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 5, 2017

Abstract

TODO: Do at the end

The purpose of this research is to identify different techniques that existing software uses to visualise extremely large networks in a browser, and then compare them and review which methods are the most successful. This is good because ??. I did it by doing ??. I found out ??. Conclusion.

This project is about researching and implementing some way for networks of hundreds of thousands of nodes and millions of edges to be viewed by a user in a browser. This involves challenges such as getting over the issues such as bandwidth, processing power and memory for machines running on a network, along with the difficulty of actually visualising the network after it is rendered.

Acknowledgements

I would like to thank my project supervisor, Helen Purchase, my advisor of studies, Dimitris Pezaros, and my friends and family for supporting me throughout my final year at university.

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

1 Introduction **1**

2 Motivation **2**

3 Review of the Field **3**

3.1 Data Visualisation 3

3.2 Network Visualisation 3

3.3 Challenges of the Visualisation of Massive Networks 3

3.4 Challenges of the Visualisation of Massive Networks in a Browser 4

3.5 Example within the Field 4

3.5.1 Visual Investigator 4

4 Research **7**

4.1 Background 7

4.1.1 Node grouping/bundling 7

4.1.2 Edge bundling 7

4.1.3 Local Edge Lens 9

4.1.4 Deleting unnecessary content smartly 9

4.2 Current Graphing Software 9

4.3 Systematic Review of Existing Software 10

4.3.1 Aim 10

4.3.2 Methodology 10

4.3.3 Data and Results 13

4.3.4 Conclusion 15

5 Potential Project Directions **17**

5.1 Potential Directions . . . 17

5.1.1 Writing Massive Network Algorithms . . . 17

5.1.2 Evaluation of existing systems . . . 18

5.1.3 Exploration of database visualisation systems . . . 18

5.1.4 Creation of a web application using Tulip . . . 18

5.2 Chosen Direction . . . 18

6 Design **19**

6.1 Scope of the Tulip Web Wrapper . . . 19

6.1.1 Overview . . . 19

6.1.2 Back-end . . . 19

6.1.3 Front-end . . . 20

6.2 Design Decisions . . . 20

6.2.1 Back-end . . . 21

6.2.2 Front-end . . . 21

6.3 Technical Infrastructure . . . 22

6.3.1 Sample Data Flow . . . 22

7 Implementation **25**

7.1 Tulip Python API Web Wrapper . . . 25

7.2 JavaScript Library . . . 25

7.3 Interface and Vis.js visualisation . . . 25

8 Challenges Encountered **26**

8.1 Research Difficulties . . . 26

8.1.1 Lack of Information . . . 26

8.1.2 Conducting a Systematic Evaluation . . . 26

8.1.3 Lack of available software . . . 26

8.2 Time Constraints . . . 26

8.3 Finding out how JavaScript libraries/Python APIs are written 27

8.4 Lack of resources 27

9 Evaluation and Testing **28**

9.1 User Evaluation 28

9.1.1 Methodology 28

9.1.2 Results 28

9.1.3 Conclusion 28

9.2 Performance Testing 28

9.3 Testing 28

10 Conclusion **29**

10.1 Achievements 29

10.2 Recommendations 29

10.3 Future Ideas 29

10.4 Lessons Learnt 29

Appendices **30**

A Additional Information **31**

A.1 MoSCoW Requirements 31

A.2 Technical Infrastructure Sketch 32

B Using the Software **33**

B.1 Running the Server 33

B.2 Using the Interface 33

C API Calls **34**

C.1 Tulip Web Wrapper Python Back-end 34

C.2 JavaScript Library for Tulip Web Wrapper 34

Chapter 1

Introduction

TODO: At the end

Rendering massive networks (tens or hundreds of thousands of nodes and edges) in a browser is a frequent problem encountered in visualisation software due to limits in browser performance and the lack of ability to render the information in a meaningful way that the user can make sense of it. This dissertation aims to analyse many different software packages available and outline different approaches which can be taken to help minimize the clutter and performance required to display these large networks, and then to describe, analyse and evaluate a system created for the purpose of this project.

There are several barriers to easily visualising massive amounts of information, including the huge amount of data that needs to be passed to the user from the server, requiring a lot of bandwidth, processing power and RAM, and, once the data is finally rendered, the result is a mass of nodes of which no useless information can be taken from.

A successful outcome of the project would be a system where, instead of the data being passed straight to the client that is to be visualised, that data is analysed, and a modified and reduced version of the data is sent to the client. This could be done by: sending an image to the client; removing unnecessary nodes/edges; or node bundling/edge bundling. This would result in the information being far more useful to the client, with them being able to make informed decisions based on the network presented to them, as opposed to before where they were shown a huge mass of nodes that could not be easily deciphered. This would also reduce the load times of the network, resulting in networks displaying far faster, and less bandwidth and processing requirements on the client's side.

Chapter 2

Motivation

There is limited research and development of software into the visualisation of massive networks in a browser. Some companies have developed in-house network visualisation software (e.g. the SAS product, Visual Investigator [1]) which works well for up to a few thousand notes, but this begins to struggles with many more nodes than that, in relation to system memory, bandwidth and processing power.

