodologies utilized in this project and the test for example code generated using the relevant techniques. After that, the third chapter present the process of implementing the overall shortest path planning algorithm and the whole code program design. Chapter 4 focuses on the result for this algorithm when testing on two different layouts of routes, *concentric boxes* and *'A' character* respectively. Both of the triangle area and minimum angles are analyzed to make

efficient and accurate triangulation in these two layouts. Besides, the performances of the shortest path planning operating on these two layouts are compared. Then, some possible future works and improvements as well as time management issues will be discussed in chapter 5. The final chapter concludes the process of this experiment and the content of this report. Finally, the list of references is presented at the end of this report.

Chapter 2: Methodologies and Examples Test

The background and need of the shortest path planning algorithm have been introduced in the above section, this section will give the descriptions of the working principles of two methodologies used to design the whole algorithm (Delaunay triangulation & Dijkstra's algorithm). Besides, the examples for both of these two algorithms are tested to check the accuracy and make sure the codes on them could be used to construct the new algorithm.

2.1. Delaunay Triangulation

This section will discuss the role and properties of the Delaunay Triangulation on how the triangular meshes could be generated and why this could useful to make shortest path planning. As this is essential background to understanding, this section will give detailed explanation on the working principles of this algorithm and the criteria to justify the quality of the meshes created.

Delaunay refinement is a technique to generate unstructured meshes of triangles or tetrahedra which is applied to perform finite element method and other numerical methods to solve partial different equations. [14] Delaunay triangulation used to generate unstructured meshes for test layouts is a well-known geometric structure applying Delaunay refinement. In two dimensions, the Delaunay triangulation of a vertex set maximizes the minimum angle of all triangles in the triangulation to avoid the existence of skinny triangles. [6]

In two dimensions, a triangulation is a set T of triangles with a set V of vertices without the intersections between interiors and whose union completely fills the convex hull of V. [14] The Delaunay triangulation D of V, is proposed by Delaunay [16] in 1934, and an example of generating meshes by this triangulation is presented in Figure 6. If two vertices in this triangulation is called u and v, the edge of uv is in the set of D and if there exists an empty circle which is able to passes through u and v, this edge is called Delaunay. [14]

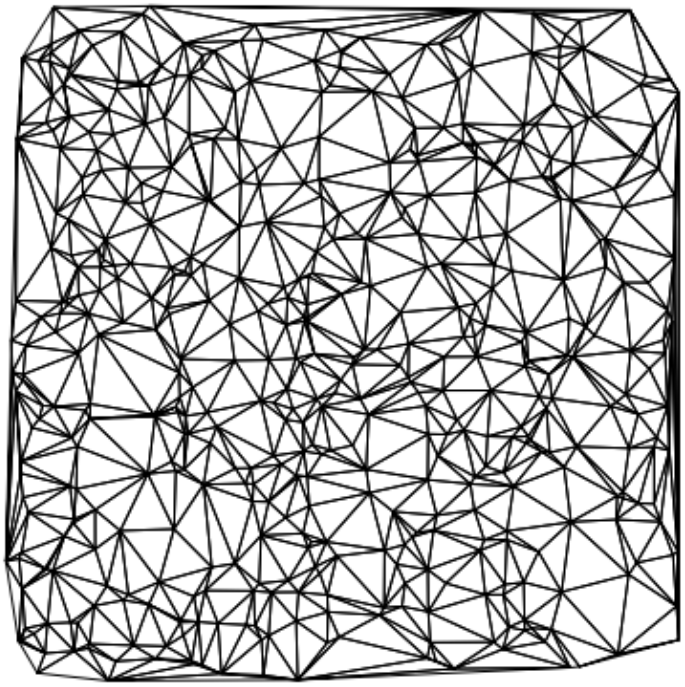


Figure 6: A Delaunay triangulation [14]

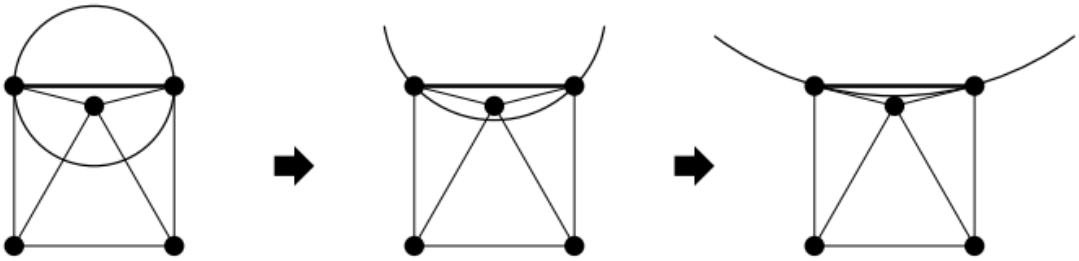


Figure 7: Delaunay of each edge on the convex hull

Figure 7 illustrates how every edge of the convex hull of a vertex set is Delaunay. For a convex hull edge *e*, there is always an empty circle existed which contains *e* from the smallest one to one away from the triangulation. [14] However, this triangulation constructed by the set of Delaunay edges is failed to form in some conditions, for example when four vertices lie on a common circle, because the vertices of *V* are needed to be located in *general position*. [14]

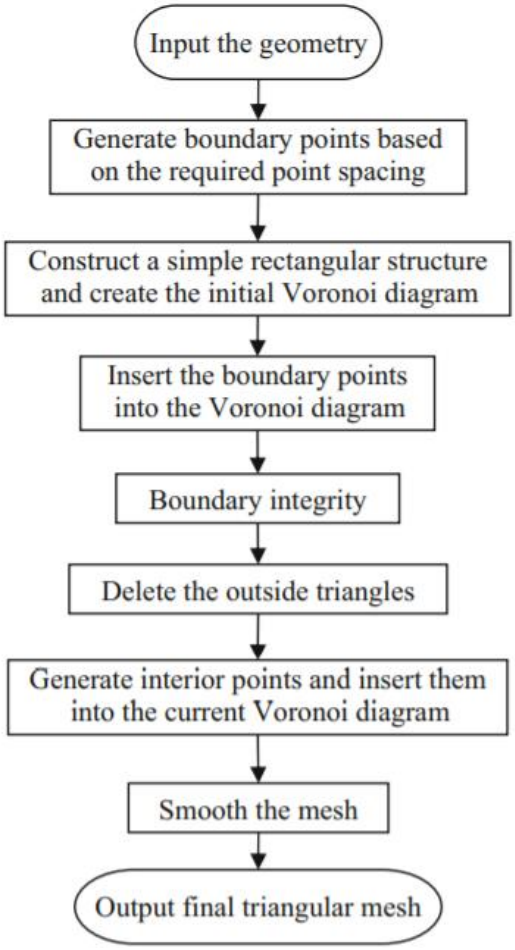


Figure 8: The procedures of 2-D finite element Delaunay triangular mesh generation [17]

The block model for the process of implementing Delaunay triangulation is shown in Figure 8. In this process, the geometric models are input into the analyze and the boundary points according to the spacing requirements of users are generated. [17] The initial Voronoi diagram and a basic rectangular structure are constructed, then the boundary points are inserted into the diagram. [17] The lost boundary edges are identified and these are replaced by adding pseudo-points [18], and the triangles outside are deleted. Next, the interior points are created based on the certain spacing boundary points and they are inserted into the generated Voronoi diagram. [17] Finally, Laplacian method is applied to smooth the mesh nodes and the construction of Delaunay triangulation is completed.

To check the quality of the triangle, two criteria are employed to evaluate the performance of triangulation: (1) the determinant value of Jacobian matrix, which is always called “Jacobian” for short; (2) the minimum angle; [17]

(1) The Jacobian

the Jacobian matrix is examined as an index for the evaluation of the quality of meshes generated by the triangulation. [19, 20] Three points on one triangle are located at

$P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$, and the Jacobian matrix for each point P_i is expressed,

$$A = \begin{pmatrix} x_i - x_k & y_i - y_k \\ x_j - x_i & y_j - y_i \end{pmatrix} \quad \text{Eqn.1}$$

$$\text{Jacobian} = \det(A) \quad \text{Eqn.2}$$

where i, j, k = 1, 2, 3. With j = 2 and k = 3 if i = 1, j = 3 and k = 1 if i = 2, and j = 1 and k = 2 if i = 3.

The minimum value of these three Jacobian values of these three points is defined as the Jacobian of the triangle. If the Jacobian>0, the area of the triangle is a positive value and it satisfies the requirements of mesh generation; if Jacobian=0, the shape of figure generated is a straight line; if Jacobian<0, the triangle is distorted or overlapped. Therefore, the triangles with Jacobian≥0 are able to constitute the triangulation of good quality.

(2) The minimum angle

The minimum angle value is another index to indicate the triangular mesh quality. The minimum angle of a certain triangle is defined as the minimum value of all these three angles:

$$l_i = \min(l_i) \qquad i = 1,2,3 \quad \text{Eqn.3}$$

where l_i is the angle of each triangle i.
When the minimum angle of the triangle is larger, the quality of the triangle is higher. [17]

In this section, the working principle and theory of the Delaunay triangulation have been discussed in details and the next section will focus on how this could be used as an available code for practical work.

2.2. Shewchuck's Program - "a Two-Dimensional Quality Mesh Generator and Delaunay Triangulator"

This section will guide the user through using an available code to generate Delaunay triangles for testing with the route planning software developed in this project. The source code is available as a *.zip* file on the website and it could be embedded, but the license is needed for commercial use due to the copyright of the author.

The program used in this project is provided by Jonathan Shewchuck in 2005, this aims at finding the Delaunay triangulation of a set of vertices. The geometry of the region which is expected to mesh is described in *.poly* file. After the operating process of the program, the

vertices are stored in the .node file and the triangles are stored in the *.ele* file. The Shewchuck's program is prepared to make Delaunay triangulation for the geometric features in the *.poly* file by the command make triangle, as shown in Figure 9.

```
HehongLi@Hehong MINGW64 /e/lhh/FYP/materials/triangle
$ make triangle
make: 'triangle' is up to date.
```

Figure 9: Commands of preparing for making triangulation

Considering the quality of the triangulation, switches are configured to give constraints on the features for the triangle. The switch of -q sets the minimum angle for triangles, which is analyzed above to be an index to evaluate the quality of triangular meshes. Besides, the switch of -a decides the maximum area of triangles effecting the accuracy of the following shortest path planning. The denser lines in triangulation, the smaller error between the experimental values and ideal values. Figure 10 presents an example Delaunay triangulation for the layout of *'A' character*, with the maximum triangle area of 0.01 and the minimum angle of 20 degrees.

```
/e/lhh/FYP/materials/triangle
  Mesh vertices: 107
  Mesh triangles: 132
  Mesh edges: 239
  Mesh exterior boundary edges: 82
  Mesh interior boundary edges: 0
  Mesh subsegments (constrained edges): 82


HehongLi@Hehong MINGW64 /e/lhh/FYP/materials/triangle
$  ./triangle -P A.poly -a0.001 -q20
Opening A.poly.
Constructing Delaunay triangulation by divide-and-conquer method.
Delaunay milliseconds:  0
Recovering segments in Delaunay triangulation.
Segment milliseconds:  0
Removing unwanted triangles.
Hole milliseconds:  0
Adding Steiner points to enforce quality.
Quality milliseconds:  0

Writing A.1.node.
Writing A.1.ele.
NOT writing a .poly file.

Output milliseconds:  1
Total running milliseconds:  2

Statistics:

  Input vertices: 29
  Input segments: 29
  Input holes: 1

  Mesh vertices: 107
  Mesh triangles: 132
  Mesh edges: 239
  Mesh exterior boundary edges: 82
  Mesh interior boundary edges: 0
  Mesh subsegments (constrained edges): 82
```

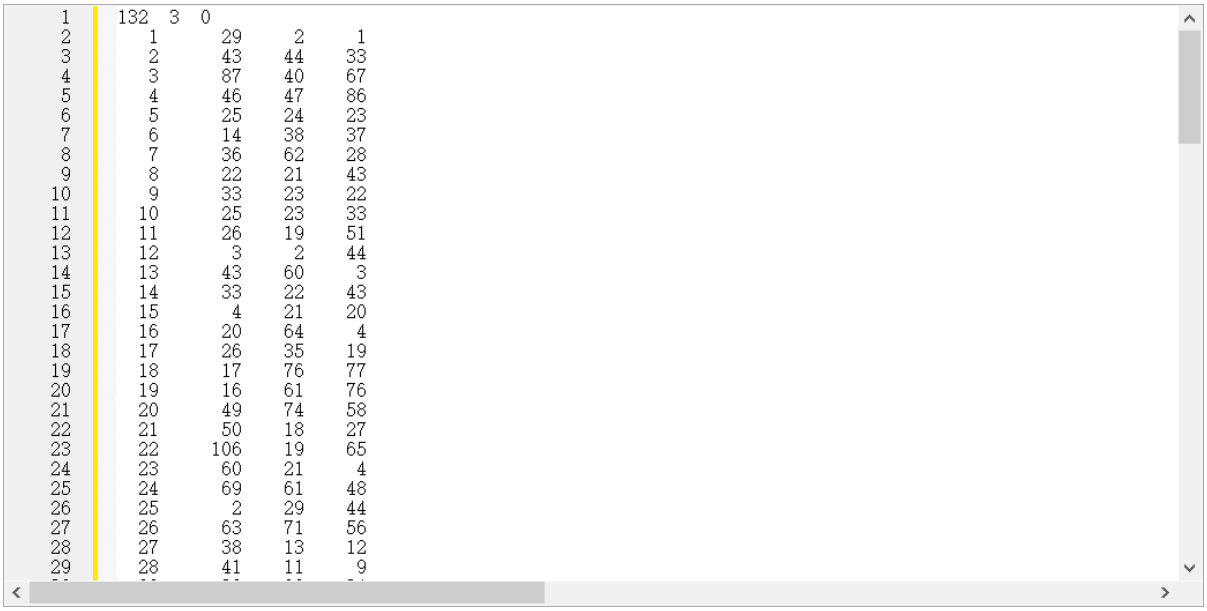
Figure 10: Details of an example triangulation layout of *'A' character*

After operating the Delaunay triangulation on the layout of *'A' character*, two essential files are created. Figure 11 shows the data in *.node* file which is created by the triangulation, and each line gives the information for one vertex excepting for the first line. In the first line, these four values are representing the number vertices, the dimension number which is 2 in this project, attributes and the boundary markers (0 or 1) respectively. The boundary markers are used to identify boundary vertices and vertices on PSLG segments. Besides, the first three values on each line are the data necessary for the following path planning algorithm. The first one is the vertex number, and the next two values are the x and y coordinates of each vertex.

