A comparison of visualisation techniques for complex networks

En jämförelse av visualiseringsmetoder för komplexa nätverk

VIKTOR GUMMESSON
A comparison of visualisation techniques for complex networks
Abstract

The need to visualise sets of data and networks within a company is a well-known task. In this thesis, research has been done of techniques used to visualize complex networks in order to find out if there is a generalized optimal technique that can visualize complex networks.

For this purpose an application was implemented containing three different views, which were selected from the research done on the subject. As it turns out, it points toward that there is no generalized optimal technique that can be used to default visualize complex networks in a satisfactory way. A definite conclusion could not be given due to the fact that all visualization techniques which could not be evaluated within this thesis timeframe.
Referat

Contents

1 Introduction 1
  1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 Arising problems with growing data . . . . . . . . . . . . . . . . . 1
    1.2.1 Edge and vertex crossing . . . . . . . . . . . . . . . . . . . . 1
    1.2.2 Labeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.2.3 Situation awareness . . . . . . . . . . . . . . . . . . . . . . . 2
  1.3 This thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Visualization techniques and their theory 5
  2.1 Fundamental techniques . . . . . . . . . . . . . . . . . . . . . . . . . 5
    2.1.1 Force-Directed . . . . . . . . . . . . . . . . . . . . . . . . . . 5
    2.1.2 Navigation through zooming . . . . . . . . . . . . . . . . . . . 7
  2.2 Two-dimensional space . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.2.1 BioFabric . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.2.2 HivePlots . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.2.3 TreeMap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.3 Three-dimensional space . . . . . . . . . . . . . . . . . . . . . . . . . 15
    2.3.1 GerbilSphere . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.3.2 H3: laying out large directed graphs in 3d hyperbolic space . 17
  2.4 Looking forward . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3 Method 21
  3.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    3.1.1 Programming environment . . . . . . . . . . . . . . . . . . . . 21
  3.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
    3.2.1 Programming libraries . . . . . . . . . . . . . . . . . . . . . . . 22
    3.2.2 Layout views . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
    3.2.3 Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4 Results 25
  4.1 Library performance . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    4.1.1 Attribute matrix . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    4.1.2 Results from library tests . . . . . . . . . . . . . . . . . . . . 26
4.2 Application .................................................. 27
  4.2.1 Main application ...................................... 27
  4.2.2 Force-Directed (FD) based view ...................... 28
  4.2.3 Two-dimensional view - BioFabric .................. 32
  4.2.4 Three-dimensional view - GerbilSphere .............. 35

5 Discussion and Conclusions ................................. 39
  5.1 Which type of visualisation technique is then best suited to visualize big and complex networks? ......................... 39
  5.2 Environmental aspects .................................. 40

A Library evaluation ........................................... 41
  A.0.1 Libraries of interest ................................ 41
  A.0.2 Library selection for evaluation .................... 48
  A.0.3 Test set up ........................................ 48
  A.0.4 Results from library tests ........................... 49

B Implementation details ....................................... 53
  B.1 Views .................................................. 53
    B.1.1 Force-directed(FD) based view .................... 53
    B.1.2 Two-dimensional view - BioFabric ................. 53
    B.1.3 Three-dimensional view - GerbilSphere .......... 54
  B.2 Data representation .................................... 54
    B.2.1 GraphElement ..................................... 54
    B.2.2 Data representation within application - DataManager ..... 54
    B.2.3 Data parsers ...................................... 54

References .................................................. 57
Chapter 1

Introduction

This chapter is intended to give an introduction to the subject of visualizing networks.

1.1 Background

To be able to visualize different networks is an important part in many fields, such as science and technology. For example, computer science that deals with complex networks of relationships between system components, displaying relations in a social network, molecular biology that studies the interactions between various systems of cells, e.t.c.

There are different approaches to take when visualizing networks. The most traditional approach is to represent the network as some kind of graph, because many structures in different scientific fields can be represented as a vertice-link graph. The vertices represent different components which are visualized with a shape and edges represent different component relations which are visualized by a connecting line between two vertices.

1.2 Arising problems with growing data

Though the traditional ways of visualizing graphs are pleasing and give an intuitive way of looking at relations [26], there are problems which can arise when the networks that need to be visualized are of a larger size. The traditional way may be sufficient when dealing with networks of small sizes of vertices and relations, but what will happen when the networks become complex and have hundreds or thousands of vertices?

1.2.1 Edge and vertice crossing

When the vertice count becomes larger, the available area dedicated to layout these vertices becomes relatively smaller. This can contribute to vertices starting to
overlap each other, making it hard to distinguish between a set of different vertices.

A similar problem arises concerning edges. Depending on the layout of the vertices a different amount of edges may overlap, crossing each other. This may not be a problem if the number of crossings is low or the angle between two edges is high. But when this angle decreases and the number of crossings increases it, becomes harder to distinguish between specific edges, to see which edge connects to which vertice. If the relations are of a large enough size, the cluster of edges may become one big black area.

When dealing with layout techniques one strives to layout the vertices in a way that minimize vertice- and edge crossings.

1.2.2 Labeling

Labeling vertices and edges in a network becomes more challenging as the network grows. In fact the optimal label placement of a graph has been shown to be NP-Complete [29]. The task of labeling can be divided into three different labeling tasks:

- Labeling area features (clusters).
- Labeling line features (edges).
- Labeling point features (vertices).

1.2.3 Situation awareness

Human and psychology factors play a role when visualizing a network. Situation awareness is a term in this aspect. Endsley [17] defines situation awareness as:

\[\text{Situation Awareness is the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status into the near future}\]

Situation awareness becomes important to consider when choosing a visualisation technique.

Figure 1.1 shows what can happen when trying to visualize big networks.
1.2. ARISING PROBLEMS WITH GROWING DATA

Figure 1.1. Example of a graph where the problem of vertex- and edge crossing becomes obvious
1.3 This thesis

This thesis revolves around the question:

Which types of visualization techniques are suitable for visualizing large and complex networks?

With the corresponding hypothesis that:

One can conclude that some visualization techniques are better suited than others and that one or several may be best for the task at hand.

In chapter 2 different common visualisation techniques are described. Chapter 3 goes through the methodology used to evaluate a set of different techniques. Chapter 4 provides the results from chapter 3. Chapter 5 discusses the results and conclusions.
Chapter 2

Visualization techniques and their theory

There exists a number of different approaches and techniques used to visualize large and complex networks. This chapter is intended to introduce some of these and explain how they work.

2.1 Fundamental techniques

Though there exists a number of different approaches many of these are based on some fundamental technique or concept. Two major aspects are important to consider when trying to visualize a network. The first is about the part that most probably relate to graph visualization, the actual layout algorithm that decides where each vertice is to be placed and how the edge routing is made. The second is the aspect of how one is to navigate a graph when it has been generated, that is navigation such as zooming and panning.

2.1.1 Force-Directed

Force-directed is a popular class for a type of algorithm for calculating layouts of graphs. They are constructed to strive towards generating graphs with vertice positions so that edges in the graph are of equal length and the layout displays as much symmetry as possible. One of the pros with these algorithms is that they are flexible, they do not rely on domain specific knowledge but instead only use the information contained within the structure of the graph. Graphs produced by these algorithms tend to be aesthetically pleasing and exhibit symmetries [26]. Figure 2.1 shows an example of a graph drawn with a force-directed algorithm.

These algorithms are based on assigning forces between vertices and edges in a graph, simulating the motion of the edges and vertice or minimize their energy. One of the first force-directed algorithm dates back to 1963 with the algorithm of Tutte [56] and is based on barycentric representation [26]. Though the more commonly
used algorithms such as vy Eades [16], and Fruchterman and Reingold [18] both rely on spring forces similar to those in Hooke’s law. Here there are repulsive forces between all the vertices in a graph while in the same time attractive forces between vertices and their neighbours. As in the Eades algorithm [26] where they have vertices represented as steel strings and edges as springs. They start from an initial random layout and then lets the system move towards a state where minimal energy between vertices is achieved.