There are several topics of interest here:

- The performance of currently available software could be improved so it can handle far more data and still load relatively quickly.
- Processing data server side could be developed to have less stress put on the user's machine.
- Joining nodes or edges together that are similar (bundling) in order to reduce loading times could be explored.

Following discussion with peers and then conducting initial research, it was found that there was little research done into 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)" [2]. Another article, by Chen and Chaomei, declared that there is "A prolonged lack of low-cost, ready-to-use, and reconfigurable information visualization systems" [3].

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." [4], 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 data able to give a user more information by, for example, selecting parts of the visualisation and getting more information on it, 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" [5].

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" [6]. All of these tasks would be far more difficult without network visualisation.

3.3 Challenges of the 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 [7]. These challenges generally present themselves in the form of computational challenges or visualisation challenges. Computational challenges are when the 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" [8]. 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." [9], meaning that huge amounts of data can be 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 Challenges of the 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 ran on a single thread and hence performance very bad when working which large data sets. 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, Visual Investigator (a SAS product TODO: insert link for who SAS are) 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 any useful 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).

As can be seen from the pictures below, the network starts out clear and load times are instant, but it soon becomes both meaningless and really 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 useful.

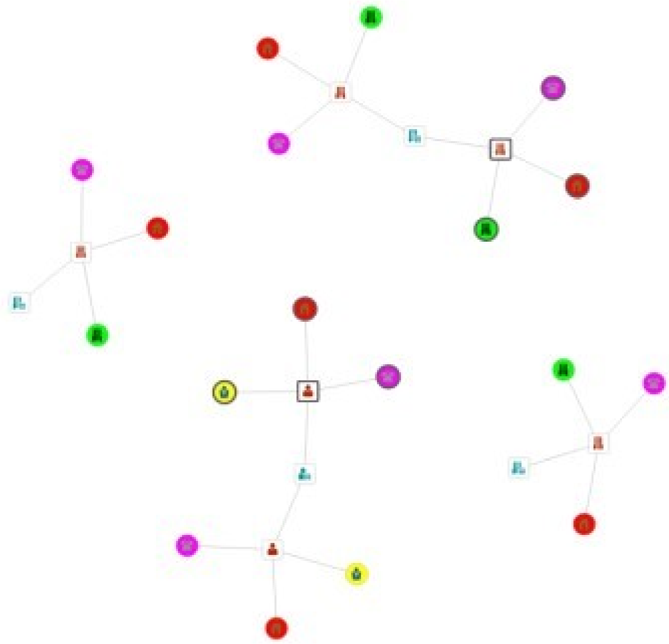

Figure 3.1: 30 nodes, Load Time less than 1 second

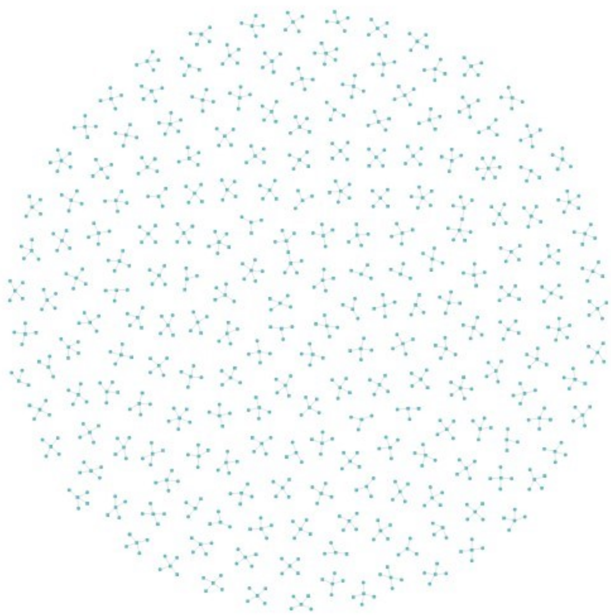

Figure 3.2: 1000 nodes, Load Time 10 seconds

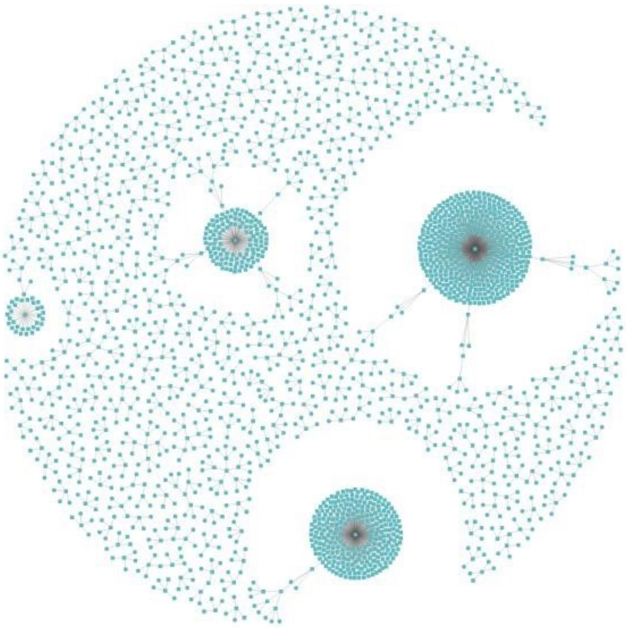

Figure 3.3: 2500 nodes, Load Time 30 seconds

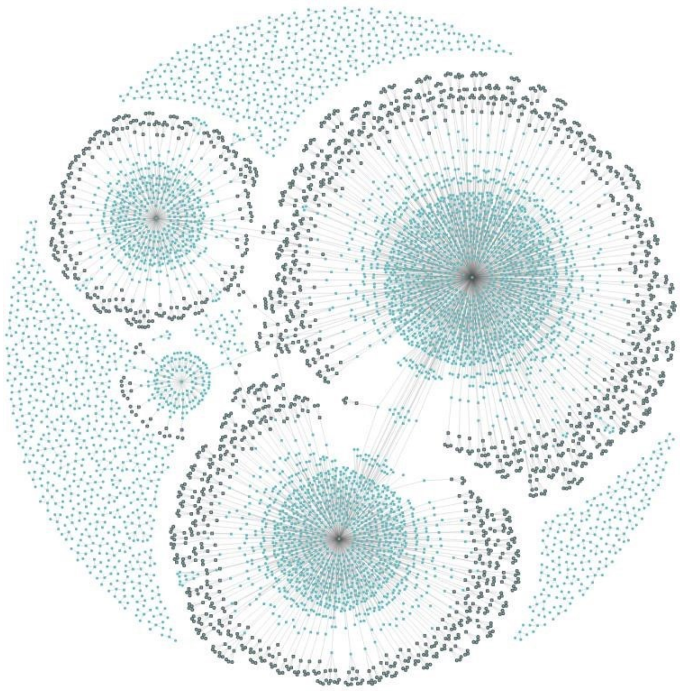

Figure 3.4: 7500 nodes, Load Time 300 seconds

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 reason so much time was spent investigating the field was due to the dearth of available research. 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 grouping/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” [10]. 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 grouping but applied to edges. It ”trade clutter for overdraw by routing related edges along similar paths” [11], and ”can be seen as sharpening the edge spatial density, by making it high along bundles and low elsewhere”[11]. If node grouping is done then this process will have less of an impact as all of the edges will already be bundled together assuming the node grouping 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.

