

## Preface

Raw airborne laser scanning data gives hardly any information about the area observed. It has to be modified by a computer program before it can be used. This fact leads to the assignment of this project. In this report the development of a program is treated which can produce useful data out of the raw airborne laser scanning data.

The report is written for an audience with some background knowledge about earth observation and computer programming. However, everything is explained from a basic level so except for certain details it should also be clear for everyone with minor knowledge in these areas. The reader who is familiar with airborne laser scanning, the data it produces or Dijkstra's algorithm can skip the corresponding theory in this report.

The help which the teaching assistant provided during these weeks is greatly appreciated, the wealth of knowledge he provided was essential for us to produce this report. For the assignment itself, along with a detailed description and explanation, we would like to thank Alexander Bucksch. We would also like to thank the TU Delft for allowing the use of the project rooms and the computer hard- and software. More importantly, the knowledge about Java programming has been of great value to us, while making the program. Therefore, we would like to thank all the teachers and student assistants who have taught us this very important knowledge. Also, the course of technical writing has helped us making this report. We would like to thank Angeniet Kam for the lectures she gave us.

Delft, March 2009

# Table of contents

Preface.....	ii
Summary.....	iv
1 Introduction .....	1
2 Airborne Laser Scanning.....	2
2.1 Remote sensing .....	2
2.2 LIDAR aircraft application.....	3
2.3 Raw data processing .....	6
3 Theory of algorithms.....	9
3.1 Density algorithm: Trunk finder.....	9
3.2 Shortest path algorithm: Dijkstra.....	10
3.3 Local density algorithm: Separation.....	12
4 Program structure.....	14
4.1 Global program structure.....	14
4.2 Filereader .....	16
4.3 Vertex connections.....	18
4.4 Trunk Finder.....	20
4.5 Shortest path.....	22
4.6 Separation.....	24
4.7 Output .....	26
5 Results.....	27
5.1 Test set .....	27
5.2 Forests .....	29
6 Conclusions and recommendations.....	35
6.1 Conclusions.....	35
6.2 Recommendations.....	35
Bibliography .....	37
Appendix A Java program code.....	I

## Summary

This report describes how a Java based program sorts point data obtained from airborne laser scanning over a forested area and how it presents this in such a way that individual trees can be distinguished. This data can then be used to obtain forest inventory parameters.

The data is acquired using an aircraft with a laser scanning device mounted on it. Airborne Laser Scanning is capable to obtain points within a xyz-coordinate system. Due to the large amount of data, processing the points demands the development of a software program. The implementation of the software makes use of the Java programming language.

The program finds the location of the individual trees by using the density of measurement points on the xy-plane. These locations are then used as starting points for an algorithm which calculates the shortest path between two trees. This algorithm is based on the Dijkstra's algorithm. Using the normal Hesse Form the trees are separated at some point along the shortest path. Boundary conditions have been set to reduce the calculations while analyzing the shortest path. The last step is to plot the forest and show the individual trees by giving them a different color.

Three input files were given. Only one file gives a clear result, the results of the other two files still need to be improved. This can be done by improving the Java code. Also varying the different parameters will give better results. The difficulty is finding the optimal values, since they are different for each file. They need to be found with trial and error. This is very time-consuming, since one run will take about five to eight hours.

In order to obtain better results, three things can be done. First of all, the forest can be split in small pieces before analyzing. Second, the separation class can be executed in a specific direction instead of randomly choosing starting points. Third, the running time can be decreased by using a computer with a higher clock rate. With a lower running time it is easier to optimize the results by varying the parameters, such as the radius to neighboring trees and the separation height.

# 1 Introduction

Plant geometry can give us information about chemical processes inside of this plant. Hence, simply knowing how areas are physically built up can yield valuable information about the ecological processes (Fleck, 2002). Airborne laser scanning is a technology to measure the earth's surface. It is a method that yields vast amounts of bulk data in the form of thousands of reflections signifying points in an area; this can be used for example over a patch of forest. The problem is to simplify these points in such a way that individual trees can be recognized and their geometry defined, allowing biologists to draw conclusions about this area.

This report describes how a Java based program structures point data obtained from airborne laser scanning over a forest area and how it presents this in such a way that individual trees can be distinguished. The general approach is to simplify millions of points into a set of tens of thousands of points called vertices. These vertices can be connected with straight lines in such a way that the forest can be represented as a skeleton. When using the point raw data to find densities combined with a path finding method called the Dijkstra algorithm, individual trees can be separated from each other. The program is a compromise between the accuracy of the tree definition and the size and speed of the java program which calculates this.

The report consists of five chapters. In chapter 2 general information about airborne laser scanning will be given. In chapter 3 the theory behind the methods used in the program will be explained, so the reader can understand the decisions made during the development of the Java program. This program will be explained in detail and visualized using a flowchart in chapter 4. The results of the simulation can be found in chapter 5. In chapter 6 the conclusion and some recommendations are given. The actual code can be found in Appendix A.

## 2 Airborne Laser Scanning

In this chapter, the use of light in remote sensing is discussed. First a short description about the principle of this type of remote sensing is given in section 2.1. Next, in section 2.2, the application on flying platforms is described. This kind of application can result in a large amount of data. In section 2.3 the processing of this raw data is explained.

### 2.1 Remote sensing

Light Detection And Ranging or Laser Imaging Detection And Ranging (LIDAR) is remote sensing technique. Laser scanning consists of coherent 'laser' light which is transmitted from a sensor in a continuous stream of 100 pulses per second. Some of the light reflects on the surface of a remote object and this reflection is captured by a receiver inside the sensor. The light is of various visible or near-infrared wavelengths and the laser pulses travel with the speed of light.

The time delay between the emitting and receiving of the pulse in combination with the known speed of the pulse is used to derive the range between the device and the object. Figure 1 shows this LIDAR principle. In one millisecond the pulse can travel 150 km. From the calculated range and the LIDAR system position the map coordinates  $P(x, y, z)$  of the surface point hit by the pulse can be determined. By changing the position of the LIDAR with respect to a reference plane, the surface profile of a remote object can then be distinguished.

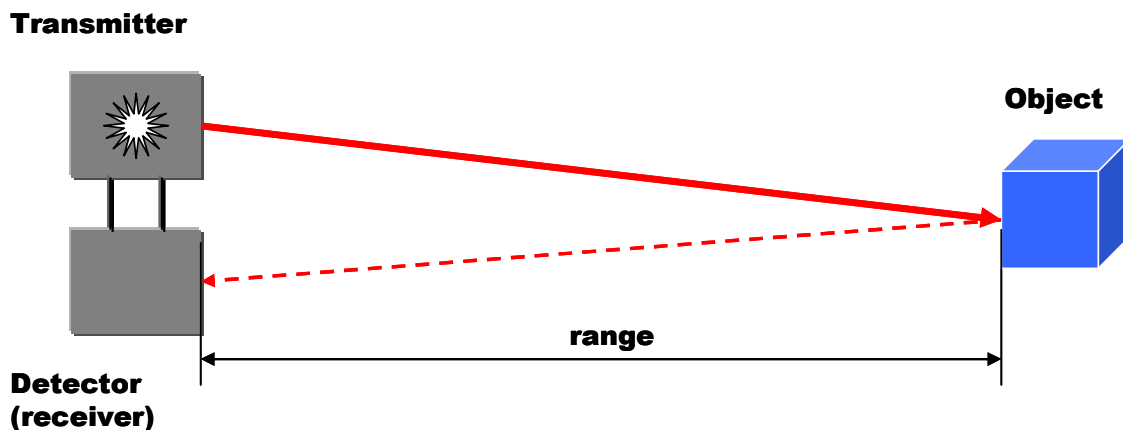


Figure 1: LIDAR principle

In addition, LIDAR can also be used to determine speed, rotation or chemical composition and concentration of a remote object. This can be a clearly defined object, such as a vehicle, or a diffuse object such as a smoke plume or clouds. As LIDAR is light near the infrared region it can be absorbed by water which causes less reflection and less data. For shallow waters one can compare the strength of the reflection from the surface and the much weaker reflection from the bottom of the lake. The difference in time delay between these two returns can be used to determine the depth of the water at any point.

Limitations are that except in areas where there are gaps in the canopy, the light cannot penetrate tree covers which prevents accurate readings at the forest floor. Also possible unwanted reflections from small moving objects can be a limitation.

## **2.2 LIDAR aircraft application**

Mounted on a flying platform such as an aircraft or helicopter, the position of a LIDAR system can be continuously changed such that large areas can be scanned. This application of LIDAR is called Airborne Laser Scanning (ALS). The range of altitude can be chosen to be a few tens of meters where atmospheric propagation effects are negligible, to even many thousands of meters. The choice of this altitude has an impact on the scale, spatial coverage and spatial resolution of the data collected. Figure 2 shows remote sensing platforms with typical altitude potentials.

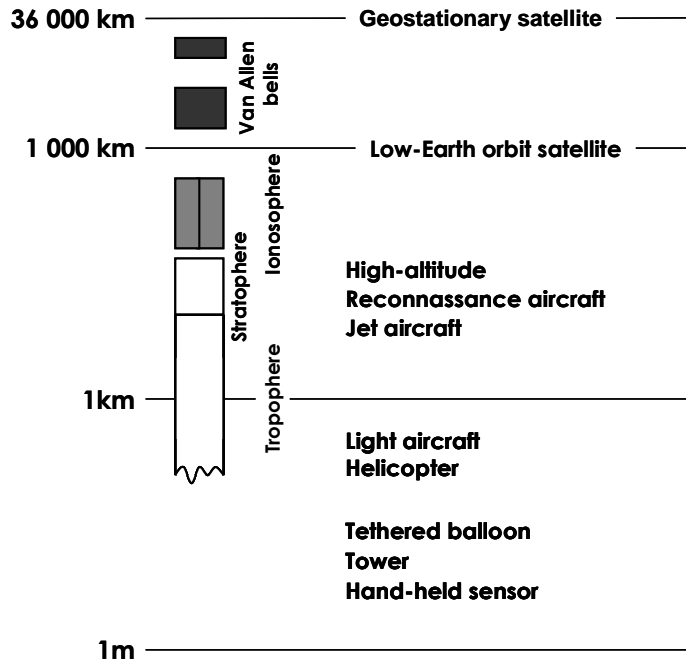


Figure 2: Platforms arranged by typical altitude above Earth's surface (Rees, 2004)

The platform speed ranges from zero for a hovering helicopter to several hundreds of meters per second. It is important that this speed matches the characteristics of the LIDAR system used, since LIDAR works with speed and time determination of pulses. The position of the sensor with respect to a reference plane can be determined in two ways. The first option is that the Global Positioning System (GPS) determines the position of the platform with accuracy of the order of one meter or better. The second approach is the use of ground control points of which the exact location is known. They can be included in the image collected by the airborne sensor.

In addition, a platform can be subjected to variation in its motion which results in a distorting effect on the image. For an airplane the most important are roll, pitch and yaw. Figure 3 shows the distortions of a scanned image of a square grid due to these unwanted motions. The scanner needs to be corrected for these oscillatory motions. In Figure 4, a distorted image before and after correction is shown.