The last two values which are standing for attributes and boundary number do not have effect on shortest path planning algorithm, but they are important to perform Delaunay triangulation and visualize the data for the shortest path planning.

```
107  2  2
   1    0.20000000000000001  -0.77639999999999998  -0.56999999999999995    1
   2    0.22  -0.7732  -0.55000000000000004    1
   3    0.24560000000000001  -0.75639999999999996  -0.51000000000000001    1
   4    0.27760000000000001  -0.70199999999999996  -0.53000000000000003    1
   5    0.48880000000000001  -0.20760000000000001  0.28000000000000003    1
   6    0.50480000000000003  -0.20760000000000001  0.29999999999999999    1
   7    0.74080000000000001  -0.73960000000000004  0    1
   8    0.75600000000000001  -0.76119999999999999  -0.01    1
   9    0.77439999999999998  -0.77239999999999998  0    1
  10    0.80000000000000004  -0.77639999999999998  0.02    1
  11    0.80000000000000004  -0.79239999999999999  0.01    1
  12    0.57920000000000005  -0.79239999999999999  -0.20999999999999999    1
  13    0.57920000000000005  -0.77639999999999998  -0.20000000000000001    1
  14    0.62160000000000004  -0.77159999999999995  -0.14999999999999999    1
  15    0.63360000000000005  -0.76280000000000003  -0.13    1
  16    0.63919999999999999  -0.74439999999999995  -0.10000000000000001    1
  17    0.62080000000000002  -0.68440000000000001  -0.059999999999999998    1
  18    0.58720000000000006  -0.60440000000000005  -0.01    1
  19    0.36080000000000001  -0.60440000000000005  -0.23999999999999999    1
  20    0.31919999999999998  -0.70679999999999998  -0.39000000000000001    1
  21    0.312  -0.73960000000000004  -0.42999999999999999    1
  22    0.31840000000000002  -0.76119999999999999  -0.44    1
  23    0.33439999999999998  -0.77159999999999995  -0.44    1
  24    0.37119999999999997  -0.76239999999999998  -0.40999999999999998    1
  25    0.37119999999999997  -0.79239999999999999  -0.41999999999999998    1
  26    0.37440000000000001  -0.56999999999999995  -0.20000000000000001    1
  27    0.57440000000000002  -0.56999999999999995  0    1
  28    0.47360000000000002  -0.33079999999999998  0.14000000000000001    1
```

Figure 11: Data in *.node* file



```
132  3  0
   1    29    2    1
   2    43    44    33
   3    87    40    67
   4    46    47    86
   5    25    24    23
   6    14    38    37
   7    36    62    28
   8    22    21    43
   9    33    23    22
  10    25    23    33
  11    26    19    51
  12     3     2    44
  13    43    60     3
  14    33    22    43
  15     4    21    20
  16    20    64     4
  17    26    35    19
  18    17    76    77
  19    16    61    76
  20    49    74    58
  21    50    18    27
  22   106    19    65
  23    60    21     4
  24    69    61    48
  25     2    29    44
  26    63    71    56
  27    38    13    12
  28    41    11     9
```

Figure 11: Data in *.ele* file

The first line in the *.ele* file gives the values of the number of triangles, the number of nodes in per triangle which is 3 and the attributes. Similar with *.node* file, the attributes are float-point values of physical quantities (such as mass and conductivity) related to the triangles of a finite element mesh. Each remaining line presents the triangle number as the first value and three vertices of this triangle which are listed in counterclockwise as the next three values.

2.3. Mesh Generation

According to the features of triangles and vertices, a *.tri* file is created and the Delaunay triangulation for the layout of *'A' character*. This .tri file is generated by the combination of *.node* file and *.ele* file with the changes on the values of attributes and boundary markers. The features for the triangulation which are in *.tri* file are visualized through the viewer software in Figure 12.

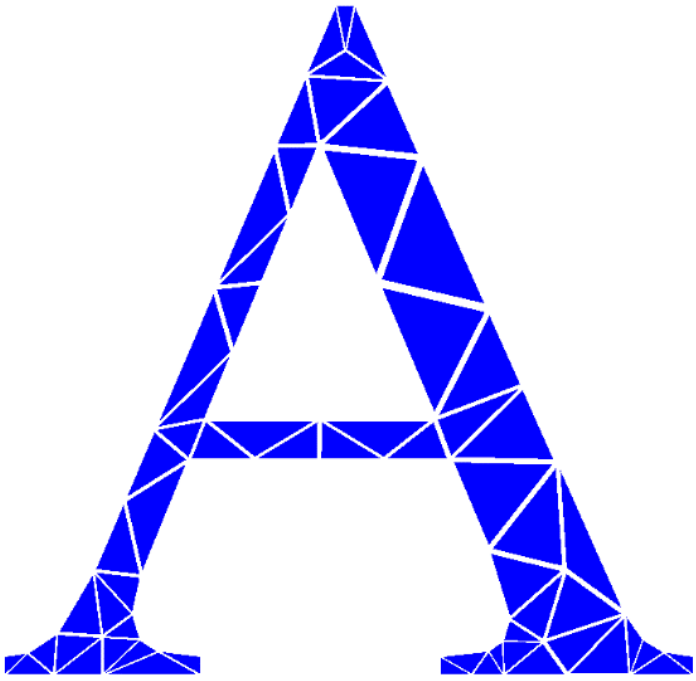


Figure 12: Delaunay triangulation on an example layout

In this Delaunay triangulation, the maximum area of triangle is set to be 0.01 and the minimum angle is 20. As shown in Figure 12, this parameter setting has high quality meshes due to the relatively higher minimum angle used. There are few lines in the triangulation which means when the mobile robot is operating shortest path algorithm, it has fewer choices to select the routes. Therefore, to obtain shortest path planning in high quality, the maximum area is required to be as small as possible according to the limited computational effort by a laptop. After constructing the Delaunay triangulation on the layout of route, the algorithm applied to perform shortest path planning which is Dijkstra's algorithm is analyzed and implemented.

2.4. Dijkstra's Algorithm – the core of the algorithm

After the explanation above, the Delaunay triangulation description of the environment has been obtained, and this section will move on to the next stage in the process – how to find the piece wise linear edge-following approximation of the shortest path. The key algorithm used is the Dijkstra's algorithm. The theory and working principle of this algorithm will be analyzed.

Dijkstra's algorithm is a breadth-first-search (BFS) algorithm to search the shortest path from the source to the destination points. [5, 40, 15] The general principle of Dijkstra's algorithm is constructing the shortest path tree edge by edge based on the distances of each vertex from the source vertex and one edge is added in each process stage. [5] This algorithm is analyzed in an example layout with 6 vertices, as in Figure 13, the distance of each edge and the vertex name are marked on the figure. This algorithm to find shortest path from point A to other vertices is accomplished in the following steps:

(1) Two matrixes are created to record the distances from each vertex to the source vertex, with one (S) containing the name of vertex which has been calculated and the other set (U) of vertices not calculated.

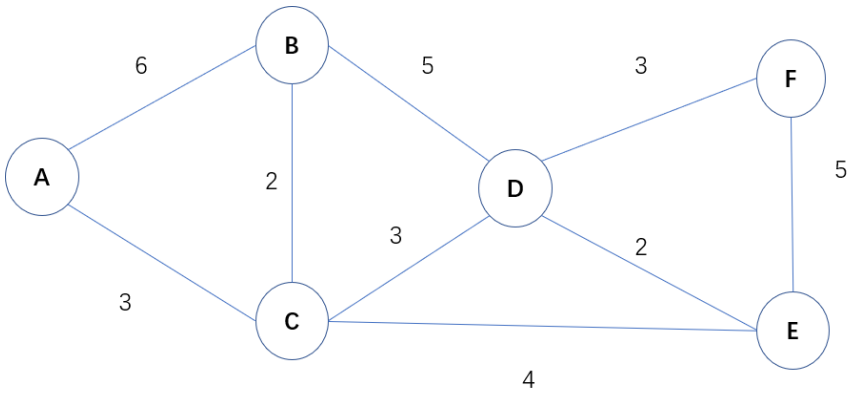


Figure 13: An example layout for Dijkstra's algorithm

(2) In the set of S, when reaching the source vertex, the shortest paths are $A \to A = 0$ and A is the starting point to analyze the surrounding vertices. Hence, $S =< A >$.
In the set of U, vertex A is connected with point B and C and the distances to each of these two points are added into the set. Hence,$U =< B(6), C(3), D(\infty), E(\infty), F(\infty) >$ and the path which has the weights of the shortest is $A \to C = 3$.

(3) In the set of S, when reaching vertex C, the shortest paths are $A \to A = 0$, $A \to C = 3$ and C is the starting point to analyze the surrounding vertices. Hence, $S =< A, C >$.
In the set of U, vertex C is connected with point B, D and E and the distances to each of these two points are added into the set. Because the distance of the path $A \to C \to B = 5 < A \to B = 6$. Hence, $U =< B(5), D(6), E(7), F(\infty) >$ and the path which has the weights

of the shortest is $A \rightarrow C \rightarrow B = 5$.

(4) In the set of S, when reaching vertex B, the shortest paths are $A \rightarrow A = 0$, $A \rightarrow C = 3$, $A \rightarrow C \rightarrow B = 5$ and B is the starting point to analyze the surrounding vertices. Hence, $S = < A, C, B >$.
In the set of U, vertex B is connected with point A and D and the distances to each of these two points are added into the set. Because the distance of the path $A \rightarrow C \rightarrow B \rightarrow D = 10 > A \rightarrow C \rightarrow D = 6$. Hence,$U =< D(6), E(\infty), F(\infty) >$ and the path which has the weights of the shortest is $A \rightarrow C \rightarrow D = 6$.

(5) In the set of S, when reaching vertex D, the shortest paths are $A \rightarrow A = 0$, $A \rightarrow C = 3$, $A \rightarrow C \rightarrow B = 5$, $A \rightarrow C \rightarrow D = 6$ and D is the starting point to analyze the surrounding vertices. Hence, $S =< A, C, B, D >$.
In the set of U, vertex B is connected with point A and D and the distances to each of these two points are added into the set. Because the distance of the path $A \rightarrow C \rightarrow D \rightarrow E = 8 > A \rightarrow C \rightarrow E = 7$. Hence,$U =< E(7), F(9) >$ and the path which has the weights of the shortest is $A \rightarrow C \rightarrow E = 7$.

(6) In the set of S, when reaching vertex D, the shortest paths are $A \rightarrow A = 0$, $A \rightarrow C = 3$, $A \rightarrow C \rightarrow B = 5$, $A \rightarrow C \rightarrow D = 6$, $A \rightarrow C \rightarrow E = 7$. and D is the starting point to analyze the surrounding vertices. Hence, $S =< A, C, B, D, E >$.
In the set of U, vertex B is connected with point A and D and the distances to each of these two points are added into the set. Because the distance of the path $A \rightarrow C \rightarrow E \rightarrow F = 12 > A \rightarrow C \rightarrow D \rightarrow F = 9$. Hence,$U =< F(9) >$ and the path which has the weights of the shortest is $A \rightarrow C \rightarrow D \rightarrow F = 9$.

(7) The set of U is empty, all the vertices are completing the calculations, and the shortest paths are $A \rightarrow A = 0$, $A \rightarrow C = 3$, $A \rightarrow C \rightarrow B = 5$, $A \rightarrow C \rightarrow D = 6$, $A \rightarrow C \rightarrow E = 7$, $A \rightarrow C \rightarrow D \rightarrow F = 9$.

Form the analysis, it can be deduced that the overall program using Dijkstra's shortest path algorithm is required to loop (n-1) times for a layout with n vertices, due to in each loop, it can only determine the shortest path for only one vertex. When performing the Dijkstra's algorithm based on the triangulation for a layout, two main issues need to be considered and analyzed in the experiment. One issue is that when operating the Dijkstra's algorithm, the program executes through all the vertices in the layout, therefore, the computational memory limitations and the running time are the factors have restrictions on the triangulation. Besides, the other issue is the reflection on if the parallelization be able to be utilized in Dijkstra's algorithm.

This section has described the theory and working principle of Dijkstra's algorithm and an example of a layout including six vertices are used to explain how this algorithm work step by step. The next section will give an example program of Dijkstra's algorithm and the results of that program are going to be compared with the results by theory analysis to check accuracy of that program.

2.5 A Basic Shortest Path Example Test

This section focuses on the result testing for an example program of Dijkstra's shortest path algorithm which is generated according to the complete codes online. In order to test the feasibility of Dijkstra algorithm and the code structure of this code structure, a simple example of Dijkstra's algorithm is applied to test. The example is operating by using the matrix of distances from each vertex to the surrounding ones, then the Dijkstra algorithm is designed to work through all values by the steps above. Each two vertices and distance of each edge are given by the user and the shortest path algorithm is applied based on these values. To check the accuracy of this example program, the details of the layout (with vertices A-F) are applied and the vertex names are changed to 0-5.

Figure 14: Details of edges input into the program

In Figure 14, the program allows the user to enter the values for each two vertices of one edge and the distance, and the numbers of overall vertices and edges. When the source is vertex *0*, shortest path from the source to each other point is searched by the program.

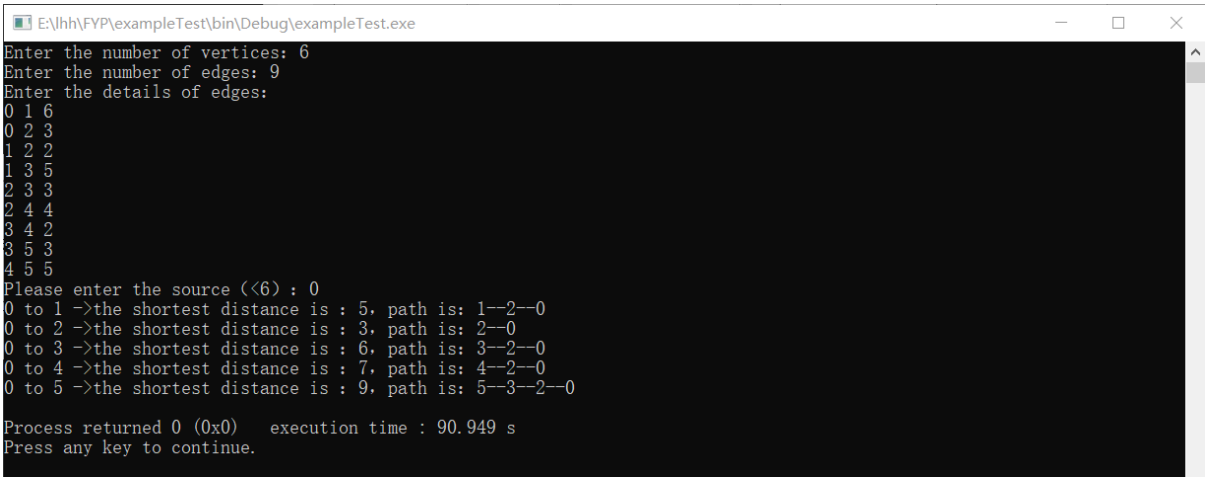


Figure 15: Display of the result of the shortest path algorithm

Each shortest path from source to its corresponding vertex is the same as that in the analysis in 2.4. Therefore, the accuracy of this program is proved and code for the Dijkstra's algorithm is improved and modified for the final path planning algorithm. In this example, the details for

edges in the layout is entered into the program by the user and the shortest paths from source vertex 0 to other vertices (1-5) are calculated by the program in Figure 15.

However, in this project, the coordinates of each vertex and each three vertices for each triangle given by the Shewchuck's program are the input for Dijkstra's program. In other words, the given layout is applying Delaunay triangulation at first, and then the values of vertices and edges generated by the triangulation are used to process the following Dijkstra's shortest path algorithm. Therefore, the connection between files generated by the triangulation and Dijkstra's algorithm is required to be designed and built. Next section will focus on how to construct this and implement the whole shortest path algorithm program.

Chapter 3: Shortest Path Planning Program Design

Having established the theory and description of the algorithms, this section will discuss the practical implement of software design for shortest path planning for navigated mobile robots and the bottlenecks in the process. This design is established on the criteria of efficiency, maintainability, etc. The possible optimization methods of the program will also be analyzed in this section.

3.1 Software Programming

The software program will be designed and constructed in this section in the aspects of program structure and the bottlenecks met during the construction of the shortest path planning algorithm on programming.

3.1.1. Program Structure

The basic program structure of the whole shortest path planning will be analyzed in this part. As discussed before, the Shewchuck's program which is applied to perform Delaunay triangulation generates the coordinates of each vertex and the name of three vertices in each triangle. After operation, the coordinates of each vertex are stored in the *.node* file and the names of three vertices in each triangle are recorded in *.ele* file. In the program, vectors of *vertex* and *triangle* are used to store the elements of these two files respectively.