Besides striving towards equal edge length and displaying symmetry, one can argue that the graph layout also should strive to have an even vertex distribution for a more pleasing layout. The algorithm of Fruchterman and Reingold covers this by using a bit of a different physical model, seeing the vertices in a graph as atomic particles or as celestial bodies, where the attractive forces are defined as [26]:

\[ f_a(d) = \frac{d^2}{k} \]

Repulsive as:
2.1. FUNDAMENTAL TECHNIQUES

\[ f_r(d) = -\frac{k^2}{d} \]

\(d\) is the actual distance between two vertices and \(k\) is the optimal distance. \(k\) is defined as:

\[ k = C \sqrt{\frac{\text{area}}{\text{number of vertices}}} \]

Besides this, the algorithm also uses the notion of temperature as a refining step. This works so that when the algorithm improves the layout, the adjustments become smaller from the last iteration of the algorithm.

As the graph size grows larger, graphs with more than a few hundred vertices, a problem arises with the basic force-directed algorithms. The fact is that the used physical model has multiple local minima, and a graph produced with only a local minimum can be much worse than would it be produced with a global minimum. Algorithms have been developed trying to avoid local minima, such as the Hadany and Harel algorithm [23], which is based on a multi-level layout technique that works with graphs containing 15000 vertices.

In multi-level techniques the graph structure that is to be drawn is viewed in substructures where each substructure has less complexity than the whole. These substructures are then laid out in order from the most simple structure to the most complex one. Hadany and Haler [23] said that a natural strategy for drawing graphs in a pleasant manner is to first consider an abstraction, disregarding some of the graphs fine details, and then add details to correct the layout. They also take up the importance of preserving the essential features of a graph in the abstraction, in other words it is important to be able to point out the essential features in a graph.

2.1.2 Navigation through zooming

The way one zooms becomes a large part when navigating a graph, how one does this greatly affects the situational awareness. When navigating through a graph, both global context and local details are of importance. Global context is provided when one can navigate through a graph and still be able to orient oneself according to the graph. This most often requires one to be far zoomed out to see the whole. Though, when zoomed out, the local details are not on a high enough level to give any real information. So to get out more detailed information, one is needed to zoom in the graph to a specific area, which is when a tunnel vision problem arises, causing one to easier lose orientation and information of the overall dependencies when the context is lost.

FishEye view is a technique that addresses this problem of tunnel vision. One can compare the technique to a fisheye lens used by cameras for the creation of wide panoramic images. The technique allows for displaying larger areas than in a standard image. It has a higher number of details in the focus, which diminish with a growing distance from the focus. Figure 2.2 shows an example of this.

Doug Schaffer et al. [47] used the FishEye method while navigating networks.
where vertices were represented by squares and vertices did not overlap each other. The data used was clustered in a way that one vertex could contain a subset of different vertices. Figure 2.3 shows an example of such a graph and a possible way to cluster it.

When zooming, one actually zooms a vertex/vertices (cluster/clusters). This translates into the FishEye view where the zoomed vertices are in focus, getting larger, and the other vertices being outside focus get shrunken. Figure 2.4 shows an example of a graph before a zooming action and after.
2.2. TWO-DIMENSIONAL SPACE

Figure 2.4. Example of a graph when zoomed (element b and a selected for zooming) [47]. Left hand side shows the graph before zooming and which segments are being enlarged and which are being shrunken. Right hand side shows the graph after zooming.

2.2 Two-dimensional space

This section introduces methods used to visualize big networks in a two-dimensional space.

2.2.1 BioFabric

BioFabric [28] is a method that uses a different approach to represent a graph than the traditional way where one represents vertices as a shape, like a circle or a rectangle, and edges as lines between vertices. Instead, vertices are represented as one-dimensional horizontal lines and edges as one-dimensional vertical lines. These vertical lines start at one of the horizontal lines (one specific vertex) and end at another, representing a connection between these lines (vertices). This different approach lets one get away from the problem of vertice and edge crossings. It guarantees no edge overlapping and no vertice overlapping.

One difficulty that can arise with many methods is when one handles updates of graphs, for example when adding or removing vertices to a graph. This can result in major alterations for a graph’s layout when only a few vertices are introduced. This problem exists in BioFabric as well, but because of the way BioFabric handles additions to a graph (adding vertical- and horizontal-lines), this helps to not having too much of an impact on a graphs layout. Though how much it alters the graph is dependent on how many vertices are added and how many connections said vertices has.

As for how to layout the vertices and edges there are different approaches one can take. One basic approach is to do a breadth first traversal of the data to be displayed, where neighbouring vertices are visited in the order determined by their degree (the data are structured by degree of vertices). Next follows an example of a way of assigning vertices and edges that uses this approach [28]. Here the vertices
is referred to as rows and edges as columns.

Vertex assignment:

1. Set row 1 as the next available row.

2. Find the highest degree vertex not yet processed, and assign it to the next available row. Make that row the current row; increment the next available row.

3. Take the vertex assigned to the current row and order its neighbours based upon their degree, highest degree first.

4. Traversing the neighbour vertices using that order, if the vertex has not yet been assigned, assign it to the next available row and increment the next available row.

5. Increment the current row. If a vertex has been assigned to that row, go to step 3. If not, go to step 2.

Edge assignment:

1. Set column 1 as the next available column. Make row 1 the current row c.

2. For current row c, get all the unassigned edges for the vertex in that row. Note that since we are not dealing with shadow links [28], all unassigned edges must connect to rows \( \Omega \) c.

3. For each row \( r \geq c \), create a set S of edges incident on c and r. Order these sets by increasing row number r, so that edges will be assigned in order of increasing length.

4. Iterating through the ordered list of sets, for each set S, order those edges in S based on lexicographic ordering of the link relation description, and assign them to the next available columns in this order; increment next available column appropriately. If there is a pair of directed edges with the same link relation description, downward links are assigned before upward links.

5. Increment the current row, and go to step 2.

Figure 2.5 is an example of a big network visualized with BioFabric using this approach.

Other approaches that can be used are to try to group vertices based on similarity and difference between their connectivity. The way to represent similarity could be to use cosine similarity [50] or Jaccard similarity [52]. Figure 2.6 shows a network visualized using similarity weights, resulting in a less compact layout than the basic approach.
Figure 2.5. This is a depiction of the yeastHighQuality.sif data set [3-5] containing over 3000 vertices and 6,800 edges. The key feature of the BioFabric presentation is that vertices are depicted as horizontal lines, one per row; edges are presented as vertical lines, each arranged in a unique column. Note how the use of darker colors for rendering edges and lighter colors for rendering vertices assures that the former stand out despite the crossover. A) The view of the full network, laid out with the default algorithm. B) Detail of network shown boxed in network A, which highlights one advantage of the BioFabric presentation technique: similarities, and differences, in the connectivity of different vertices are immediately apparent. C) The six vertices and first neighbours depicted in a subset view, where all extra space has been squeezed out, creating a compact presentation that still retains all the relative positioning from the full view. Note how the full inventory of edges incident on the six vertices also includes those on the left originating from higher vertex rows.
BioFabric has one release which is an open-source Java application with some documentation [3]. Though BioFabric is built on a relatively easy and intuitive algorithm, one could take the option of implementing an own version, having own customized features that suits one’s purpose.

Figure 2.6. Layout that tries to place vertices with similar connectivity next to each other in the linear ordering of vertices.

2.2.2 HivePlots

HivePlots is a visualisation algorithm that uses a number of radially oriented linear axes that have a coordinate system based on vertices properties. A network’s vertices are layed out on these axes. Connecting vertices are shown with edges between them, visualized as curves between vertices. Figure 2.7 shows an example of a HivePlot.

Initially before the layout is made, a number of structural parameters are calculated such as degree, flow, pagerank, clustering coefficient etc. Which parameters to use is up to the user to decide, they need to be appropriate for the network being visualized. For example one could use the clustering coefficient to distinguish between hubs and clusters. Next these parameters are used to set up rules that are used to assign vertices to an axis and decide its coordinate. These rules are often boolean rules. Example of rules could be:

- Is the vertice a sink?
- Is the vertice a source?
- Clustering coefficient < 0.5?

If a HivePlot can be created with three axes this is preferred [27], laying the axes with a uniform radial distribution. Because with three axes you get a layout were no pair of edges cross each other and no edge connected to two axes will cross
2.2. TWO-DIMENSIONAL SPACE

Figure 2.7. Example of a HivePlot containing 2500 vertices and 5900 edges.

another axis. Though this is not restrained to only three axes, it becomes hard to assign vertices to axes so these features become obtained for a different number of axis.

For HivePlots there are some choices of use, one used is a Java based library [2]. There are also libraries for R [45], HiveR [1], that support HivePlots in the two-dimensional and three-dimensional space. The framework D3.JS [14] is another option that is a JavaScript to create hive plots. And pyveplot [43] is a library for HivePlots in Python [44].