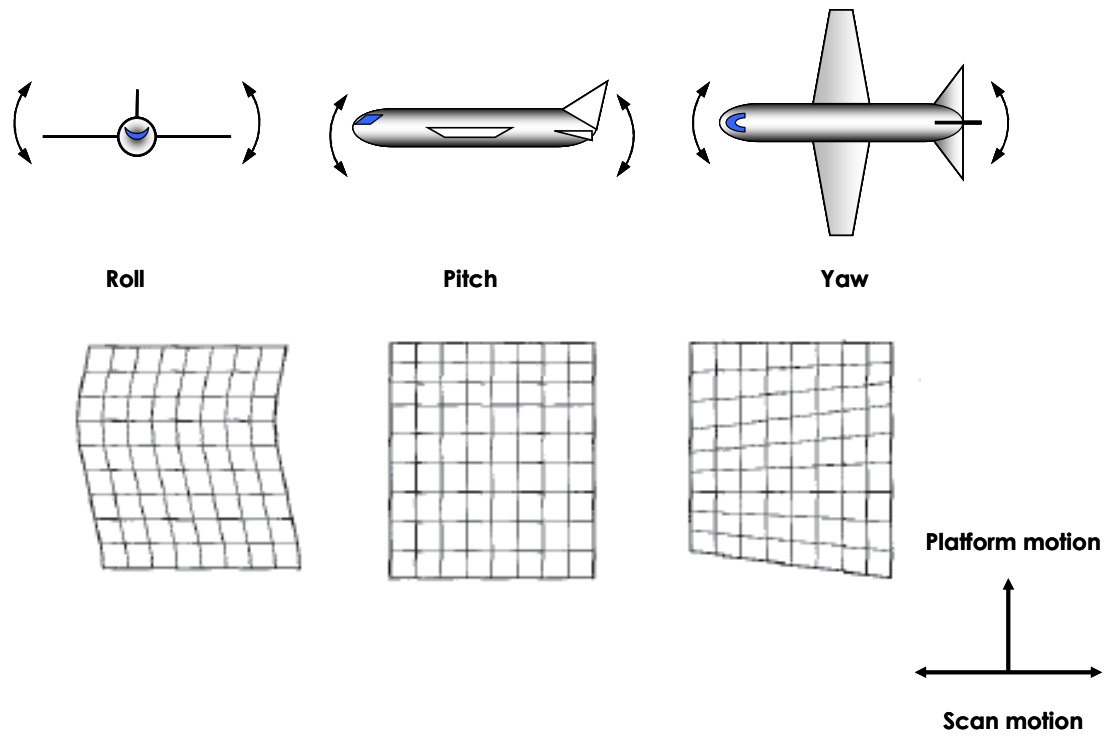


Figure 3: Roll, pitch and yaw distortions of a scanned image (Rees, 2004)

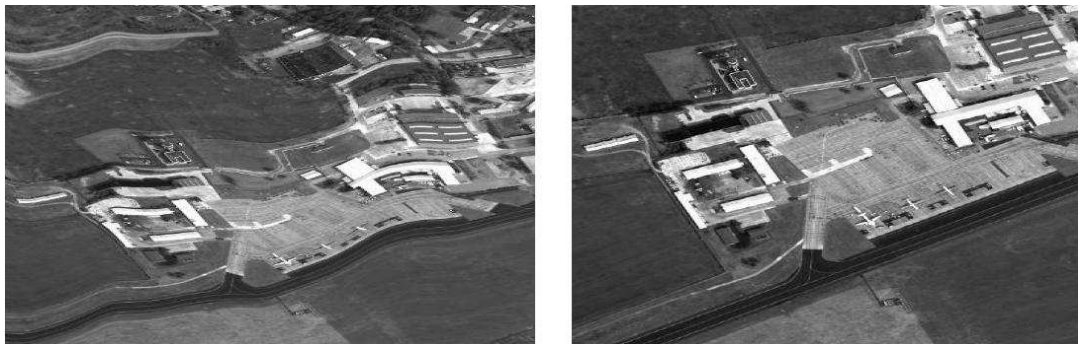


Figure 4: Distortion of a scanned image, before and after processing (Gunter, 2009)

Figure 5 shows the three reference frames of importance; namely that of the GPS system on board, the airplane itself, and the reference frame of the earth.

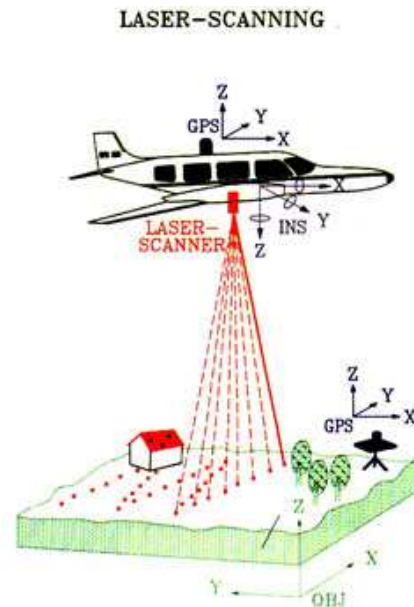


Figure 5: Laser scanning airplane (Spencer B. Gross, 2003)

### 2.3 Raw data processing

By combining the measurement data with the aircraft data, like height, position and attitude, the precise coordinates of every single point in absolute xyz-coordinates can be determined. However, the large amount of points is difficult to process. Therefore, points in close proximity, a point cloud, are merged together to form a so called vertex. The vertices are linked by vectors, also known as edges. A graph is an abstract representation of these connected edges and vertices. Figure 6 shows such a representation of a graph. The vertices are represented by circles and the edges by the straight lines.

Like trees and connected branches, graphs may be used as diagrams to illustrate the relationship between pairs of elements in a set of objects. It can be seen that use of graph theory can structure this large amount of data and relate the elements to each other. The edges give an indication about the relative positions of the vertices with respect to each other. The length of the edges also gives an indication about the density of the vertices. By use of software, recorded intensity data and digital image information properties of objects, can automatically be identified and determined for a large amount of data.

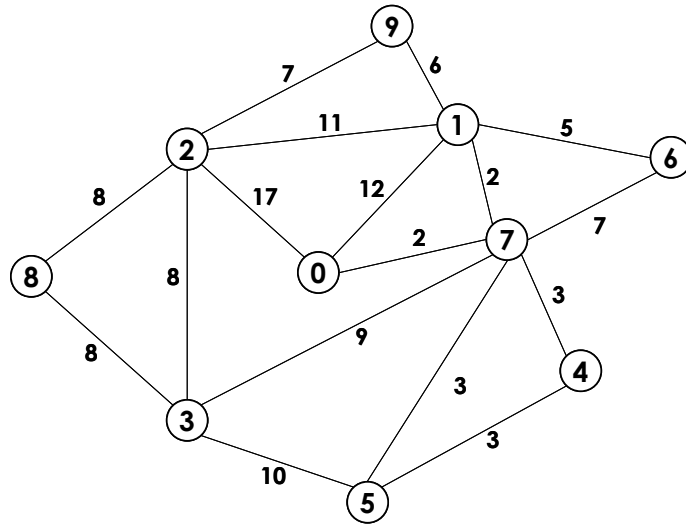


Figure 6: Graph of vertices and edges

The file containing these vertices and edges is presented in a VRML file. VRML stands for Virtual Reality Modeling Language and is used for three dimensional computer graphics. As one can see from Figure 7, when viewing a file containing forest data, some trees can already be recognized. However, all trees are still attached to each other and the data gives hardly any information about the forest. The trees have to be separated from each other to make the data useful. This can be done using a computer program. The splitting of individual trees is called tree delineation.

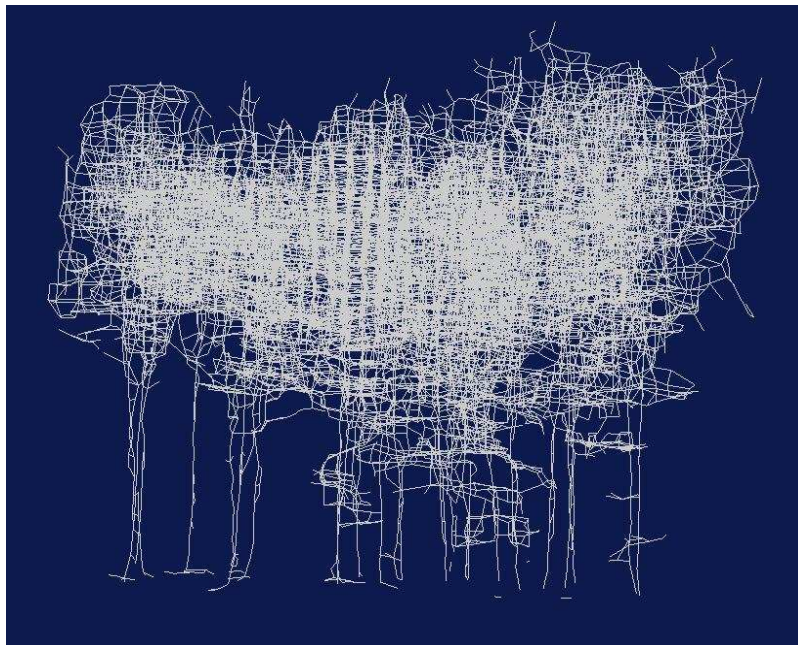


Figure 7: view of a VRML file (TU Delft, 2009)

Table 1 shows the xyz-coordinates of four vertices. This gives an indication about the data that needs to be processed and how it is related.

*Table 1: xyz-coordinates of vertex in meters*

Number vertex	X-coordinate vertex	Y-coordinate vertex	Z-coordinate vertex
1	202274,1875000000	486924,9375000000	6,5599975586
2	202273,1875000000	486926,3437500000	7,9199981689
3	202274,0625000000	486926,4375000000	8,1600036621
4	202273,8437500000	486925,1562500000	7,9799957275

In **Error! Not a valid bookmark self-reference.** the connection between several vertices is illustrated. The edge length and direction between two vertices can be determined from the relative difference between the xyz-coordinates of two connected vertices. The direction edge is the main direction of the vector between two connected vertices. This direction edge is used, like the North, West and South direction on a map, to give an indication about the relative position between two vertices without the need to first determine the exact distance.

*Table 2: Vertex to vertex relation*

from vertex	edge (direction) [x y z]	to vertex
0	[0 0 1]	3
1	[1 0 0]	2
1	[0 -1 0]	4
1	[0 1 0]	-1

## 3 Theory of algorithms

As mentioned in chapter 2, a large amount of data is retrieved from laser scanning. In order to make processing easier, the points are converted to vertices and vectors. In this chapter, several techniques to process the data, which are used by the Java program, will be introduced. This chapter will also give a thorough understanding of the methodology of the program. For a detailed description of the Java code please refer to chapter 4. For the full version of the code, refer to Appendix A.

Several algorithms will be applied such that the final objective, separation of trees, is achieved. In section 3.1, a density algorithm, named trunk finder, is applied to localize the individual trees. An algorithm which will reveal the shortest path between the individual trees is clarified in section 3.2. From this path, the program will be able to determine the actual connection between the trees. The final step is presented in section 3.3. This is the application of a so called separation algorithm. This algorithm determines which point belongs to which tree.

