University of Glasgow | School of Computing Science

Visualisation of Massive Networks in a Browser

Benjamin Jackson

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 13, 2017

Abstract

Visualising massive networks in a browser causes both developers and users numerous difficulties. Challenges are the high amount of bandwidth and computational power required to render a massive network, along with the difficulty of getting useful information from the network after it is displayed. After completing background research into the field, a systematic review of existing software was conducted, and a network visualisation system was designed and implemented. Finally, the system was evaluated and successfully confirmed techniques to improve performance and network visualisability.

Acknowledgements

I would like to thank my project supervisor, Helen Purchase, and my advisor of studies, Dimitris Pezaros, for supporting me academically throughout my final year. Additionally, I would also like to thank Rory MacKenzie and all of SAS for motivating and supporting the project.

Finally, I would like to thank my friends and family for their continued care and support.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

Rendering massive networks (upwards of thousands of nodes and edges) in a browser is a frequent problem encountered in visualisation software. This is due to limits in browser performance as well as the fact that rendering information in a clear and meaningful way for the user is so difficult. This dissertation aims to analyse many different software packages available and outline different approaches which could be taken to help minimise the clutter and performance required to display these large networks. A network visualisation system is then designed, implemented and evaluated.

There are two main barriers to easily visualising massive amounts of data. The first is the sheer amount of data that needs to be visualised by the client, which requires a lot of bandwidth, processing power and RAM. Secondly, once the data is finally rendered, the result is a mass of nodes and edges of which no useful information can be taken.

A successful outcome of the project would be a system where, instead of data being passed straight to the client to be visualised, that data is analysed, modified, and a reduced version of the data is sent. This could possibly be done by: removing unnecessary nodes or edges, node bundling or edge bundling, or even by sending an image to the client. This would result in the data being far more useful to the user, with them being able to make informed decisions based on the network presented to them, as opposed to seeing an illegible mass of nodes. Additionally. this would reduce the processing required to render a network, resulting in lower loading times and bandwidth requirements.

Chapter 2

Motivation

There is limited research and development of software in regards to the visualisation of massive networks in a browser. A solution for companies who require network visualisation can be to use existing software (whether open source or with a license attached), or to develop their own software. SAS [1] have developed their own network visualisation software (for example, within the SAS product, Visual Investigator [2]) which performs well for up to a few thousand nodes, but begins to struggle when the number of nodes exceeds that - in relation to processing power, system memory and bandwidth. As a result, SAS, who explicitly support this project, are keen to find ways for larger networks to be visualised both with high performance and high information throughput.

The solution to this problem is either to enhance the existing software or develop new solutions. If the current software was to be improved, then the goal would be to find out what bottlenecks exist throughout the process of loading a network, and then try to minimise the impact of the bottlenecks. If new solutions were to be created, then there are several possible ways the software's performance could be improved:

- The network could be processed server-side, before it is passed to the client in order to:
  - Put less stress on the user's machine, so the transfer of the network and its rendering becomes quicker.
  - Make visualisation clearer, by removing unnecessary information.
- Joining nodes or edges together that are similar (bundling) could be done in order to reduce loading times. This could be done on either the client or server-side.
- Different tools could be explored that could make visualisations clearer to the user

Following discussion with peers and then conducting initial research, it was found that there was little research done into massive network visualisation. Csardi and Nepusz have stated that there is a "lack of network analysis software which can handle large graphs efficiently, and can be embedded into a higher level program or programming language (like Python, Perl or GNU R)" [3]. Another article, by Chen and Chaomei, declared that there is "A prolonged lack of low-cost, ready-to-use, and reconfigurable information visualization systems" [4].

There are two important metrics to consider for massive network visualisation: the time it will take to visualise the networks from the user asking for it, and the amount of useful information the user will get from the displayed network. If neither of these criteria are met then the visualisation software is unusable. If the system is slow but displays useful information at the end then the system is not very user friendly, but can still be useful. Finally, if the system is both fast and displays useful information then it an optimal network visualisation system.

Chapter 3

Review of the Field

3.1 Data Visualisation

The visualisation of data is a field of critical importance that many huge companies rely on in order to make predictions, improve themselves, and get an edge over competitors. Data Visualisation is "the presentation of data in a pictorial or graphical format [which] enables decision makers to see analytics presented visually, so they can grasp difficult concepts or identify new patterns." [5], as defined by SAS - a world leader in the field. This clearly defines Data Visualisation and how it is useful.

Visualisation of data can be static or interactive, with interactive visualisations able to give a user more information. This can be done by: letting a user select part of a visualisation and hence gain a greater understanding of the data, changing parameters for the visualisation and observing changes, or the moving of nodes in a network and seeing how the rest of the network responds. This can greatly improve the usefulness of the visualisation for users, with new trends or vulnerabilities becoming clearer faster through interacting with the visualisation. Theus stated that, "Interactive statistical data visualization is a powerful tool which reaches beyond the limits of static graphs" [6].

3.2 Network Visualisation

Network Visualisation is a branch of Data Visualisation involving the ability to display nodes and edges in a meaningful way to a user. Nodes can represent any object, from people to countries to laws, and edges are used to display when there is a connection between two nodes. This can be used for a large variety of purposes, such as to "identify nodes with the most links, nodes straddling different subgroups, and nodes isolated by their lack of connections" [7]. All of these tasks would be far more difficult without network visualisation.

3.3 Visualisation of Massive Networks

Visualisation of massive networks can lead to many challenges. Massive is a very vague term, but is generally considered as anything above about a thousand nodes [8]. These challenges generally present themselves in the form of computational challenges or visualisation challenges. Computational challenges are when a system struggles to display a network visualisation due to lack of speed or performance from the CPU, RAM or backing storage. If a machine has a slow processor then the amount of time to create and then render the network will

be increased greatly. Similarly, "data sets are often too massive to fit completely inside the computer's internal memory" [9]. If the RAM is not big enough to fit the network, then the performance of the network visualisation will degrade significantly due to external memory needing to be used.

The other computational challenge is the ability to store the network. Normally data is stored locally on a machine and loaded into a system in order to visualise it. However, when storing multiple massive networks, it can be nearly impossible to store this data on one machine. Hence, storing data on the cloud is often turned to, whether internally hosted within a company or externally hosted. This has a benefit as "Cloud computing is a powerful technology to perform massive-scale and complex computing." [10], meaning that huge amounts of data can be efficiently stored and processed.

Assuming that the machine is powerful enough in all of the above ways, and is capable of visualising massive networks, the next challenge is how to present that data in any meaningful way. When visualising a few hundred or thousand nodes, it is generally fairly easy to get useful information out of the visualisation. However, when a user is faced with a few hundred thousand nodes or even several million, it can be very difficult to gain any sort of insight from the visualisation.

3.4 Visualisation of Massive Networks in a Browser

Visualising massive networks in a browser adds several more constraints. Firstly, it requires all data to be sent over the internet so high bandwidth is required and even then, depending on the network size, a huge amount of time could be taken waiting for the network to download. Secondly, it requires any client-side processing to be done in JavaScript, which means that all code is run on a single thread and hence performance is poor when working with large datasets. Furthermore, JavaScript is necessary in order to render the network which is, again, a very slow process for large amounts of information.

A goal of the project was to research techniques currently used to visualise massive networks, look into how successful they were, and then make a system that allows for the visualisation of massive networks in a browser and across a network.

3.5 Example within the Field

3.5.1 Visual Investigator

Currently, the SAS product Visual Investigator [2] lets users view a network of entities in the system to see how they connect (for example: names, phone numbers and addresses). This works perfectly for a few entities, with useful information being shown clearly. However, with potentially millions of nodes, the network no longer shows much useful or practical information - just a mass of nodes - and becomes unusably slow (Visual Investigator is a web app and both processing power and memory are hard limits which, depending on network size, will easily be hit). However, it is worth nothing that the "mass of nodes" can still show useful information despite initially looking meaningless. The overall shape of the network can highlight bottlenecks of the system, or potential threats or holes in a system.

As can be seen in Figures 3.1, 3.2, 3.3, 3.4, the network starts out clear and loading times are instant, but it soon becomes both meaningless and impractically slow. If there was a way to both speed up the loading times of huge networks, and make them convey meaningful information to the user, this would be very beneficial.