2.2.3 TreeMap

TreeMap is a technique to present graphs in sequences of nested boxes [24]. TreeMap requires the data to be hierarchy structured as a tree. Figure 2.8 shows an example. The size of individual boxes becomes significant in a TreeMap layout, where the user specifies how they should grow. Take for example if Figure 2.8 shows data that represents a file system. The size of a box could then be proportional to the size of the file it represents. The colors of the boxes represent the hierarchy, same color of boxes belong to the same file.

For TreeMaps there are some choices to make. For .Net, which is this thesis’s working environment, there is the WPF TreeMaps & SquarifiedTreeMaps control library [11], though it has poor documentation. Another alternative is the .NET Treemap Control library [15]. Here again the problem lies in little documentation.
and it is hard to get information about the library when it was part of an old Microsoft research project called Netscan.

Figure 2.8. Example of a Tree-map
2.3. THREE-DIMENSIONAL SPACE

2.3 Three-dimensional space

In the hope of acquiring more space for the layout of a network one can take the approach to go from a 2D to a 3D environment. In other words one can strive to get away from the Euclidean space to another space that provides more dimensions. An important aspect that follows from going to a 3D view is that the system should be navigatable. This because in a 3D view, vertex and edge occlusion is bound to happen. Being able to alter a view by navigating becomes an important tool that helps to find a view where no occlusions occur.

Hyperbolic space

The hyperbolic space has the property that it has more room compared to the familiar Euclidean space [37]. [60] states that the fifth postulate in the Euclidean plane geometry can be formulated as:

“Through a given point, not on a given line, one and only one line can be drawn which does not intersect the given line.”

As in the hyperbolic plane geometry they introduce the Characteristic Postulate:

“Through a given point, not on a given line, more than one line can be drawn not intersecting the given line.”

Moreover two lines that are parallel in the Euclidean space are always the same distance apart. As in the hyperbolic space parallel lines are not equidistant. For instance two parallel lines in the hyperbolic space that do not intersect can be separated by increasing distance the further away one moves from the origin. Figure 2.9 shows this compared to the Euclidean geometry.

![Figure 2.9](image)

**Figure 2.9.** Parallel lines in Euclidean space are always the same distance apart. In hyperbolic space the distance between two lines that never meet does indeed change. Here we show two geodesics which never meet but are not equidistant: the further they extend away from the origin, the more room there is between them.
Normally to make use of the hyperbolic space, to use the extra space, one goes about to perform a layout algorithm in the hyperbolic plane or space and then display the results in the Euclidean plane or space. Some models to do this have been created. Best known are the Klein and the Poincaré models [24].

2.3.1 GerbilSphere

There have been studies on 2D vs 3D user interfaces that have shown that in many cases 2D exceeds 3D. Though the more space in 3D is still compelling. GerbilSphere is an inner sphere 2D system that tries to use the benefits from both a 2D approach as well as a 3D approach.

GerbilSphere works in a way that it places the observer inside a sphere while projecting the network on to the surface of the sphere. As part of the layout, GerbilSphere uses an extended version of the Fruchterman and Reingold force-directed algorithm to apply to the three dimensional space. However this is not enough to work on the surface of a sphere. To apply the forces to the surface of a sphere, GerbilSphere uses an algorithm described by Kobeourov and Wampler [25]. For more technical information about the data structure and how their layout algorithm works see [48].

Zooming in GerbilSphere is viewed as having a world camera attached to one end of a tether and having the other end attached to the center of the sphere. When zooming in and out it can be seen as moving the world camera along this tether. Figure 2.10 shows when zoomed out respectively zoomed in.

GerbilSphere implements a 2 1/2D interface, advocated by Ware [58]. When a user is positioned inside the sphere and zooms in, the part of the network when zoomed in will be visualized on a flat 2D surface, as seen in Figure 2.11. When zooming out one can still have the point of interest in view, trying to gain more global context of the network. Lastly one can zoom out enough to place the view outside the sphere, seeing the network on a 3D sphere.

GerbilSphere is an open-source project. No API is available, though good documentation is presented within the code.
2.3. THREE-DIMENSIONAL SPACE

Figure 2.11. Spherical volume grid based

2.3.2 H3: laying out large directed graphs in 3d hyperbolic space

T. Munzner [37] visualizes graphs in the three-dimensional hyperbolic space by placing the network, represented as a spanning trees, inside a sphere. It is exploiting the property that the amount of space covered by a sphere in the three-dimensional hyperbolic space increases exponentially with respect to the radius of the sphere, rather than polynomially. They compare using the traditional cone trees with their use of a layout on spherical caps, see Figure 2.12. Figure 2.13 shows an example of a network being displayed from [37].

Figure 2.12. Comparison of the traditional cone tree layout along the circumference of a circle with the H3 layout on the surface of the spherical cap. Both pictures show 54 child vertices in hyperbolic space, represented by pyramids of the same size. Left: The traditional perimeter layout requires a large cone radius and is quite sparse. Right: A quite small cone radius suffices for the H3 spherical cap, so the layout is reasonably dense.
CHAPTER 2. VISUALIZATION TECHNIQUES AND THEIR THEORY

Figure 2.13. Link structure of a Web site laid out in Three-dimensional hyperbolic space by [37]. The vertices represent documents, which are coloured according to MIME type: HTML is cyan, images are purple, and so on.

2.4 Looking forward

When this thesis has been done within a time constraint all the relative methods of visualisation could not be implemented and evaluated due to the great number of existing methods. Still measures needed to be taken so that no major or relevant visualisation method gets overlooked. With the information presented in this chapter as a basis, the visualisation techniques of the highest relevance will be chosen for evaluation.

Space

One important choice of consideration is in which space should the networks be displayed? In this chapter the reader was introduced to a number of different spaces where the two most common spaces, the two- and three-dimensional space, are of great importance and need to be included for this thesis purpose.

Different visualisation techniques will behave differently and result in having different aspects and characteristics depending on which space one uses. These can
2.4. LOOKING FORWARD

Then be compared for evaluation purposes, such as how layout of vertices and edges differs and what impact the space has on zooming and panning. Also how label clutter behave in different spaces may be of interest.

**Layout method**

When it comes to choosing which layout methods to evaluate, the same aspects mentioned in the previous section need to be considered here as well. Which layout method one chooses to use can have an immense impact on these aspects. Their results may also differ depending on which space one use and it needs to be taken into consideration.

**Chosen views**

From the research and study in this chapter a selection narrowed down to three different views were made to be used for a implementation of an application. In this selection both the two-dimensional and the three-dimensional space were covered. We will show the resulting application and for each view give an account of why those views where selected and how they were implemented in section 4.2.

**Technical aspects**

In addition to the effects on visualisation from the above described aspects there are other more practical aspects to be considered. Aspects revolving around one enforcing some pretension on the performance of the chosen methods implemented. This to ensure smooth usage so that a slow visualisation application will not impact the result in a negative way when evaluating a visualisation technique. More about this in section in the next chapter and appendix A.
Chapter 3

Method

When one is to display data using graphs one is confronted with the task of choosing between a large range of different visualisation techniques. This chapter is intended to show the process used to decide which methods and areas are to be investigated for this thesis. In section 3.1 the way found to implement these choices for further evaluation is shown. In section 3.2 the way taken for evaluation is described.

3.1 Implementation

It is difficult to compare and evaluate different visualisation techniques only on the information found in scientific thesis and books concerning them. One cause to this arises when one looks at the data used, different theses use different data. In some cases data might seem bias as having been chosen to fit better with the visualisation method concerned for the purpose of that particular thesis, making it hard to compare performance between techniques. Different visualisation methods perform differently on different data, making it only helpful if one wants to establish some form of knowledge around that a specific technique can be good on a specific kind of data. For this thesis it was necessary to have a more generalized unbiased approach.

To work around this an implementation was to be made that incorporates a selection of the studied techniques. Following the methodologies of selection for spaces and layout methods discussed earlier. This in order to make it possible to display the same networks (same data), using these different techniques and then be able to compare and evaluate performance on these techniques.

3.1.1 Programming environment

The implementation was to be developed in the programming language C# (C-sharp) [33] within Visual Studio [34]. The main application was to be made as an WPF (Windows Presentation Foundation) [36]. Other programming languages that have been used and incorporated in to the main WPF application is C++ [13] and
Java [53]. For database retrieval LINQ (Language-Integrated Query) [32] has been used.