### 3.1 Density algorithm: Trunk finder

The obtained data, as described in chapter 2, is first written in arrays. The transformation process is described in 4.2 by means of a flowchart. The result of this transformation is the input data for the density algorithm.

The density algorithm is applied to localize the start points for the Dijkstra algorithm (Dijkstra, 1959). By counting the xyz-coordinates of the vertices, a so called density field can be created. It shows the amount of vertices per area, called  $ds$ . The density grid is then used to determine the position of the trunk of each tree. The assumption is made that there is a trunk at every place with a high density. The number of points, and thus vertices, per area is higher when there is also a trunk instead of only canopy. The algorithm defines a trunk if the density of an area is higher than the density of surrounding areas.

To make results even more accurate, only the points within a certain range of heights are considered. This filters out the canopy and bushes on the ground. Thus, for this determination, the setting of the step size of the area  $ds$  and the height from which the counting will be applied are important. The coordinates of the trunks are made available for the next part of the program.

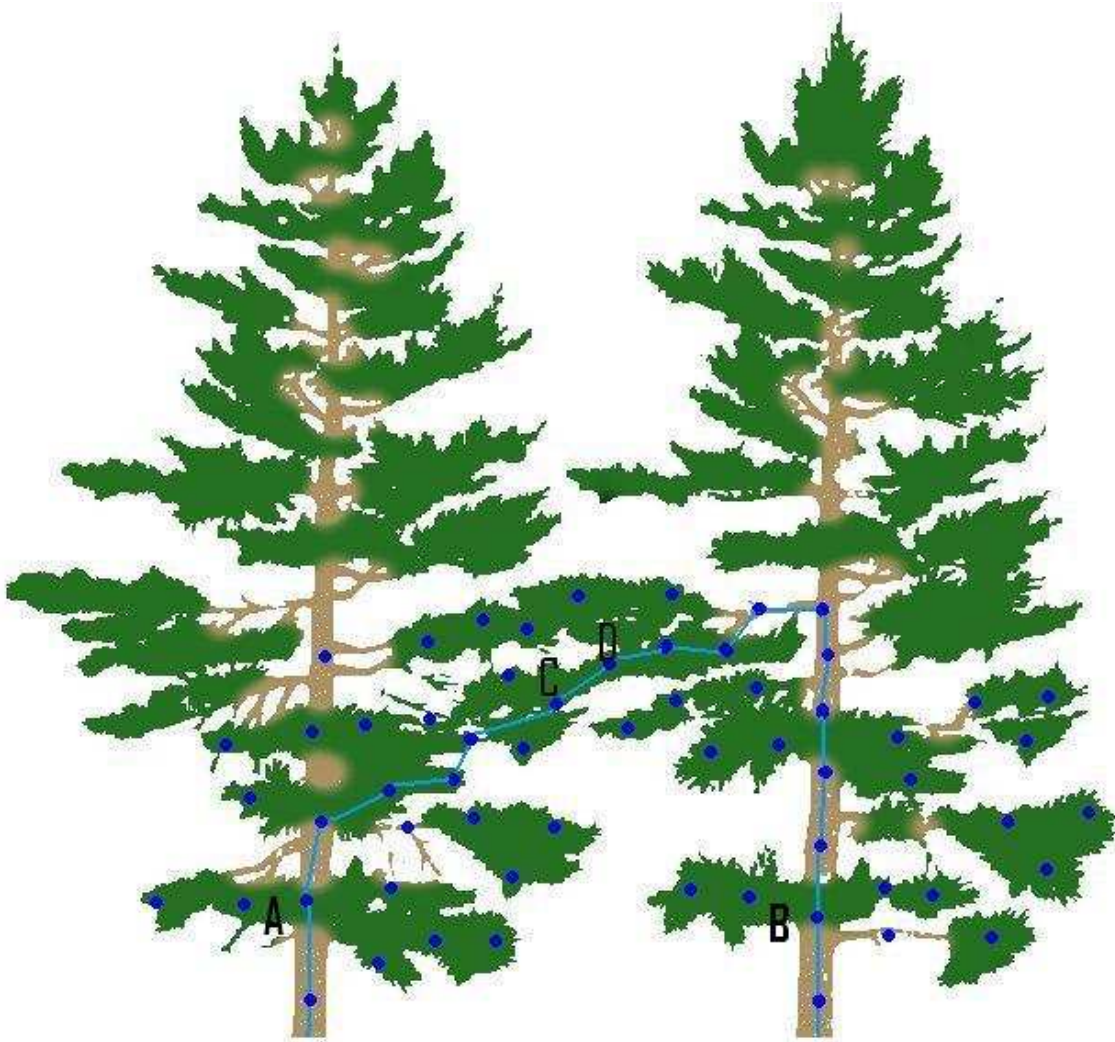
### 3.2 Shortest path algorithm: Dijkstra

For the given data package, the algorithm which is suited best to determine the shortest path between the trees is the Dijkstra algorithm (Cormen, 2001). The Bellman–Ford algorithm computes also shortest paths where even some of the edges may be negative (Cormen, 2001). The Dijkstra algorithm solves the same problem with a lower running time, and requires edge weights to be non-negative. The data package consists of no negative edges and Bellman–Ford algorithm is usually used only when there are negative edge weights. For this case the Dijkstra algorithm was the best option and contained no limitations.

Now that the coordinates of the trunks of the trees are determined, Dijkstra's algorithm can be applied. This algorithm will find the shortest path between the vertices in a graph like Figure 6 in section 2.3. The algorithm travels from node 1 to node 5 in every possible way. Every time the distance travelled is compared to the stored distance. The initial stored distance is infinity and all nodes are marked as unvisited. If the new distance is shorter, the new distance is stored. Then the algorithm considers the node which has the shortest total distance from the starting point, and calculates the distance to all connected nodes. Every node which has been visited is marked as visited. All nodes are visited this way. All possible routes are considered, and the distance stored is the shortest distance to that node. To avoid unnecessary tracks the assumption is made that the shortest path is always less than 1.5 times the horizontal distance between two trees. Else the assumption is taken that there are no connections with the tree.

As mentioned in section 2.3 a graph can be seen as trees connected to each other with their branches. Figure 8 illustrates the Dijkstra's algorithm used to calculate the shortest path from one tree to another. This path consists of the trunk and the branches of both trees. This means that there can be more than one path connecting two trees. Only the x- and y-coordinate of the trunk are known, so the algorithm will start at a random height on the trunk. For example, in Figure 8, the algorithm will start at point A and travels to point B. Somewhere in this path the two trees are linked by their branches. It is unknown at this point where the trees are connected, but one of the vectors in the shortest path has to be removed to separate the trees. This is done by finding the lowest density which will be explained in the next section of this chapter. At the point where the density is lowest, for example in Figure 8 between point C and D, the

corresponding edge is removed from the array. After that, the shortest path is determined again. This time it will take a different route, since the last one does not exist anymore. The program continues doing this, until there are no more routes remaining to the other tree. This is done for all surrounding trees and for each tree until all trees are free.



*Figure 8: The shortest path algorithm is applied from point A to point B. This is a simplified drawing, not all nodes are visible*

### 3.3 Local density algorithm: Separation

The final step to separate all trees is the use of the local density algorithm. This algorithm determines the density on each vertex in the shortest path. An assumption is made that at the connection between the trees the density will be lowest because at the profile edge of the tree the leaves are minimal. At the vertex where the density is minimal, the path may thus be disconnected. In addition, all edges and vertices on each shortest path are considered except the vertical edges. When the difference between the y-coordinate of two vertices is less than horizontal height  $hh$ , then the edge is considered horizontal.

The method makes use of the direction vectors, shown in Table 2 in section 2.3, to find its way through the shortest path. A limitation of this method is the use of the step size of the area  $ds$  since the distance between the vertices is not defined and has to be calculated for every vertex pair. Also  $ds$  needs to be adapted every time. To avoid complexity in the process, a plane specified in Hesse normal form (Taylor, n.d.) is applied on each vertex to determine the density. A point and vector is enough to determine this plane normal to the edge between the two vertices. The direction of the plane is given by the three values  $a$ ,  $b$ ,  $c$ . The value  $d$  determines the intersection between the plane and the normal line. Submitting these values into the equation for the distance  $D$ , the distance between any point and the plane can be determined. The coordinates of the data point are  $x_0$ ,  $y_0$  and  $z_0$ .

$$D = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$$

On both vertices and on the midpoint of the edge respectively three planes are applied. These three points, two vertices and a midpoint will also be the points necessary for the normal Hesse form. For all planes the edge is used for the normal vector. Instead of  $ds$  the plane on the Hesse form is used as the reference. Figure 9 shows the normal Hesse form applied to vertex one, midpoint  $S$  and vertex two which result in a normal plane  $A$ ,  $B$  and  $C$  respectively on the path between vertex one and two.

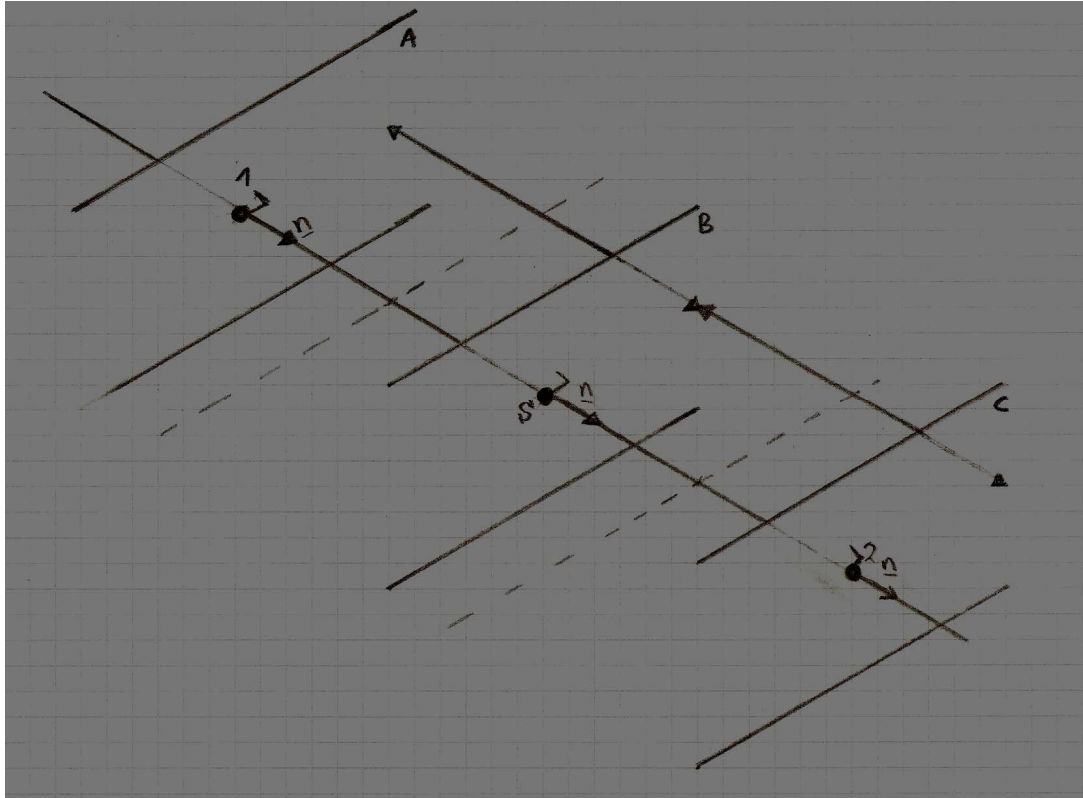


Figure 9: normal Hesse form: vertex A, vertex B and reference point S

The number of xyz-coordinates and their distance with respect to the planes will determine the density on each of the three measurement points. The median of these three points is taken and compared. The point S is taken to be a reference point.