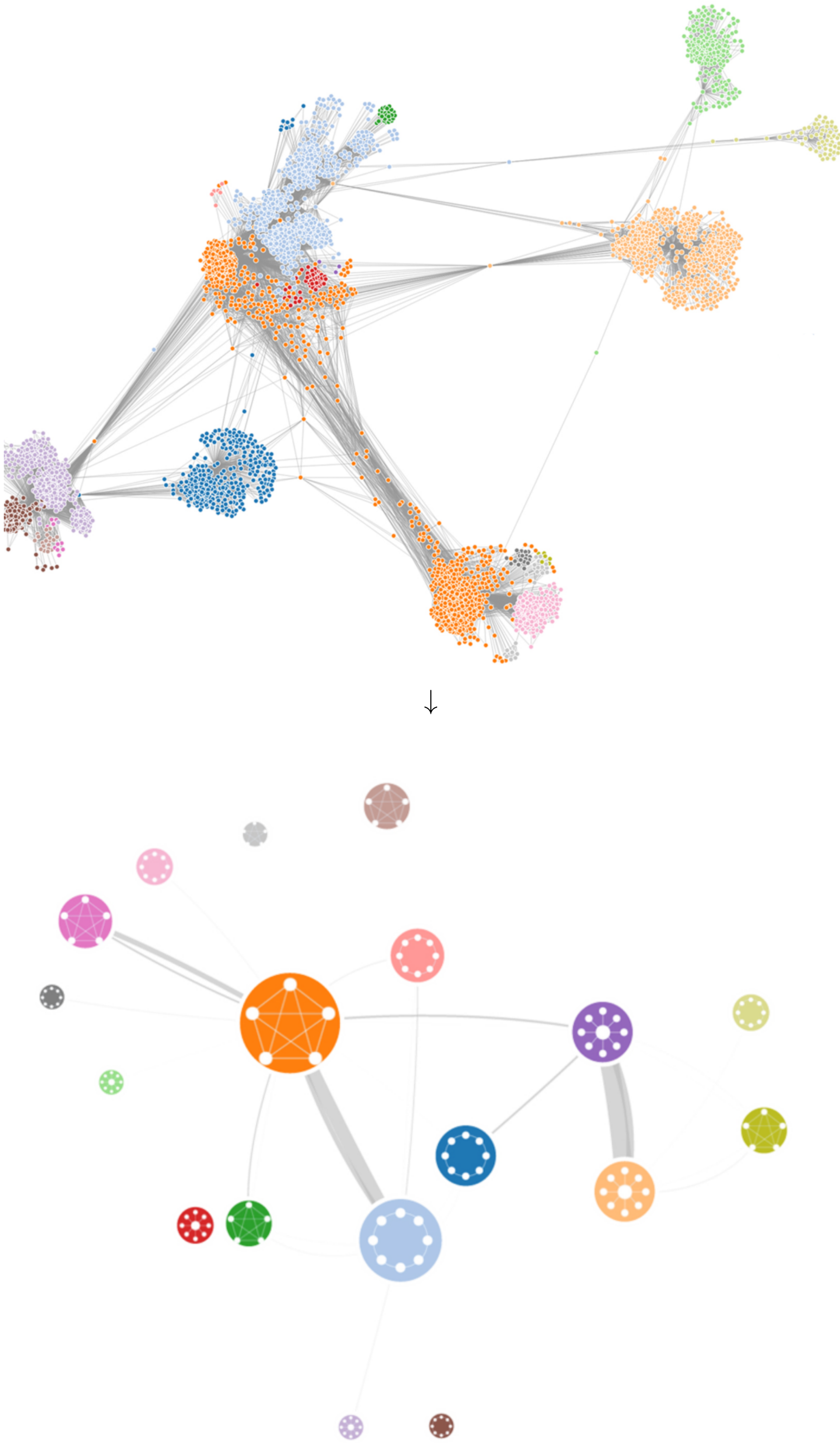

Figure 4.1: Demonstrates both node and edge bundling, where thickness of edge displays how many edges there are, and size of node indicates number of nodes group. On top of that, each node displays how nodes within it were connected together. This particular example display the comparison between the common graph visualization and ModulGraph-based graph visualization. [12]

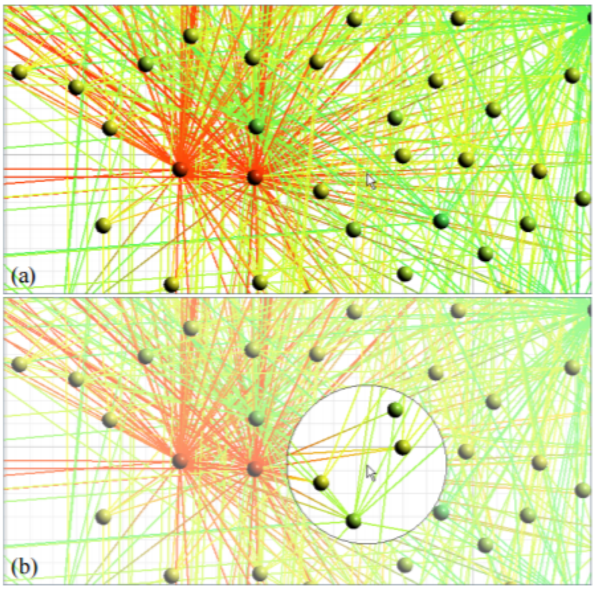

Figure 4.2: If a network is cluttered (a), then it can be hard to get useful information from it. A local edge lens (b) lets users easily identify which edges are connected to which nodes. [13]

4.1.3 Local Edge Lens