3.2 Evaluation

In order to draw some relevant results from this thesis it is necessary to evaluate the different visualisation techniques chosen to be investigated. Section 3.2.1 takes up the necessity to adding tests concerning the prerequisites for implementing the layout views, which programming libraries to use. Section 3.2.2 will explain different aspects that need to be considered when evaluating the different layout methods.

3.2.1 Programming libraries

An evaluation of different programming libraries was made to find a good starting point for the implementation of an application. First a study that investigates what different libraries exists that supports visualisation for different networks was made. In this research a comparison of which different functionalities these libraries support was retrieved in form of data structures and algorithms, layout algorithms and so on.

From this initial study a selection of possible libraries to use for implementation was needed to be made. From here one could then evaluate the different selections to be able to make a final choice of which libraries to use.

The library evaluation will test for the libraries capacity in speed, how much time does it take to set up and draw networks of different sizes? And then as the last step was to evaluate the performance of the libraries after the networks had been drawn. This by taking measurements of smoothness while traversing a network. Smoothness was represented by the applications FPS (Frames Per Second) while navigating the network at different stages. The evaluation of programming libraries can be found in appendix A.

3.2.2 Layout views

Based on the study done in chapter 2 the different views will be evaluated on the following characteristics that are of great importance for a general visualisation system:

- Navigation
- Situational awareness
- Ease of recognizing important parts as sub-networks, vertices and connections
- Labeling

The evaluation will be executed by using the implemented application to try and complete tasks similar to the following:
3.2. EVALUATION

- Identify cluster/vertex X. Navigate to cluster/vertex Y. Can one navigate back to X?
- Identify the biggest cluster
- Identify vertex X
- Identify the vertex with highest degree

The tasks will be viewed as completed correctly, incorrectly or uncompleted. In addition to this, time will be taken into consideration as a measurement of performance. The results and drawn conclusions from these evaluations can be found in chapter 5.

3.2.3 Data

To be able to perform these evaluations some data needs to be at hand, data that fits this thesis purpose. This thesis has been performed at a company that provided the necessary data to be visualized.

Data for evaluation of programming libraries

For the performance evaluation of programming libraries data were needed to be generated. To generate this data the program Gephi [19] was used. This program was to be used to generate a number of different graphs of different sizes which were then saved as a plain text file. The following graphs were generated:

- Graph with 50 vertices with 592 edges. Referred as G50.
- Graph with 100 vertices with 3941 edges. Referred as G100.
- Graph with 500 vertices with 99641 edges. Referred as G500.
- Graph with 1000 vertices with 399969 edges. Referred as G1000.

On the account that different libraries use different ways of representing data this text file cannot be presumed to work as input for all libraries. On that fact parsers were needed to be developed to attend that the input was on the right format for the corresponding library.

Data for layout views

For the layout views, data in the form of electronical units found inside trucks called ECUs (Electronic Control Units) are used. Also variables used within these systems called AEs (allocation elements) are used to be visualized.

Their data have a complex form with a great number of relations and communications, thus making it suitable as data for this thesis purpose.
Chapter 4

Results

This chapter narrates the results from the literature studies and based on the application that was implemented. The results from an evaluation of existing programming libraries are given in section 4.1. For a more ingoing review of these libraries and how they were evaluated see appendix A. Section 4.2 shows the resulting application from the implementation and its views.

4.1 Library performance

To be able to make a solid application that can be used for this thesis evaluations, not only the need of choosing what layout methods to be used needs to be considered. Which prerequisites one chooses to use while implementing said application needs to be considered. Thereby a consideration of which libraries one can use when implementing the different visualisation methods is needed.

4.1.1 Attribute matrix

The following matrix, figure 4.1, lists important functionality one looks for when considering a programming library for implementing visualisation methods. The matrix gives an overview of what the different libraries supports from the start. This matrix helps as a basis when selecting libraries for the implementation.
CHAPTER 4. RESULTS

Table 4.1. Attribute matrix

<table>
<thead>
<tr>
<th>Library</th>
<th>Own layout algorithms</th>
<th>Documentation</th>
<th>External layout algorithm</th>
<th>Zoom functionality</th>
<th>Highlight functionality</th>
<th>Nested Graph support</th>
<th>User editable graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickGraph</td>
<td>✓</td>
<td>Good</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>yFiles</td>
<td>✓</td>
<td>Good</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GraphX</td>
<td>✓</td>
<td>Good</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MSGAL</td>
<td>✓</td>
<td>Good</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GraphV2</td>
<td>✓</td>
<td>Decent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VTK</td>
<td>✓</td>
<td>Good</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4.1.2 Results from library tests

Speed

Each graph has been drawn ten times and the arithmetic mean value of the time has been calculated and given as result. Time is displayed on the form of minutes:seconds.

Table 4.2. Speed results

<table>
<thead>
<tr>
<th>Library</th>
<th>G50</th>
<th>G100</th>
<th>G500</th>
<th>G1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphX</td>
<td>0:0.515</td>
<td>0:7.415</td>
<td>22:33.457</td>
<td>Out of memory</td>
</tr>
<tr>
<td>VTK</td>
<td>0:0.0155</td>
<td>0:0.0414</td>
<td>0:0.883</td>
<td>0:3.530</td>
</tr>
<tr>
<td>yFiles</td>
<td>0:0.540</td>
<td>0:1.682</td>
<td>0:49.605</td>
<td>12:11.536</td>
</tr>
</tbody>
</table>

Smoothness - FPS (Frames per second)

There are three FPS measurements for each library in which panning actions were being performed. One zoomed out far away, giving an overview of the graph (Z1), one zoomed in halfway to the centre of the graph (Z2) and a third in which the user is zoomed far in to the graph, being able to distinguish between vertices (Z3). When the measurement states undetectable the fps value have been too low to measure, stalling out the program. The procedure this was performed in is the same as for the speed values, an arithmetic mean value is given.
4.2. APPLICATION

Table 4.3. GraphX

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>17 fps</td>
<td>23 fps</td>
<td>35 fps</td>
</tr>
<tr>
<td>G100</td>
<td>1 fps</td>
<td>3 fps</td>
<td>5 fps</td>
</tr>
<tr>
<td>G500</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>Undetectable</td>
</tr>
<tr>
<td>G1000</td>
<td>Out of memory</td>
<td>Out of memory</td>
<td>Out of memory</td>
</tr>
</tbody>
</table>

Table 4.4. VTK

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>1000 fps</td>
<td>1000 fps</td>
<td>1000 fps</td>
</tr>
<tr>
<td>G100</td>
<td>500 fps</td>
<td>500 fps</td>
<td>500 fps</td>
</tr>
<tr>
<td>G500</td>
<td>35 fps</td>
<td>11 fps</td>
<td>3 fps</td>
</tr>
<tr>
<td>G1000</td>
<td>11 fps</td>
<td>7 fps</td>
<td>2 fps</td>
</tr>
</tbody>
</table>

Table 4.5. yFiles

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>22 fps</td>
<td>20 fps</td>
<td>25 fps</td>
</tr>
<tr>
<td>G100</td>
<td>2 fps</td>
<td>3 fps</td>
<td>5 fps</td>
</tr>
<tr>
<td>G500</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>Undetectable</td>
</tr>
<tr>
<td>G1000</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>2 fps</td>
</tr>
</tbody>
</table>

From the tables above one can see that The Visualization Toolkit (VTK) [57] often outperforms the other libraries by a large factor, both in speed and fps measurements. Therefore this will be the library of use when implementing own views.

4.2 Application

This section shows the resulting application implemented with the corresponding views chosen from the research done in chapter 2.

4.2.1 Main application

At the start up of this application the user will be taken to the main application where the user will be prompted to make some choices before being able to visualize data. These choices are concerning which data to be used.

In this application the user is constrained to choose between visualising ECUs or AEs, see section 3.2.3. The user is also prompted to choose which SOP date to use. Next the user needs to load the data from the database by pressing a button, labeled 'Load Data'. After that is done the user can go on and choose which view to display. The user is not restrained to one view at the time but can bring up multiple views at the same time, making it easy to compare views.
4.2.2 Force-Directed (FD) based view

Force-directed algorithms are, as shown in chapter 2, an important part when it comes to visualising networks. There are many visualisation techniques based solely on force-directed algorithms and other techniques that use different approaches do often incorporate some kind of force-directed algorithm to their visualisation method. As for example by computing a base graph using a force-directed algorithm. Therefore the first view of the implementation is a view that uses a force-directed algorithm for the vertices layout and edge routing. Figures 4.1 to 4.6 show examples of the application using this view.