Because the length of the edges between the vertices on the shortest path are not equal, the distance between the vertices needs to be normalized, else the results cannot be compared. The distance between the vertex and reference point s is  $1/2$ . All points with a distance equal or less than  $1/4$  of the total distance between the two vertices, from plane A and C respectively, are assumed to belong to that plane. The results are squared to weigh them as well as the distances. The quarter distance will then become  $1/16$ . Therefore, for the comparison of the results the smaller of the medians corresponding to the vertices needs to be  $1/16$ th smaller than the median corresponding to the distance to the reference plane s.

The density along the total path is determined like this. The lowest density will be assumed to be the point of separation, followed by disconnecting the path.

## 4 Program structure

In this chapter the Java program structure will be explained. It consists of several sections, all about a specific part of the program. In every section a flowchart can be found.

The first section briefly explains how the whole program works. In section 4.2 the class *fileReader*, which reads in the data to the program, is described. Section 4.3 elaborates on the class *vertexConnections*, which formats the input data to ease the use of it in the rest of the program. The class *trunkFinder* uses the density algorithm to identify trunks and is explained in section 4.4. In section 4.5 the shortest path algorithm is explained. This algorithm iterates the different paths between two trunks and finds the shortest one. The next section, section 4.6, elaborates on the separation of the path between two trees. In the final section, section 4.7, the output of the program is clarified.

### 4.1 Global program structure

The program starts by reading the input data, which consists of three important parts, the xyz-coordinates, the vertices and the edges. These parts are put into arrays. After that, a density field is created, (see section 3.1). Then for each area the number of measured points is counted, this is used to determine the position of the trunk of each tree.

Next, the shortest path between two trees is determined using Dijkstra's algorithm (see section 3.2). Thus, the program 'walks' from vertex to vertex via the edges. At the point where the density of points is the lowest the corresponding edge will be removed from the array.

After that, the shortest path has to be determined again. This time the program will follow a different route, since the last one does not exist anymore. The program continues doing this, until there are no more routes remaining to the other tree. This is done for all of the surrounding trees and for each tree. If everything is done correctly, each vertex is only connected to one tree. A number can be given to each vertex to indicate to which tree it belongs.

Next, all of the vertices are sorted according to each tree and then a file is printed, which can be viewed by VRML-viewer. All trees will have a different color.

The flowchart of the Main class can be found in Figure 10. The class calls all other methods so the flowchart of this class represents the high level flowchart of the entire program.

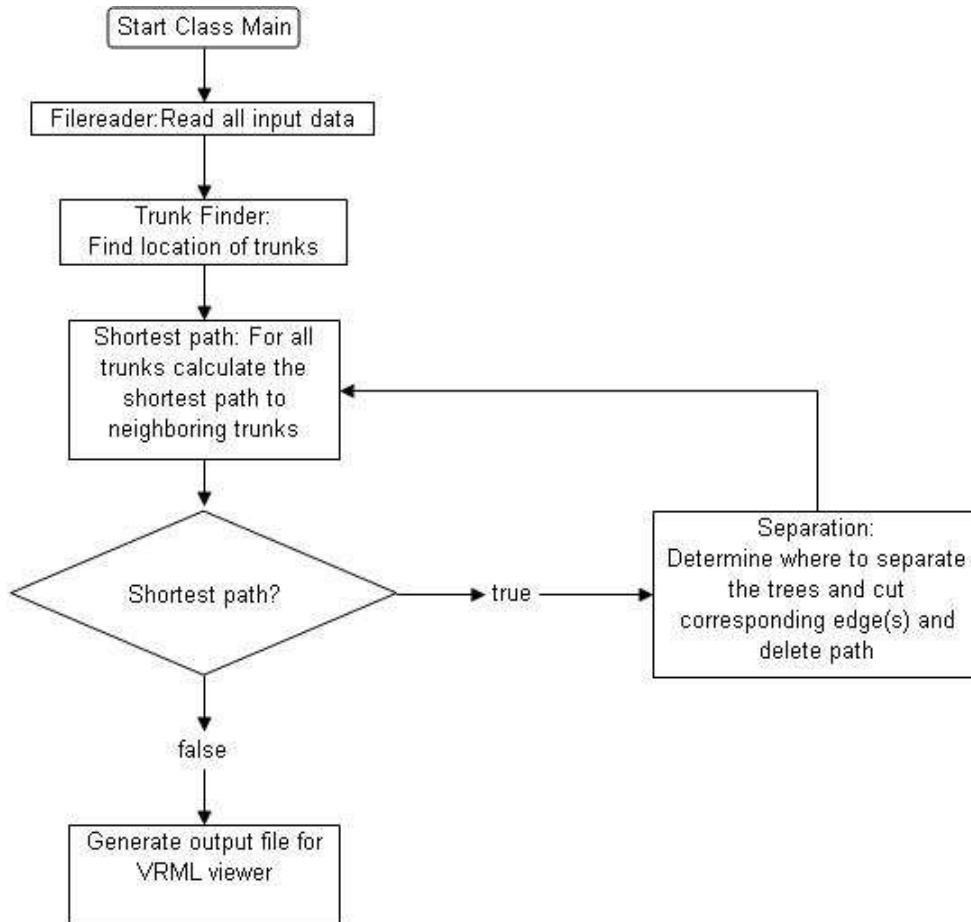


Figure 10: Flowchart Main

## 4.2 Filereader

The file reader is the part of the program which reads the input data and stores it into arrays so the data can be used. This part is explained in this section, Figure 11 is the corresponding flowchart which can be found on the next page.

The input data consists of four parts. First there are xyz-coordinates given from the measurements of the laser scanning. The coordinates of the vertices are the second part of the input data. Next, the position of these vertices with respect to each other is expressed in vectors. The list of the vectors and the vertices they connect is the third part of the input data. The last part is a list of the vertices and the corresponding group of xyz-coordinates.

The file reader contains several methods which can run parallel. There are three methods which calculate the number of elements in each part of the input data. This is done because each set of data has a different size. To create an array the length needs to be known in advance. Another reason to calculate the number of elements is to make the program adaptable to files with different sizes.

The other methods write all of the data in arrays by using a stream tokenizer. The tokenizer makes a difference between numbers and symbols and stores only the numbers. The xyz-coordinates, the vertices and the list of vertices with the corresponding xyz-coordinates can be put in the array without adaptations. The list of vertices which are linked and their vector is filtered and put in two different arrays. This is done by using two methods. In the first method, called *returnXyzVertexVectors*, the last two values of each row are removed since they are irrelevant. The *vrmf*-file reader needs a “-1” between the pair of vertices and the vector and at the end of each line to separate data. The “-1” is not needed to process the data, thus it is also removed from the list. In the other method, called *returnPointConnections*, only the first two columns, thus the pair of vertices, are returned in an array. This array is used in the method *returnVertexConnection*, which calculates which vertices are linked to every vertex.

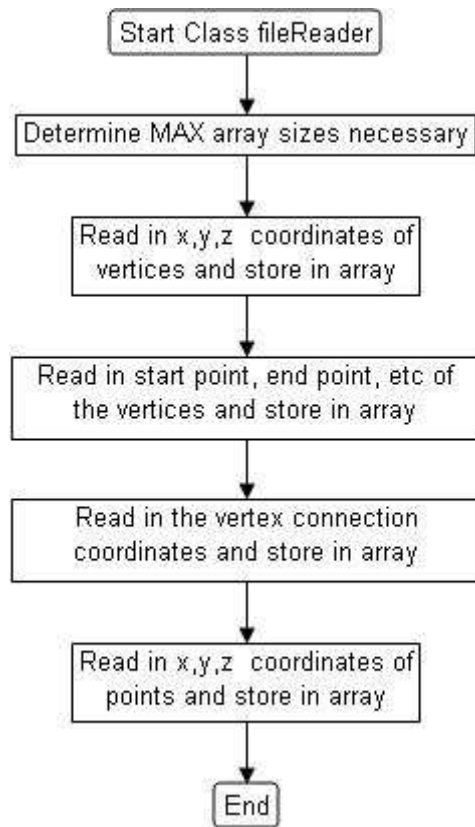


Figure 11: Flowchart of fileReader

### 4.3 Vertex connections

The *returnVertexConnections* class contains a method to generate a multi dimensional array, which contains all vertex connections in another format. The class *fileReader* has a method to collect the vertex data, but for this method only the connections between the points of the vertices are needed, so *fileReader* also has a method which returns only these points. In the following paragraph, the working of the vertex connections class will be explained with help of the flowchart (see Figure 12).

First the size of the output array is determined. The number of rows is equal to the number of points that has a connection. The number of columns is determined by the maximum number of connections that one point has.

The next step is to create the output array with the size determined in the previous step. Then the whole array is filled with “-1”. This is needed for the right processing of this array by another method.

The last step is to fill this array with the read data from the method from the class *fileReader* which returns only the connections between the points of the vertices. If a point is connected to more than one point, all of these points will be stored in the same row. So the first column of every row contains a point and the next columns contain the connected points. When this is done, the array is returned to the calling method.