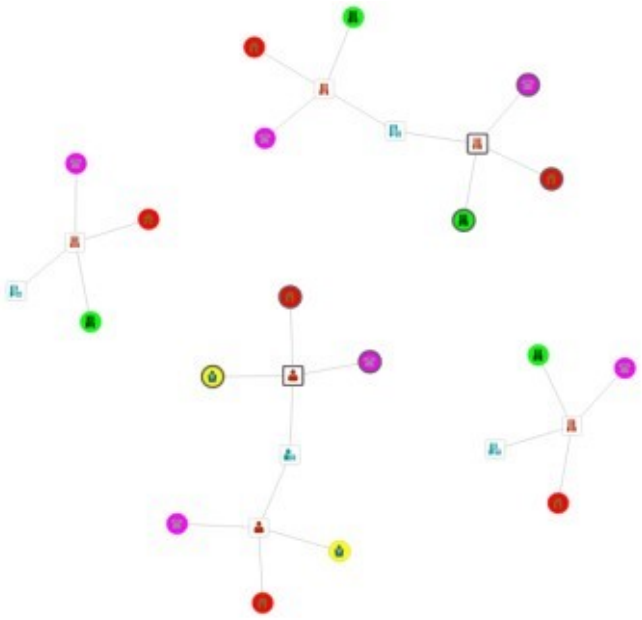

Figure 3.1: 30 nodes, Loading Time less than 1 second

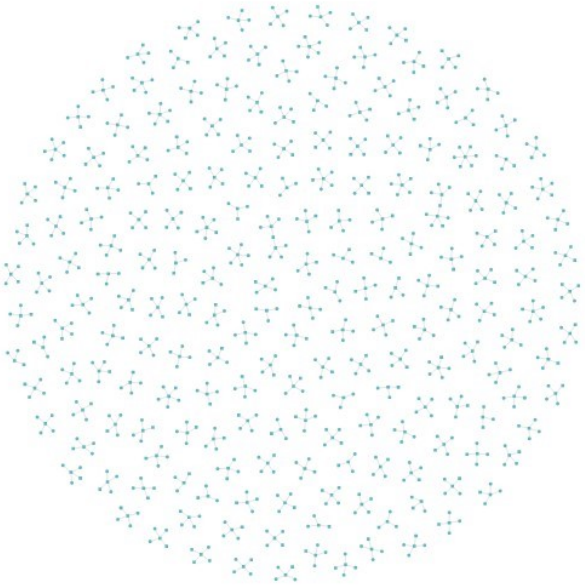

Figure 3.2: 1000 nodes, Loading Time 10 seconds

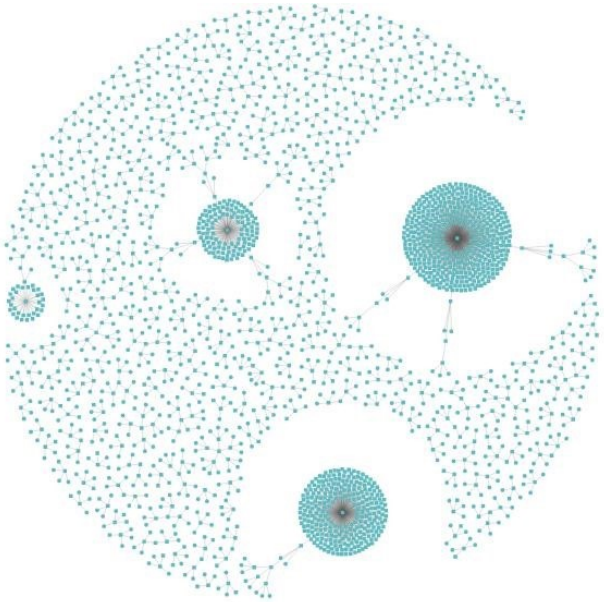

Figure 3.3: 2500 nodes, Loading Time 30 seconds

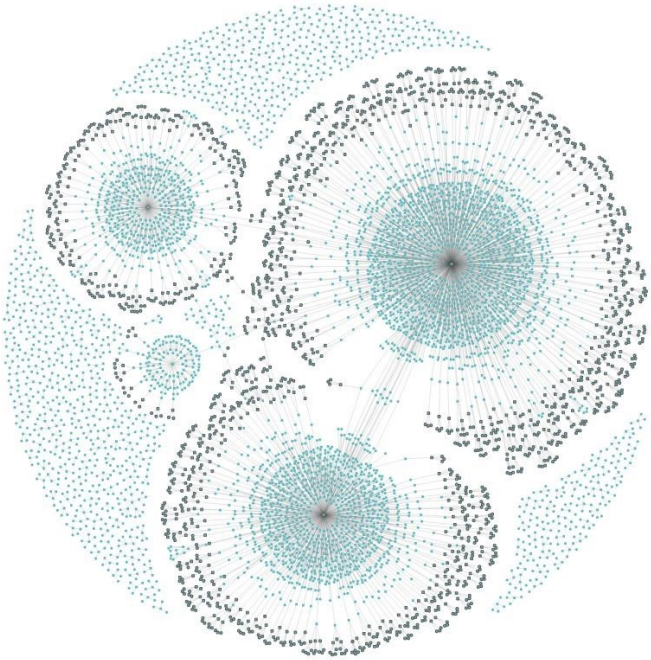

Figure 3.4: 7500 nodes, Loading Time 300 seconds

Chapter 4

Research

4.1 Background

Initial research was completed over a three month period and covered network visualisation, specifically massive network visualisation, and additionally looked into software that existed which visualised networks. The dearth of previous research confirmed that this is an area that was worth exploring further. This meant that exploration had to be done from scratch as there was no previous studies to build upon. However, this proved advantageous as the wealth of information gained during this period informed future decisions throughout the project.

In regards to the visualisation of massive networks, some important techniques learnt were:

4.1.1 Node bundling

This is when software uses algorithms to decide (with or without some form of user input) which nodes are very similar to each other, and upon establishing this relationship, groups them in to one node, generally visually different in some way (often by size or colour). It "creates a less complicated visualization without losing connectivity information by automatically abstracting small [...] subgraphs" [11]. This allows for hundreds or thousands of nodes to be grouped into one node and hence take up for less memory, processing power, bandwidth and screen space on the users machine. A way to make this even more helpful is to show how the nodes being bundled interact with themselves, for example if they are heavily connected or all connected to just a few nodes, or if they are in a ring formation. See Figure 4.1 for an example of both node and edge bundling.

4.1.2 Edge bundling

This is similar to node bundling but applied to edges. It "trade[s] clutter for overdraw by routing related edges along similar paths" [12], and "can be seen as sharpening the edge spatial density, by making it high along bundles and low elsewhere" [12]. If node bundling is done then this process will have less of an impact as all of the edges will already be bundled together, assuming the node bundling algorithm was optimal. Like node bundling, it is often shown through the change of the size of the edge or the colour. See Figure 4.1 for an example of both node and edge bundling.

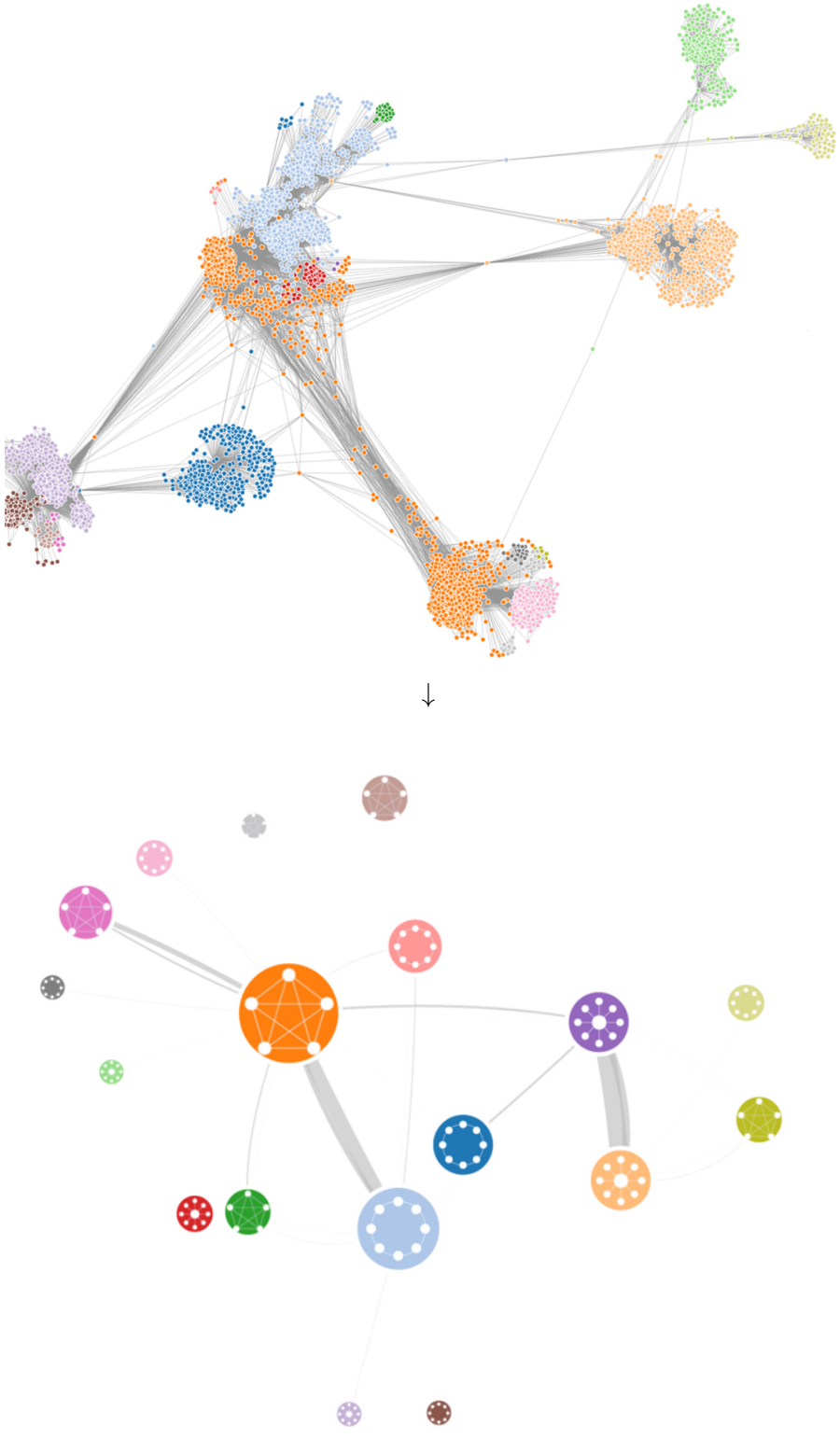

Figure 4.1: Demonstrates both node and edge bundling, where thickness of edge displays how many edges there are, and size of node indicates number of nodes group. On top of that, each node displays how nodes within it were connected together. This particular example displays a comparison between a common network visualisation and a ModulGraph-based network visualisation. [13]