Figure 4.1. An unlabled overview of a network when all AE was chosen.
4.2. APPLICATION

Figure 4.2. Same network as in 4.1 after zooming actions were preformed.

Figure 4.3. A labeled overview of a network when all ECUs were chosen.
CHAPTER 4. RESULTS

Figure 4.4. Same network as in 4.3 after zooming actions were preformed.

Figure 4.5. Showing data with labels enabled.
4.2. APPLICATION

Figure 4.6. Showing same data as in picture 4.5 but with labels unenabled.

Layout

The force-directed view results in an even distribution of all the vertices and is good at separating subgraphs from each other, as seen in figure 4.3. An evident problem that arises is with the edges of a network. The edges quickly converge to a hairball of edges, making it impossible to distinguish between individual edges. This can be seen in figure 4.2.

Labeling

As mentioned in section B.1.1, the VTK library displays labels according to their weight, the higher the weight the higher priority a given vertex label will have. This combined with the force-directed algorithm by nature is good at evenly spreading out vertices, label obscuration becomes less of a problem. An example of this is shown in figure 4.4 where a fairly large network is being displayed.

Of course this comes at a price. By allowing a prioritizing of which labels to display one is left with a loss of information, in this case a loss of labels. This can have a large effect on the outcome result when using this application. Depending on what tasks one wants to solve by using this visualization technique, important data may be missed and/or be more difficult to locate. It comes down to what data one is attempting to extract from the view. If for instance one is out to identify the vertices with a large number of connections and greater subnets this view might work terrific. On the other hand if one is to find a specific data part that may be
smaller weighted this view can be difficult and uneasy to use.

**Navigation - Situational awareness**

The effect on the situational awareness of a user in this view depends highly on what stage in a task the user is on and what type of task said user is performing. Because the view provides a highly zoomable network a user can with a far out zoomed view, combined with the labeling enabled, acquire a good overlook of a given network.

Though when zooming in using this view there comes a point when some data are lost, there are only so much data that can fit on a screen at the same time. This can diminish the situational awareness for a user and result in a loss of orientation, forcing a user to zoom out to try and see correlations from one part of the network to another. The user might even have a loss in orientation trying to get back to the same spot as before a given zooming action was made. One can draw the conclusion that this view provides a poor solution for problems having a need for global context and local details at the same time. This results in a user losing orientation or specific information at different stages performing tasks.

In this view one has the option to navigate through the x- and y-axis or x-, y- and z-axis. This can be helpful when looking closer on how a subpart of a network joins with another. But one needs to be careful when going from two axises to three, due to the fact that it can result in a loss of orientation.

### 4.2.3 Two-dimensional view - BioFabric

For a view in the two dimensional space BioFabric was used. BioFabric provides a non conventional approach to visualize data and have worthy attributes to be evaluated for visualization. Such as the layout algorithm for vertices and edges, the labeling of vertices and how one navigate the network. Figures 4.7 to 4.9 show examples when using the Biofabric view.
4.2. APPLICATION

Figure 4.7. An overview of a network using the BioFabric view when all ECUs were chosen.

Figure 4.8. Same network as in 4.7 after zooming and selection actions were performed.
CHAPTER 4. RESULTS

Because of the static layout function BioFabric uses it results in a consistency while visualizing graphs, a specific graph will be drawn the same way from one time to another. One benefit with BioFabric's layout is to get away from the problem of edge crossings, as seen in the top part of figure 4.8.

BioFabric produces compact graphs, which makes it more difficult to identify individual elements without deeper zooms. Comparing the main view with the network overview in figure 4.8, a situation occurs where a greater zoom is needed to see individual elements in a small selection.

Labeling

BioFabric also uses a weighted label solution. Here the vertices with highest degree, as in BioFabric becomes the horizontal lines at the top of the view, will be of the greatest size. In figure 4.7 one can see this clearly. The reason behind weighting labels becomes fairly obvious when one starts to think about how it would turn out if BioFabric would try to show all their labels of all vertices at the same size. There would be so many labels occluding each other that one would not be able to distinguish which label belongs to which vertex. Furthermore most part of the graph would have labels occluding each other, making it difficult to see what label says what.
4.2. APPLICATION

Navigation - Situational awareness

BioFabric’s approach is to have more active views in their application at the same time, they have three different views. One main view of the network, a second view that gives an overview of the network and shows selections made and a third view called the network magnifier. The network magnifier displays a sub selection of the network made by the user (a selection) where the user can see all the labels and connections of vertices in their selection. Figure 4.9 shows all three views where an selection has been made.

Having these different views, especially the relation between the overview and network magnifier, tries to help toward good situational awareness. The user can get a more detailed view of a subpart of a network using the network magnifier while trying to maintain a global context by simultaneously looking at the network overview and main view.

Having this setup one can argue that their approach goes toward data on demand. Requiring the user to have good knowledge about the network being displayed but also to have a good idea of the where and how the information searched is localized. Putting the pressure on the user to have a good situational awareness orientation beforehand, which is not always the case.

4.2.4 Three-dimensional view - GerbilSphere

The direction many new visualisation techniques are taking is attempting to take advantage of the extra space in the three-dimensional space. And thus it is of significance for this thesis to incorporate a tree-dimensional visualisation technique. Many of these tree-dimensional techniques use a sphere shape to visualize networks in/on. And it is common that they use some sort of force-directed algorithm to layout the vertices and edges in this spherical space.

For this application a tree-dimensional view with a layout method that uses the spherical space was used. The view is based on GerbilSphere [48] where the vertices of a network are laid out on the surface of a sphere and one navigates from a point of view inside the sphere (though one can zoom out to see the sphere from the outside). For more details about GerbilSphere see chapter 2 section 2.3.1. Figure 4.10 and 4.11 show the view in use zoomed outside the sphere while figure 4.12 shows the view from inside the sphere.
CHAPTER 4. RESULTS

Figure 4.10. An overview with GirbilSphere of a network when all AEs were chosen.

Figure 4.11. Same network as in Figure 4.10 rotated ninety degrees.
4.2. APPLICATION

Figure 4.12. View of the network in Figure 4.10 and 4.11 from inside of the sphere.

Layout

When GerbilSphere uses a force-directed algorithm as part of its layout it produces, an even distribution of vertices and produce a good view of different subgraphs. One feature that emerges is being able to easy identify hubs. An example of this can be seen in the right side of Figure 4.11, where a hub (a single vertice in this case) that connects to many different subgraphs can be seen.

The greatest advantage with GirbilSphere is the extra space one gains. Even in our example where we are displaying a large network we have plenty of space left over. However problematics still exists, as with edges converging towards a hairball effect because GerbilSphere displays subnetworks in focus as a two-dimensional view when inside the sphere. Figure 4.13 shows an example of this. Vertice obscuration becomes a side effect in this three-dimensional view when outside the sphere, forcing a user to change orientation to obtain desirable views. Compare Figure 4.11 with 4.12 where a rotation of ninety degrees of the sphere has been made.

Labeling

The greatest advantage to this approach is the extra space one gains by using the hyperbolic space. So one might be enticed to draw the conclusion that this is all beneficial thinking, having more space for both labels and vertices lets the application lay out all data points in a satisfactory way. While the fact about more space might be true one will soon see that obscuration of data becomes a much more
severe problem compared to two-dimensional solutions. Now labels and vertices can be obscuring each other in a wider range, laying in front or behind each other.

Presumably GerbilSphereAlpha has attempted to work around obscurance taken a information on demand approach. Choosing not to display labels on the sphere and instead having a separate window that displays information about a vertice when that specific vertice is chosen by the user. This results in a lower risk of obscuration of labels but at the same time demands more from the user. Instead it demands the user to have a greater knowledge about the network and its structure beforehand to be able to navigate and retrieve desirable data from the visualized network. It also causes a delay in identifying vertices when a user needs to click on a specific vertice to get information about it. Finally a user might loose orientation if the user forgets which vertice where clicked on in previous steps.