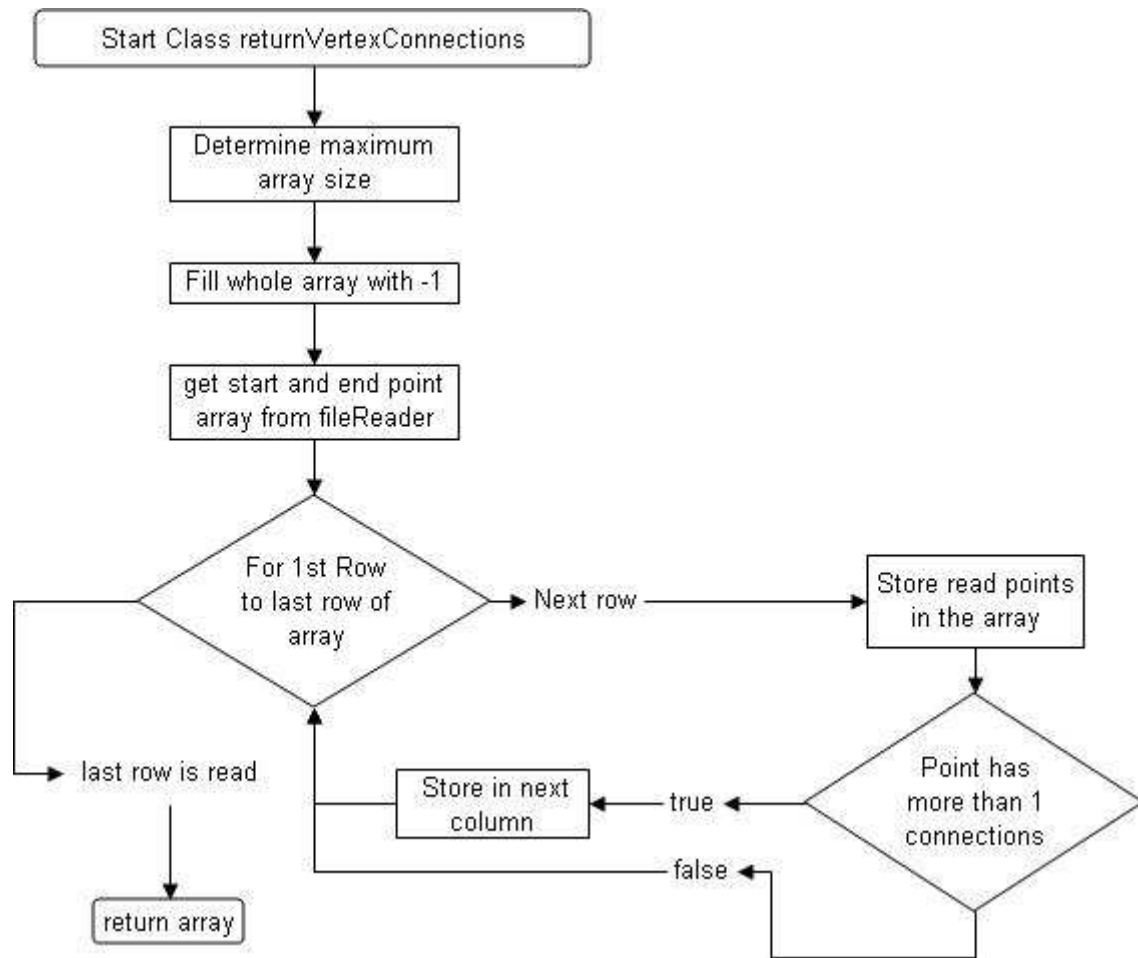


Figure 12: Flowchart of `returnVertexConnections`

## 4.4 Trunk Finder

The *Trunk Finder* class first creates a density grid. This density grid is made by dividing the total area in cells, where each cell is a square with dimension 0.5 x 0.5 meters. For each cell the number of measured points is counted.

The density grid is then used to determine the position of the trunk of each tree by comparing each value with its surrounding values. If the value is higher than all other values in a radius of two cells, then most likely a trunk is located here and the cell is classified as such. In Figure 13 a small fragment of this density grid is shown, all points which are recognized as trunks have been multiplied by “-1”. The flowchart can be found in Figure 14.

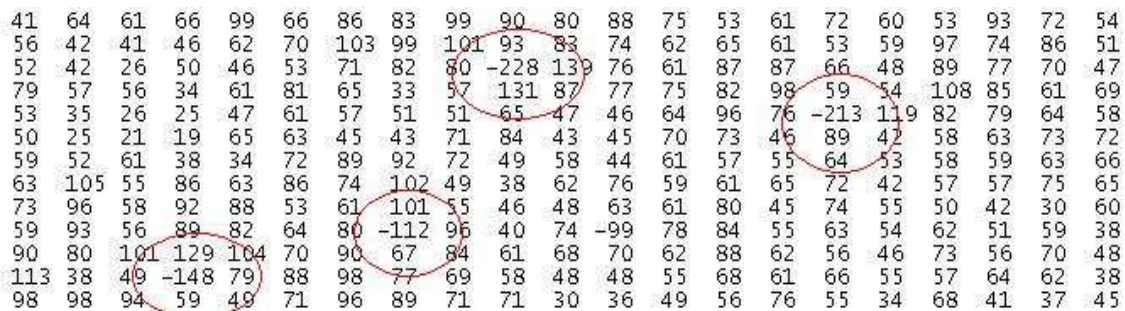


Figure 13: Density grid

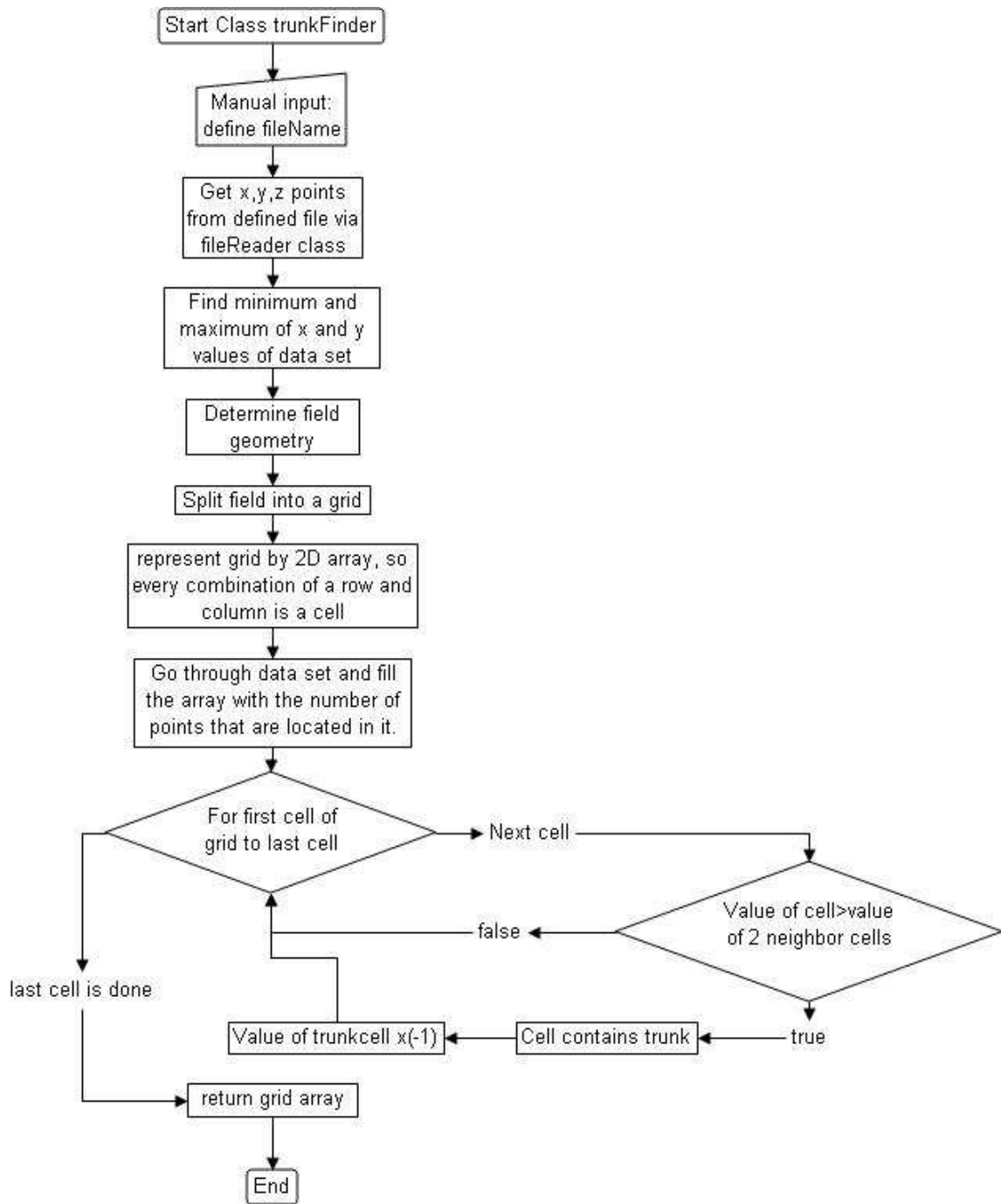


Figure 14: Flowchart of trunkFinder

## 4.5 Shortest path

The shortest path class is the part of the Java program that determines the shortest path between two vertices by implementing Dijkstra's algorithm. Figure 15 is the corresponding flowchart. The flowchart follows the steps of Dijkstra's algorithm as described in section 3.2.

To implement Dijkstra's algorithm (Dijkstra, 1959) into the Java program a small alteration of the algorithm is made. Instead of setting the initial distance to all vertices to infinity, the initial distances are set to -1. The Java program takes this into account.

The two vertices that are the input values of the shortest path method will be two vertices on neighboring trunks. These vertices are found by the *TrunkFinder* class. The shortest path method then returns the shortest path between these two vertices and the two neighboring trees can be separated by using the separation class (see section 4.6).

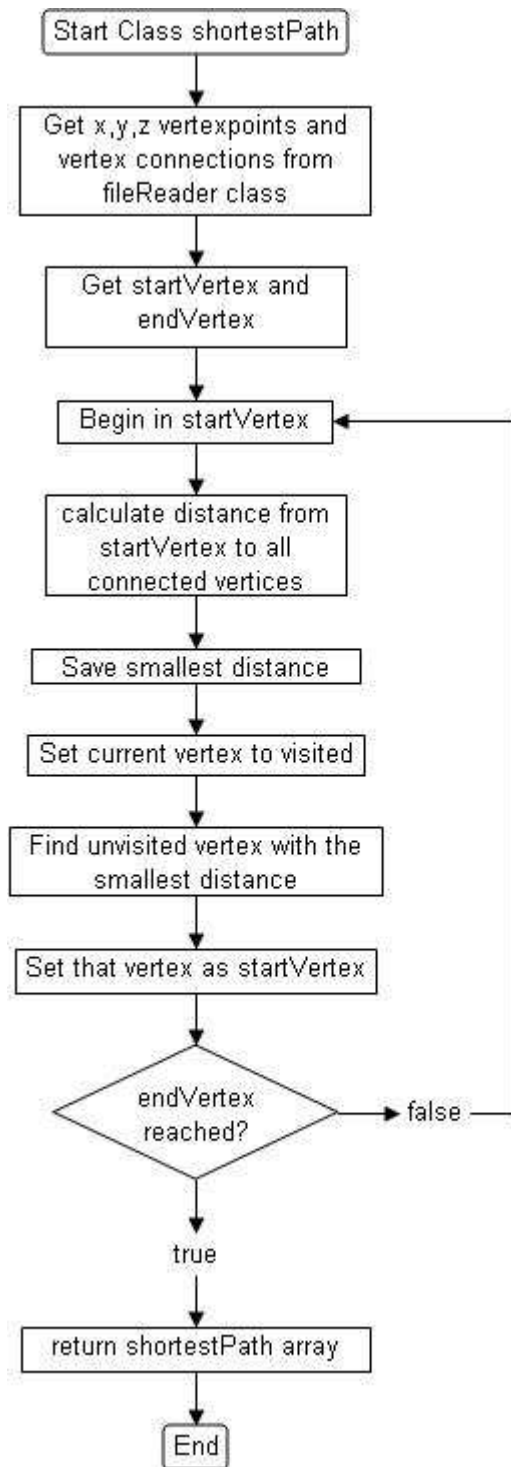


Figure 15: Flowchart shortest path

## 4.6 Separation

The separation class takes a series of vertex points and the vectors between these vertices and checks where a vector can be removed. This has the effect of cutting the path in two, which is needed to separate trees from each other. This class also has a method to find all points belonging to one tree when it has been completely separated from its neighbors. The flowchart of this class is shown in Figure 16.