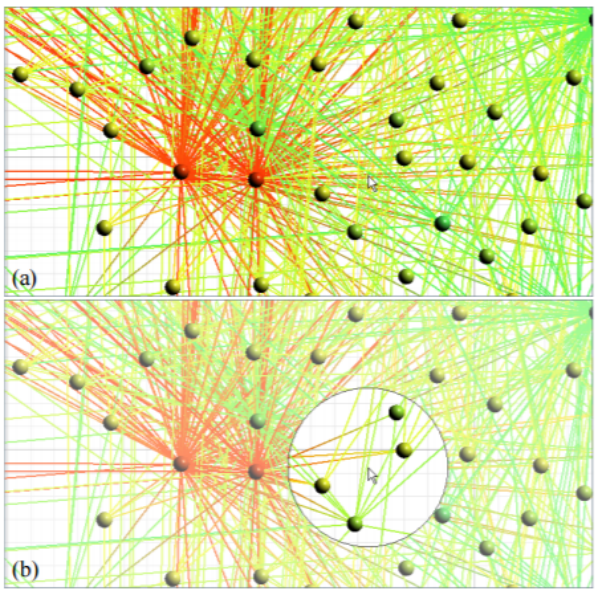

Figure 4.2: If a network is cluttered (a), then it can be hard to get useful information from it. A local edge lens (b) lets users easily identify which edges are connected to which nodes. [14]

4.1.3 Local Edge Lens

This involves rendering all or part of the network, and upon focusing on a section of the network, that part is zoomed in or clarified as if through a lens and that part of the network is zoomed in, becoming more clear. Another way of implementing this can be that when the lens is applied, it can render more information that was not shown before in order to save computational power. "By doing so, the cluttering of edges is removed for a local focus region defined by the position and scope of the lens" [14] See Figure 4.2.

4.1.4 Deleting unnecessary content smartly

This involves using an algorithm to remove certain nodes or edges that are deemed to be adding little or no useful information for the user. This algorithm can take no parameters, which would then make it a variant on Node/Edge bundling, or work off a user's input, in which they specify what they are interested in, so other information is partially or fully removed. It differs to Node and Edge Bundling as a result of the amount of information that is removed, which is far greater in this case. See Figure 4.3.

4.2 Current Network Visualisation Software

On top of finding out techniques used by software for large network visualisation, a list was also created of all software that was mentioned in papers or found during research that was related to large network visualisation. Quite often, the software mentioned had been hand-made for that project and was not feature complete, or the software could only be used for a price, but the below list contains the names of software that seemed reasonable to look further into, found by searching online and through mentions from papers:

- GUESS [16]
- SNAP [17]

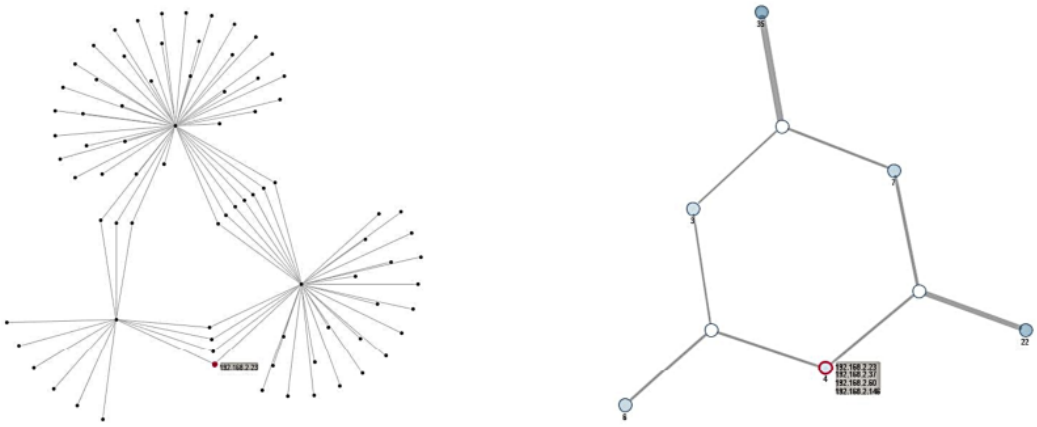

Figure 4.3: This figure clearly shoes how the deletion of many nodes mean that, although information is lost, the visualisation can become more clear as a result of the operation. [15]

- Gephi [18]
- GraphViz [19]
- Tulip [20]
- D3.js [21]
- Vis.js [22]

This software is reviewed in Chapter 5, below.

Chapter 5

Systematic Review of Existing Software

5.1 Aim

The goal of the review was to look into several different types of network visualisation software, and as a result, find out more about how it works. The software would be analysed for several characteristics, both related to the visualising of networks and the visualisation of specifically huge networks.

5.2 Methodology

Upon finishing background research and defining a clear aim for the review, a list of criteria and a process had to be created. These would include what would be searched for and tested for in each software package, and instructions on how to go about that testing. The initial task was to split the criteria into several categories and weigh each category as to its importance.

The first criteria to assess would be how easy it is to get a basic network (5 or 10 nodes) displayed from scratch, from downloading the software and installing it, to understanding it's documentation and finally to getting a network displayed.

Additionally, another criteria was how easy it is to load a network from a file or save a network to a file. These were important as a system would have limited use if it did not allow a user to upload networks, and allowing users to save networks (potentially after it was modified by the system or the user) would mean the network could be loaded in from another system or at a later date.

The next features to be analysed were based on interactivity - how easily can a user edit the network to get more useful information out of it. This would include the ability to zoom into a part of the network or being able to pan. A more complex feature (that would be rarely used but would occasionally add valuable information to the user) would be the ability to make it dynamic - showing the change in a network over time (or another parameter).

A major criteria would then be how well the software would scale up to massive networks - this would include running tests on increasingly large networks and seeing how long the software took, and also if it includes node or edge bundling, or other ways of increasing clarity of massive networks. Time taken to render is also an important statistic, but less critical than the clarity of the massive amounts being displayed. This is perhaps the most vital category as no matter the above features, if the network is not clear for huge numbers of nodes or edges then it does not provide a solution to the problem at hand. An additional feature to increase user clarity would be the

ability to view the network in many different layouts, which would give users more flexibility and would often lead to a better understanding of the network.

This set of criteria, both qualitative and quantitative (to maximise information gathered), was compiled into a list to act as the process for the experiment. This enabled a systematic analysis of the software to take place.

1. **Standalone or library?**
   If a software package is standalone then can it be executed (often after it has been installed) without any programming on the user's behalf? If the software comes as a library package then that library will provide an API so one can call functions from the library in their own programs.

2. **Documentation**:

   (a) Is it easy to understand: is it explained clearly and concisely?

   (b) Is it comprehensive: are all aspects of the software fully documented?

   (c) Length of time spent reading until confident in using the system: how long was spent reading documentation before programming using the system could begin?

3. **Time until a simple network could be displayed?**
   After finishing reading the documentation and starting to use the software, how long did it take to create a hard-coded network with three nodes and two edges? These values were chosen as they represent a very basic network that should be instantaneous to load in nearly all systems, but an understanding of the software is still required to display the network.

4. **Can a file be loaded into the system?**
   Is there a way to load a file (whether in a standard file format such as JSON or XML, or in a custom file format) into the software? If this is possible, even only in one file format, then a script could be written in order to convert from that file format to whatever necessary file format was needed.

5. **Can a network be exported to a file?**
   Is there a way to export a network to a file, in the form of a representation of the network, such as JSON or XML? This would mean that the network could be easily passed between software applications and across a network.

6. **Can a network be exported to an image?**
   Could the network be exported to a file as an image, such as a PNG or a JPEG? This is useful as images will generally be far smaller than representations of the image in code, such as JSON or XML, and hence are far easier and quicker to send or store on a machine.

7. **How long it took to render a network of:**

   (a) 30 nodes

   (b) 200 nodes

   (c) 1000 nodes

   (d) 3000 nodes

   These numbers were chosen as it was expected that all software could handle all of the values, with all software having no trouble with thirty nodes, but software that performs less well being expected to struggle as the number of nodes got into the range of the thousands.

8. **If the software allowed for:**

   (a) Zooming: if certain parts of a network could be focused on and viewed in more detail.

(b) Panning: the network could be traversed in the x or y axis.

9. **Can a network be dynamic?**
   This would allow for information to be viewed across multiple points in time? This allows certain types of information to be visualised far more easily, in particular when data has been been collected over a long time period over which relationships between the data evolve.

10. **Does the software included features for displaying huge networks:**

    (a) Node Bundling: See section 4.1.1.

    (b) Edge Bundling: See section 4.1.2.

    (c) Alternative layout options: This includes the ability to change what algorithms decide how the network is laid out.

11. **Can a network be multivariate?**
    Does the software support nodes being grouped in some way? This could be in many ways, such as based on shape or colour, and would allow users to distinguish different catagories of data.

12. **Is there a possibility of changing graphical settings for the network?**
    Can nodes or edges be coloured differently at the users request? This could be done by changing their shape, size, colour or outline?