This involves rendering all or part of the graph, and upon focusing on a section of the graph, that part is zoomed in or clarified as if through a lens and that part of the graph 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” [13] 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 Graphing 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 [15]
- SNAP [16]

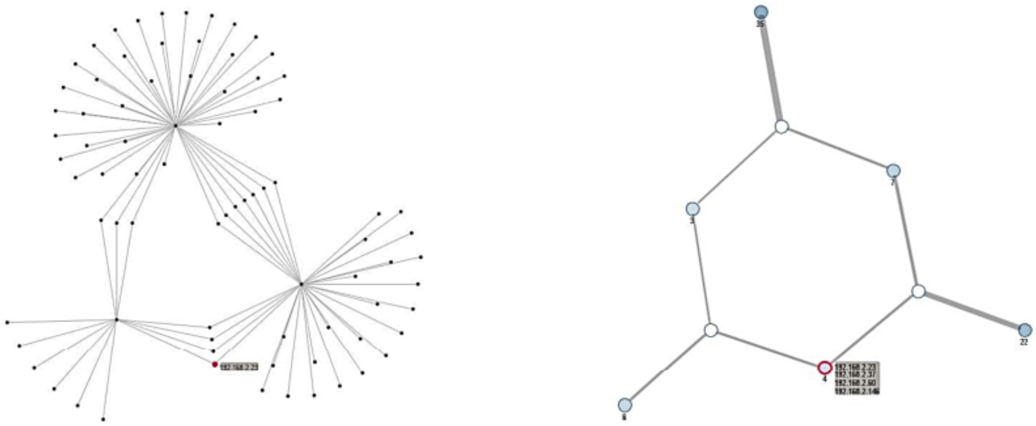

Figure 4.3: This figure clearly shoes how the deletion of many nodes mean that, although information is lost, the visualisation can become more clear as a result of the operation. [14]

- Gephi [17]
- GraphViz [18]
- Tulip [19]
- D3.js [20]
- Vis.js [21]

This software is reviewed below.