The program starts by running the path through a method which removes any points between which logically cannot be cut, for example in the trunk. The method finds all the points between which the vector is more or less horizontal, also these points must be relatively far off the ground. This method ensures that fewer calculations are needed later on in the program.

Next, the overlap method to construct the three normal planes explained in section 3.3 and shown in Figure 9. After that, the program looks at the points which are related to the vertices. The normal distance from the point to the plane belonging to its vertex is found by the distance equation, see section 3.3. This is done for both vertices, the distance is stored in an array. Next, the same procedure is applied for all points to the reference mid-plane, again the results are stored in an array.

The following step is to normalize the results by dividing by the length of the vector and squaring the result to weigh it. Taking the medians of each of the three sets of results leads to three values which need to be compared. At the minimal density the path can be cut. When it is found that a path can be cut, the separation method is stopped and a method is called which removes the vector from the array which stores the vectors.

Once all possible paths from one tree to the next have been looked at and split, there will still be some trees left. Using the tree-finder method all points belonging to one tree are found and stored in an array. This is done by taking a known point on the trunk (needed in the first place for the shortest path method) and finding all points connected to this. Once these points have been found, the results can be passed on to the output section of the program.

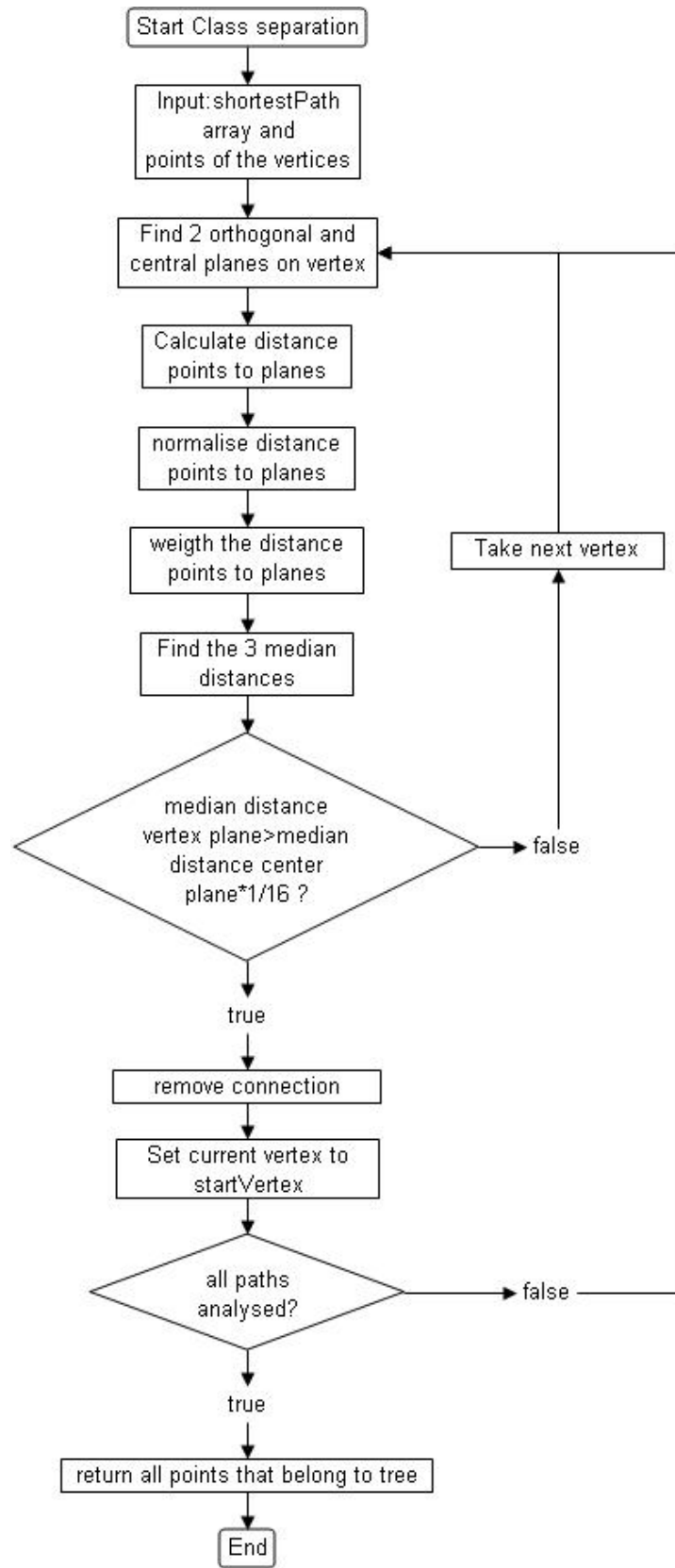


Figure 16: Flowchart separation

## 4.7 Output

The Output class contains two methods. They are called *ConvertIO* and *makeWRLfilePoints*. As can be seen from Figure 17, the last method receives a four-dimensional array from the *ConvertIO* method. This array contains all the data it needs to make a VRML file, which can be read by a program called VRML-viewer.

The *ConvertIO* method receives from the separation class an array, with the same data as originally obtained from the laser scanning. Additionally, the tree number is given next to the xyz-coordinates of the vertices. The *ConvertIO* method also needs the array with the point connections (see section 2.3). It then sorts all these vertices per tree.

After that, the *makeWRLfilePoints* method writes all of the points belonging to a tree to a file and gives each tree a different color. The output file will show all of the points.

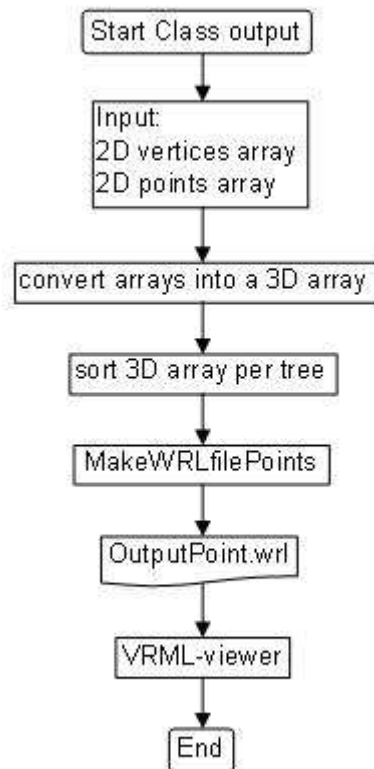


Figure 17: Flowchart output class

## 5 Results

In this chapter the results of the program are illustrated and discussed. Three different forests are processed by the program. From the results of dataset one and three it can be clearly seen that several parameters will be determining for the results. These parameters are height  $h$ , step size area  $ds$ , minimum density, radius-1, radius-2 and horizontal height  $hh$ . Radius-1 and Radius-2 act as boundaries to reduce the calculations on separating the shortest path. The minimum density is set to 99 for all cases to avoid bushes being seen as trees. The step size value for the area is set to 0.5 meter for all cases, resulting in an area size of  $0.25 \text{ m}^2$ . This guarantees the best results. Setting a higher value decreases the accuracy of the density and for a lower value it is more difficult to distinguish the trunks. All other parameters are determined by means of trial and error to get the best results.

First, in section 5.1, the program is successfully tested using a test dataset containing only three trees. In section 5.2 the results of processing of the forests are discussed followed by forest two with illustrations.

### 5.1 Test set

The test set contains three trees. To find the correct number of trunks with the *Trunkfinder* for this file, the height parameter has to be set on 27 meters. Deviation of this value will result in an incorrect number of trunks. From trial and error it follows that Radius-1 is best set to six meters and Radius-2 to ten meters. Also, the horizontal height is set to 0.7 meters.

Figure 18 shows the three trees from the test set. They are distinguished in different colors. The trunk and canopy of the different trees can clearly be seen. Figure 19 illustrates the top view of the same test set. Again three trees are clearly visualized. However, in the top of the figure a yellow spot can be seen, probably the crown of a fourth tree of which the trunk is not given in the test data set. Therefore, it is colored yellow because the program relates it to the yellow tree next to it.

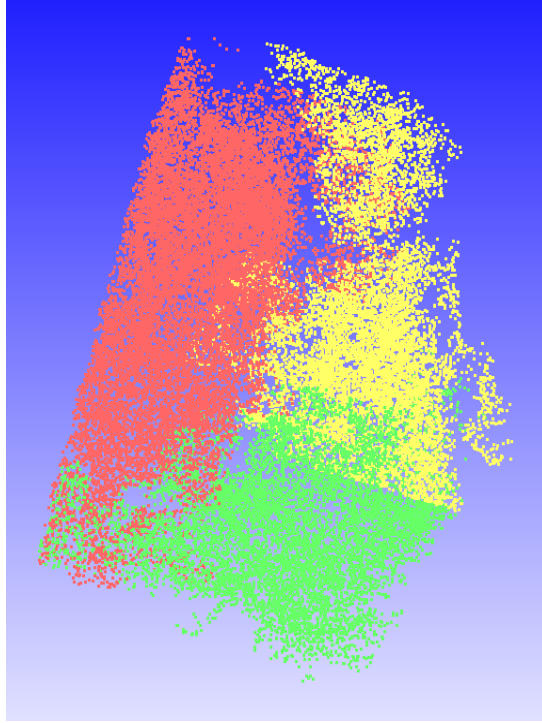


Figure 18: Top view test set ( $h=27$ , separation height=13.5m, Radius-1=6m, Radius-2=10m,  $hh=0.7m$ )

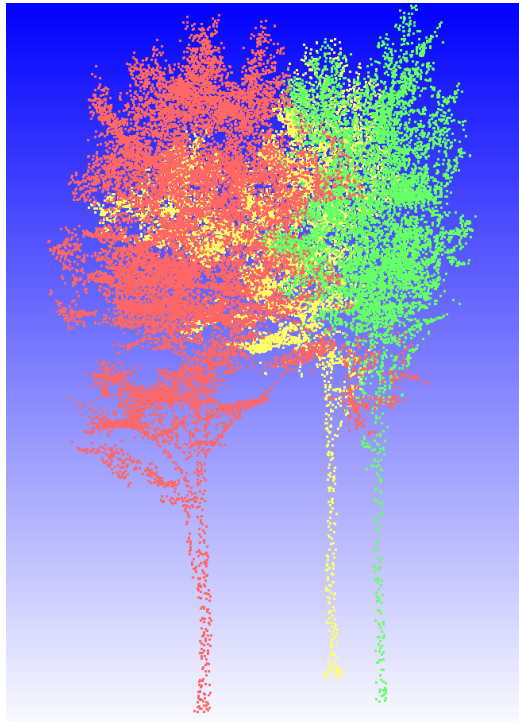


Figure 19: Side view test set ( $h=27m$ , separation height=13.5m, Radius-1=6m, Radius-2=10m,  $hh=0.7m$ )

## 5.2 Forests

### ***First data set and the third data set***

This calculation failed because there were too many connections. The program gets stuck while running this file. Changing the parameters had no positive effect.