This path planning algorithm is based on object-oriented programming (OOP) according to its four main principles, polymorphism, encapsulation, inheritance and abstraction. The language used in this project is C++ which supports Object Oriented Design, i.e. it provides the facilities to produce code in the OOP way while C cannot. The object-oriented programming has lots of benefits on the software programs, such as code reusability/extensibility, flexibility,

information hiding and complexity hiding. [21] There are three main advantages: 1. The modular classes allow a relative level of parallel development. 2. It provides more flexibility to make the codes reusable. 3. It is easier to create a maintainable procedure code which could keep the data accessible when it becomes necessary to perform an upgrade. [21]

The properties of OOP play an important role in this shortest path planning algorithm which is reflected in classes. For example, the member functions in class *mesh* are defined to create the matrix, print the shortest path, check input values and perform Dijkstra's algorithm. These functions always work reliably and this design makes the program more robust and less prone to human error.

The program stores the details of each edge in an adjacency matrix and the data is processed through each row and each column.

```
while (count_row != this->numberOfvertex) {
    count_col = 0;
    while (count_col != this->numberOfvertex) {
      if (arc[count_row][count_col] == INT_MAX)
        cout << "-" << " ";
      else
      cout << arc[count_row][count_col] << " ";
      ++count_col;
    }
    cout << endl;
    ++count_row;
    }
```

To construct the connection between two algorithms, the details of each edge (endpoints & length) are calculated to be the input of Dijkstra's algorithm. In *.ele* file, each line expresses three vertices in one triangle, hence each two vertices set one edge. The length of each edge is considered as the distance between two vertices and this distance can be calculated as,

$$length = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} \qquad \text{Eqn.4}$$

where A and B are endpoints of an edge.

Then the value of each corresponding element in this matrix is assigned,

```
        startValue=theTriangleName[i].first;
  endValue=theTriangleName[i].second;

        length=sqrt(pow((theTriangleVertex[startValue-1].x-theTriangleVertex[endValue-
        1].x),2)+pow((theTriangleVertex[startValue-1].y-theTriangleVertex[endValue-
        1].y),2));

  //give values to the matrix
```

```
arc[startValue - 1][endValue - 1] = length;
arc[endValue - 1][startValue - 1] = length;
```

In this matrix, each row number and column number both represent one vertex number from 0 to n, and each value in this matrix is the distance between two corresponding vertices. The Dijkstra's program performs the calculations through each point in this matrix and search for the shortest path from the start point. An array of *dis* is created to store the distances from each point to its surrounding points and the shortest path as following,

```
dis[i].path = to_string(source) + " " + to_string(i + 1)+" 1\n"+to_string(i + 1);
dis[i].value = arc[source - 1][i];
```

The shortest path for the start point is set to be 0 and that for the rest vertices (*this-> vexnum-1*) are calculated in a *while* loop. A variable *temp* is used to save the vertex number with shortest distance in the *dis* array and a variable *min* is to record the shortest distance value.

```
for (i = 0; i < this->numberOfvertex; i++) {
    if (!dis[i].visit && dis[i].value<min) {
        min = dis[i].value;
        temp = i;

    }
}
```

Then the vertices corresponding to *temp* are put into the set of visited points, by setting *dis[temp].visit = true*. As the process of Dijkstra's algorithm analyzed before, if the new edge obtained in the next step has effect on the shortest paths for unvisited points, the shortest paths and lengths are updated.

```
for (i = 0; i < this->numberOfvertex; i++) {

    if (!dis[i].visit && arc[temp][i]!=INT_MAX && (dis[temp].value + arc[temp][i]) <
            dis[i].value) {

        dis[i].value = dis[temp].value + arc[temp][i];
        dis[i].path = dis[temp].path + " " + to_string(i + 1)+ " 1\n"  +to_string(i + 1);

    }
}
```

It is worth noting that the condition of *arc[temp][i]!=INT_MAX* is required to avoid data overflow which will cause a program exception. After the execution of the whole program, the shortest paths from start point to point *i* are given by *dis[i].path* and the shortest distances are presented by *dis[i].value*. To visualize the results and make the error checking, the details for

the meshes with shortest path of special colors are plotted in a viewer software.

3.1.2. Bottlenecks

While constructing the structures for this software programming, the main bottlenecks met is how to build connection for these two algorithms. The inputs for Dijkstra's algorithm are distances from each two vertices, while the outputs provided by triangulation are x,y coordinates for each vertex and three vertices for each triangle. This problem when building this connection is how to store the values in vector *vertex* and *triangle* and use these values to calculate the distance of each edge for Dijkstra's algorithm.
To store the values in vectors, values are read from two files of *.ele* and *.node* by *eleFile.open ("A.1.ele")* and *nodeFile.open("A.1.node")*. As shown in section 2.2, in *.node* file, five values in each line are vertex number, x coordinate, y coordinate, attributes and boundary markers respectively. On each line, only first three values for vertex number, x and y coordinates are required to calculate the distances for edges, hence the last two values are deleted by hand and then values of each line are stored in a vector using *nodeFile.get()*. In *.ele* file, four values for each line are triangle number and three vertices on the corresponding triangle, and these values are stored in a vector by *eleFile.get()*.

To obtain the values of details for each edge, each two values in vector *triangleName[i]* are two vertices (e.g. *a* and *b*) of one edge and their coordinates in vectors *triangleVertex[a]* and *triangleVertex[b]* are used to calculate the distance of this edge. Then the value of its corresponding point in this matrix are assigned by *arc[a - 1][b - 1] = length,* and because there is no direction in this graph, another point is also assigned by *arc[b - 1][a - 1] = length*. After that, each point in this matrix is analyzed and the shortest path for each point are calculated.

Code debugging is also a time-consuming difficulty when testing the program and the whole program is divided in part to find and correct the errors. During the process of constructing this program, most errors happen due to the wrong methods of using vectors. Different functions to create and make use of vectors are applied in the program to test the feasibility and availability to perfect the codes. Another error which has been figured out is the mistakes in reading values from the *.node* file due to the long space from coordinate values to attribute values. To avoid this, the useless values which are attribute values and boundary markers are deleted by hand.

After breaking through these bottlenecks, this software program is designed and implemented and provides the desired values. The whole shortest path planning program is designed to produce a *.txt* file of edges presenting each two vertices of one edge and the edges on the shortest path are shown using different color value, as Figure 16.

```
1 1 2 1
2 1 29 0
3 2 3 1
4 2 29 0
5 2 44 0
6 3 43 0
7 3 44 0
8 3 60 1
9 4 20 0
10 4 21 0
11 4 60 1
12 4 64 1
13 5 6 0
14 5 40 0
15 5 59 0
16 6 59 0
17 6 68 0
18 7 8 0
19 7 48 0
20 7 52 0
21 7 69 0
22 8 9 0
23 8 41 0
24 8 48 0
25 9 10 0
26 9 11 0
27 9 41 0
28 10 11 0
```

Figure 16: Few example data in the *.txt* file produced

In this file, the final value for each line is representing the color for each edge in the meshes by triangulation which is 1 on the shortest path while others are 0. These data are added to the end of *.node* file to create a *.plsg* file which could be plotted in the viewer software for testing. An example of *.plsg* file which is used for testing the results is in Figure 17, in which the red line represents the shortest path while other edges are plotted using blue lines.

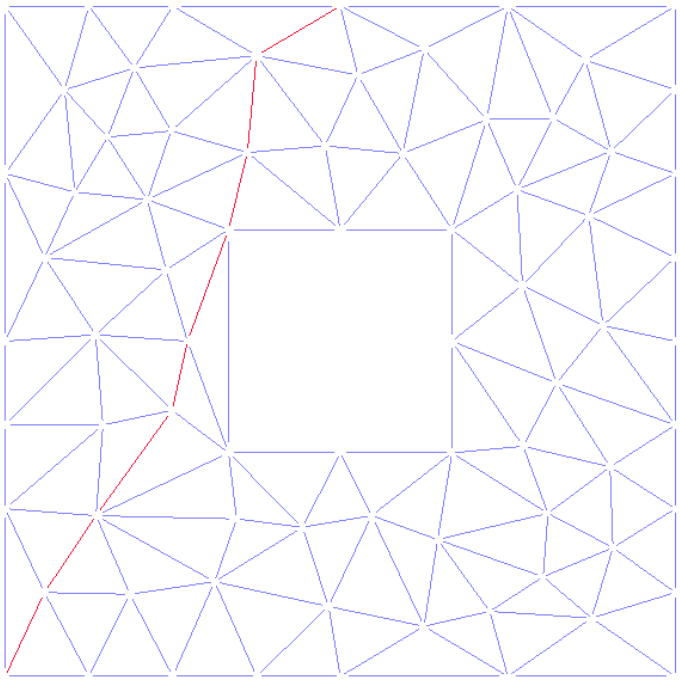


Figure 17: An example of *.plsg* file for result testing

3.2. Optimization Analysis

Code optimization is a necessary step in the program development which makes modification on codes to improve quality and efficiency. In this section, the code optimization will be discussed through three main aspects, STL data structures, parallelization and platform portability for MSYS2 or Linux.

3.2.1. STL Data Structures

The Standard Template Library (STL) [23, 24, 25] is a C++ library which is a set of C++ template classes providing well-structured generic C++ components, such as lists, stacks, arrays, etc. [22] STL has four main components which are algorithms, containers, functions and iterators. Algorithm is defined as a collection of functions which is acting on containers and providing operation methods for the content of containers. The containers, also known as container classes, are used to store objects and data, and four classical containers are vector, list, set and map. The STL has classes which could overload the function call operator and the instances of such classes called functors customize the associated function. The iterators are used to work upon a sequence of values which are the major features that allow generality in STL.
The STL vector implements a dynamically resizable array which allows push, pop, and indexing operations. [27] The push and pop operations take amortized constant time and the indexing operation constant time. The same as C++ arrays, linear work and span are required for the initialization of vectors. [27] The removal of an element from a vector takes constant time, because it has no need to resize the vector. However, the insertion or deletion of an element at the beginning of a vector takes linear time.

The differences between vector and array are shown in following table,

vector	array
A sequential container to store elements and not index based	Storing a fixed-size sequential collection of elements of the same type and index based
Dynamic, increasing size with insertion of elements	Fixed size, cannot be resize after initialization
More memory occupation	Memory efficient data structure
More time cost in accessing elements	Accessing elements in constant time

From the table above, the iterator vector allows the code to be more reusable, solid and robust.

When reading the values in the *.node* and *.ele* files, the maximum sizes of vectors cannot be fixed in the program due to the unpredictable files produced by Delaunay triangulation. C++ vectors could be resized which provides better flexibility to handle dynamic elements [26], hence in this project, vectors are more suitable used to store the details of vertices and triangles.

In this project, the containers of vectors are used to store the details for vertices and triangles in meshes. Vectors work similar with dynamic arrays which has the ability to resize itself automatically when inserting or deleting an element, with automatically changed storage by the container. The elements are accessed and traversed into vectors through iterators, such as *begin( )*, *end( )*, *cbegin( )*, *rbegin( )*, etc. The path planning program uses the function *push_back( )* to put new elements in vectors and *.size( )* to limit the maximum number of elements that vectors can hold when accessing elements. In the member functions of class *mesh*, to call the vectors and use their elements in further calculations, the operators are added as *vector<vertex> &theTriangleVertex* and *vector<triangle> &theTriangleName,* while the values of elements in vectors are assigned in *main* function.

3.2.2. Parallelization

In the simplest sense, parallel computing [28, 29] is the simultaneous use of multiple computing resources to solve a computational problem. The principle is breaking a problem into discrete parts to solve the concurrently, and each part is broken down to a series of instructions which execute simultaneously on different processors. [28] The overall coordination mechanism of parallel computing is in Figure 18.

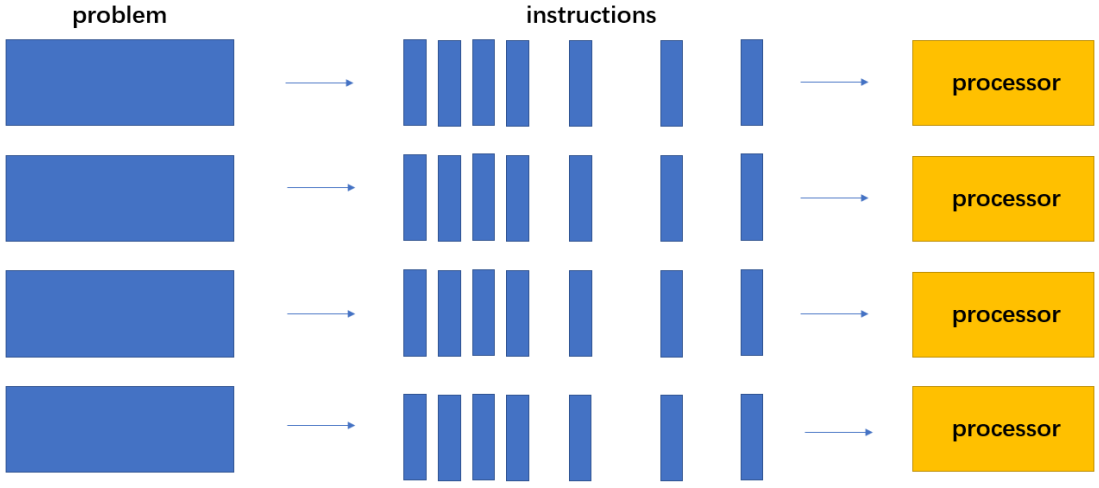


Figure 18: Parallel computing mechanism

The whole shortest path planning program completed above is a serial shortest path, to save time and improve efficiency, the parallel computation for Dijkstra's algorithm is analyzed. Two possible parallelization strategies are considered: 1. Using a different processor to execute each of the n shortest path problems. 2. Increasing concurrency which is the source parallel by a parallel formulation of the shortest path algorithm. [30] The more popular one for parallelization of Dijkstra's algorithm is using OpenMP which starts with a single thread, the

master thread. [30] The OpenMP application operates based on the hybrid model of parallel programming which is able to continue running on a computer cluster by OpenMP, so that it is utilized for parallelism inside a (multi-core) hub. [32] During the execution of the program, the application may work through the parallel regions where the thread teams are produced by the master thread. [30] After the parallel region, the program continues execute on the master thread and the thread teams are stopped. [30]

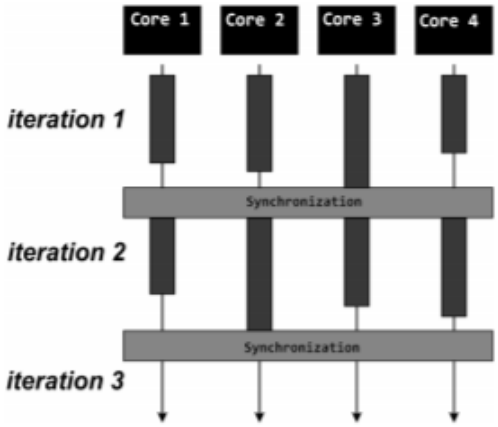


Figure 19: Example of iterations during execution of parallelized Dijkstra's algorithm on four processors using OpenMP [30]

In the sequential shortest path algorithm completed above, it costs long time to search for the shortest distances along all sets of vertices in the mesh graph. [31] Therefore, it's a difficulty to assign the locations of the nearest nodes from source to the goal. [31] When the graph solving problems increase in size, developing effective parallel shortest path has more demands due to the increment on computational and memory prerequisites. [31] The availability of the parallelization of Dijkstra's algorithm was indicated in the analysis above, however, it hasn't been practiced in this project as the limitation of time.

3.2.2. Portability for MSYS2 or Linux

MSYS2 is an emulator of Linux that is a collection of tools and libraries which provides an environment for building, installing and running native Windows software. For independent software vendors (ISVs) who would like to support Linux platform in their products, the portability of applications across various Linux distributions becomes a major problem. [33] The *#undef LINUX* directive is added at the beginning of the whole shortest path program which enable the program compile on Linux environment. Then the main .c file for this shortest path planning algorithm is tested to be running on MSYS2, the result is the same as that works on windows through the software *CodeBlocks*. Therefore, this program is proved to be able to compile on Linux operating system and the portability of the program is improved.