13. **Can a network be displayed in 3D?**
    Is there a way to view the network in three dimensions, for example as a globe? This allows for a unique insight into the data and can often make the data a lot easier to comprehend and/or reveal more subtle relationships in the data.

14. **Any other idiosyncrasies?**
    Is there any other aspects of the software of worth that is not identified above?

Each piece of software was evaluated against this criteria, with the goal of both finding out which pieces of software were more successful over many parameters, and also to become more familiar with network visualisation software used in industry.

5.3 Data and Results

After starting to run the experiments, it became clear that two pieces of software from the list above would not be capable of being compared against the criteria previously created, which were GUESS and SNAP.

GUESS was software that never left beta, and was last developed in 2007. Having never been released meant that the software had fairly little documentation and what it did have was not comprehensive. On top of this, the wiki created for it was no longer hosted online which meant many of the links to documentation on the website were broken. Despite it looking promising, it became clear it would not be usable.

SNAP could not be compared to much of the criteria as the purpose of it is to analyse and manipulate networks, as opposed to visualising them. However, time was spent reading its documentation and going through all the sample code snippets to understanding how the software worked to some degree. The reason this was done was that it both felt important to have some knowledge of how network manipulation software worked, and also depending on the direction the project took, network manipulation may well be used in order to bundle edges or nodes in order to increase performance of massive networks.

The rest of the software however was analysed against the above criteria and the results are documented below.

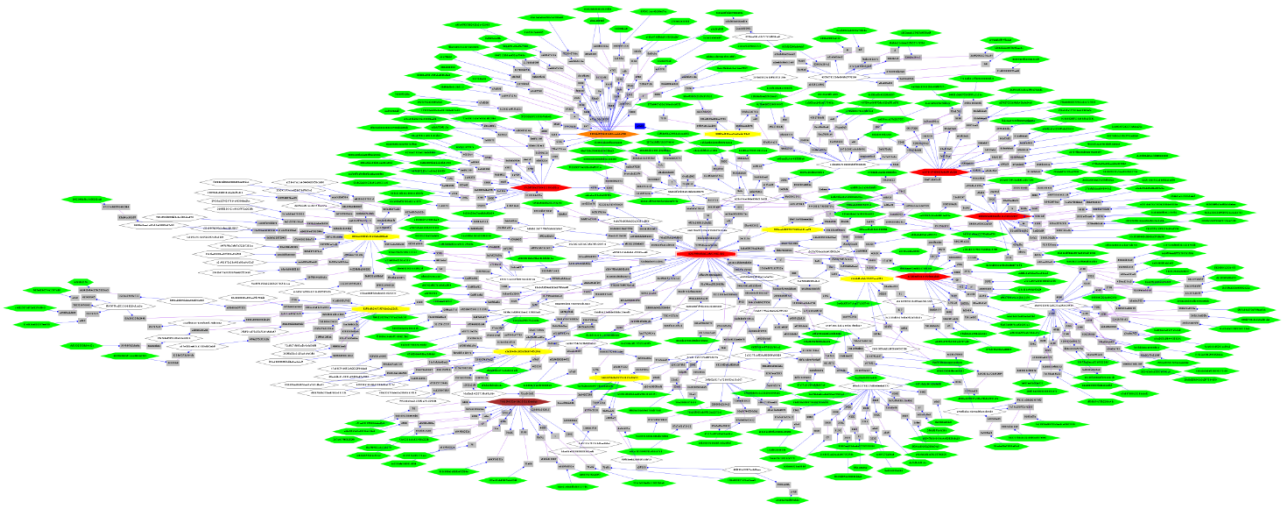

Figure 5.1: this shows how the User Interface is clearly focused towards smaller networks of up to a thousand nodes or so.

5.3.1 Gephi

This was the first software package that was as expected, and was fairly easy to test, use and was well documented. After about half an hour I was comfortable with how it worked and was able to make simple networks. Additionally, I could load and save networks from many formats (CSV, GDF, QML, GraphML, net, etc.), and also export to an image or document (PNG, PDF or SVG). It supported zooming, panning, and displaying information dynamically. It also included customisable layout options, some of which would be beneficial for visualising massive networks, although there was no explicit support for them. Additionally, there was the ability to change graphical settings, have the data visualised in 3D, and have a multivariate network.

5.3.2 Graphviz

Graphviz was difficult to test, being a collection of software as opposed to a single package. Most of the packages were not fit for purpose, often displaying data in charts, for example. The package I ended up testing was “sfdp”. Although it fitted most of the criteria and claimed to support large networks, it turned out to be very focused on visualising far smaller networks with little support for anything of a few thousand nodes or more. The documentation was clear, and the software had the ability to import and export a to very large amount of file formats. It also supported zooming/panning, but the User Interface clearly was not build around supporting large networks, with information becoming very unclear quickly - see Figure 5.1. There was also no explicit support for visualising massive networks.

5.3.3 Tulip

Tulip is an information visualisation framework that lets users both visualise and analyse the data. It was very easy to use and thoroughly documented, and allowed for easy importing and exporting of networks. Additionally, it performed very well, allowing for 3000 nodes to be visualised nearly instantly. It also included edge bundling and ’clustering’ (a type of node bundling) to improve visualisations of massive amounts of data.

5.3.4 D3.js

D3.js is a JavaScript library for data visualisation. The library is far more flexible than the rest of the software tested, but requires far more setup to be done by the user. The documentation is good although it is more complicated than most of the other programs. Importing and exporting data is simple using JSON, and other file formats with slightly greater difficulty. It supports zooming and panning. Also, there are many ways to make it effective at showing massive networks, but this would require research into many different packages and potentially writing a lot of code.

5.3.5 Vis.js

Vis.js is, like D3.js, a JavaScript library for data visualisation. However, unlike D3.js, it is not hugely flexible but much easier to get a basic network setup. The documentation is very good and importing and exporting is easily possible via JSON. It also includes some node bundling options which could be looked into at a future date.

5.4 Software Comparison

For the majority of the criteria, all of the software packages performed similarly. All five of the fully analysed network visualisation software provided the functionality of zooming and panning, along with all of the standalone packages (Gephi, GraphViz and Tulip) allowing for users to export to/import from a file, and the two libraries (D3.js and Vis.js) had functions which made it easy to export a network to a file or import from a file. On top of exporting to a standard file format for a network, all of the software allowed for exporting to an image.

See Table 5.1 for render the times for each of the different pieces of software.

	30 nodes	200 nodes	1000 nodes	3000 nodes
Gephi	n/a	n/a	0.2	0.6
Graphviz	n/a	0.3	1.1	4.5
Tulip	n/a	n/a	n/a	0.1
D3.js	n/a	0.2	1.1	5.3
Vis.js	n/a	1.8	4.4	13.6

Table 5.1: This shows how long (in seconds) it took each of the different pieces of software to render the specified number of nodes. n/a symbolises that it was too fast to calculate.

See Table 5.2 for details on the quality of documentation.

	Easy to Understand	Comprehensive	Time spent until confident
Gephi	✓	✓	30 minutes
GraphViz	✗	✗	1 hour
Tulip	✓	✓	15 minutes
D3.js	✗	✓	1 hour, 30 minutes
Vis.js	✓	✓	5 minutes

Table 5.2: This shows the quality of the documentation for the different pieces of software.

See Table 5.3 for details what features the software had that support visualising massive networks.

	Massive Network Layout Options	Node bundling	Edge bundling
Gephi	✓	✗	✗
GraphViz	✗	✗	✗
Tulip	✓	✓	✓
D3.js	n/a	n/a	n/a
Vis.js	✗	✓	✗

Table 5.3: This shows what support each of the software packages had for massive networks. D3.js has n/a in all cells as it does not support these features natively. However, D3 is built on using third party packages and libraries and some of these packages support several different massive network solutions, including the ones analysed for above.

See Table 5.4 for details on how long it took to create a simple hardcoded network, whether the network visualisation software supported dynamic networks, if the software included graphical options, if networks were viewable in 3D and if a network could be multivariate.

	Gephi	GraphViz	Tulip	D3.js	Vis.js
Time until hardcoded network could be created (minutes)	5	30	1	30	5
Visualisation could be dynamic	✓	✗	✓	n/a	✗
Graphical Options	✓	✓	✓	✓	✗
Ability to view in 3D	✓	✗	✓	✓	✓
Network could be multivariate	✓	✓	✓	✓	✗

Table 5.4: This shows additional information about the network visualisation software. For whether the visualisation could be dynamic, D3 - again - does not support this natively, but with additional libraries, it could.

5.5 Conclusion

Following the process outlined above was difficult as a collection of APIs was expected, which could have been easily compared by minor differences, but most of the software were standalone packages and were more difficult to compare. This highlighted the difficultly in conducting a systematic evaluation as, due to the complexities in comparing such diverse software, the criteria changed several times throughout the evaluation and continually evolved, meaning all previously tested software had to be analysed again. Additionally, only the JavaScript libraries would be directly suitable for SAS as the other pieces of software were standalone applications and could not be be utilised by a web application.

Despite the aforementioned challenges, a lot of useful information was gained throughout the evaluation, from how individual packages work to what techniques software uses to deal with large networks. Given all of the above data and comparisons, Tulip was clearly the best performing software. Like all of the other software, it allows for:

- Importing from a file (many file formats supported, including it's own .tlp format)
- Exporting to a file (to a similarly large number of supported formats)
- Exporting to an image
- High quality documentation
- Graphical options for the network exist

- Networks can be visualised in 3D
- Networks can be multivariate

However, it also rendered networks significantly faster than the other software, supported displaying massive networks in many ways (such as advanced layout options and node/edge bundling), could visualise dynamic information and the software makes it very easy to generate networks which made testing easier and more informative.