### ***Second data set***

The second data set gives the best results. It has been run several times, every time with other parameters. For the first run Radius-1 is set to 10 meters and Radius-2 is set to 15 meters, the minimum separation height is 13.5 meters and the horizontal height is 0.8 meters.

Figure 20 and Figure 21 show the results of the processing of the second file. The purpose of the assignment is to represent every tree in a different color. However, there are more trees than different colors, this is either that the input data is corrupt and that there are some connections missing for some trees. Another possibility is the *Trunkfinder* class not finding enough trunks or the *Separation* class assuming that two trunks belong to the same tree. This can be seen in Figure 20, at multiple locations where two trunks have the same color.

For the second run Radius-1 is set to 3 meters and Radius-2 is set to 6 meters, the minimum separation height is 14.5 meters and the horizontal height is 0.5 meters.

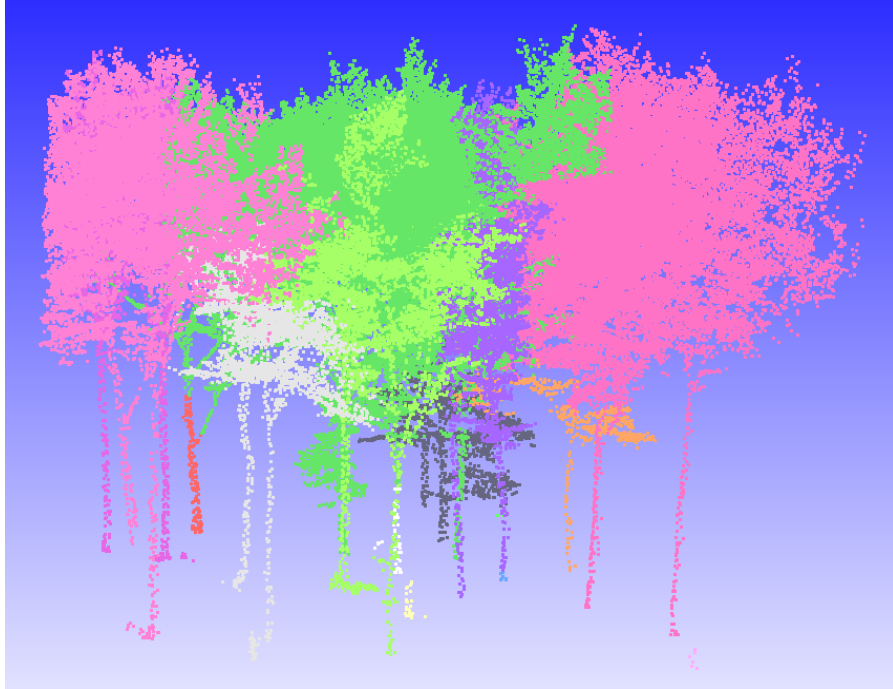


Figure 20: Side view data set 2 ( $h=99\text{m}$ , separation height=13.5 m, Radius-1=10m, Radius-2=15m,  $hh=0.7\text{m}$ )

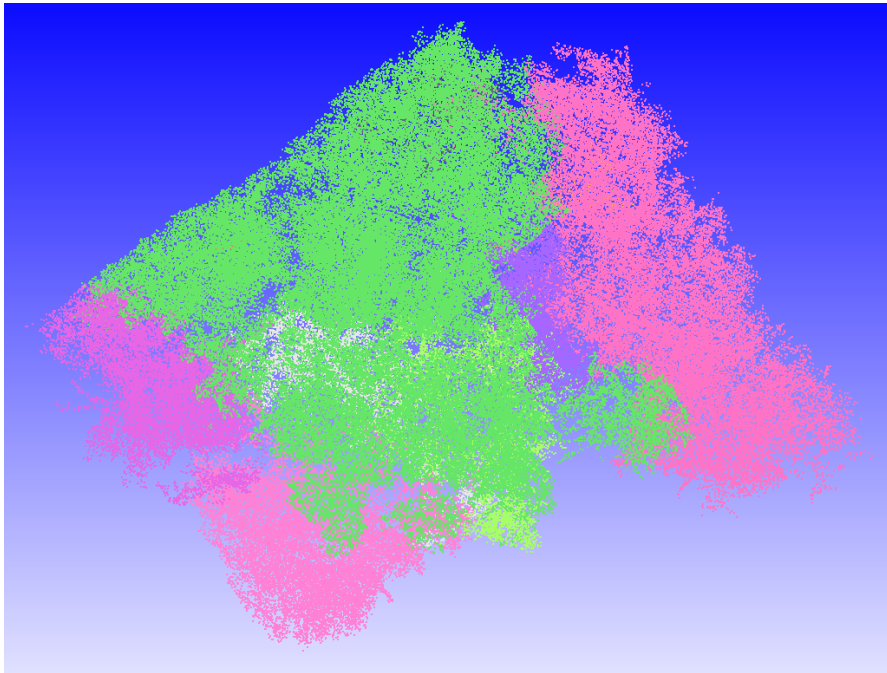
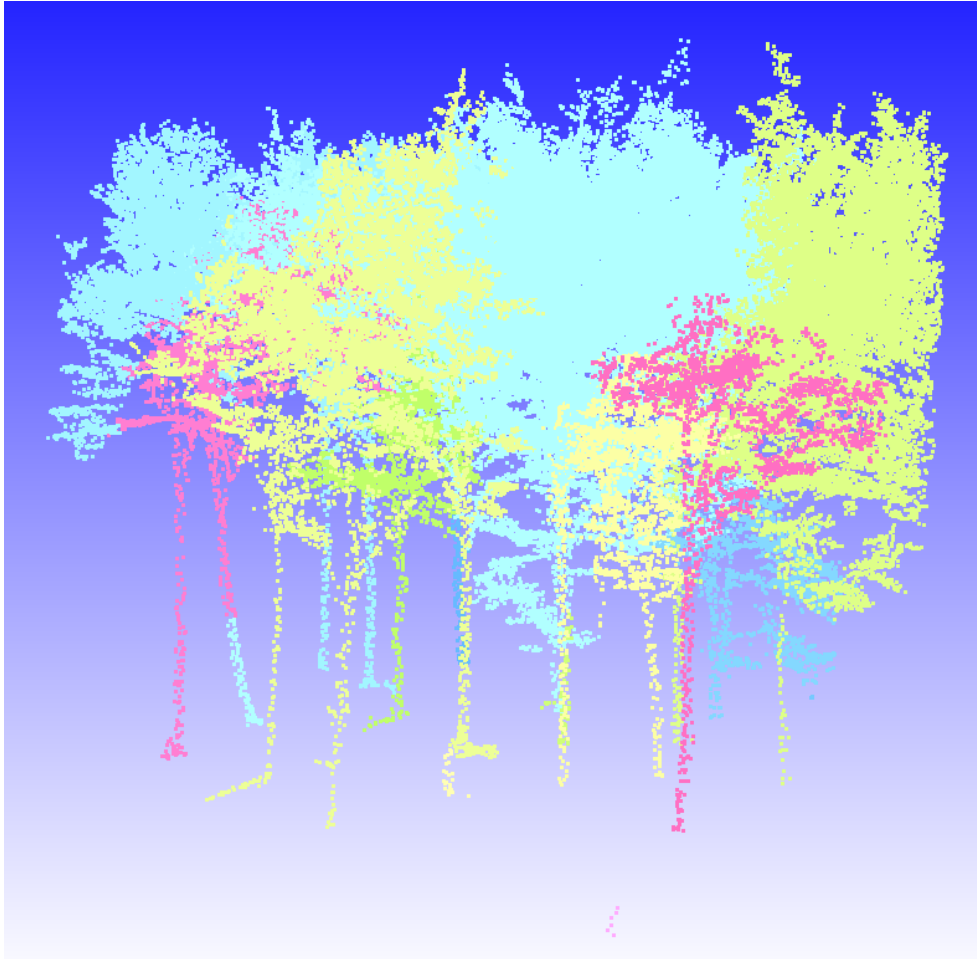


Figure 21: Top view data set 2 ( $h=99\text{m}$ , separation height=13.5 m, Radius-1=10m, Radius-2=15m,  $hh=0.7$ )

Figure 22 and Figure 23 show the results of the processing of the second file with other parameters. As can be seen the results are somewhat similar to Figure 20 and Figure 21, in that there are multiple locations where two trunks are considered to be part of the same tree.



*Figure 22: Side view data set 2 ( $h=99m$ , separation height=14.5 m, Radius-1=3m, Radius-2=6m,  $hh=0.5$ )*