Chapter 4: Result Analysis

After working out the overall software program for shortest path planning algorithm, the results for different layouts are illustrated in this section. The performances of the results of two different layouts (one simply and the other complicated) are compared to verify the generality of this path planning algorithm. Besides, the practical parameters' effect on errors of results will also be discussed.

4.1. Layout of *concentric boxes*

A simply layout of two concentric boxes is defined to examine the feasibility and accuracy of this path planning algorithm by testing the shortest path from source to different destination nodes. The parameters used for Delaunay triangulation which are typically the maximum triangle area and the minimum triangle angle are evaluated based on the computational efforts.

4.1.1. Parameter Analysis

In Delaunay triangulation, more vertices are generated when the maximum triangle area is assigned a smaller value which is demonstrated to cost more time and memory to process. As shown in Figure 20, the number of vertices generated by Shewchuck's program is scaled with the maximum triangle area by approximately -*O(N)*.

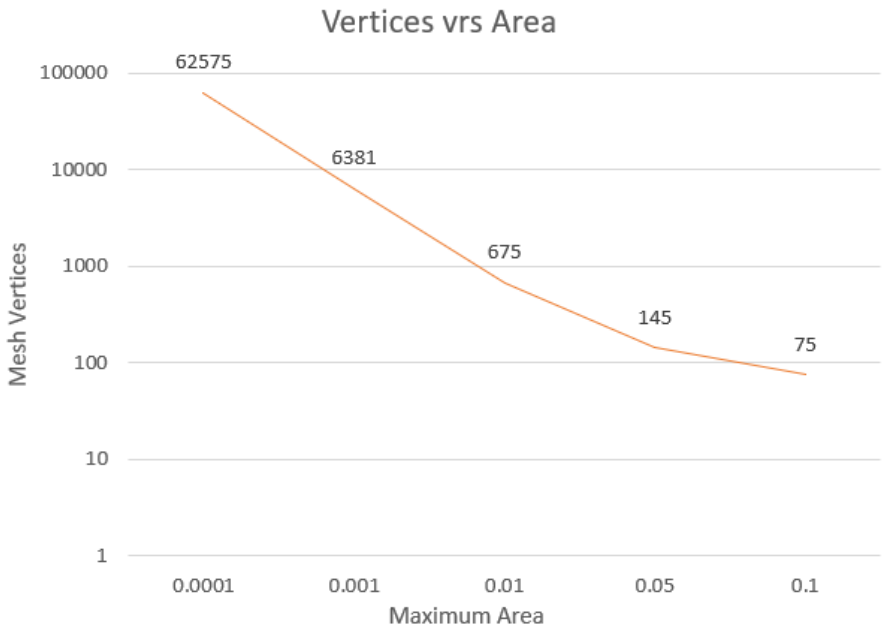


Figure 20: Number of vertices generated versus maximum triangle area (for *concentric boxes* layout)

As the consideration of limitation on time cost and computer memory, the maximum triangle

area for the triangulation needs to be set in a suitable range. The layouts for Delaunay triangulation and their computational effort requirements using different maximum triangle area are compared to obtain a suitable value for the further experiments. In this project, the properties of the laptop used are 16GB memory and 6 cores. When the minimum triangle angle is fixed to be 20 degrees, the Delaunay triangulation of the *concentric boxes* layout using four various maximum triangle areas (0.001, 0.01, 0.05 and 0.1) for are compared and the value has the highest efficiency of path planning will be used in the further analysis. These layouts can be simply generated by Shewchuck's program and the maximum triangle area parameter is changed by setting various values on the *-a* switch.

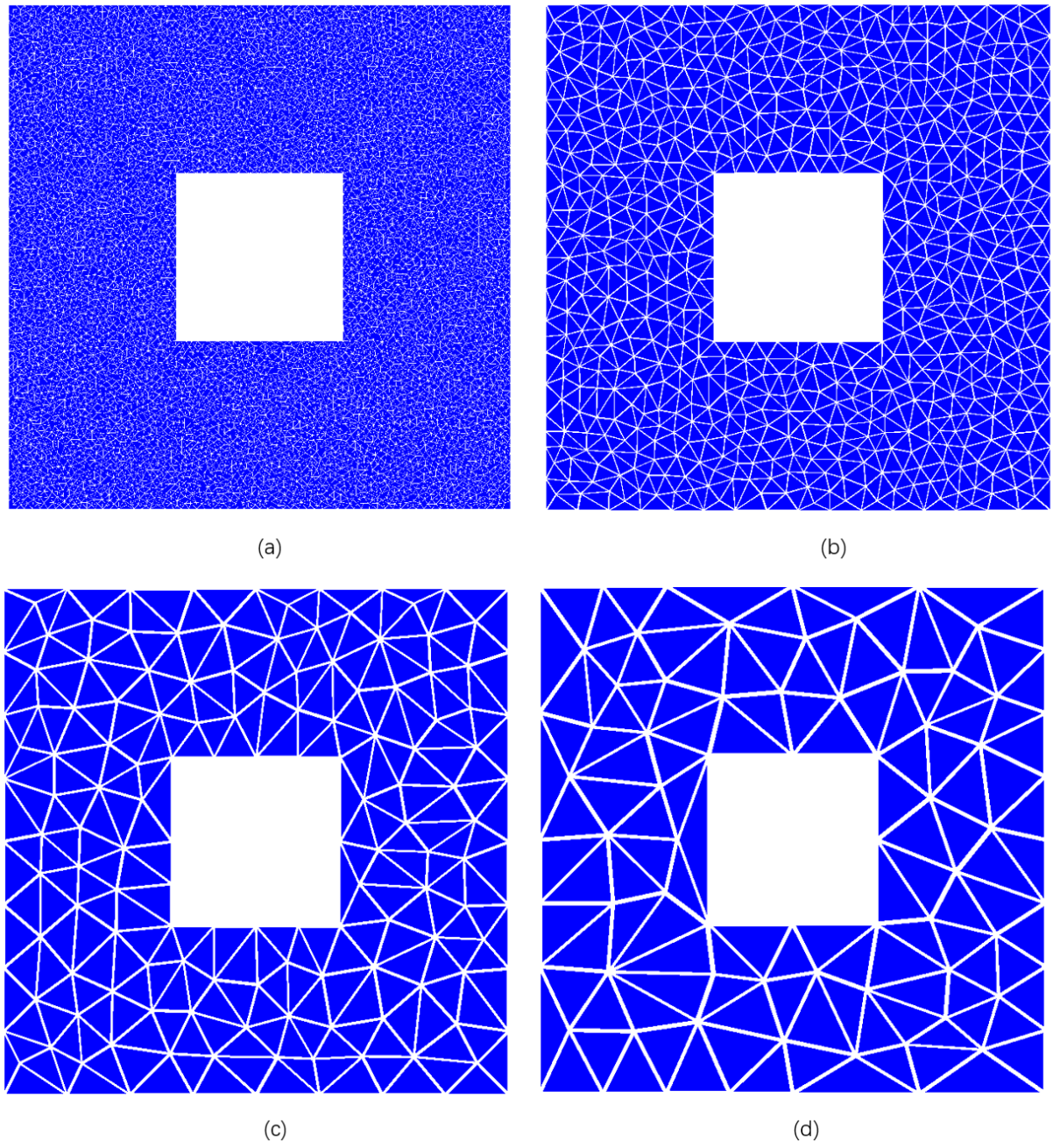


Figure 21: Delaunay triangulation using various maximum triangle areas with fixed minimum triangle angle of 20 degrees: (a) a=0.001; (b) a=0.01; (c) a=0.05; (d) a=0.1 (for *concentric*

boxes layout)

When the parameter of maximum triangle area becomes smaller, the graph is more complicated to be processed due to the increment on the number of vertices as in Figure 21. The triangle area of triangulation is an essential factor of the accuracy of the shortest path planning algorithm that the increase of triangles in meshes is predicted to improve precision of the results. Due to a few handworks required for the files created by Shewchuck's program to implement the combination between Delaunay triangulation and Dijkstra's algorithm, the number of vertices has the limitation of the maximum value as the consideration of time costing.

Figure 21(a) and 21(d) have the highest and lowest number of triangles in the meshes, hence they are obviously not the best choice for the value of triangle area used to create the triangulation. Because mesh with massive vertices requires much more calculation work according to the working principle of Dijkstra's algorithm which process the shortest path searching on each vertex and as shown the mesh generated using *-a0.1* has few edges for the mobile robot to travel which will result in low accuracy for the path planning. For the other two layouts with the maximum triangle area of 0.01 and 0.05, after processing the whole shortest path algorithm, time cost on calculation for Figure 21(b) is more than 30 minutes which is not an adaptive choice to test the results for many destinations. Therefore, the constraint of maximum triangle area is set to be 0.05 in the further analysis on the other influential parameter of triangulation and results for different destinations.

The other parameter has impact on the Delaunay triangulation is the minimum triangle angle value, an index to indicate the triangle mesh quality, which has been demonstrated in Section 2.1. As discussed before, the larger value of the minimum triangle angle causes lower quality of triangulation, hence mostly the range of minimum triangle angle constraint is 20-30 degrees. The line chart in Figure 22 illustrates the relationship between the number of vertices in meshes generated and the minimum triangle angle values.

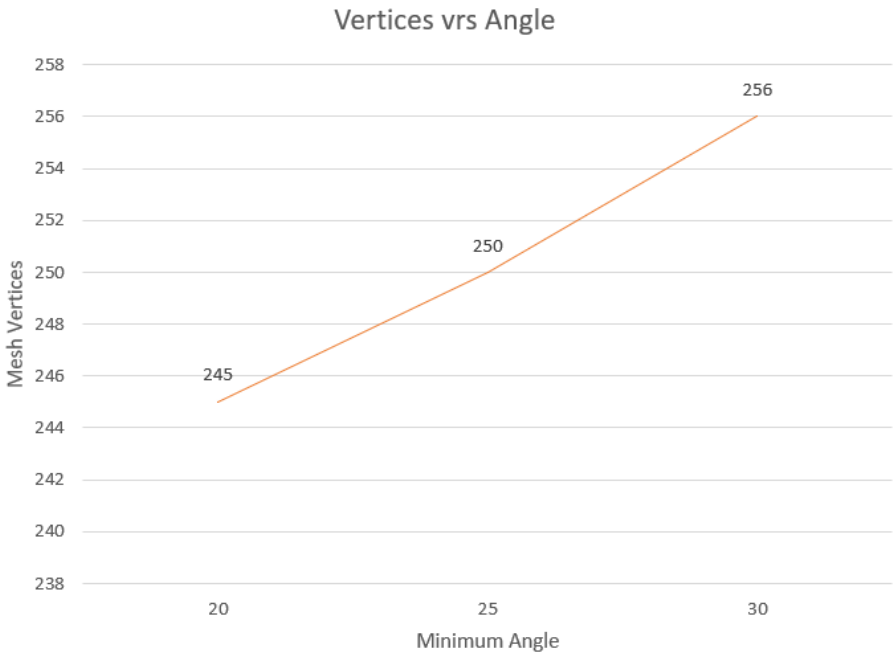


Figure 22: Number of vertices generated versus minimum triangle angle (for *concentric boxes* layout)

In Figure 22, three examples of triangulation with the minimum triangle angles of 20, 25 and 30 degrees are analyzed. The number of vertices is observed to increase proportional to increment on the minimum triangle angle constraint in the generation of meshes. However, the increasing number of the number of vertices generated is approximately the same value as the increment on minimum triangle angle which are small values, hence the differences on the number of vertices when the angle value varies from 20 to 30 are not obvious. But the discussion on this parameter should be carried out in conjunction with the layouts of their triangulation in Figure 23, where the other parameter, maximum triangle area, is set to be 0.05.

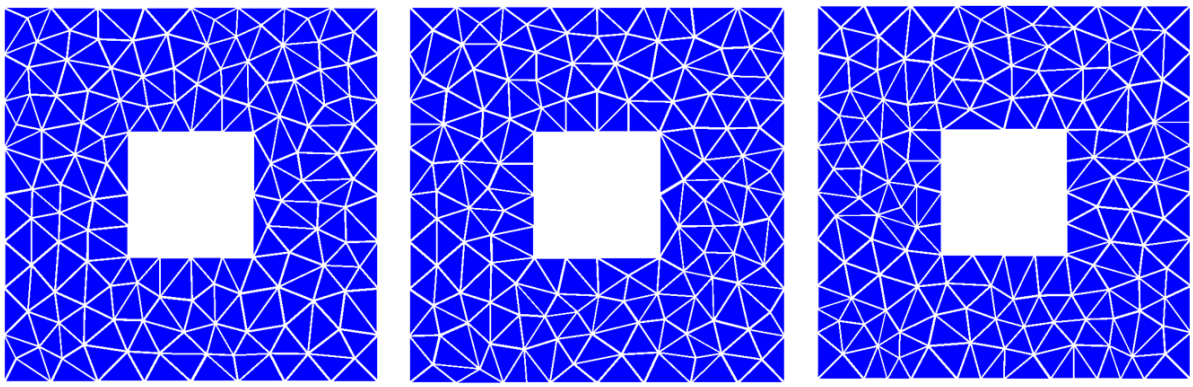


Figure 23: Delaunay triangulation using various minimum triangle angles with fixed maximum triangle area of 0.05: (a) q=20; (b) q=25; (c) q=30 (for *concentric boxes* layout)

As shown in Figure 23, these layouts for triangulation using minimum triangle angle of 20, 25

and 30 degrees show tiny difference among each other due to the similar values of the number of vertices generated. However, mesh with more vertices takes more time for the program to process, hence for these three similar layouts, the one with fewer vertices which has the minimum triangle angle constraint of 20 degrees is used in the calculations of shortest path planning algorithm next.

The most suitable values of two parameters which are maximum triangle area *a* and minimum triangle angle *q* are obtained in this section. Therefore, when creating Delaunay triangulation for *concentric boxes* layout, the commands on MSYS2 are *-a0.05* and *-q20*. In the next section, the overall shortest path algorithm will be applied this triangular mesh and the result for the path planning will be presented on the layouts.

4.1.2. Shortest Path Layout

After constructing the whole shortest path planning algorithm, the program is expected to find the shortest path from source to each other vertex using the triangulation with certain triangle properties. In the first step of this program, it asks the user to enter the values of the number of vertices, edges and triangles which are all obtained from Shewchuck's program and these values are shown on MSYS2. For the *concentric boxes* layout, these three values used are 145, 390 and 245 respectively as shown in Figure 24.

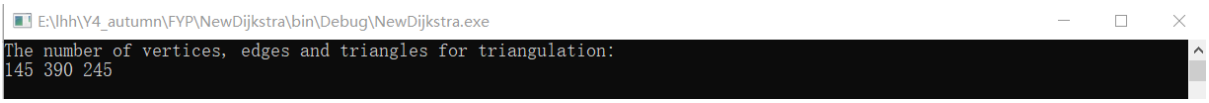


Figure 24: The display of the first step of this shortest path planning program (for *concentric boxes* layout)

To confirm the generality of this shortest path algorithm that it could be used between any two vertices, four different destinations are tested to generate the shortest path from the source. Four layouts in Figure 25 illustrate the shortest paths from vertex No.1 to four vertices of the interior square box from the bottom two to the top two. From the *.txt* file generated by the program, the final values of the lines which represent the edges on shortest path are changed to "1" producing the red lines by hand, while other lines are all blue lines with final value of "0". With the combination of its *.node* file and this *.txt* file, a *.plsg* file is created to visualize the shortest path planning, where the red lines represent the shortest path calculated.