4.3 Systematic Review of Existing Software

4.3.1 Aim

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

4.3.2 Methodology

Upon finishing background research and defining a clear aim for the research, the next step was to come up with a list of criteria that each of the pieces of software would be compared against, and a robust process for how the software would be tested against the criteria. The process and criteria would ideally be based off an established way to systematically evaluate information visualisation. However, even after a long time was spent trying to find papers on that subject, it turned out to be surprisingly difficult to find out about testing software as a developer. Nearly all of the results found were about evaluating a system you had made and tended to be focused on getting professional testers to use the application or trialing the software on potential future end-users, as opposed to systematically evaluating systems.

The initial plan 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, to getting a network displayed.

After that, some more important criteria were how easy it is to load in a network from a file, and also, if the network was modified in any way between loading it in and displaying it to the used, or even while the user was viewing it, then saving the network to a file.

The next features to be analysed for will be based on interactivity - how easily a user can 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 graph 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 takes, 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 important than clarity for massive amounts of data for this project. This is perhaps the most important category as no matter the above features, if the network is not clear for huge numbers of nodes / edges then it does not answer the problem at hand. A feature that would be desired 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 better understanding.

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 to take place of the software.

1. **Standalone or library?**
   If a software package is standalone then it can 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 graph could be displayed?**
   After finishing reading the documentation and starting to use the software, how long did it take to create a hard-coded graph with three nodes and two edges. These values were chosen as they represent a very basic graph 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 the graph 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 has the allowing 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 constantly evolve.

10. **Does the software included features for displaying huge graphs:**
    (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 graph be multivariate?**
Does the software support nodes being labeled in more than one way, such as each node having a label and being part of a larger group?

12. **Is there a possibility of changing graphical settings for the network?**
Can nodes or edges be coloured differently at the users request, can they be enlarged or shrunk, or can they the shape of nodes or the background colour be set?

13. **Can a graph 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 that is not identified above of worth.

Each piece of software was to be 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 graphing software used in industry.

4.3.3 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 graphs, as opposed to also visualise them, despite the fact that it had looked promising on several papers. 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 results are documented below.

TODO: Should I remove some or all duplication in below text and further below tables?

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, able to make simple graphs, and to import graphs from many formats (CSV, GDF, QML, GraphML, net, etc.) and export to all of them, along with as a PNG, PDF or SVG. It supported zooming, panning and scrolling, and having data being dynamic was a possibility. There was also multiple heavily customisable layout options, some of which would suit large networks more than others, although there was no additional support for large networks, other than the system being very efficient (bundling was not supported, along with partial viewing of the data). Additionally, there was the ability to change graphical settings, have the data graphed in 3D, and have a multivariate graph.

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 graphing far smaller networks with little support for anything of a few thousand nodes or more. The documentation was okay, and the software had the ability to import and export as a very large amount of file formats. It also supported zooming/panning/scrolling, but the User Interface clearly was not build around supporting large networks, with information becoming very unclear quickly - see Figure 4.4. There was also no explicit support for visualising massive networks.

Tulip

Tulip is an information visualisation framework that also lets you both visualise and analyse the data. It is very easy to use and is well documented, allows for easy importing and exporting of networks, and is really fast,

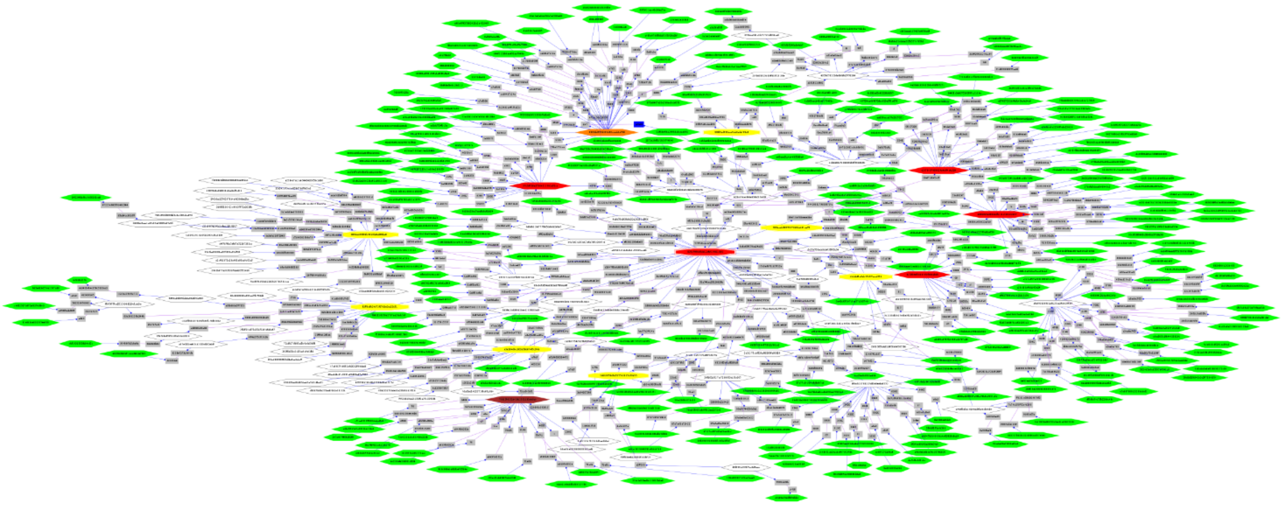

Figure 4.4: this shows how the User Interface is clearly focused towards smaller networks of up to a thousand nodes or so.

allowing for 3000 nodes to be visualised nearly instantly.