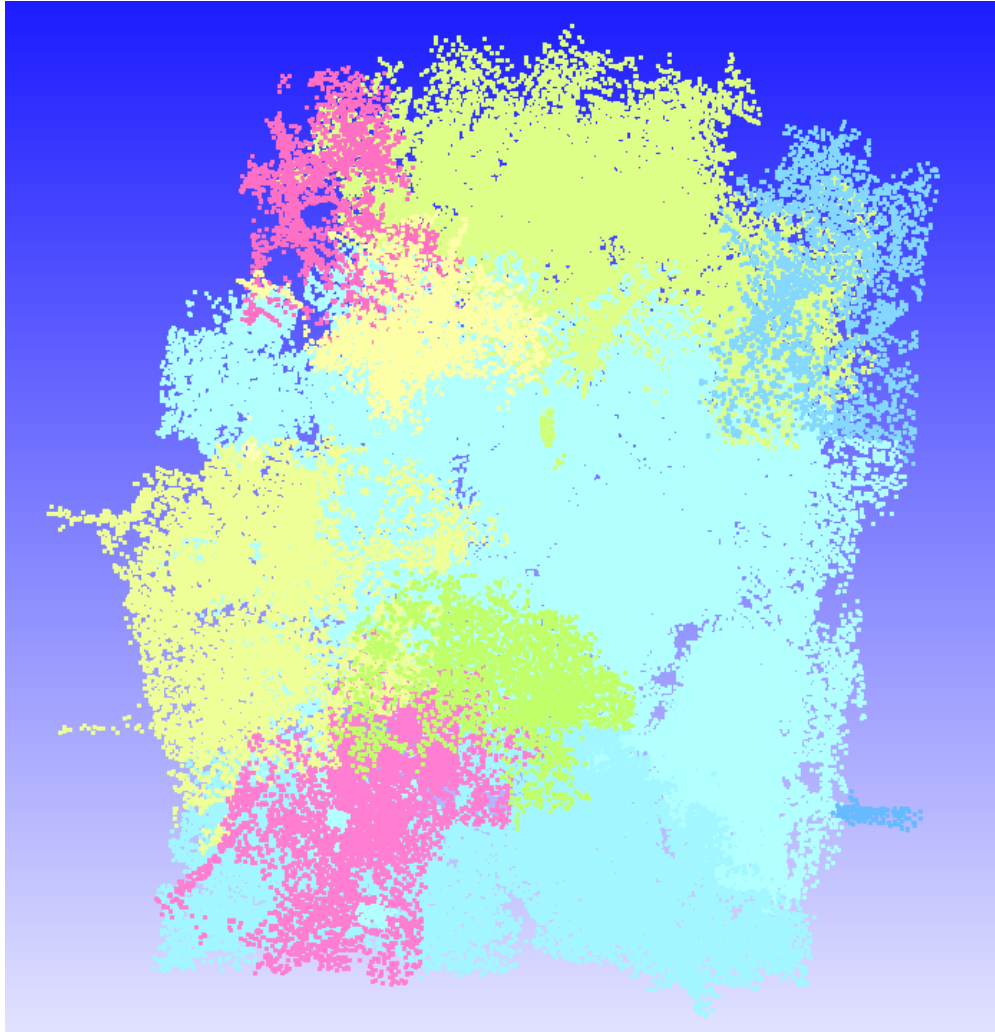
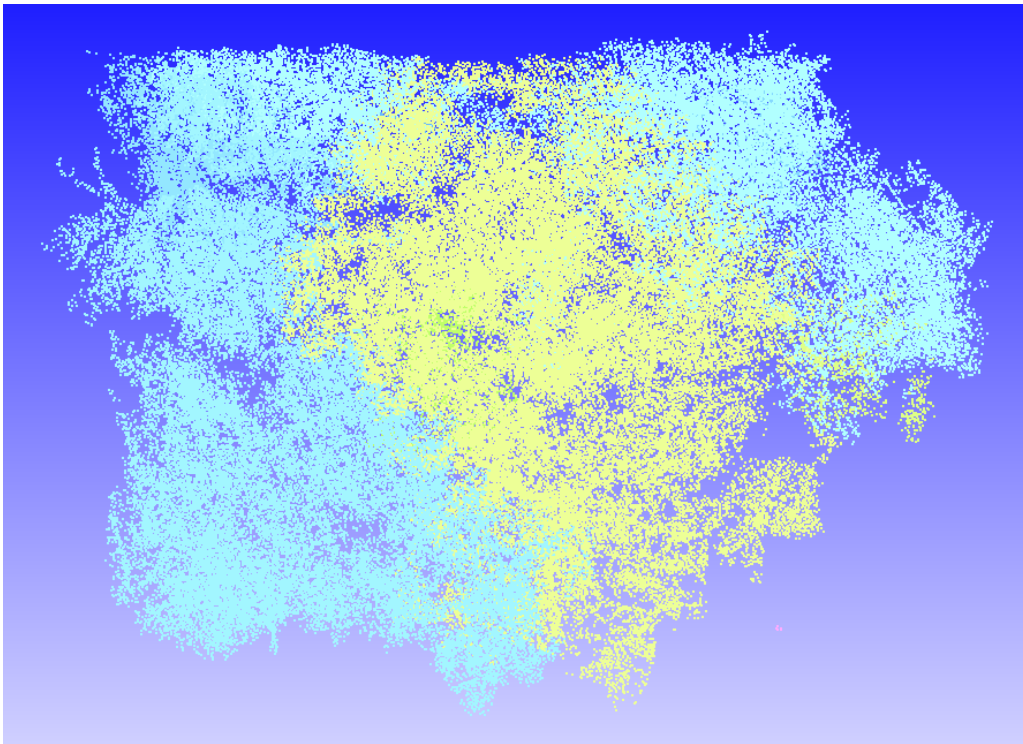


Figure 23: Bottom view data set 2 ( $h=99\text{m}$ , separation height= $14.5\text{ m}$ , Radius-1= $3\text{m}$ , Radius-2= $6\text{m}$ ,  $hh=0.5$ )

For the third run Radius-1 is set to 4 meters and Radius-2 is set to 6 meters, the minimum separation height is 20 meters and the horizontal height is 0.5 meters.

As can be seen from Figure 24 and Figure 25, with these parameters some trees can not be seen in the top view. These trees are cut too early. Also when the color difference is small, it is difficult to see the different trees.

As can be seen from Figure 24 and Figure 25, with these parameters some trees can not be identified in the top view. These trees are cut too early. Also when the color difference is small, it is difficult to see the different trees.



*Figure 24 Top view data set 2 ( $h=99m$ , separation height=20 m, Radius-1=4m, Radius-2=6m,  $hh=0.5$ )*

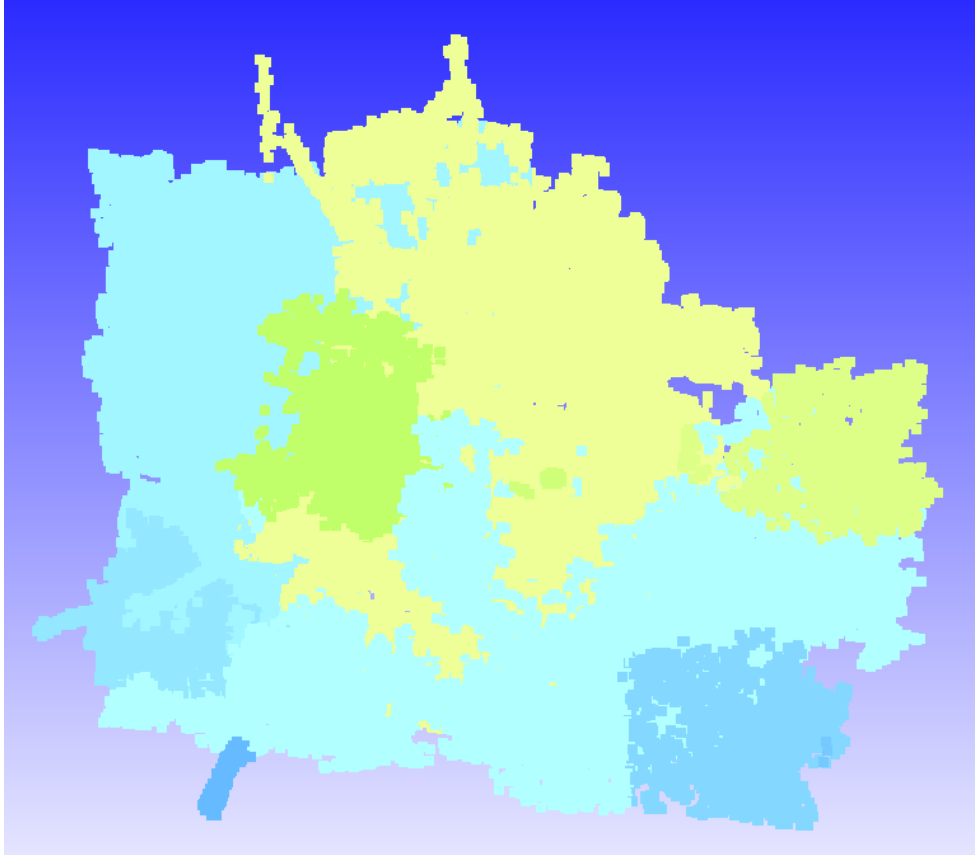


Figure 25: Bottom view data set 2 ( $h=99\text{m}$ , separation height=20 m, Radius-1=4m, Radius-2=6m,  $hh=0.5$ )

## **6 Conclusions and recommendations**

### **6.1 Conclusions**

This report describes how a Java based program was developed that structures point data obtained from airborne laser scanning over a forest area, and that visualizes the results in such a way that individual trees can be distinguished. Due to the time limit and the size of the program, it was not possible to optimize the program. The program is able to distinguish some, but not all, trees from each other. When only three trees of a forest are considered, the program splits the trees correctly.

The ability of the program to separate the trees correctly depends on several parameters that have to be manually put in. These parameters determine where the program looks for trunks and where to separate the trees. If the height to look for trunks is set too high, the program will find too many trees. If this height is set too low, the program will not find any trees. The parameters vary for different data files and have to be found by trial and error. Therefore, this will need more time.

Three data files were given to be processed. Only the processing of one file gave satisfying results. The results of the other two files can still be improved. To conclude, the program looks promising, but more time is needed to optimize it.

### **6.2 Recommendations**

To optimize the results, a few recommendations are presented in this chapter. Java is used to separate the different trees. Since the running time of the program varies between five and eight hours, the choice of Java might not be optimal. However, the use of Java was required for this assignment.

To obtain better results, several other methods to separate the trees can be used. After finding the trunks, the forest can be cut into small pieces of three to four trees using a Voronoi decomposition (Weisstein, n.d.). Each of these small regions of the forest can be separated in about six minutes. These results can be combined again to get the original forest. This method will yield better results and the processing will require less time.

Another possibility to separate the trees is by executing the class in a certain direction, instead of randomly considering trees. This way, when a tree is separated from

its nearest neighbors, the tree will not be considered for separation anymore. The running time of the program will be reduced by using this method, as trees are not considered for separation multiple times.

Another way to decrease the running time, is to use a computer with a higher clock rate. With a lower running time it is easier to optimize the results by varying the parameters, such as the radius to neighboring trees and the separation height.

## Bibliography

Dijkstra, E. W. (1959). *A note on two problems in connexion with graphs*. In *Numerische Mathematik 1*

Fleck, S (2002): Integrated analysis of relationships between 3D-structure, leaf photosynthesis, and branch transpiration of mature *Fagus sylvatica* and *Quercus petraea* trees in a mixed forest stand, University of Bayreuth: Bayreuth Institute for terrestrial ecosystems research

Gunter, Brian C. (2009). *Introduction to Earth Observation*. Retrieved on 10-03-2009, from [http://blackboard.tudelft.nl/webapps/portal/frameset.jsp?tab\\_id=\\_2\\_1&url=%2fwebapps%2fblackboard%2fexecute%2flauncher%3ftype%3dCourse%26id%3d\\_21808\\_1%26url%3d](http://blackboard.tudelft.nl/webapps/portal/frameset.jsp?tab_id=_2_1&url=%2fwebapps%2fblackboard%2fexecute%2flauncher%3ftype%3dCourse%26id%3d_21808_1%26url%3d)

Lay, David C. (2006). *Linear algebra and its applications*. Third edition. Boston: Pearson Education.

Rees, W.G.(2004). *Physical Principles of Remote Sensing*, Cambridge: Cambridge University Press

Spencer B. Gross, Inc. (2003). *LIDAR (Light detection and ranging)*, Retrieved on 03-02-2009, from <http://www.sbgmaps.com/lidar.htm>

Taylor, J. (n.d.). *Plane through a point with a given normal vector*. Retrieved on 16-3-2009, from <http://www.jtaylor1142001.net/calclat/Solutions/VPlanes/VPtNorm.htm>

Thomas H. Cormen et al (2001). *Introduction to Algorithms*. Second Edition. MIT Press and McGraw-Hill.

Weisstein, Eric W. (n.d.). *Point-Plane Distance*. Retrieved on 27-2-2009, from MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/Point-PlaneDistance.html>

Weisstein, Eric W. (n.d.). *Voronoi Diagram*. Retrieved on 17-3-2009, from MathWorld--A Wolfram Web Resource. <http://mathworld.wolfram.com/VoronoiDiagram.html>