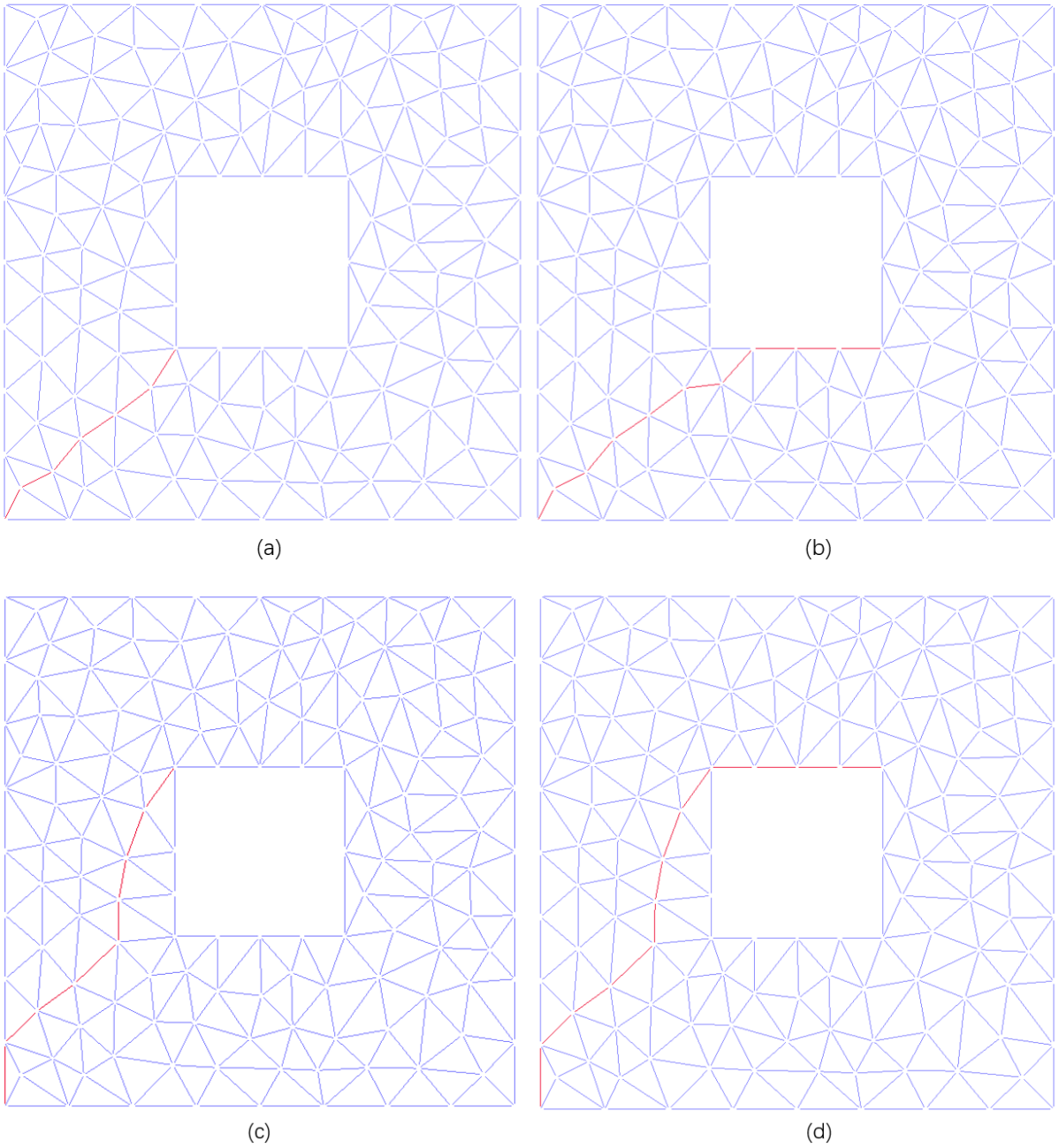


Figure 25: Shortest path planning from source (vertex No.1) to four other vertices (for *concentric boxes* layout) with maximum area of 0.05

According to the common sense, the shortest path between two vertices without obstructions on the way is the straight line. As shown in Figure 25(a) and (c), there are no obstacles between the source and destinations, but the shortest paths generated are obviously not straight lines. This error is considered to be caused by the lack of triangles produced by the triangulation due to the limitation of time cost and computer memory. As the maximum triangle area of 0.05 is the minimum value could be used which has been discussed above, and to prove the prediction of the source of this error, the maximum triangle area of 0.1 is used to compare.

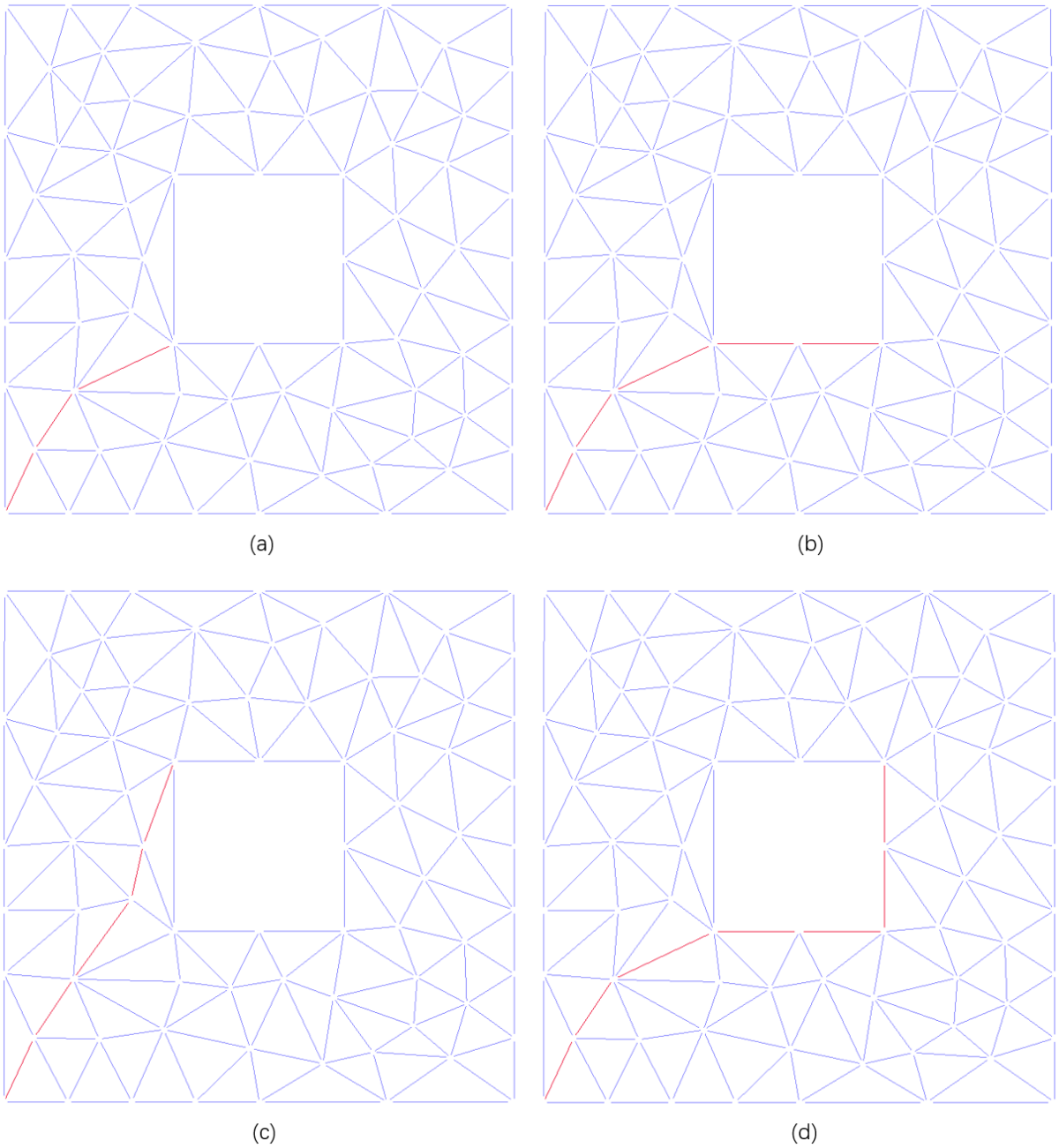


Figure 26: Shortest path planning from source (vertex No.1) to four other vertices (for *concentric boxes* layout) with maximum area of 0.1

When the factor of maximum triangle area is 0.1, the shortest distances calculated by the program from the source to four various destinations are 1.485, 2.485, 2.26 and 3.485 for layouts in Figure 26(a)-(d) respectively. These distances for layouts with the maximum triangle angle of 0.05 are 1.45, 2.417, 2.143 and 3.382 in Figure 25(a)-(d) respectively. It is indicated from both these two figures and the values of distances that smaller triangle area provides more accuracy of the results of shortest path planning, hence the source of the error is probably the lack of triangles in Delaunay triangulation. The processing time for shortest path algorithm on triangulation with maximum triangle area of 0.05 is 16.839s, while that for area of 0.1 is 16.283s, and it is consistent with the previous speculation that more vertices will require more

time for the operation of the shortest path planning program.
This section discussed the results of the shortest path planning on the layout of *concentric boxes* by analyzing the effect caused by two practical parameters and the possible sources of errors. To ensure this program is feasible for any route with various kinds of obstacles, another layout is defined to test the accuracy of the shortest path planning results.

4.2. Layout of *'A' character*

The other layout defined for testing this shortest path planning algorithm is a more complicated one which is the *'A' character* and the versatility of this program is examined by applying various kinds of layout. The practical parameters used to generate Delaunay triangulation and possible sources of errors of the results will also be analyzed in this section.

4.2.1. Parameter Analysis

It's the same as the layout of *concentric boxes* that two main parameters have significant effect on the Delaunay triangulation which is generated by the Shewchuck's program using simple commands. The first factor is the maximum triangle area in the meshes and the relationship between the value of this factor and the number of vertices generated are illustrated in Figure 26. As discussed before, Dijkstra's algorithm needs to process each vertex to find the shortest path from a single source to another node destination. Therefore, the number of vertices in this analysis represents the computational effort cost during the execution of this shortest path planning program to demonstrate how it is related to the parameter of maximum triangle area.

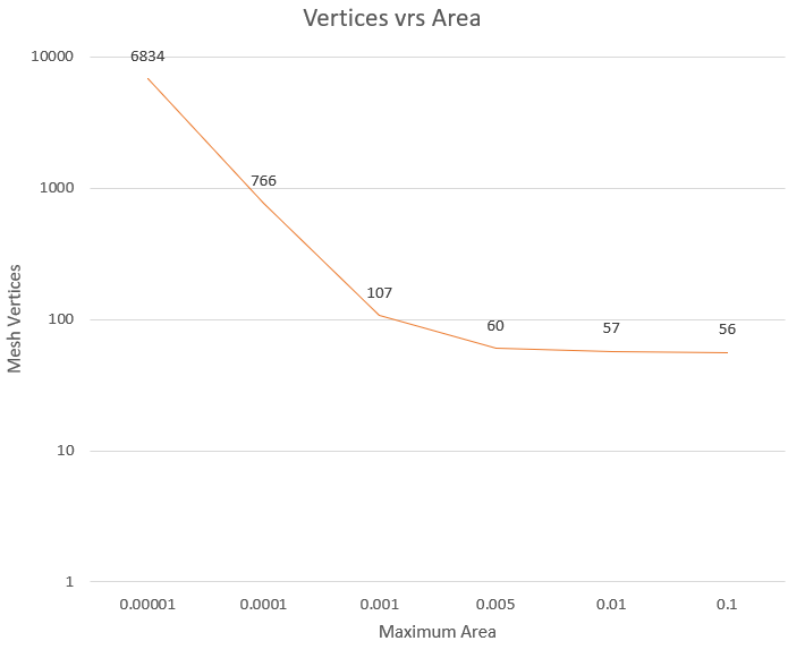
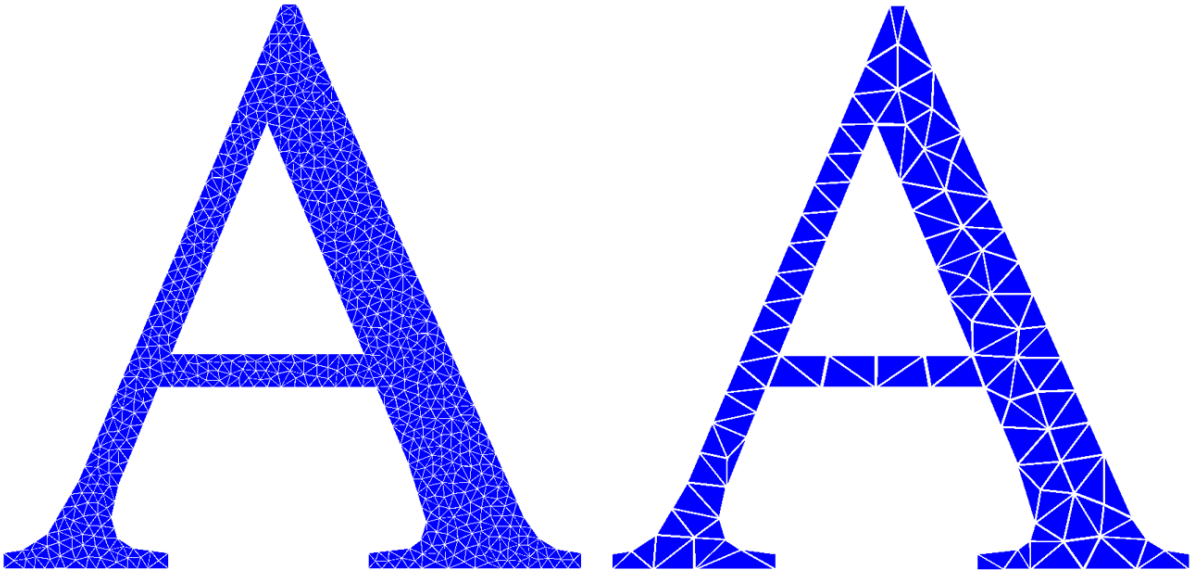


Figure 26: Number of vertices generated versus maximum triangle area (for *'A' character* layout)

As shown in Figure 26, when the maximum triangle area is smaller than 0.05, the number of vertices generated increases as the maximum triangle area decreases scaled by approximately -*O(N)*. The numbers of vertices created by Shewchuck's mesh generator using the values of this parameter larger than 0.05 have similar values at about 60, hence these values are not suitable choices used in further analysis. The computational effort requirement – computer memory and time cost are also important factors when processing the shortest path algorithm. At first, the value of the other effective parameter, minimum triangle angle, is fixed to be 20 degrees and four layouts using various maximum triangle areas (0.0001, 0.001, 0.005 and 0.01) to perform Delaunay triangulation are compared to decide a proper value for testing results. In the commands for running Shewchuck's program on MSYS2, the value of maximum triangle area is changed according how the switch -*a* is set.