Additionally, Tulip has a full C++ API, and a full Python wrapper around the C++ API, meaning one can interface with it fully in either of the two languages, and a very simple file format, .tlp (of which more information can be found on their website [23]), which means that networks can easily be created by hand or converted to and from it's format to other formats such as JSON.

5.6 Potential Areas of Development

As a result of the above research and review, there were several different directions that the project could go. Four ideas that were considered are listed below:

5.6.1 Writing Massive Network Algorithms

One possible choice would be to write several different programs in JavaScript (or Python, as several of the packages explored earlier have Python APIs) that alter massive networks by any or all of the techniques highlighted in Section 4.1. These collections of algorithms would then be compared for how they make it easier to visualise and/or render networks using D3.js or Vis.js.

Upon writing these algorithms, they would be tested:

- On their own - what effect that function has on a network
- In conjunction with other algorithms - if combining certain algorithms together leads to a greater improvement
- With a large variety of different sizes of networks - if the algorithm hits a bottleneck after the network contains more than a certain number of nodes, or continues to perform as well as expected.
- With a variety of sparse or highly interconnected networks - some algorithms, for example bundling, may work particularly well with highly interconnected networks, and analysing performance for different levels of connectedness could reveal when it is most often best to use a certain algorithm.

For each of the above tests, each run would have it's performance evaluated (from time taken to how much processing power it required and the amount of RAM used) and then what effect it had on the end visualisation (if spending the time to run the algorithm gave any minor or notable benefit to the visualisability of the network).

5.6.2 Evaluation of existing systems

For each of the standalone systems evaluated in Section 5 - which were Gephi, GraphViz and Tulip - explore where each of the packages begin to struggle to display networks clearly or quickly and why. This would include

analysing the amount of time taken to run on different datasets, and for massive datasets, discover what task in particular makes it take as long as it would to display. Possible reasons could be that the network can't fit in RAM so data is constantly being put onto and pulled off of backing storage, or that algorithms have exponential performance which is negligible for a while but as networks get massive this performance starts to take it's toll.

Additionally, it would look at how effective the visualisations that created were at imparting knowledge to the user. This analysis would both take into account the quality of the visualisation baring in mind how long it took, and also how clear the visualisation was, regardless of time taken to create it.

5.6.3 Exploration of database visualisation systems

Another option would be to look into different database visualisation systems. Many database software solutions incorporate visualisation into them, and/or support writing your own visualisation programs and linking them to the information in the database. This path could be interesting as most databases can handle a very large amount of data, and hence storing the data would be an issue that does not need to be considered.

5.6.4 Creation of a web application using Tulip

Given that Tulip was considered the best software under the criteria tested for in Section 5, another possible idea would be to create a network visualisation web application using the APIs that Tulip provides. Given that Tulip provides both a C++ API [24] and Python API [25] (of which the Python API is a wrapper around the C++ API), the web application could be made using either of the languages. Given that "Python is very flexible and makes experimentation easy" [26], and also that Python has a popular web framework, Django [27], which is "a powerful Web application framework that lets you do everything rapidly from designing and developing the original application to updating its features" [28], these seemed a reasonable choice of stack to develop on. It is worth noting that Tulip has covered web visualisations before, when its creators visualised the Firefox Dataset as part of a competition [29]. However, this was an isolated project and currently there is no supported web visualisation framework made by Tulip.

5.7 Chosen Path

After much discussion and thought with both peers and my supervisor, it was decided that **creating a web application based on the Tulip Python API** is how the project should progress. This would result in users being able to access a wealth of information without owning a huge amount of computational power. This could benefit both employees of a business (allowing them to work from home on less powerful devices and without the need for huge data connections) and also clients of a business with lower specification machines, which could potentially lead to more business.

Chapter 6

Design

6.1 Scope of the Tulip Web Wrapper

6.1.1 Overview

The application that was to be created was planned to have:

- A Django Web Framework
- A database that was to be created and maintained by Django
- A Tulip Python API Web Wrapper that would be called by Django and make calls to Tulip
- A front-end JavaScript library to interface with the Django web framework, allowing for the uploading, viewing and deletion of networks
- A JavaScript network visualisation frameworks

6.1.2 Back-end

The back-end would consist of Django, the database it creates, the Tulip Python API (which is a wrapper around the Tulip C++ API) Web Wrapper, and potentially the ability to connect to a file host. This would enable it to be expandable to as many users as the host deemed necessary.

The main Django app and database it creates would be fairly simple to create, and would include the logic for the server.

The Tulip Python API Web Wrapper (itself an API) would include a large amount of functions that can be called from the front-end of the app, and would take in any necessary parameters and pass them to the Tulip Python API. The main purpose of this would be so that when a network gets loaded, instead of sending it directly to the front-end to be rendered, it is optimised in some way first (via for example node bundling or edge bundling) and then sent to the client in a reduced form which will both mean the network will be transferred faster to the client and then visualised faster too.

The web wrapper could be made exactly to match what functions the front-end requires, but a far more elegant and reusable solution would be to make it as flexible and as full as possible, allowing developers to have as much

freedom as possible, and not have their options limited by the web wrapper's API. For this project it is likely that the amount of flexibility that the web wrapper would have would be prioritised over the amount of features, given time constraints, although at a later date additional functions could be added with ease. Another advantage of creating an API is that it would mean that developers could make their own client, whether in JavaScript or in another language such as Java or Python.

A necessity for the web wrapper would be that in the vast majority of (or all) cases that the system works better using the web wrapper than just passing the nodes directly to the client. There should be a large increase in performance as the amount of data gets higher.

Allowing the data to be hosted, either internally on a network drive within a business, or on an external data host such as Amazon's S3 [30], would allow the user to store multiple massive networks on a low power and low storage machine. Although this was not implemented during the project (both given time constraints and the fact that external storage would have to be rented), it was kept in mind throughout development and the option to expand the application to hosted data remains and minimal development would be required in order to set this up.

6.1.3 Front-end

The front-end main aim would be to: make calls to the Django back-end, get back a network, and render that network, along with additional functionality such as uploading new networks and deleting current networks.

Although it would be possible to have single JavaScript file that took in input from the web page, called the back-end and then received the result of that call and rendered it, it would be preferable for a number of reasons to split the JavaScript into a library and an interface. The benefits of splitting the data into a library and an interface include the fact that it makes the code far clearer, and that it means that other developers would be able to develop their own interface that calls the JavaScript library created for this project.

When the user wants to upload a file, the user will be given the choice of the file name, what file to upload, and whether it is a .tlp or an industry style .json file (based upon sample SAS data).

The visualisation of the network would be done using either D3.js or Vis.js which were explored in the systematic review of visualisation software in Section 5. They both have advantages and disadvantages that are discussed in Section 6.2.

Ideally, the interface would be polished, fully-featured and user tested - with the results of those tests implemented into design. However, given the amount of time allocated for the project, it was likely that a relatively simple interface would be created that exposed all necessary features that the back-end provided, but was not as refined as it could be with more development time.

After defining the scope of the project, MoSCoW requirements were created that can be read in Appendix A.1.

6.2 Design Decisions

Justification for including or not including the following features:

6.2.1 Back-end

Web Endpoint

The core part of the project was creating a web wrapper for the Tulip Python API. The justification for making this is to enable low specification clients (low processing power / RAM / storage) to visualise data quickly. Currently alternatives include using spreadsheet software which take a long time to both create and render visualisation and is hard to make meaningful, or using graphing software, which has the drawback that it requires installing the software locally. Making a web wrapper for Tulip would allow a user to log in to a system and visualise a potentially massive network quickly and easily due to all of the processing being done on a powerful server. Also, assuming that the software will be dealing with massive data (a user could have access to terrabytes of data) then it is not feasible that any client could store that locally.

Making the web wrapper flexible

Ensuring that the web wrapper is flexible is not essential for allowing the creation of the whole application, but ideally an API could be created that other developers can utilise easily. This involves making all the API calls as open as possible, covering as much of the Tulip Python API as possible, and making sure it is thoroughly documented.

Return an image

As opposed to returning a JSON object that includes nodes and edges, and then visualising them using JavaScript, an option could be added to the client to allow for the user to choose to return just an image of the network. This would allow for the system to work on very low power machines or machines with a bad network connection. However, this is not critical functionality and was not explored in the project.

6.2.2 Front-end

A Basic Interface

It is critical to create an interface as it will let users interact with the back-end of the system. However, this need not be complicated and a very simple user interface would still convey what is required.

Fully fledged interface

Having a more full-featured and finished interface was not be required in for the project, as although it would make the system look better, all functionality would be displayed in the basic interface. Additionally, developers could easily make their own interface for the system.

Uploading files

When uploading a file, along with a name and the file itself, the decision was made to allow the user to select whether the file is a .tlp of an industry standard .json. If .tlp is selected then the file is uploaded as is. However, if

industry standard is selected, then a conversion will be done on the file, and a python script will scrape the .json file for relevant information and a .tlp will be created based on that information. The .json files are provided by SAS and a .json file that is to be uploaded needs to be structured as SAS have in order for the conversion to work.

However, the fact that a conversion will be done on the SAS .json files will prove that it is simple for developers to create their own scripts to convert from one file type to .tlp, and insert that script into the system.