**Navigation - Situational awareness**

It has already been indicated that drawing networks in the hyperbolic space offers more space compared to the two-dimensional space, which is a positive thing. Though this comes with a price, which has to be paid by the end user with a worsened sense of situational awareness. Due to the fact that with more space and axes to navigate through it becomes more difficult to keep orientation. For example a user might take too large of a step in the sphere, losing track of where they came from which might result in forcing the user to go further back or even start over in the task currently being performed.

This becomes a problematic downside when going from a two-dimensional - to a three-dimensional space view, and it needs to be considered and handled. GerbilSphereAlpha tries to solve this by using a developed navigation feature that is displayed at the bottom of the sphere. See [48] for further details. This navigation feature becomes a good first step in trying to help the situational awareness of a user, though it is not too intuitive to use at first and it takes some practice to become familiar with.

Going over to the hyperbolic space with a need for more space, having larger networks to display, can result in a large number of small vertices being displayed, which when seeking individual vertices can become infeasable. Also worth mentioning is that while the hyperbolic space provides more space which is good for larger networks it can have an negative effect if it is used on smaller networks. This could result in a small number of vertices on a greater surface that could produce unnecessary long edges and also vertices may become further apart then necessary.
Chapter 5

Discussion and Conclusions

In this chapter plausible conclusions from the studies and results in previous chapter will be discussed to answer this thesis question. *Which type of visualisation technique is best suited to visualize big and complex networks?*

5.1 Which type of visualisation technique is then best suited to visualize big and complex networks?

One thing that becomes fairly clear when reading the research from this thesis is that answering this question is not an easy task. Due to the fact that this thesis does not go through all existing visualisation techniques, the evidence from this thesis points toward the conclusion that there is no correct solution answering this question. It points more towards that there is no best choice of technique to display a complex network that is independent from the kind of and form of data being used.

It boils down to that when one wants to visualize a complex network one must consider all the features and demands that the given network needs. Do we need more space for the data? A clean and clear structural way of traversing the network? How does the need for labeling our vertices look? Do we need to find out how subnetworks looks like or get more information about specific vertices and connections? And so on. There exists an indefinite number of different complex networks and one cannot find a generalisation saying that one visualisation technique satisfies all the needs of all networks and displays them in an optimal way.

Instead one needs to tailor a specified solution for a specific network and the accompanied requirements of visualisation with it. One needs to become very familiar with the data that is going to be displayed along with getting an excellent knowledge of the visualisation needs for the specific application. Then use this knowledge to develop features that comply with these needs. Features such as complex filters that trim the network to specific data, search functions, highlights etc. It all boils down too the needs of the specific application.
For further research one could be interested in seeing how these different visualisation approaches would cooperate. To see how the situational awareness is affected when navigating through one view and seeing at the same time that change in the different views. How it would affect if one takes a sub-selection of a network in one view and sees the same sub-selection marked in the other views simultaneously. Filters may become important features when it is often hard to draw any good information from looking at a entire network at once. Often one is looking for a subgraph and needs a more detailed view of said graph. Such filters could be developed that take keywords describing features one are looking for. Or maybe it is better for the user to first see the whole network and manually make selections? Being able to create filters for not only vertices but also edges could be evaluated. Would letting the user decide which types of connections to display help getting away from the hairball of edges problem? These kind of questions needs to be asked and answered when developing a visualization technique or application.

5.2 Environmental aspects

It is hard to draw any definite conclusions on the impact this thesis work has on environmental aspects, at the time this thesis was completed no mentionable effects had occurred. Though it is known that visualisation of complex networks can help too facilitate work with analysis and verification of the systems the networks represent. In this case this could lead to safer and more cost-effective vehicles. This becomes particular important in the case of autonomous heavy vehicles. Therefore if work is continued from this thesis one could see these impacts in the future.
Appendix A

Library evaluation

A.0.1 Libraries of interest

For the considered libraries of use for the implementation four different categories is used to classify them. Open-source libraries, closed-source libraries, 3D-approaches and non-conventional graph visualisation approaches.

Open-source libraries

QuickGraph

QuickGraph [9] is a library containing generic graph data structures and algorithms for a range of graph problems, developed for .Net [55] use. Such as classical problems like maximum flow, topological sort, shortest path, depth search, etc.

Supports: [8]

1. Graph data structures
   1.1. Directed graph
   1.2. Undirected graph
   1.3. Dense graph
   1.4. Sparse graph
2. Graph computational Algorithms
   2.1. Topological sort
   2.2. Strongly connected components
   2.3. Minimum spanning tree

Notes:
QuickGraph does not support an option to use an own solution for visualisation of graphs but points to the layout library Graphviz [22] that QuickGraph claims work well with their data structures [10]. This applies to layout algorithms as well.

**Graphviz**

Graphviz [22] - graph visualization software. Graphviz provides different layout algorithms and take descriptions of graphs in a text language as input. Normal usage with Graphviz is by using DOT (graph description language) [51].

It can be used with QuickGraph as a C# wrapper. Other wrappers for C# and .Net one could use are graphviznet [6] and graphviz4net [7].

**Supports:**

1. Layout algorithms [22]
   
   1.1. Dot
   1.2. Neato
   1.3. Fdp
   1.4. Sfdp
   1.5. Twopi
   1.6. Circo
   1.7. Osage

**GraphX**

GraphX is an advanced open-source .Net library for graph visualization with capabilities to rend large graphs with large amount of vertices and edges which depends on the QuickGraph library [42]. It also uses partial code from Graph# [21], WPFExtensions [12], NodeXL [38] and Extended WPF Toolkit [61], which are open-source projects.

It is based on WPF for rendering graphs and can be seen as the successor too Graph#. GraphX is the new Apache Sparks API for graphs and graph-parallel computation. It introduces a new API that operates on both tables and graphs and incorporate this API as a library using graph parallel techniques to be as fast as specialized systems (such as GraphLab, Giraph and Pregel). By embedding this graph-parallel model in Spark it enables GraphX to integrate easily with RDDS (Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing) and perform data parallel operation while also enabling the speed of specialize graph systems [62].
Supports: [42]

1. Layout algorithms
   1.1. BoundedFR (Fruchterman Reingold).
   1.2. Circular.
   1.3. CompoundFDP.
   1.4. EfficientSugiyama.
   1.5. Sugiyama.
   1.6. FR.
   1.7. ISOM.
   1.8. KK (Kamanda and Kawai).
   1.9. LinLog.
   1.10. Tree.

2. Possibility to implement an own layout algorithm (external layout algorithm).

3. Visual control
   3.1. Delete animation of vertices and edges.
   3.2. Mouse over control animation
   3.3. Custom animations
   3.4. Highlighting of vertices and edges.
   3.5. Zoom control.
   3.6. Area selection of vertices.
   3.7. Area zooming and smooth animations.

Notes:

There is no documentation of implemented functions for nested graphs. A nested graph is a graph were vertices can contain subgraphs within themselves.

MSAGL

MSAGL [46], Microsoft Automatic Graph Layout, is a .Net tool for graph layout and viewing. It is built on the Sugiyama scheme [54] that produce hierarchical layouts. Where the vertices are drawn in horizontal layers and the edges are often drawn in a downward fashion between vertices. MSAGL contains its own layout engine.

Supports: [46]
APPENDIX A. LIBRARY EVALUATION

1. Layout algorithms.
   1.1. Sugiyama.

2. Editable layout after initial layout.

3. Navigation of graph
   3.1. Zoom.
   3.2. Pan.
   3.3. Search and focus function.

4. Visual control.
   4.2. Zoom.

Closed-source libraries

**yFiles (yWorks)**

yFiles [66] provides data structures and algorithms for graph operations. Including automatically layouts for graphs and visualization controls for those graphs. yFiles are supported for different platforms, including .Net where one can either use a library for Windows Forms [35], WPF [36] or Silverlight [49].

*Supports: [65] [64] [63]*

1. Layout algorithms
   1.1. Circular layout.
   1.2. Hierarchical layout.
   1.3. Organic layout.
   1.4. Orthogonal layout.
   1.5. Tree layout.
   1.6. Incremental layout.

2. Edge routing algorithms.
   2.1. Organic routing.
   2.2. Orthogonal routing.

3.1. Highlighting controls.
3.2. Zoom.
3.3. Smooth change animations.

Notes:

yFiles also supports incremental layout, meaning that when a graph needs to be updated (addition/removal of vertices or edges) the whole graph is not re-computed but tries to maintain as much as possible of the same layout as before. Also yFiles tries to supports nested graphs, but has some restrains on the data to be able to use it. Their function is called collapsible tree, which requires the data to be able to be structured as a tree.