It is worth noting that on the university lab machines, without admin access, it was not possible to install Tulip, which meant I ended up installing it on Windows (the computer that Tulip was ran on is far more powerful than the university machines and there is a very reasonable chance that on the lab machines Tulip would not perform as well). However, it seemed very good quality software. It also included edge bundling and 'clustering' (a type of node bundling).

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 that other most of the other programs, and importing and exporting is easily possible via JSON, and other file formats with slightly greater difficulty. It supports zooming / panning / scrolling as expected. 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 code to do it.

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.

Comparisons

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 4.1 for render times.

See Table 4.2 for details on the quality of documentation.

See Table 4.3 for details what features the software had that support visualising massive networks.

See Table 4.4 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.

	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 4.1: 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.

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

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

Table 4.3: 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 but D3 is built upon importing third party packages and libraries in and some of these packages support several different massive network solutions, including the ones analysed for above.

4.3.4 Conclusion

Following the process outlined above was difficult as several software APIs were expected which could be easily compared by minor differences, but most of the software were standalone packages and were more difficult to compare. Also, none of the software would be directly suitable for SAS apart from the JavaScript libraries as the other pieces of software were standalone applications as opposed to libraries that could be utilised by a web application. Considering that the results were difficult to both gain and easily differentiate, this highlighted the

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

Table 4.4: 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.

complexities comparing such diverse software, as well as the lack of software suitable to compare against the criteria.

However, 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

Tulip also rendered networks faster than the other software significantly, 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 graphs 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 [22]), which means that networks can easily be created by hand or converted to and from it's format to other formats such as JSON.

Chapter 5

Potential Project Directions

5.1 Potential Directions

As a result of the outcome of the research done previously (in Section 4), there were several different directions that the project could go. A substantial amount of time was spent exploring some of these options, and below is the outcome of that exploration:

5.1.1 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).

5.1.2 Evaluation of existing systems

For each of the standalone systems evaluated in Section 4.3 - which were Gephi, GraphViz and Tulip - explore where each of the packages begin to struggle and why. This analysis would include analysing the amount of time taken to run on different data sets, and for massive data sets, what task in particular makes it take as long as it would. 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 how long it took in mind, and also how good the visualisation was, regardless of how long it took to create.

5.1.3 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 no longer be a problem in which a solution needs to be sought.

5.1.4 Creation of a web application using Tulip

Given that Tulip was considered the best software under the criteria tested for in Section 4.3, 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 [23] and Python API [24] (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” [25], and also that Python has a popular web framework, Django [26], which is ”a powerful Web application framework that lets you do everything rapidly from designing and developing the original application to updating its features” [27], these seemed a reasonable choice of stack to develop on.

5.2 Chosen Direction

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, which 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 leads to more possible clients for a 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 graph gets loaded, instead of sending the graph 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 an 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 [28], 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 webpage, 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 for 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 4.3. 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 time limitations 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 flawless 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 takes a long time and is not very meaningful, or using graphing software, but that 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 of the graph done on a powerful server. Also, assuming that the software will be dealing with a 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 it would make sense to create an API 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 blob of data including 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 given time constraints may not be included.

TODO: either make the last sentence end : However, this is not critical and not going to be explored in this project OR However, this is not critical, but was still explored due to usefulness for the user.

6.2.2 Front-end

A Basic Interface

Although the interface will probably not be very complicated, it is important to have some form of interface in order to demonstrate the product and show whole system working.

Fully fledged interface

This would make the product look neat and well-finished, but is not required in order for the project to be complete as the developers for users of the system would be expected to make their own interface.

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 needs to be structured as they 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

Figure 6.2 shows a sample data flow for the system.

TODO: Do I need to go into more detail here.