Network Visualisation

The front-end will obviously need some way to render the networks it gets passed from the back-end and it was previously discussed above that either D3.js or Vis.js would be used. There are a number of advantages and disadvantages with each of the systems. D3.js is far more powerful and can do so much more in regards to visualisation of any sort of data than Vis.js, and on top of that is a lot more flexible, giving the developer far more control over what they create. However, these benefits come at a cost - that it is far more complicated to set up and work with. On the other hand, Vis.js is very simple to set up, and it takes no more than two minutes to run the sample code and let one visualise a network.

After extensive consideration it was decided that **Vis.js** would be used to visualise the networks created. This was because several days were spent trying to get D3.js to work as desired but a satisfactory implementation was not made, and the fact that Vis.js does not perform as well as D3.js will make it clearer to see if development done on the web endpoint has made an improvement on loading times.

Creating a JavaScript Library

In a similar vein to making the web wrapper flexible, as opposed to just making an interface that has all of the JavaScript calling the web endpoint as part of it, it would be preferable to make a JavaScript library that would interface with the web endpoint, and then the client would call the JS library. This would make it far easier for developers to develop their own client using the library and the web endpoint, which could be hooked into their own data source easily.

6.3 Technical Infrastructure

Figure 6.1 shows the infrastructure for the system.

6.3.1 Sample Data Flow

Based on the data flow shown in Figure 6.1, the below list was created to show an example of ahow a user could interact with the system, and what the result of those actions would be.

1. The user selects a network to open from a list of all networks on the system and clicks on “Open Network”.
2. The ID of that network and the request for the network to be loaded is then caught by the JavaScript interface which calls the `loadGraph` function from the JavaScript library, passing the network ID and a success and error callback.
3. The JavaScript library makes the appropriate AJAX call to the web endpoint, the Django server.

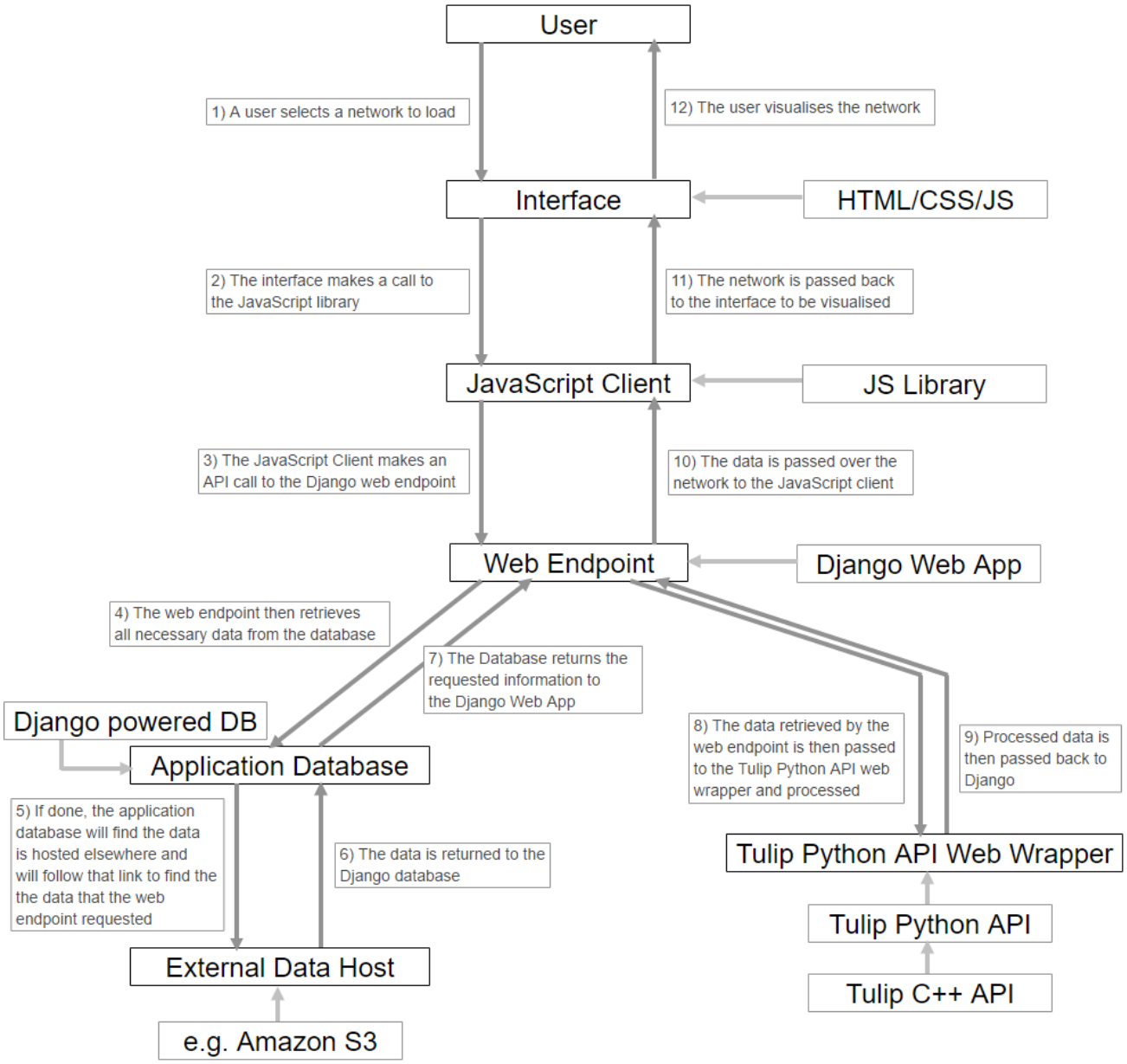

Figure 6.1: The Technical Infrastructure of the whole system, based off of the sketch in Appendix A.2

4. The Django web application queries the database for the file to load.
5. If an external data host is set up, then at this point the database would find the location of where the network is stored as opposed to the path the the file locally.
6. Again, if an external data host is set up, the network would be returned to the application database.
7. The network is then passed to the web endpoint.
8. Once the network is loaded into Django, it then calls a function from the Tulip Python API Web Wrapper while passing in the network, with the network is loaded from a .tlp and then TODO: describe what happens next.
9. After the network has been processed and minimised, it is then passed back to Django again.
10. From here Django passes either success or failure back to the JavaScript client, with success including the network that is to be visualised, and with failure including a message as to why the network could not be sent back.
11. The JavaScript library checks if the interface passed in an error callback function. If not then either nothing is returned if the call failed and the network is returned if it is successful. If as error callback function is passed in then if the call failed then JavaScript interface throws an error explaining why the call failed. If everything was successful then Vis.js renders the network.
12. The user can then examine and explore the visualisation of the network.

Chapter 7

Implementation

The Django Web App project contained 3 applications:

- The core Django app
- The Tulip Python API Wrapper, which controls all interactions between Django and Tulip
- The documentation app, which includes all of the front-end: the HTML/CSS, the interface, the Vis.js library and code, and the JavaScript library.

The core Django app included the URL paths for the whole project and the project settings, and no more.

7.1 Tulip Python API Web Wrapper

The web wrapper for the Tulip Python API was a major part of the project, and contained all interactions that Django had with Tulip. Tulip has a python package [31] that can be installed using pip [32]. Calls can then be made to the Python API from within Tulip, such as loading .tlp's and running code against the loaded networks. The end result of this was enabling Tulip to be called by Django (after a user asked to visualise a network), and then creating a network in memory and running operations against it.

Several critical design decisions had to be made throughout the creation of the web wrapper, listed below.

7.1.1 How to handle throwing errors - server side

A decision had to be made as to how the client would be made aware that the server had caught an error. Simply, there are three states which the server can return: success, expected error and unexpected error. From within Django there are many ways in which the errors can be dealt with. The page can simply be refreshed, flushing the error, a custom error page can be shown displaying the error code, such as 404, or an error can be passed back to the interface which can then be caught and displayed as appropriate. As one of the goals of the project was try to make the web wrapper a reusable API as oppposed to a single use system, it was decided that each return would include a JSON object, which would include a success boolean, and then either the expected information (such as the network) or a message with why the error had occurred. It would then be down to the library that called the API to catch the JSON appropriately, and then the interface to act on them accordingly. For this application, it can be seen in Section 7.2 how the library and interface handled errors.

7.1.2 How to transfer a network to JavaScript

There were two ways in which the network could be sent across a network to the user. These were:

- Serialising the network and then recreating it from within JavaScript
- Transforming the network into JSON and then sending it over the network

The advantage of serialising the network and recreating the whole tlp in JavaScript is that the client then has access to the whole network and there is no data loss in the transaction. However, the advantage of transforming the network into JSON and then sending it across the network is both that the size of the transfer is less as only the required data needs to be sent, superfluous data can be cut out, and also that the client does not need to then recreate the network after receiving it. It was decided that the data would be converted into JSON and then transferred over the network. This decision was made as high performance for massive networks was one of the main goals of the system. Transferring only what is necessary both reduces the amount of data that needs to be sent over the network, and also minimises the amount of processing required by the client. These are both ways to increase performance.

7.1.3 How to store the networks on the server

Another critical design decision had to be made relating to how the networks were to be stored on the server. Upon further research, three possibilities presented themselves as to how this could be completed:

- The structure of the network could be stored directly in a database's table. This could be done by having a table in the database for networks which would include a network ID, all of the nodes edges in a network.
- The .tlp of the network could be uploaded to the server and a link to that location could be saved in the database.
- The networks could all be saved on Hadoop's HDFS (explained in Appendix A.3) so that working with Big Data would be supported.