An comparison of given libraries.

Attribute matrix:

The following matrix, figure A.1, lists important functionality one looks for when considering a programming library for implementing visualisation methods. The matrix gives an overview of what the different libraries supports from the start. This matrix helps as a basis when selecting libraries for the implementation.

<table>
<thead>
<tr>
<th>Library</th>
<th>Own layout algorithm</th>
<th>Documentation</th>
<th>External layout algorithm</th>
<th>Zoom functionality</th>
<th>Highlight functionality</th>
<th>Nested Graph support</th>
<th>User editable graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuickGraph</td>
<td></td>
<td>Good</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yFiles</td>
<td>Yes</td>
<td>Good</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphX</td>
<td>Yes</td>
<td>Good</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGAL</td>
<td></td>
<td>Good</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphViz</td>
<td></td>
<td>Decent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1. Attribute matrix

3D approaches

Libraries

For developing advanced 3D graphics the two most common approaches is to use either OpenGL [41] or Direct3D [31]. To use OpenGL in windows one has a few options, one can for example use OpenTK that wraps OpenGL, OpenCL [39] and OpenAL [40].
Direct3D is part of the DirectX API that uses hardware acceleration (if available on the graphics card).

**Differences between OpenGL and Direct3D**

OpenGL is being developed largely by a consortium of different parties and follows a largely open standard. While Direct3D are being developed and maintained by Microsoft and is completely proprietary in its implementation. Resulting in a big platform difference, where OpenGL is supported on a wide range of platforms and languages, while Direct3D is bound to Microsoft Windows systems.

At last there are different methods used to introduce new hardware and features. OpenGL do this by allowing hardware manufactures to be able to implement special functions, called extensions that give immediate access to features of new hardware. As for Direct3D, Microsoft needs to process theses features and then release access to these in forms of new functions. Here OpenGL allows new features to be accessible quicker then Direct3D, though reduces the overall compatibility of a program using the extensions. And the other way around, Direct3D takes longer to give access to the new features but compatibility across different systems is being maintained.

The aspect of hardware managing differs in these two libraries. OpenGL hides the hardware and works so that the implementation handles hardware resources, users of OpenGL use functions for drawing which relies on drivers to directly access the hardware. Direct3D on the other hand lets the application handle the hardware resources. With OpenGL it becomes easier to write applications but hard to see the status of hardware resources and thus must hope that the implementation uses the resources in a way that suits the related application. While with Direct3D the writing of the application may be more complex but have the possibility to use hardware resources in the most efficient way for the application [59].

Both these differs a bit form previous discussed libraries in the sense that there are no (known) support for graph visualization. So for instance no layout algorithms for graphs like force directed or data structures to represent a graph. This implies that all has to be implemented from scratch, taking more time but open up for less restrictions on what one can do.

**VTK**

VTK (Visualization Toolkit) [57] is an open-source software system for 3D computer graphics, image processing and visualization. At its core VTK is implemented as a C++ toolkit and it supports parallel processing which help in performance. VTK is a popular library when it comes to visualize scientific data, which is often big and complex.

There are a number of wrapper languages making it possible in addition to C++ use VTK through for instance Python, Java and .NET.
Non conventional graph visualization approaches

When it comes to the visualisation techniques like BioFabric, Hive Plots and Treemap (described in chapter 2) there are few libraries that support this compared to the amount of libraries there are to visualize the classical vertice/edge graphs. How these approaches can be used are shown below.

**BioFabric:**

BioFabric has one release which is an open-source Java application with some documentation that can be found at [3]. Though BioFabric is built on a relatively easy and intuitive algorithm, one could take the option of implementing an own version. Having their own customized features that suits one’s purpose.

**Hive Plots:**

For hive plots there are some choices, one used is a Java based library [2]. There are also libraries for R [45], HiveR [1], that supports hive plots in the two dimensional and three dimensional space. The framework D3.JS [14] is an other option that is a JavaScript to create hive plots. And pyveplot [43] that is a library for hiveplots in Python [44].

**Tree Map:**

As for Treemaps there are some choices as well. For .Net, which is this thesis working environment, there is the WPF Treemaps & SquarifiedTreeMaps control library [11], though it has poor documentation. Another alternative is the .NET Treemap Control library [15]. Again the problem lies in little documentation and it is hard to get information about the library when this was part of an old Microsoft research project called Netscan.

**GerbilSphere:**

GerbilSphere visualize graphs by projecting vertices and edges to the surface of a sphere, defined as an inner sphere 2D system. It differs from other graph visualisation techniques that also uses spheres in the way that in GerbilSphere the observers point of view is from inside the sphere.

GerbilSphere is an open source project. No API is available, though good documentation is presented within the code.

**Supports:** [48]

1. Specialised layout algorithm that is based on force-directed algorithms.
APPENDIX A. LIBRARY EVALUATION

1.1. Static layout when adding/removing vertices.

2. Labeling.

3. Menus when choosing specific vertices.

4. Visual control.
   4.1. Zoom.
   4.2. Paning.
   4.3. Fisheye view.
   4.4. Variablezoom.

Notes:

While GerbilSphere does not support nested graphs this becomes of less interest when the whole graph is being visualized.

A.0.2 Library selection for evaluation

From the matrix in figure A.1 one conclude that GraphX and yFiles are two possible candidates for usable libraries. They both support more functionality than QuickGraph, MSGAL and GraphViz. Therefore they will be included in the evaluation.

Needed is also an option for implementing three dimensional views. For this purpose the VTK library is included when it has support for graph visualization.

A.0.3 Test set up

The libraries have been tested on two different aspects, rendering speed and smoothness. Rendering speed is measured as the time it takes a specific library to draw up a given graph. As for smoothness the frames per second (fps) rate has been measured during navigation through corresponding graphs. Navigation actions such as panning and zooming.

Data

The data used for these tests has been produced with the use of Gephi [19]. Four different graphs were created for testing:

- 50 vertices with 592 edges. Referred as G50.
- 100 vertices with 3941 edges. Referred as G100.
- 500 vertices with 99641 edges. Referred as G500.
- 1000 vertices with 399969 edges. Referred as G1000.
Hardware

The evaluations were run on the same computer for each graph and library with the following hardware:

- **Processor:** Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
- **Video Card:** Intel(R) HD Graphics. 64 MB Dedicated Memory, 1.7 GB Total Memory
- **Memory:** 6.1 GB
- **Operating System:** Microsoft Windows 7 Enterprise Edition Service Pack 1 (build 7601), 64-bit

A.0.4 Results from library tests

Two kinds of users need to be considered when talking about the results from the library evaluations. The users that will indirectly use the libraries by using an application built on these, we will call these users for end users. Second is the kind of user that will use these libraries to build an application for visualisation, we call these users for developer users.

The needs one have on the libraries differs depending on what kind of user one are. For an end user the speed and smoothness are the two things that shows most. Developer users must also take into consideration what the different libraries supports and not.

**Speed**

Each graph have been drawn ten times and the arithmetic mean value of the time has been calculated and given as result. Time is displayed on the form of minutes:seconds.

<table>
<thead>
<tr>
<th></th>
<th>G50</th>
<th>G100</th>
<th>G500</th>
<th>G1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphX</td>
<td>00:00:51461281</td>
<td>00:07:41482771</td>
<td>22:33:4573175</td>
<td>Out of memory</td>
</tr>
<tr>
<td>VTK</td>
<td>00:00:01546258</td>
<td>00:00:04138581</td>
<td>00:00:88301525</td>
<td>00:03:53977976</td>
</tr>
<tr>
<td>yFiles</td>
<td>00:00:54974544</td>
<td>00:01:68249026</td>
<td>00:49:60542746</td>
<td>12:11:535764772</td>
</tr>
</tbody>
</table>

Here one can see that the VTK library is far superior to the other libraries, especially on the larger sized graphs. On the smallest graphs the difference are not as significant. For the end users the difference on drawing speed on graphs of size with 50 vertices would not be to notable. Though the developer users might want to take this into consideration. On larger than 50 vertices graphs the difference between these libraries shows fairly clear. GraphX drawing speed decreases rapidly and are not able to draw graphs with a size of 1000 vertices with the hardware used. yFiles do manage to draw this graph, though it took over 12 minutes compared to VTKs
3 seconds.