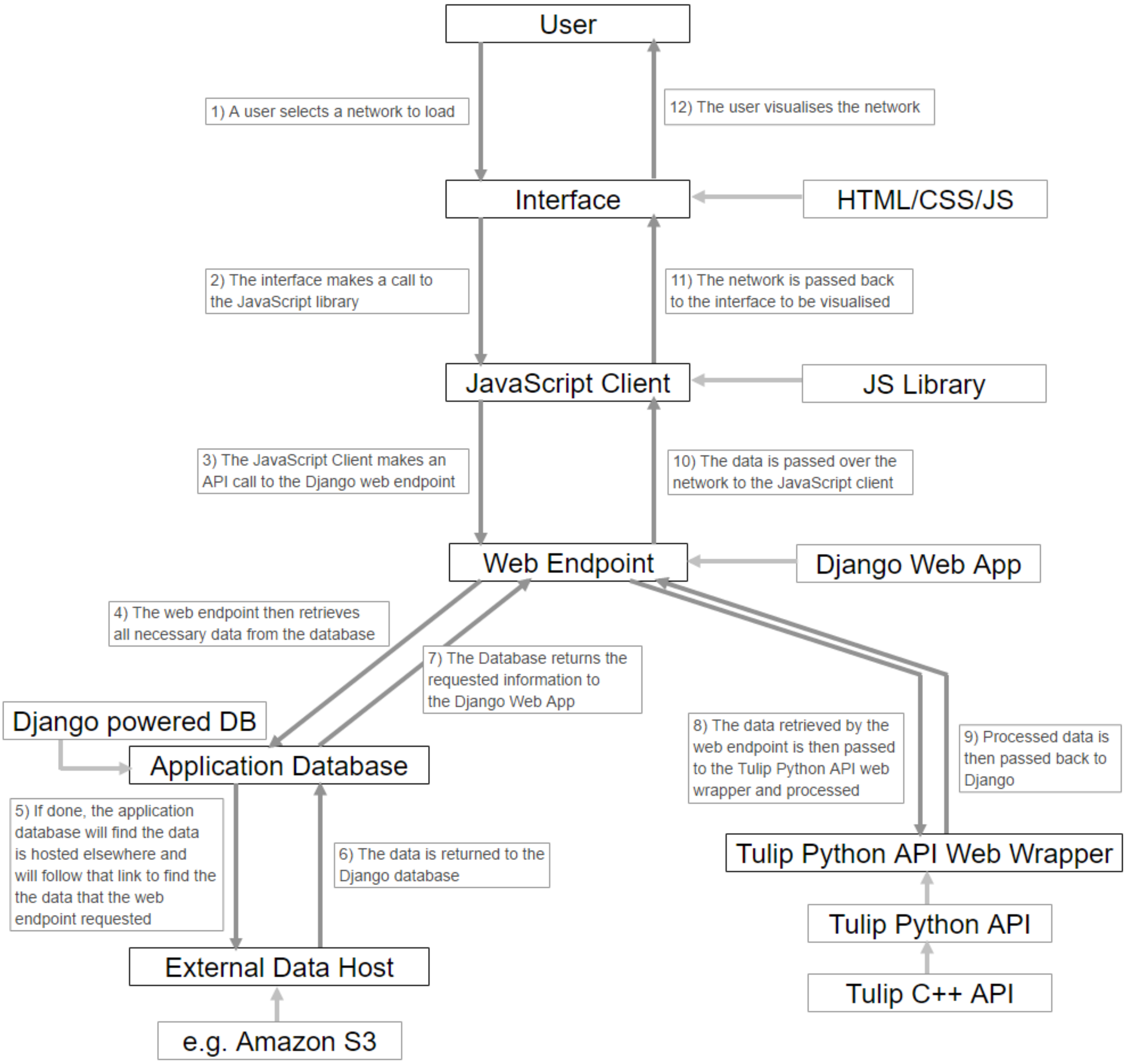

Figure 6.1: The Technical Infrastructure of the whole system, based off of the sketch in Appendix A.2

The user selects a netowrk to open and clicks on ”Open Network”

↓

Request caught by the JavaScript interface which calls the JavaScript library

↓

The JavaScript library makes the appropriate AJAX call to the web endpoint, the Django server

↓

Database queried for network and potentially where the network is hosted on the external data host

↓

The web endpoint uses the Tulip Python API Web Wrapper to call the Tulip Python API

↓

The Tulip Python API (using the Tulip C++ API) loads in the network and shrinks the data and make it easier to send over a network

↓

The new smaller network then given back to the web endpoint

↓

It is then transferred to the JavaScript Client

↓

Vis.js then renders the network

Figure 6.2: This highlights a possible way of a user interacting with the system and what the system does as a result of the interaction.

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 [29] that can be installed using pip [30]. Calls can then be made to the Python API from within Tulip, such as loading .tlp's and running code against the loaded networks.

7.2 JavaScript Library

The JavaScript library involved creating functions, each of which could be called from an interface (with or without an error callback), and then call functions from the Tulip Python API. Talk about why and how I handled errors.

7.3 Interface and Vis.js visualisation

Talk about the implementation of the client, including wireframes etc. and how Vis.js worked.

Chapter 8

Challenges Encountered

8.1 Research Difficulties

During the initial research phase of the project several research difficulties arose.

8.1.1 Lack of Information

The initial problem faced was that when attempting to do background research on network visualisation in a browser, massive network visualisation, or a combination of both the aforementioned terms, there were limited resources available online. As a result, a lot of time was dedicated to background research and then doing an in-depth systematic evaluation of existing software that supported network visualisation.

8.1.2 Conducting a Systematic Evaluation

This was challenging due to the qualitative data included in the criteria. However, this data influenced my decision in selecting Tulip as the most appropriate software package for the next stage of the project.