The advantage of storing the network directly is that all data would be stored in a single database. For a small scale application, this would make managing the system easier and have no notable drawback. However, as many massive networks may be saved to the system, the database would quickly become very large and therefore its performance would suffer greatly. As a requirement for the system was that it would work on massive datasets, an alternative solution would have to be found. Out of the other two solutions, one involved saving all of the networks on the server, and the other on a HDFS cluster.

The advantage of storing the networks on the server means that you can still store a massive amount of data on the server without the overhead and complications of setting up a Hadoop cluster. However, the advantage of using Hadoop is that it would allow for far more data to be stored in HDFS, and then MapReduce would allow for considerably faster processing of massive networks.

For this project, it was decided that the network would be stored on the server as this was a simpler approach that would be far quicker to get up and running, and additionally it would not be possible to acquire the resources needed to utilise Hadoop effectively.

This links in to another challenge faced - if a large amount of backing storage had been available to utilise then more experimentation could have been done into how the software interacts with massive amounts of data.

However, as a result of not having this, the project was tailored in a way which meant that having large datasets was not required, and design decisions such as using Vis.js as opposed to D3.js were made, so differences in performance were more apparent.

7.2 JavaScript Library

The JavaScript library involved creating a function to match each function in the Tulip Python API Web Wrapper, which could be called from an interface. The main aim in creating the library was to make it as open and flexible as possible, allowing for developers to have access to exactly what they need, and on top of that be able to handle errors easily. Given that I had not developed a library before, research was done into the best way to take errors from the web wrapper and then pass them on to the interface. The outcome of this research was that the preferred way of error handling was to:

1. Make the interface call a function with any parameters, a success callback, and an *optional* error callback. An example of this can be seen in Listing 7.1

```
tulipWebApi.loadGraph(graphToLoad, function(result) {
    networkCreator.drawSimpleGraph(result);
}, function(error) {
    console.error(error)
});

```

Listing 7.1: How the interface would call the JavaScript library

2. The library then receives the call and makes the appropriate AJAX call to the server, and upon getting a result it checks if the call was successful. Line 10 of Listing 7.2 shows this check

   (a) If the AJAX call was successful then the callback function passed in by the interface is called with the data requested, such as the network requested. This is shown on line 11 of Listing 7.2.

   (b) If the AJAX call failed then the library checks if an `errorCallback` function was passed in by the interface. If so, then that function is called and the error message is passed. If not, then nothing is called. This means that the library has the option of failing silently if the developer making the interface wishes. This is shown on lines 12 to 16 of Listing 7.2.

```
loadGraph: function(fileName, callback, errorCallback) {
    $.ajax({
        url: ''/api/loadGraph'',
        data: {
            ''network_name'': fileName,
        },
        cache: false,
        type: ''GET'',
        success: function(result) {
            if (result.success) {
                callback(result.data);
            } else {
                if (typeof errorCallback === ''function'') {
                    errorCallback(result.message);
                }
            }
```

```
        },
        error: function() {
            if (typeof errorCallback === ``function'') {
                errorCallback(``Unknown server error.'');
            }
        }
    });
}
```

Listing 7.2: How the JavaScript library catches and passes errors

7.3 Interface and Vis.js visualisation

The interface involved catching events (such as button clicks) and then calling the appropriate JavaScript library function. If the call is successful then a success message appears and/or an action is executed, and if not then an error is outputted to the console and potentially to the DOM.

One of the core actions the front end was required to do is to actually render a network. This was done by taking in all the nodes passed to it by the JavaScript library, then converting all of the nodes and edges into two arrays, and then initialising the network with the container of where it would go, the data it would contain and any optional parameters. This can be seen in Listing 7.3.

```
drawSimpleGraph: function(result) {

    arrayOfNodes = [];
    arrayOfEdges = [];

    for (var i = 0; i < result.nodes.length; i++) {
        arrayOfNodes.push({id: i, label: i});
        result.nodes[i].outEdges.forEach(function(element) {
            arrayOfEdges.push({from: i, to: element})
        });
    }

    var nodes = new vis.DataSet(arrayOfNodes);
    var edges = new vis.DataSet(arrayOfEdges);

    var container = document.getElementById(``main_network'');

    // provide the data in the required vis.js format
    var data = {
        nodes: nodes,
        edges: edges
    };

    var options = { /* Setting many layout and physics parameters */ }

    // initialise the network
    var network = new vis.Network(container, data, options);
}
```

Listing 7.3: How to create a network using Vis.js

TODO: either specify that this is the bare bones or include new stuff i added.

There are several different variables that can be set in `options` on line 24 of Listing 7.3. A full list of options can be seen on the Vis.js website [33]. Several options were experimented with for this project, specifically within the layout and physics module.

In `layout` [34], if `improvedLayout` was set to false, then Kamada Kawai Algorithm [35] is *not* applied, which reduces stability of the network but improved rendering times.

In `physics` [36], there were several different parameters which could be finely tuned. Stabilisation was a major parameter experimented on for this project which sets if the network should be stabilised and, if so, by what means. Additionally, the maximum and minimum speed can be set, the time between steps can be changed, and the physics solver can be selected. This can be one of four preset solver: `barnesHut` [37], `forceAtlas2Based` (based on the Force Atlas 2 algorithm [38]), `repulsion` or `hierarchicalRepulsion`, or alternatively a custom model can be supplied. A custom solver could take in the following arguments: `gravitationalConstant`, `centralGravity`, `springLength`, `springConstant`, `damping` and `avoidOverlap`.

Although time was spent exploring the physics module and creating several custom solvers, it turned out that creating powerful physics solvers would improve the visualisation slightly at the cost of a large decrease in performance. Physics could also be turned off entirely and the network could be displayed in a ring, with nodes equally spaced.

TODO: make the above into a list and maybe include a few more options like size etc.?

Chapter 8

Testing and Evaluation

8.1 Testing

8.2 User Evaluation

8.2.1 Methodology

How I got users to evaluate it.

8.2.2 Results

What they said.

8.2.3 Conclusion

What this means.

8.3 Performance Evaluation

Chapter 9

Conclusion

This project aimed to research how existing software visualised networks, to do a systematic review of existing network visualisation software, and design, implement and evaluate software that could visualise networks in a browser.

9.1 Recommendations

As a result of the completed research and development of a prototype for this project, and despite the fact that the application developed is some way from being deployable to the public, there are several things I learnt that I feel SAS might benefit from using. Firstly, in regards to the initial research done and then the systematic review conducted, the most valuable piece of information learnt was that the best way to improve visualisability of a network is to use node and edge bundling, and that the best way to improve performance is to ensure that the server reduces the network as much as possible before sending it to the client to be visualised. These ideas were reaffirmed by the implementation of the visualisation system created for this project.

9.2 Future Ideas

9.2.1 Supporting Big Data

A possible way that the application could be extended woul d be to allow it to support Big Data. As a result of the node and edge bundling being done across the whole network, which may have hundreds of thousands or millions of nodes, the network could be pre-processed on the server and split off into multiple different sub-networks. Then Hadoop [39] (explained in Appendix A.3) could be utilised and each of these sub-networks could then have its nodes and edges bundled using different `map` job, and then `reduce` jobs would combine all of the sub-networks back into a single network. This would considerably increase the performance of the system.

9.2.2 External Data Host

Tied into supporting Big Data, another way the application could be improved is by allowing users of the system to link external data hosts to the database. This would enable users to link their own data host (whether stored

locally on company wide network storage or externally on Amazon S3 [30] or Microsoft Azure Storage [40]) to the application. Hence, all of a companies data could be visualised with relative ease.

Appendices

Appendix A

Additional Information

A.1 MoSCoW Requirements

After defining the scope of the project in Section 6.1, the below MoSCoW requirements [41] were made.

Must Have:

- A web endpoint that calls the Tulip Python API
- To be able to deal with large data (that is too large to just send over the network and be visualised by the client without manipulation)
- A basic demo interface
- To have data that can be rendered returned from the endpoint

Should Have:

- A neat JavaScript client, with neat meaning a well-documented list of functions that the client can call that then make calls to the web end-point
- A flexible web endpoint, with flexible meaning covers as many relevant API calls as possible, and making sure that as few of the parameters for the API calls are hard-coded. This has the benefit of both:
  - Making the system more useful for a larger number of clients
  - Letting users make more than just a JavaScript client, so if they want to make a C# or Java app that would not create a problem

Could Have:

- The ability to ask for an image to be returned if client is low power: This would mean that the visualisation would not be interactive, but would take very little time to transfer the image and next to no time to render it, as opposed to far more time for both tasks to send a network to be constructed on the client.