Smoothness - FPS (Frames per second)

There are three fps measurements for each library were panning actions was being performed. One zoomed out far away, giving an overview of the graph (Z1), one zoomed in half way to the centre of the graph (Z2) and a third were the user are zoomed far in to the graph, being able to distinguish between vertices (Z3).

### Table A.2. GraphX

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>17 fps</td>
<td>23 fps</td>
<td>35 fps</td>
</tr>
<tr>
<td>G100</td>
<td>1 fps</td>
<td>3 fps</td>
<td>5 fps</td>
</tr>
<tr>
<td>G500</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>Undetectable</td>
</tr>
<tr>
<td>G1000</td>
<td>Non-executable</td>
<td>Non-executable</td>
<td>Non-executable</td>
</tr>
</tbody>
</table>

### Table A.3. VTK

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>1000 fps</td>
<td>1000 fps</td>
<td>1000 fps</td>
</tr>
<tr>
<td>G100</td>
<td>500 fps</td>
<td>500 fps</td>
<td>500 fps</td>
</tr>
<tr>
<td>G500</td>
<td>35 fps</td>
<td>11 fps</td>
<td>3 fps</td>
</tr>
<tr>
<td>G1000</td>
<td>11 fps</td>
<td>7 fps</td>
<td>2 fps</td>
</tr>
</tbody>
</table>

### Table A.4. yFiles

<table>
<thead>
<tr>
<th></th>
<th>Z1</th>
<th>Z2</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td>G50</td>
<td>22 fps</td>
<td>20 fps</td>
<td>25 fps</td>
</tr>
<tr>
<td>G100</td>
<td>2 fps</td>
<td>3 fps</td>
<td>5 fps</td>
</tr>
<tr>
<td>G500</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>Undetectable</td>
</tr>
<tr>
<td>G1000</td>
<td>Undetectable</td>
<td>Undetectable</td>
<td>2 fps</td>
</tr>
</tbody>
</table>

50
From these tables one can see that GraphX and yFiles perform almost equally, though VTK outperforms both these libraries with quite a larger margin. VTK is also the only one that could detect a readable FPS value when maneuvering the largest sized graph. Both GraphX and yFiles displays a better smoothness (higher FPS) at a deeper zoom, which at first can seem a bit strange. The reason for this is that once you are zoomed far enough in the graph, you come to a point where there is less vertices and edges to display on the screen then previous zoom where the low FPS value starts to increase after this point.

**To take into consideration**

One thing that becomes overlooked when talking about these different libraries is how one wants to draw graphs. For instance if one wants a small or large representation of vertices. If one is planing on just drawing smaller graphs with larger vertices one might be fine using yFiles or GraphX. Though if one wants to draw graphs of the larger kind one might be better suited with the VTK library. These aspects will of course affect the speed and smoothness when using these libraries. In this study the basic common configuration for each library has been used.

For the end user the speed and smoothness is what becomes of most importance, because that is what they see when using an application. The developer user must of course take this into account when developing an application. The developer user must also take into consideration what features that the different libraries support.
Appendix B

Implementation details.

B.1 Views

B.1.1 Force-directed (FD) based view

This view was implemented when using the VTK library for C# on the basis from the evaluation done in section 4.1.

In this view the user of the application can choose between showing labels or not. The labeling of the vertices in this view are weighted with a weight based on their degree. This in such a way that the higher the degree of a vertice, the higher weight that vertice’s label will be given. Based on the vertice’s weights, a priority of which labels to be displayed at a certain point in the network are made. Giving vertice’s with higher degree a higher priority. This also results in that depending on which zoom level in a network one are, different labels will be prioritised.

When it comes to the space this view is using, it is actually up to the user to choose. By holding down shift when navigating a network in this view the z-axle becomes fixed, making it appear as a two dimensional view. Otherwise it works in the three dimensional space, where one can navigate through the x-, y- and z-axles. This is unique to this view, the next two views works in the two- or three-dimensional space.

B.1.2 Two-dimensional view - BioFabric

BioFabric is an open-source Java application which makes it applicable to use for this thesis. Though some changes was needed to be done for it to work with the implementation. Changes concerning the code that conducts to where and how the input of data for the application were made, changes so that BioFabric becomes compatible with the application. Also additions concerning that the input data converts to the correct form for BioFabric where needed. More about this in B.2.3.
APPENDIX B. IMPLEMENTATION DETAILS.

B.1.3 Three-dimensional view - GerbilSphere

GerbilSphere is an open source application developed in C++. Again the data handled needs to be modified to match the form of data input GerbilSphere uses and more about this in B.2.3.

GerbilSphere handles labels a bit differently than previous views. Instead of trying to place visible labels in the spherical space they have taken an information on demand approach, making it up to the user to prompt for specific information at need. This by having a secondary window up that keeps track of actions made in the sphere. As when clicking on a specific vertex results in getting the information (the label) about that vertex showing in this window.

B.2 Data representation

This section goes over how to represent the data to be visualized in the different views.

B.2.1 GraphElement

To be able to represent data in the earlier stages, before displaying any views, a small class called GraphElement was created. This represents, as its name suggests, an element of a graph such as a vertex or an edge.

The class holds parameters about the name and ids and functions that makes it possible to distinguish between two different GraphElements.

B.2.2 Data representation within application - DataManager

DataManager is a class developed to do the actual data retrieval from the working database. To represent a graph internal in the application the DataManager class retrieves the data one wants to visualise and creates GraphElement objects that are put into an object of the data structure called Dictionary [30]. This in such a way that each object (can be seen as a vertice) of a network will be set as a key in the dictionary and have a tied value in form of a list of GraphElements. Thus one can see the dataset as a given key (vertice) in the dictionary (network) has a corresponding list Y (neighbours). For instance say that GraphElement X is a key and are associated to a list Y, then an object y in Y can be seen as that there exists and edge between X and y.

Then when it comes to actually visualise data one needs to convert the data to the appropriate form for the corresponding view. More about this in the next section.

B.2.3 Data parsers

This section will give account for the resulting data parser written for each of the three views. Parser for the VTK library for the force-directed view, a parser for
B.2. DATA REPRESENTATION

BioFabric and lastly a parser for GerbilSphereAlpha.

**VTK data parser**

The VTK library uses a straightforward simple way of representing a graph. It works in a way that first all the vertices are added and get sequential ids starting from 1. Then one can add edges between these by calling a function with parameters saying between which two vertices the edge connects, using the vertices ids. Making it very compatible with the DataManager class and GraphElement object.

Because VTK represents their data in such a way and one needs to build the data structure in code behind there was no need to implement a parser for VTK. One could simply retain the necessary information from your network and use it directly to functions for building graphs in the VTK library. Though a parser was implemented, reading a text file containing information about a network and storing it in a dictionary, such as in the DataManager class. This parser was implemented for the evaluation stage for the VTK library. Making it possible to take a text file, generated from Gephi, as input and returning a dictionary that could then be used to generate a graph with VTK to be evaluated.

**BioFabric data parser**

BioFabric supports tab-delimited .sif files [4] as input, therefore a parser for transforming data from our DataManager class to SIF files were created. A SIF file has the following form:

From vertex name, tab pp (indicating an edge), and tab followed by to vertex name. For example:

2 pp 4,

meaning there exists an edge between vertex 2 and vertex 4.

Where it states vertex name, the information in that field will be what displays in the BioFabric view as the labels.

**GerbilSphere data parser**

GerbilSphere supports two different file formats as input. First is Pajek [20] and the second is the Extensible Graph Markup Language (XGMML) [5]. Pajek is on a text based format with the file extension .Net. Which first states the number of vertices N, followed by N lines with the vertices id and labels. Next comes "*Edges" followed by X lines defining the networks edges. Where the vertices from id are shown followed the to vertex id. The edges can also be weighted, in that case a third value with the weight are added.

In GerbilSphereAlpha these vertices can have attributes such as positioning for the sphere, the file is then called baked. See [48] for further information.
APPENDIX B. IMPLEMENTATION DETAILS.

XGMML is based on the Graph Modelling Language, GML, which is a hierarchical ASCII-based file format for describing graphs. Using tags to describing different graph components. Though both Pajek and XGMML were usable for this thesis application the Pajek format was in the end used.

A parser for both these file formats were created, following the syntax described above.
References


REFERENCES


58


REFERENCES