(a) (b)

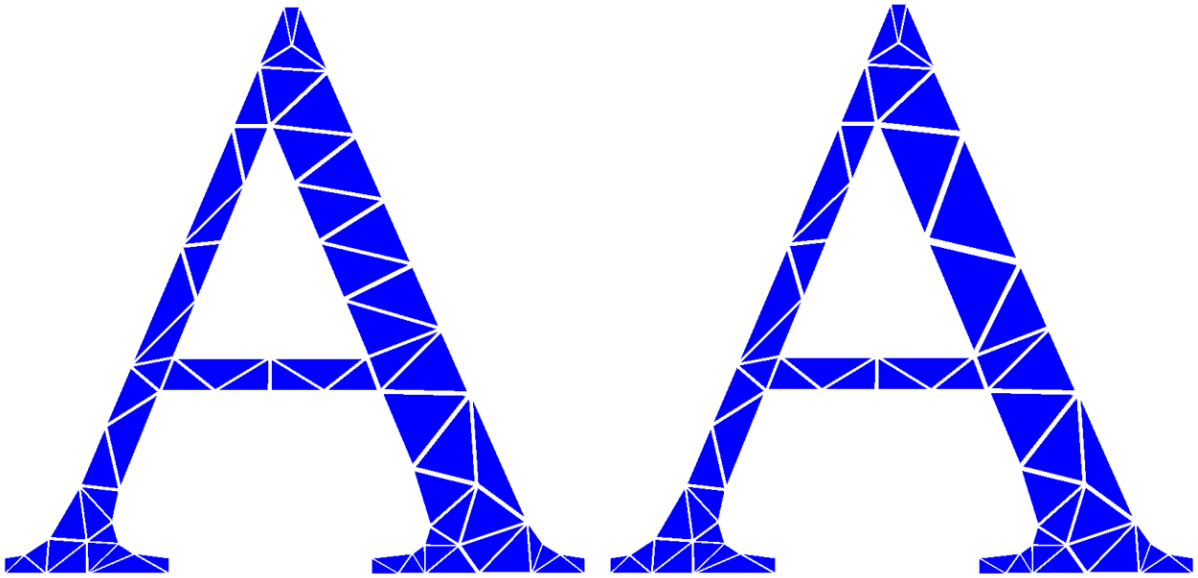


Figure 27: Delaunay triangulation using various maximum triangle areas with fixed minimum triangle angle of 20 degrees: (a) a=0.0001; (b) a=0.001; (c) a=0.005; (d) a=0.01 (for *'A' character* layout)

It is obvious that the triangulation generated using the maximum triangle areas of 0.005 (shown in Figure 27(c)) and 0.01 (shown in Figure 27(d)) have the similar meshes which is consistent with the previous recordings of similar number of vertices produced. Due to the time cost on program processing and the handworks required for the modification on related files, massive vertices in meshes will be complicated to calculated by shortest path algorithm. Therefore, the densest mesh in Figure 27(a) is not the best value used to test results of shortest path planning from one single source to any other vertices, and when applying on the shortest path planning algorithm, it takes much time to process which means lower working efficiency on this project.

As discussed before, the accuracy of the shortest path planning between two vertices based on the triangulation is proportional to the number of edges on the meshes. Layouts of Figure 27(c) and 27(d) consists of lower values of the number of edges and vertices which lack of the lines enabling the mobile robot to travel through, and consequently the results of shortest path planning have lower accuracy. Therefore, according to both computational efforts required and the accuracy of the algorithm, 0.001 is the most appropriate value for the maximum triangle area of Delaunay triangulation.

It is the same as the layout of *concentric boxes* that the minimum triangle angle used is mostly in the range of 20-30 degrees because its influence on the quality of triangle meshes. Figure 28 illustrates how the number of vertices that is describing the computation efforts is related to the parameter of the minimum triangle angle, where the maximum triangle area is fixed to be 0.001 as discussed before.

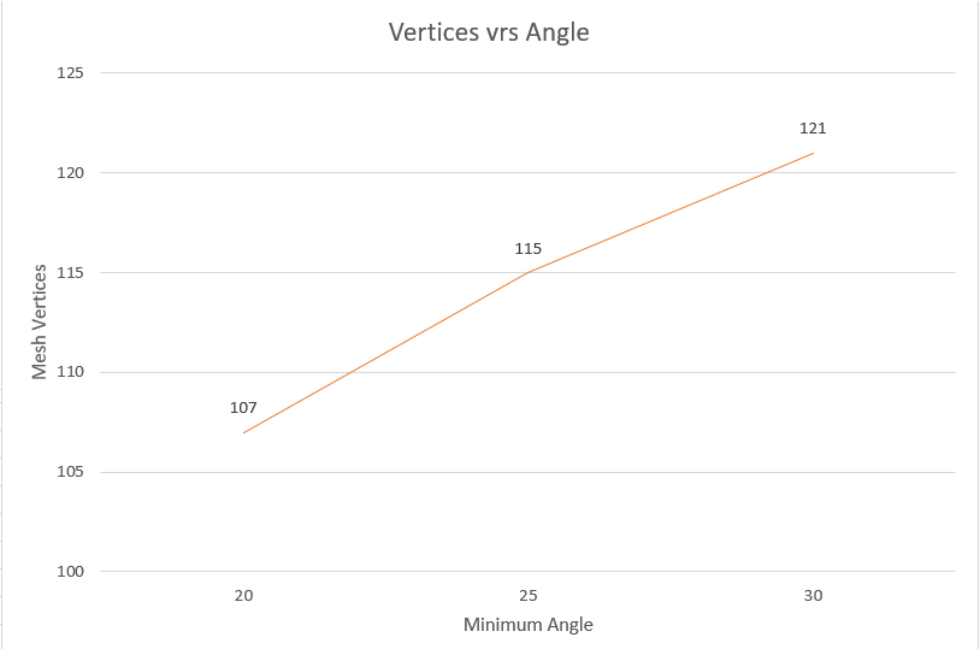


Figure 28: Number of vertices generated versus minimum triangle angle (for *'A' character* layout)

The values of the number of vertices in meshes generated by Delaunay triangulation are almost the same with little increase when the minimum triangle angle used change from 20 to 30 degrees. The analysis above suggests that the fewer vertices on meshes will causes the more computational efforts cost for the execution of this shortest path planning algorithm. The layouts for Delaunay triangulation using three various minimum triangle areas which are 20, 25 and 30 degrees are plotted by the viewer software according to the relative files generated by Shewchuck's program in Figure 29.

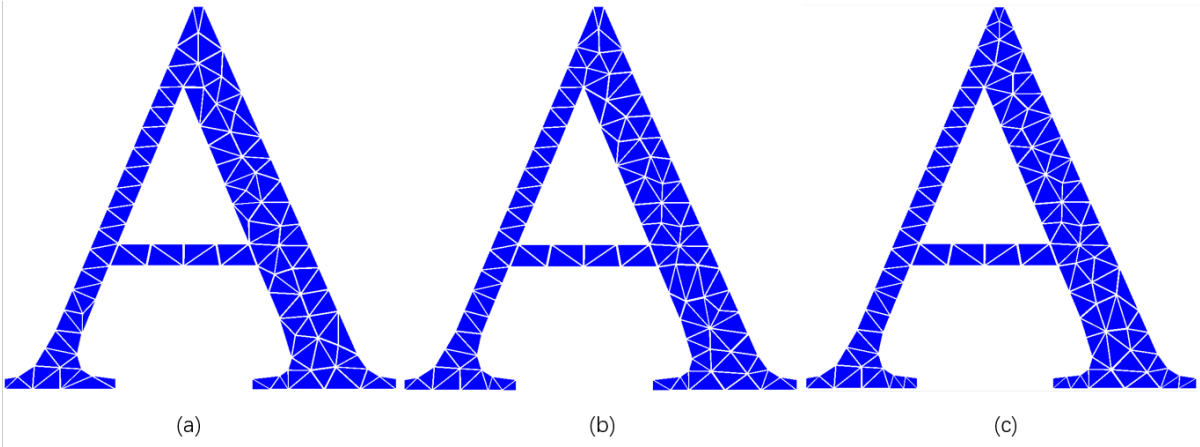


Figure 29: Delaunay triangulation using various minimum triangle angles with fixed maximum triangle area of 0.05: (a) q=20; (b) q=25; (c) q=30 (for *'A' character* layout)

These three layouts in Figure 29 show little differences among each other due to the similar

values of the number of vertices generated by triangulation, hence the parameter of minimum triangle area only has little effect on the accuracy of the results. As the similar layouts and the proportional relationship between the number of vertices and the minimum triangle angle, the value of 20 degrees is chosen to be used for result testing.

The suitable values of two effective parameters, maximum triangle area and minimum triangle angle, are figured out in this section and they are input to the layouts by setting *-a* and *-q* switches. For this *'A' character* layout, these two switches are set to be *-a0.001* and *-q20* according to the discussion before. The next section will provide the layouts for the overall shortest path planning algorithm and the results will be tested by using different destinations. Besides, based on the obtained results, the possible sources of errors will also be discussed.

4.2.2. Shortest Path Layout

When the overall shortest path algorithm is executing, the user is asked to enter the values of the number of vertices, edges and triangles at first. For the triangulation generated of *'A' character* layout by the commands on MSYS2 which are *-a0.001* and *-q20*, these three values are 107, 239 and 132 respectively as shown in Figure 30.

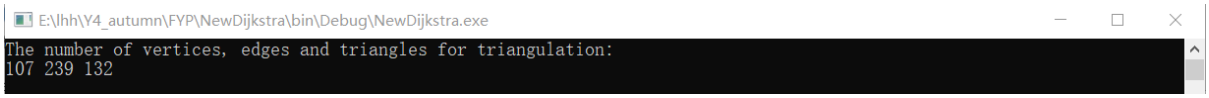


Figure 30: The display of the first step of this shortest path planning program (for *'A' character* layout)

For testing the results, four various nodes are used for the program to search for the shortest path from source to each destination along the edges of the triangular meshes created by Shewchuck's program. To visualize the results and make it easier to check errors, a *.plsg* file are made by the combination of the *.node* file and the *.txt* file generated by the program. Then the details in this file were plotted by a viewer software and the layout is presented in each graph in Figure 31. These four end points are placed on both the left leg and the right leg of *'A' character*, where the top two graphs show the shortest paths for the bottom areas of both legs in the layout and the testing points are on the top areas for both legs in the bottom two graphs. In these graphs, red lines represent the shortest path from source to these four destinations, while other edges are all blue lines.

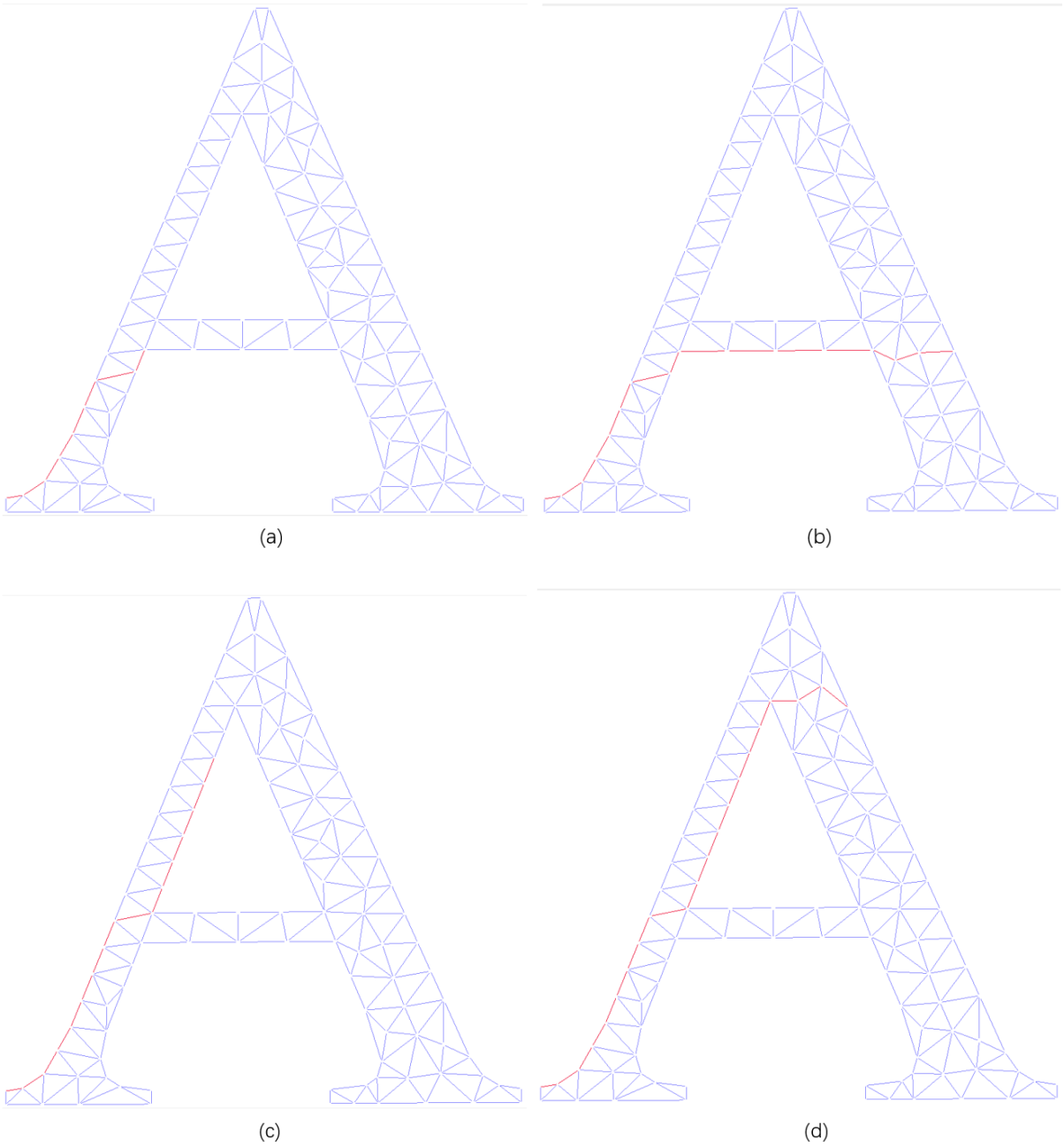
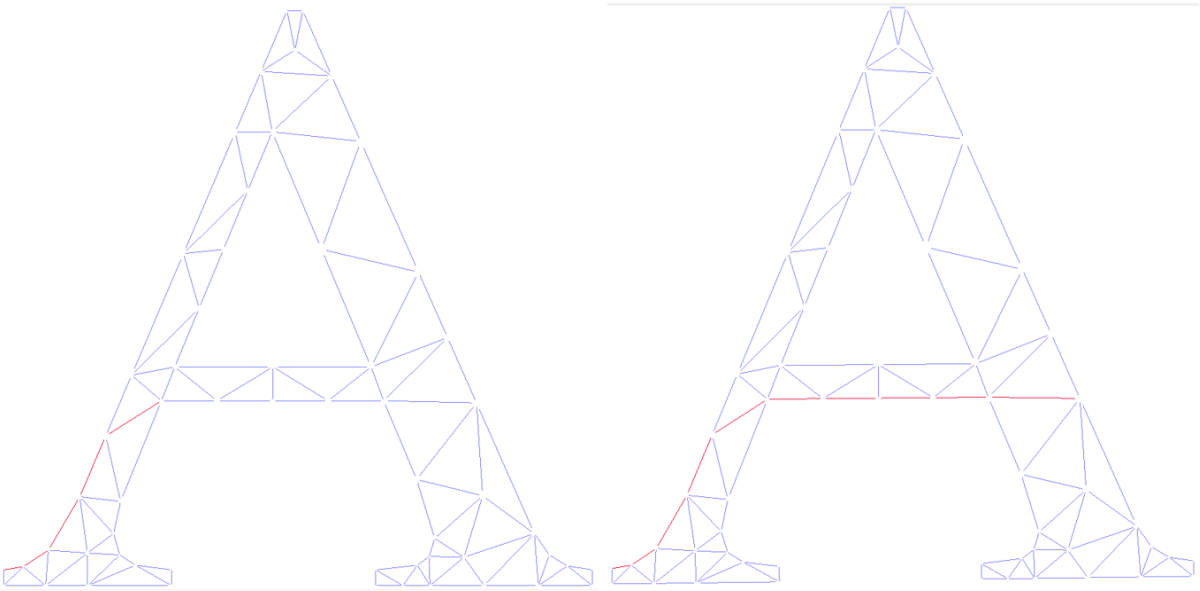


Figure 31: Shortest path planning from source (vertex No.1) to four other vertices (for *'A' character* layout) with maximum area of 0.001

As shown in these four figures in Figure 31, the shortest paths along the triangulation edges from source to these four end points have been calculated successfully. However, the same errors as in *concentric boxes* layout occur that the shortest paths are not exactly the same as the results predicted which should be straight lines. The errors are predicted to be caused by the inadequate triangles generated by the Delaunay triangulation which leads to the lack of edges, hence the mobile robots have few lines to travel along. Therefore, the same method to analyze the sources of errors is also applied on this *'A' character* layout. As excessive time cost for processing the triangulation with maximum triangle area larger than 0.001, a lower value which is 0.01 are used to perform triangulation and the shortest paths from the same four destinations

are calculated by the program to compare.