8.1.3 Lack of available software

Considering that GUESS and SNAP were rejected early in the research process, this meant that there were only five software packages that met the criteria.

8.2 Time Constraints

The project could have been developed in various directions if time had allowed. This is explored in Further Ideas, in Section 10.3.

8.3 Finding out how JavaScript libraries/Python APIs are written

Time was spent researching how to write a high quality library or API. This was challenging as it was something that I had not encountered before, and there were several different parts to it that were discovered throughout the development of the library/API. These included how to catch errors, how to pass messages and structure responses etc.

8.4 Lack of resources

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.

Chapter 9

Evaluation and Testing

9.1 User Evaluation

9.1.1 Methodology

How I got users to evaluate it.

9.1.2 Results

What they said.

9.1.3 Conclusion

What this means.

9.2 Performance Testing

9.3 Testing

Chapter 10

Conclusion

10.1 Achievements

10.2 Recommendations

For SAS.

10.3 Future Ideas

TODO: Future idea could be that at some point Big Data is the end point but right now it isn't there, but I have kept it in mind i.e. uploading files next to directly into DB

10.4 Lessons Learnt

Appendices

Appendix A

Additional Information

A.1 MoSCoW Requirements

After defining the scope of the project in Section 6.1, the below MoSCoW requirements [31] 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 of the network and be visualised be 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 be 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.

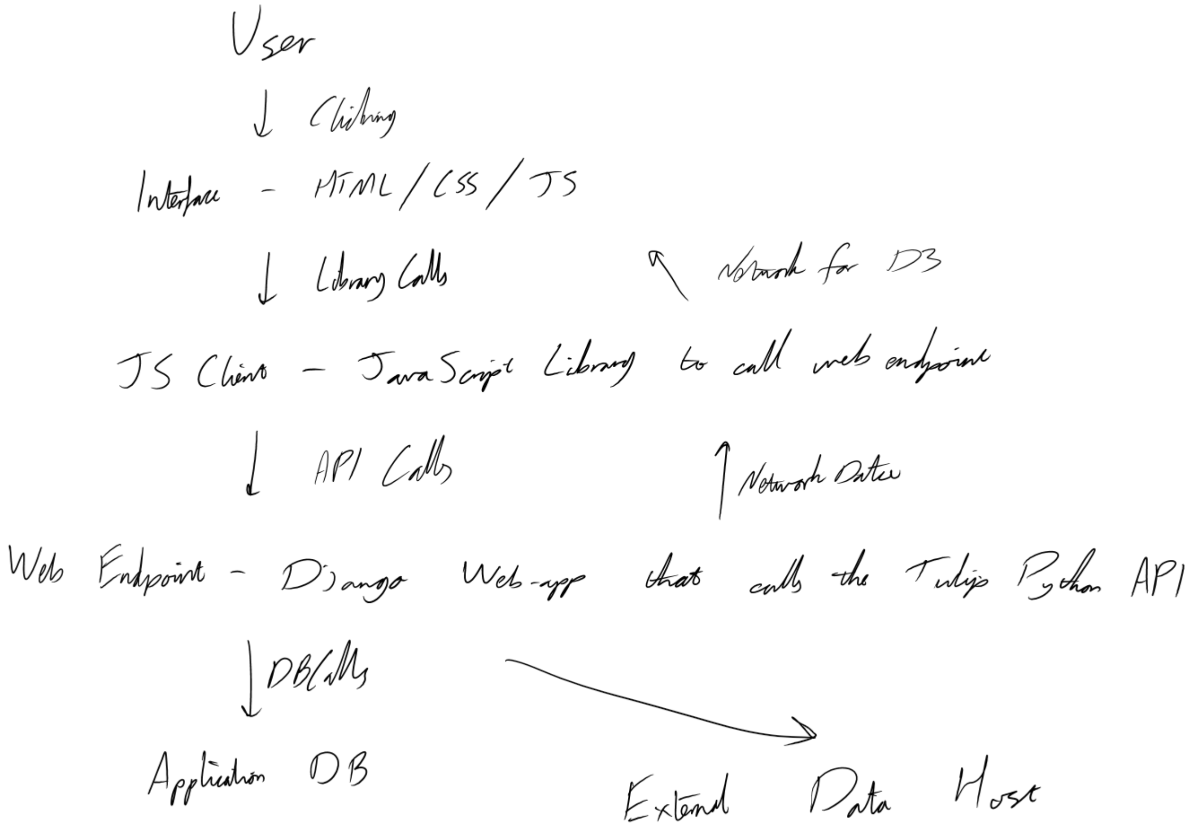

Figure A.1: Draft of Technical Infrastructure

- 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 [28]. 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.

Appendix B

Using the Software

B.1 Running the Server

How to run everything.

B.2 Using the Interface

How to run everything.

Appendix C

API Calls

C.1 Tulip Web Wrapper Python Back-end

Call x to achieve y

C.2 JavaScript Library for Tulip Web Wrapper

Call x to achieve y

Bibliography

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

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

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

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

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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