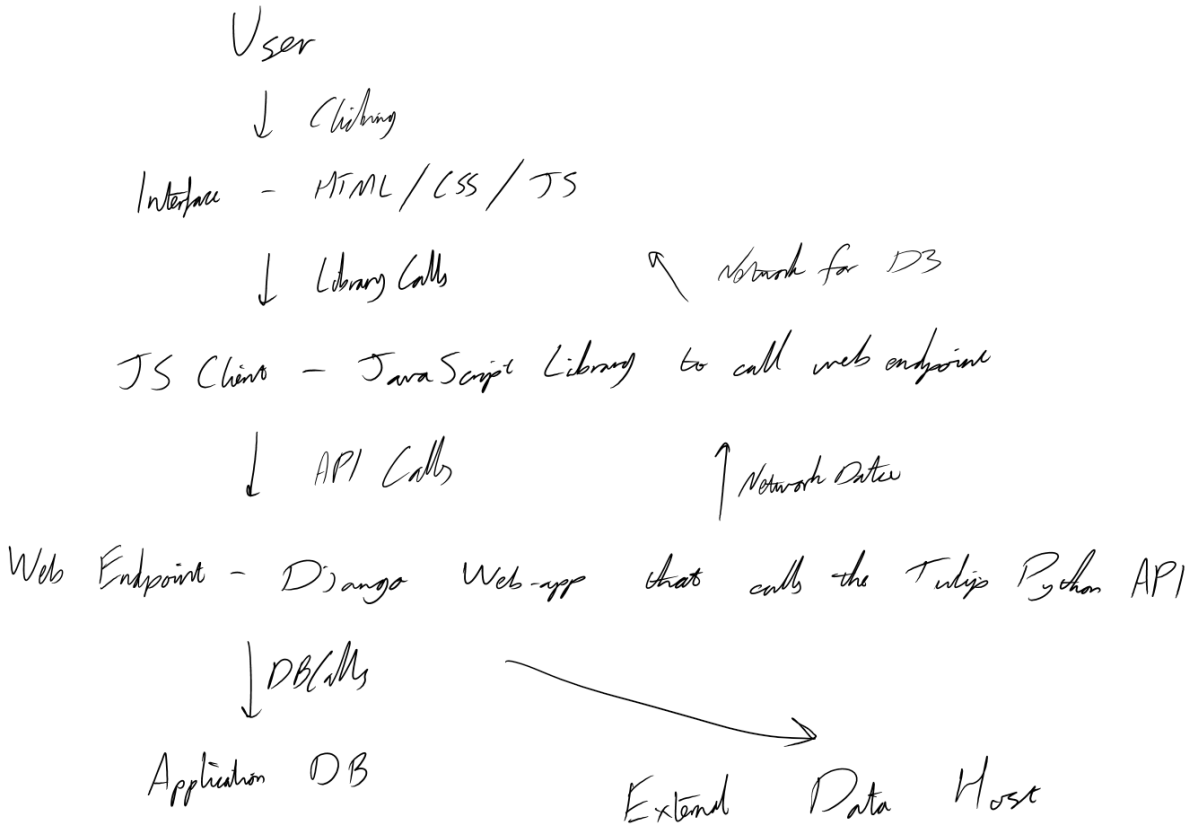

Figure A.1: Draft of Technical Infrastructure

- The ability for a user to link their hosted data to the application: This could be an in-house solution such as a a company network storage, or an external data host such as Amazon's S3 [30]. Hence, big data (hundreds of gigabytes or terrabytes) could be linked to the system with ease, each one being imported and linked to the system manually, or an XML/JSON file could be provided that allowed for batch uploading of data.

Won't have but would like:

- A fully fledged and pretty interface: It would be nice to have a fully completed front-end that is very user-friendly and has been thoroughly evaluated and tested. However, given the amount of time available for the project, it was decided that this is not as critical as having back-end functionality, and additionally that the front-end is being created to demonstrate what the back-end can do as much as to be part of the project.

A.2 Technical Infrastructure Sketch

See Figure A.1 for first draft of the technical infrastructure.

A.3 What is Hadoop?

Hadoop is a set of open source programs that contain four core modules. These are:

- Hadoop Common: The common utilities that support the other Hadoop modules.
- Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data.
- Hadoop MapReduce: A YARN-based system for parallel processing of large datasets.
- Hadoop YARN: A framework for job scheduling and cluster resource management.

These combine to give a way of storing, scheduling and processing Big Data in parallel. The way Hadoop could be utilised by the developed application is by storing all of the data that is to be visualised on HDFS. This would be done by using YARN to schedule the node and edge bundling jobs and then manage the resources available to the cluster where the jobs would be run. Next, MapReduce would be used to process each of the sub-networks and finally combine them back into a single network.

Appendix B

Using the Software

B.1 Running the Server

How to run everything.

B.2 Using the Interface

How to run everything.

Bibliography

[1] SAS. Analytics, Business Intelligence and Data Management. https://www.sas.com/en_gb/home.html. Last accessed: 4th March 2017.

[2] SAS. Intelligence Analytics — SAS Visual Investigator. https://www.sas.com/en_gb/software/intelligence-analytics-visual-investigator.html. Last accessed: 4th March 2017.

[3] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[4] Chaomei Chen. Top 10 unsolved information visualization problems. *IEEE computer graphics and applications*, 25(4):12–16, 2005.

[5] SAS. SAS Data Visualisation. http://www.sas.com/en_us/insights/big-data/data-visualization.html. Last accessed: 4th March 2017.

[6] Martin Theus et al. Interactive data visualization using mondrian. *Journal of Statistical Software*, 7(11):1–9, 2002.

[7] Nathan Yau. Why network visualization is useful. https://flowingdata.com/2010/11/17/why-network-visualization-is-useful/. Last accessed: 4th March 2017.

[8] Vladimir Batagelj and Andrej Mrvar. Pajek-program for large network analysis. *Connections*, 21(2):47–57, 1998.

[9] James Abello, Panos M Pardalos, and Mauricio GC Resende. *Handbook of massive data sets*, volume 4. Springer, 2013.

[10] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[11] Janet M Six and Ioannis G Tollis. Effective graph visualization via node grouping. In *Software Visualization*, pages 413–437. Springer, 2003.

[12] Christophe Hurter, Ozan Ersoy, and Alexandru Telea. Graph bundling by kernel density estimation. In *Computer Graphics Forum*, volume 31, pages 865–874. Wiley Online Library, 2012.

[13] Chenhui Li, George Baciu, and Yunzhe Wang. Modulgraph: modularity-based visualization of massive graphs. In *SIGGRAPH Asia 2015 Visualization in High Performance Computing*, page 11. ACM, 2015.

[14] Christian Tominski, James Abello, Frank Van Ham, and Heidrun Schumann. Fisheye tree views and lenses for graph visualization. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, pages 17–24. IEEE, 2006.

[15] Yifan Hu and Lei Shi. Visualizing large graphs. *Wiley Interdisciplinary Reviews: Computational Statistics*, 7(2):115–136, 2015.

[16] GUESS. GUESS - The Graph Exploration System. http://graphexploration.cond.org/. Last accessed: 4th March 2017.

[17] SNAP. Snap.py - SNAP for Python. https://snap.stanford.edu/snappy/. Last accessed: 4th March 2017.

[18] Gephi. Gephi - The Open Graph Viz Platform. https://gephi.org/. Last accessed: 4th March 2017.

[19] GraphViz. Graphviz - Graph Visualization Software. http://www.graphviz.org/. Last accessed: 4th March 2017.

[20] Tulip. Tulip - Data Visualization Software. http://tulip.labri.fr/TulipDrupal/. Last accessed: 4th March 2017.

[21] D3.js. D3 - Data-Driven Documents. https://d3js.org/.

[22] Vis.js. vis.js - A dynamic, browser based visualization library. http://visjs.org/.

[23] Tulip. Tulip software graph format (TLP). http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format. Last accessed: 4th March 2017.

[24] Tulip. Welcome to Tulip's documentation. http://tulip.labri.fr/Documentation/current/doxygen/html. Last accessed: 4th March 2017.

[25] Tulip. Welcome to Tulip Python documentation! http://tulip.labri.fr/Documentation/current/tulip-python/html/index.html. Last accessed: 4th March 2017.

[26] John M Zelle. *Python programming: an introduction to computer science*. Franklin, Beedle & Associates, Inc., 2004.

[27] Django. The web framework for perfectionists with deadlines. https://www.djangoproject.com/. Last accessed: 5th March 2017.

[28] Jeff Forcier, Paul Bissex, and Wesley J Chun. *Python web development with Django*. Addison-Wesley Professional, 2008.

[29] Tulip. Tulip vs Firefox Dataset. http://tulip.labri.fr/TulipDrupal/?q=node/1021. Last accessed: 7th March 2017.

[30] Amazon. Simple Storage Service. https://aws.amazon.com/s3/. Last accessed: 5th March 2017.

[31] Python Software Foundation. tulip-python 4.10.0. https://pypi.python.org/pypi/tulip-python. Last accessed: 5th March 2017.

[32] Python Software Foundation. pip 9.0.1 : Python Package Index. https://pypi.python.org/pypi/pip. Last accessed: 5th March 2017.

[33] Vis.js. Network decumentation. http://visjs.org/docs/network/#options. Last accessed: 7th March 2017.

[34] Vis.js. Layout decumentation. http://visjs.org/docs/network/layout.html. Last accessed: 7th March 2017.

[35] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[36] Vis.js. Physics decumentation. http://visjs.org/docs/network/physics.html. Last accessed: 7th March 2017.

[37] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[38] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6):e98679, 2014.

[39] Apache. Welcome to Apache Hadoop! https://hadoop.apache.org/. Last accessed: 7th March 2017.

[40] Microsoft. Azure Storage, Secure Cloud Storage — Microsoft Azure. https://azure.microsoft.com/en-gb/services/storage/. Last accessed: 7th March 2017.

[41] Agile Business Consortium. MoSCoW Prioritisation. https://www.agilebusiness.org/content/moscow-prioritisation-0. Last accessed: 5th March 2017.