When the maximum triangle area parameter is changed to 0.01, the distances of the shortest paths from source to the destinations in Figure 32(a)-(d) are 0.248, 0.569, 0.475, 0.63 respectively. In comparison with these graphs, the distances from source to the same four vertices when using the 0.001 maximum triangle area are 0.239, 0.547, 0.468, 0.61 respectively as shown in Figure 31(a)-(d). From these figures, it is observed that more triangles could shorten the distance of shortest path and this improvement can be achieved by minimizing the maximum triangle area of the meshes, hence the sources of the errors are proved to be the insufficient value of the number of triangles. For the consideration of the project efficiency, the time cost for shortest path calculations of Figure 31 and 32 are 19.889s and 19.674s. These data support the prediction that more triangles take more time to process as the time spending on making shortest path plan for triangulation layout with 0.001 maximum triangle area is a bit larger than that with 0.01 maximum triangle area. However, due to the limitations of time and memory use of the laptop for this project, triangulation layouts with smaller maximum triangle area parameters were not be tested and it is wished to be analyzed in the future researches.



(a) (b)

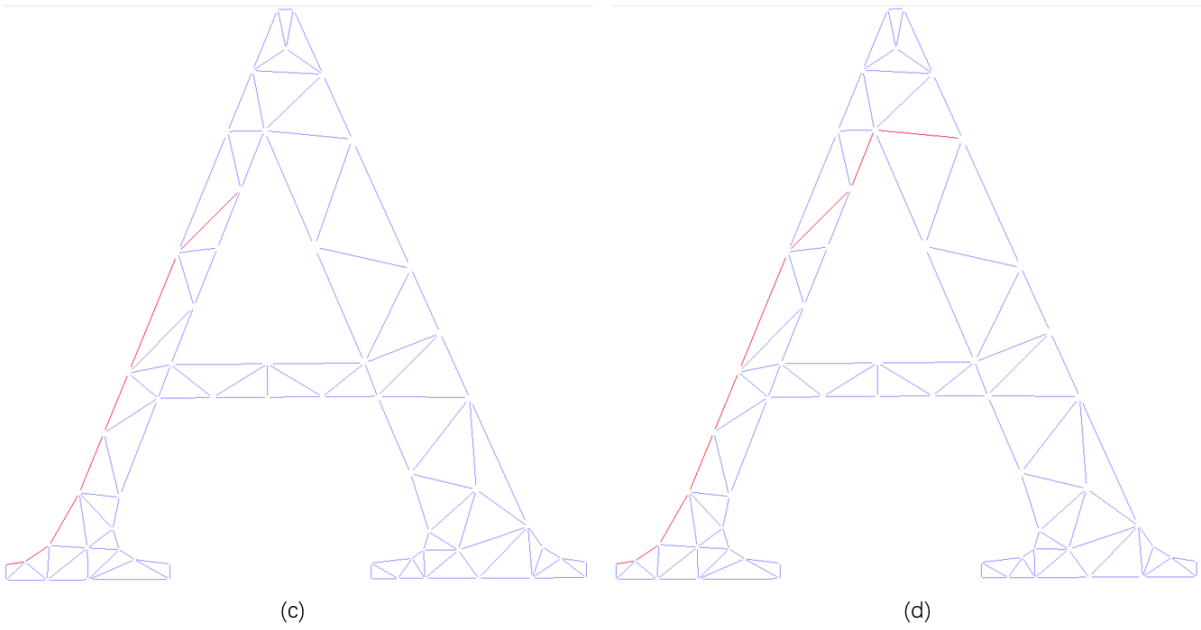


Figure 32: Shortest path planning from source (vertex No.1) to four other vertices (for *'A' character* layout) with maximum area of 0.01

The section discussed the results testing of the shortest path planning of this *'A' character* layout from one source to four different destinations and the impact of the triangulation parameter on the errors. The next section will analyze the difference between the results and parameters used for these two layouts and the factors which have effect on the accuracy of the path planning.

4.3. Comparison Between Two Layouts

At first, the difference of two parameters used for generating Delaunay triangulation for these two layouts are analyzed which are mentioned above (maximum triangle area & minimum triangle angle). For result testing, these two layouts are using the same value for the minimum triangle angle and various maximum triangle areas which are 0.001 and 0.01 for *'A' character* and 0.05 and 0.1 for *concentric boxes* layout. These deviations are reasonably caused by the different areas of these two layouts which requires various maximum triangle areas for Delaunay triangulation to achieve the similar densities for meshes. According to the computational efforts and limitation of project time, the number of vertices which could be processed by the program has a maximum value.

For the *'A' character* layout, as shown in Figure 26, the number of vertices is inversely proportional to the value of maximum triangle area. However, when the maximum triangle area in the triangular meshes is greater than 0.005, the numbers of vertices in the layouts reveal almost no differences. In the contrast, the graph (Figure 20) of the relationship between the

number of vertices and the maximum triangle area in meshes of the *concentric boxes* layout are invariable as the proportion by inversion for the range of any values of the triangle areas. This phenomenon is probably caused by the limitations of the minimum number of vertices in the Shewchuck's program and the values of triangle areas used in the *concentric boxes* layout generates more than 60 vertices, hence the effect by this limitation is not shown in Figure 20.

The time costs for this software program to search for the shortest paths for these layouts are predicted to be the same if they have the same number of vertices generated by the Delaunay triangulation. This is because of the theory of Dijkstra's algorithm discussed before that the program executes through each vertex in the meshes to find the shortest path, hence the computation efforts to operate the path planning will only related to the number of vertices no matter the kinds of shapes. As the limitation of project time, these two layouts with the triangulation including the same number of vertices were not been tested and the experiments on this deduction could be performed in the future researches.

The section illustrated the comparison between two layouts defined for result testing and analyzed the possible causes for these differences. Some expectations and discussions for the future work, the reflection on the original time plan, as well as the suggestions for the project time management will be included in the next section.

Chapter 5: Discussions of Future Works and Review of Project Time Management

The software program of shortest path algorithm has been designed, implemented and characterized in the most computationally efficient manner through the previous discussions. This Chapter 5 will suggest how this project work could be improved in the further researches from the steps to design and construct the software program and the reflection on the original project time management plan.

5.1. Discussions of Future Works

The fundamental aims and objectives of this project have been achieved by this software program of the shortest path planning algorithm for navigated mobile robots. However, there are also some shortcomings in the outcomes of this project and the process of executing. Therefore, the future works could be improved by several aspects categorized to the structural adjustment of the program and code optimizations.

5.1.1. Program Structure

The software program of shortest path planning was implemented based on object-oriented programming (OOP) making use of classes and structures of *vertex*, *triangle* and *mesh*. In the previous work, little handworks are required for modifying the *.node* file by deleting the final value in each line which represents the attributes value and changing the color value in *.txt* file generated to create a *.plsg* file to visualize the shortest path planning results. The improvements of the future works are expected to reduce the handworks by using relevant program structures to create a more complete *.txt* file. Besides, the functions in the program used to read values from the files generated by Delaunay triangulation are also expected to obtain the required values for the coordinates of vertices to calculate the details of edges. The savings on handworks will reduce the time costing on the total work and the layouts of denser triangular meshes by using smaller maximum triangle area could be tested to improve the accuracy of results and project efficiency.

5.1.2. Bottlenecks

As discussed before, the main bottlenecks during this project are developing the connection between Delaunay triangulation and Dijkstra's algorithm and the code debugging work for the software program. When implementing the whole program, solving these two problems are the most time-consuming procedure and the time cost on working is going to be saved in the future design. The combination between two algorithms has been constructed in this project, hence this will not take much time in the future researches on shortest path planning. Besides, for the code debugging work, most errors are produced by the improper manner to use vectors and they are solved by searching for the literature with the related information online. After mastering enough knowledge on the STL data structure of vector, it will cost less time to make improvements on the program. Therefore, as for the time limitations of constructing the project, time savings will give more time to test more examples of layouts using various parameters to generate and the shortest path planning from source to more different destinations.

5.1.3. STL Data Structures

The software program for shortest path planning was implemented with C++ STL data structure of *vector* in this project, which robust the code structure and allows the code to be reusable. There are also some other containers in STL also could be able to store the data of the project, which are the sequence container *list* [35, 36] and the associative container *map* [37, 38]. *List* is a doubly linked list which allows non-contiguous memory allocation. In the contrast with *vector*, *list* spend more time on traversal but less time on insertion and deletion when the positions are found. *Map* is a sorted associative container which is able to store data in pairs as a mapped form, where each item consists of key-value and a mapped value. As the ability for *map* container to store a pair of data, it could be used to save the x and y coordinates of each vertex in the triangulation layout which is generated to search for shortest path. The advantages for *maps* over both of *vectors* and *lists* are in terms of lookup time, ordered and insertion respectively: (1) The array will break down when the key space is too large and it is not an

integer. Therefore, a map is required to maintain reasonable lookup performance (*O(log(n))*) and only two spots are used to store the memory. (2) A *map* allows data of any arbitrary key type with any arbitrary ordering to be defined. (3) For a dynamic *array* or a *vector*, it is needed to be resized and the entire array is copied to new memory, while a *map* has the reasonable insertion time (*O(log(n))*). As these advantages of *maps* usage for the program, the methods of implementing the new shortest path algorithm with STL data structure of the *map* could be analyzed in the future researches.

5.1.4. Parallelization

As discussed before, the shortest path planning algorithm is running in sequential and takes much time to process, thus constructing parallelization of Dijkstra's algorithm are demanded to improve the efficiency of the program. As mostly the computer used to perform the calculations have 4 cores, the data being analyzed is split into four partitions and processing on each one simultaneously. Therefore, the results of parallel programming are expected to be 4 times more efficient than sequential programming due to 4 machines in a cluster. [34] The main challenge for parallelizing the Dijkstra's algorithm is cause by several inter-process interactions after splitting the data in meshes into subgraphs to process. [34]

To implement the parallelization of Dijkstra's algorithm, a fine-gain parallelization scheme is constructed by relaxing all outgoing edges of vertex *u* in parallel to build parallelism at the inner loop as shown in Figure 33. [39] It could be observed that if vertices have a smaller number of neighbors on average, the parallel segment (lines 6-14) of the algorithm will occupy a small part of execution time while most time is cost by the sequential part. [39] This observation suggests that the speedup of the parallelization takes the average over-degree of vertices as the boundary which is represented by the density of triangulation meshes in this project. [39]

Algorithm 2: Fine-grain parallel implementation of Dijkstra's algorithm.

Input : Directed graph $G = (V, E)$, weight function $w : E \rightarrow \mathbf{R}^+$, source vertex s, min-priority queue Q
Output : shortest distance array d, predecessor array π

```
/* Initialization phase same to the serial code */
/* Main body of the algorithm */
1  while Q ≠ ∅ do
2      Barrier
3      if tid = 0 then
4          u ← ExtractMin(Q);
5      Barrier
6      foreach v adjacent to u do in parallel
7          sum ← d[u] + w(u, v);
8          if d[v] > sum then
9              Begin-Atomic
10             DecreaseKey(Q, v, sum);
11             End-Atomic
12             d[v] ← sum;
13             π[v] ← u;
14     end
15 end
```

Figure 33: The pseudocode for lock-based parallel implementation of Dijkstra's algorithm [39]

Figure 34 shows the execution time for the Dijkstra's algorithm in parallel and sequential manner. It is obvious that when there are approximately 50-100 nodes in the meshes, the time costs do not show much difference between these two algorithms. As the increase of the number of nodes, the processing time of sequential shortest path algorithm is much larger than that of parallel algorithm. From the data in this figure, when the number of nodes is greater than 1000, the Dijkstra's algorithm in parallel is about 2 times more efficient than sequential processing which is different from the predicted value of 4. This may be caused by the processing time spending on the sequential part of the program.

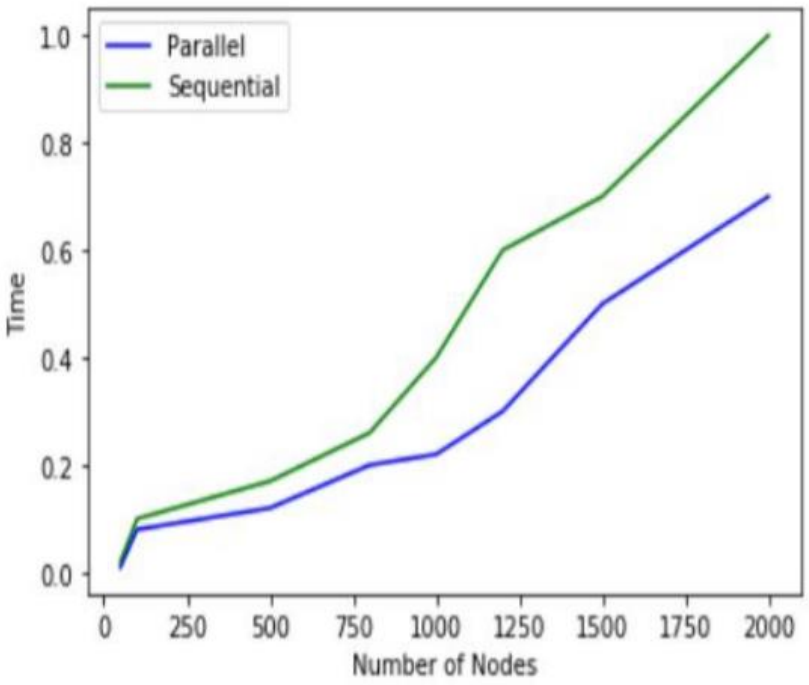


Figure 34: The time cost for Dijkstra's algorithm when running in parallel and sequential manner [34]

During the researches for the experiments of parallel programming for Dijkstra's algorithm, the speed of parallel computation is examined to be higher than the sequential one, and the degree of acceleration depends on the number of executors running on. [34] For the future work, the experiments for the shortest path algorithm using parallel programming could be performed to improve the project efficiency.

5.1.5. Portability for MSYS2 or Linux

In this project, the program was tested to be successfully running on the MSYS2 platform by the commands of *g++ -o test1 main.cpp*, hence the portability for Linux operating system has been proved. As the program is already has the ability to compile on MSYS2 or Linux, there is no further researches required to be analyzed in this area. In the future work, the tests for the program on another platform could be performed to improve the portability of this shortest path planning program.

5.2. Review of Time Management

The Gantt chart of the time plan set out at the start of the project is presented in Figure 35 in Appendix. In this time plan, the meshes of Delaunay triangulation should be generated using the suitable parameters (maximum triangle area & minimum triangle angle) and the connection between these two algorithms is expected to be implemented in schedule for the autumn

semester. Then the overall shortest path algorithm is expected to be established, and the results testing on different layouts and routes are completed in the spring semester. Besides, seven milestones for the desired achievements are shown in this chart by using blue rhombuses, but the risks are not presented on this time plan graph. Therefore, another Gantt chart based on the actual time cost for the project progress is generated to compare and make reflection on the original time plan which is shown in Figure 36 in Appendix.

There are many differences between the original project plan and the actual time spending on each objective for this project. During the procedure of this project, more time is cost on building the program structures which is mainly on the design of the combination between Delaunay triangulation and Dijkstra's algorithm and code debugging. The time for analyzing the triangulation parameters and error checking is also added in the time plan which have not been considered before. The risks of the project are shown as the yellow circles in Figure 36 in Appendix and how these risks could be mitigated are discussed in the following table, where the risks are listed in chronological order.

Risks	Avoidances
Proper Meshes not constructed on time	Work on the constructions of meshes and connections between two algorithms simultaneously
More time cost for building connections between Delaunay triangulation and Dijkstra's algorithm	● Speed up the progress of error checking of overall program construction ● Construct code structures for connections of these two algorithms and overall program simultaneously
Over-running in time for completing the whole program	● Shorten the time cost for defining layouts to balance ● Define fewer layouts than expected for result testing
Enough layouts are not defined on time	Reduce the number of points testing and fewer layouts generated for parameter analysis
Laptop crash - work cannot be completed and submitted on time	● An extra laptop prepared for this accident ● The whole program, recordings and dissertation are constantly uploaded to *Microsoft onedrive*.

Chapter 6: Conclusions

The thesis focused on the design, implementation and result testing for the software program to complete shortest path planning for navigated mobile robots between two points using two main algorithms which are Delaunay triangulation and Dijkstra's algorithm. During this project, fundamental aims and objectives listed at the beginning of this thesis have been achieved by the program completed and the influential parameters for triangulation layout and sources of errors were discussed. The software program constructed is able to perform shortest path planning from one single source point to any other point in any kinds of layouts representing the actual routes with some various obstacles.

Although most aims and objectives were completed, some possible future works to improve the project were also emphasized in this thesis. Due to the time limitations, the parallelization for this program was not implemented and tested in this project, thus this analysis is expected to be completed in the future work. Besides, though this shortest path algorithm has been tested to successfully find the shortest path from source to a few destinations in two various layouts, the universality of the program is expected to be proved by using a few more layouts (three or more). The layouts could be added to test are not only of different shapes, also with different numbers of vertices generated by using various parameters (maximum triangle area & minimum triangle angle).

The time management for this project is not rational as timeouts occur frequently when completing tasks in the project, for example excessive time cost on code debugging and construction of combination between two algorithms which causes less time spending on result testing, hence time for algorithm study and literature review will be shortened in the future work to balance. So that more kinds of layouts could be tested to improve the accuracy of the shortest path algorithm and a more detailed conclusion can be summarized.

When modifying the *.node* file and *.ele* file to enable the program read the data in these files and store it in the corresponding classes, some handworks are required to delete the useless values in these files and allow the values needed to be input. In the same way, to create the *.plsg* files for monitoring the results for the shortest path planning algorithm, handworks are also in the requirements. These handworks cause the limitations of the number of layouts could be tested in some extent, thus the design of the program for replacement of handworks is a method to improve the project efficiency in the future works.

The desired deliverables of the project are all acquired which are the characterization of the computational effort required for Dijkstra's algorithm, testing reports of the program using software to visualize triangulations and routes, a *main.c* file including the overall program and a final report to summarize the key findings of this project. Except for these obtained deliverables, the future works were also discussed to analyze some possible improvements which could be performed in this project.

References:

The references have been grouped by the topics of this thesis as following.

Chapter 1: Introduction and Background

[1] Yongguo Mei, Yung-Hsiang Lu, C. S. G. Lee and Y. C. Hu, "Energy-efficient mobile robot exploration," *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, 2006, pp. 505-511, doi: 10.1109/ROBOT.2006.1641761.

[2] N. Makariye, "Towards shortest path computation using Dijkstra algorithm," *2017 International Conference on IoT and Application (ICIOT)*, 2017, pp. 1-3, doi: 10.1109/ICIOTA.2017.8073641.

[3] Ji-Xian Xiao and Fang-Ling Lu, "An improvement of the shortest path algorithm based on Dijkstra algorithm," *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, 2010, pp. 383-385, doi: 10.1109/ICCAE.2010.5451564.

[4] Wang, Chaoqun, Wang, Jiankun, Li, Chenming, Ho, Danny, Cheng, Jiyu, Yan, Tingfang, Meng, Lili, and Meng, Max Q-H. "Safe and Robust Mobile Robot Navigation in Uneven Indoor Environments." Sensors (Basel, Switzerland) 19.13 (2019): 2993. Web.

[5] Syed Abdullah, F., et al. "Robotic Indoor Path Planning Using Dijkstra's Algorithm with Multi-Layer Dictionaries" *The 2nd International Conference on Information Science and Security 2015* (Seoul, Korean), 2015, doi: 10.1109/ICISSEC.2015.7371031

[6] G. E. Jan, C. Sun, W. C. Tsai and T. Lin, "An O(n log n) Shortest Path Algorithm Based on Delaunay Triangulation," in *IEEE/ASME Transactions on Mechatronics*, vol. 19, no. 2, pp. 660-666, April 2014, doi: 10.1109/TMECH.2013.2252076."

[7] König, M, and Neumayr, L. "Users' Resistance towards Radical Innovations: The Case of the Self-driving Car." *Transportation Research. Part F, Traffic Psychology and*

M. König, L. Neumayr, "Users' resistance towards radical innovations: The case of the self-driving car", *Transportation Research Part F: Traffic Psychology and Behavior*, vol. 44, 2017, pp. 42-52, ISSN 1369-8478

[8] T. Yuta, S. Masahiro, C. Kai, and Z. Lin. "Application of Online Supervisory Control of Discrete-event Systems to Multi-robot Warehouse Automation." *Control Engineering Practice* 81 (2018): 97-104. Web.

[9] Ophaswongse, Chawin, Murray, Rosemarie C, Santamaria, Victor, Wang, Qining, and Agrawal, Sunil K. "Human Evaluation of Wheelchair Robot for Active Postural Support (WRAPS)." *Robotica* 37.12 (2019): 2132-146. Web.

[10] Wang, Yong, Hu, Zhen, and Wang, Ying. "The Application of Markov Decision Process in Restaurant Delivery Robot." *AIP Conference Proceedings* 1839.1 (2017): AIP Conference Proceedings, 2017-05-08, Vol.1839 (1). Web.

[11] Anatolyievich, Titenko, Gennadyevich, Emelyanov, Guryevich, Dobroserdov, Borisovich, Borzov, Nikolaevich, Frolov, Alexsandrovicn, Lisitsin, Vladimirovna, Bredikhina, and Yuryevich, Sazonov. "Modified Method of a Path Planning on a Local Maps for Transport and Aerospace Robots." *Journal of Applied Engineering Science* 17.4 (2019): 585-89. Web.

[12] Rao, A.M., Ramji, K., Sundara Siva Rao, B.S.K. *et al.* Navigation of non-holonomic mobile robot using neuro-fuzzy logic with integrated safe boundary algorithm. *Int. J. Autom. Comput.* **14,** 285–294 (2017).
[13] Hassani, I., Maalej, I., and Rekik, Chokri. "Robot Path Planning with Avoiding Obstacles in Known Environment Using Free Segments and Turning Points Algorithm." *Journal of Mathematical Problems in Engineering* 10.1155(2018), 2018-06-11
[14] Shewchuk, J., "Delaunay Refinement Algorithms for Triangular Mesh Generation", *Journal of Computational Geometry* 22(1-3), 2002-05-01, pp. 21-74, doi: 10.1016/S0925-7721(01)00047-5

Chapter 2: Methodologies and Example Test

[15] Zhang, Jin-dong, Feng, Yu-jie, Shi, Fei-fei, Wang, Gang, Ma, Bin, Li, Rui-sheng, and Jia, Xiao-yan. "Vehicle Routing in Urban Areas Based on the Oil Consumption Weight -Dijkstra Algorithm." IET Intelligent Transport Systems 10.7 (2016): 495-502. Web.
[16] Boris N. Delaunay. Sur la Sphere Vide. Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk 7:793–800, 1934.
[17] Sun, L., Yeh, GT., Ma, X. *et al.* Engineering applications of 2D and 3D finite element mesh generation in hydrogeology and water resources. *Comput Geosci* **21,** 733–758 (2017). Available at: https://doi.org/10.1007/s10596-017-9654-z
[18] Weatherill, N.P.: Delaunay triangulation in computational fluid dynamics. Comput. Math. Appli. 24, 129–150 (1992)
[19] Garimella, R.V., Shashkov, M.J., Knupp, P.M.: Triangular and quadrilateral surface mesh quality optimization using local parametrization. Comput. Meth. Appl. Mech. Eng. 193, 913–928 (2004)
[20] Knupp, P.M.: A method for hexahedral mesh shape optimization. Int. J. Numer. Methods Eng. 58, 319–332 (2003)
[40] Liu, Shan, Jiang, Hai, Chen, Shuiping, Ye, Jing, He, Renqing, and Sun, Zhizhao. "Integrating Dijkstra's Algorithm into Deep Inverse Reinforcement Learning for Food Delivery Route Planning." Transportation Research. Part E, Logistics and Transportation Review 142 (2020): 102070. Web.

Chapter 3: Shortest Path Planning Program Design

[21] Seiller, N., Williem, Singhal, N. et al. "Object oriented framework for real-time image processing on GPU." Multimed Tools Appl 70, 2347–2368 (2014). Available at: https://doi.org/10.1007/s11042-013-1440-x
[22] Wu, Hong, and Nie, Xumin. "Extending STL with Efficient Data Structures." Journal of Computer Science and Technology 13.4 (1998): 317-24. Web.
[23] Hassani, I., et al. (2018). "Robot Path Planning with Avoiding Obstacles in Known Environment Using Free Segments and Turning Points Algorithm." Mathematical Problems in Engineering 2018: 2163278.
[24] Wu, H. and X. Nie (1998). "Extending STL with efficient data structures." J. Comput. Sci. Technol. 13: 317-324.
[25] Wise, G. (1996). "An Overview of the Standard Template Library." SIGPLAN Notices 31: 4-10.
[26] Plauger, P.; Stepanov, A.; Lee, M.; Musser, D.The C++Standard Template Library; Prentice-Hall PTR,Prentice-Hall Inc.: Upper Saddle River, NJ, USA, 2001.

[27] Kreft, K. and A. Langer (2021). "Iterators in the standard template library.", 1996
[28] Stewart, A. (1995). "Introduction to Parallel Programming, by Steven Brawer." Scientific Programming 4: 115-118.
[29] Philippou, A. (1994). "The Art of Parallel Programming & Parallel Programming, An Introduction." The Computer Journal 37.
[30] Jasika, Nadira, Alispahic, N, Elma, A, Ilvana, K, Elma, L, and Nosovic, N. "Dijkstra's Shortest Path Algorithm Serial and Parallel Execution Performance Analysis." 2012 Proceedings of the 35th International Convention MIPRO (2012): 1811-815. Web.
[31] Chowdhury, R., et al. (2018). "Comparative study of parallel implementation for searching algorithms with openMP." Journal of Theoretical and Applied Information Technology 5: 6329-6338.
[32] Grama A, Gupta A, Karypis G, Kumar V. Introduction to Parallel Computing, Second Edition. Communication. 2003. 856 p.
[33] Rubanov, Vladimir, and Silakov, Denis. "Ensuring Portability of Linux Applications through Standardization and Knowledge Base Driven Analysis." Science of Computer Programming 91 (2014): 234-48. Web.

Chapter 4: Result analysis

Chapter 5: Discussions of Future Works and Review of Project Time Management

[34] S. Eyvazov, “Parallel Calculation to find Shortest Path by applying Dijkstra algorithm”, 2019.
[35] J. McDonald, D. Hoffman and P. Strooper, "Programmatic testing of the Standard Template Library containers," *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*, 1998, pp. 147-156, doi: 10.1109/ASE.1998.732610.
[36] S. Alexander., et al. “The Standard Template Library,” Oct.1995.
[37] Babati, B., et al. “On the Validated Usage of the C++ Standard Template Library.” *The 9th Balkan Conference* (2019), pp.1-8, doi: 10.1145/3351556.3351570
[38] H. Gábor, P. Attila and P.Norbert. “Detecting Misusages of the C++ Standard Template Library” *The 10th International Conference on Applied Informatics*, 2018, pp.129-136, doi: 10.14794/ICAI.10.2017.129.
[39] N. Anastopoulos, K. Nikas, G. Goumas and N. Koziris, "Early experiences on accelerating Dijkstra's algorithm using transactional memory," 2009 IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1-8, doi: 10.1109/IPDPS. 2009.5161103.

Chapter 6: Conclusions

Appendix A: Comparisons of initial time plan and actual project progress

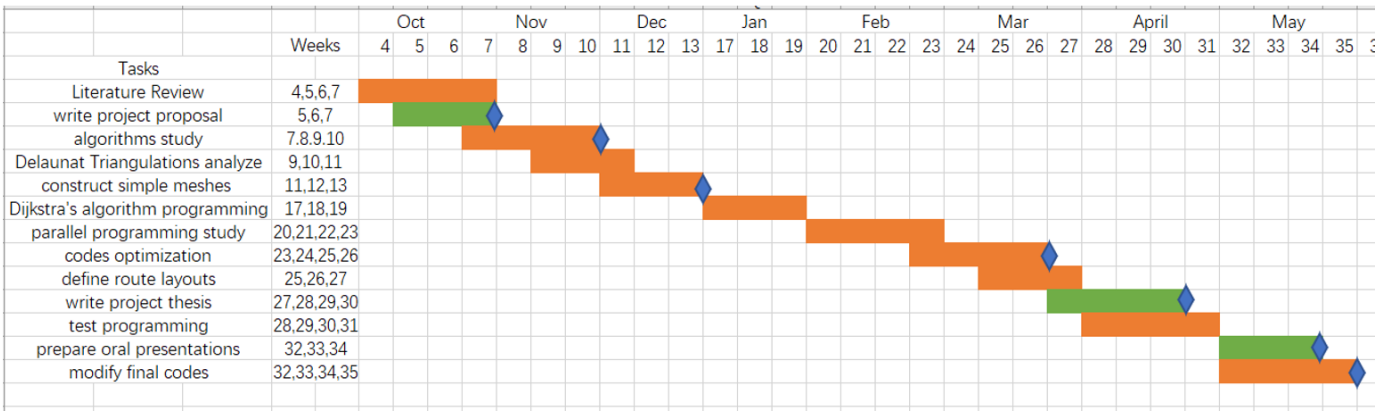


Figure 35: Project initial time plan

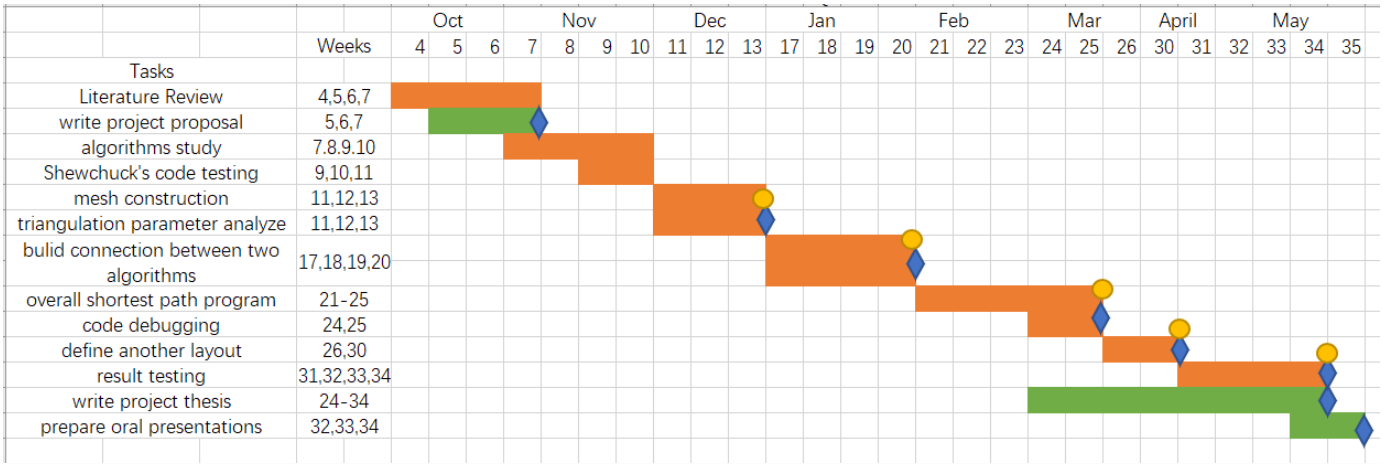


Figure 36: Actual project progress