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 21, 2017

Abstract

Visualising networks with thousands of nodes and edges in a browser causes both developers and users numerous difficulties. Challenges include 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, due to its complexity, after it is rendered. After completing background research into the field, a systematic review of existing network visualisation software was conducted. This found that Tulip was the best overall network visualisation system that was reviewed, based on its performance, documentation and techniques for handling the visualisation of massive networks. Then, a network visualisation system was designed and implemented with the goal of manipulating networks in order to reduce their size and then visualise these networks. This was done by removing as much unnecessary content as possible while keeping the structure of the network intact. Finally, the system was tested and its performance was evaluated. It was found that applying node pruning followed by node bundling based on number of edges resulted in a 99.997% decrease in loading times. The performance evaluation confirmed that node bundling techniques improve performance of network visualisation systems significantly and additionally result in the network being visualised more clearly.

Acknowledgements

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

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

Contents

Chapter 1

Introduction

1.1 Context

Rendering massive networks (thousands of nodes connected by edges) in a browser is a frequent problem encountered in visualisation software that has become more prominent as both companies and regular users want to visualise a growing amount of data. As the number of nodes increase, most visualisation systems suffer an exponential performance decrease as each node in the network interacts with every other node. Additionally, for each node that is added, it is likely that multiple edges will be added to the network causing an even greater decrease in performance. Challenges involve the limits in browser performance as well as the fact that rendering information in a clear and meaningful way for the user is so difficult.

There are two main barriers to easily visualising massive amounts of data. The first is the quantity of data that needs to be visualised by a client, which requires a lot of bandwidth, processing power and RAM. This could not be expected from a standard user with a: 3GHz quad-core processor, 4GB of RAM, no dedicated GPU and 30Mbps internet connection. Secondly, once the data is finally rendered, the result is a complex mass of nodes and edges of which minimal useful information can be taken.

1.2 Aims

The intent of this project is to analyse many different software packages and outline different approaches which could be taken to help minimise the clutter of the visualisations and increase the performance of the software package that would display these large networks. A network visualisation system would then be designed, implemented and evaluated. Furthermore, this project was proposed by SAS [1] - a world leader in data visualisation - who were interested in ways to let users visualise massive amounts of information on their own machines through a web application. Hence, another goal was to be able to provide SAS with recommendations on how they could improve current systems based on performance evaluations.

A successful outcome of the project would be a system where, instead of data being transferred to the client directly from the server to be visualised, that data is analysed and modified server-side in real time, and then 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 by sending an image to the client. This would result in the data being far more useful to the user, enabling them to make informed decisions based on the network presented to them, as opposed to users not being able to gain useful information from the visualisation due to the complex and cluttered layout.

Additionally, this would reduce the processing required to render a network, resulting in lower processing power requirements, loading times and bandwidth requirements.

1.3 Achievements

This project succeeded in many important ways:

- **A thorough review of the field of network visualisation.** This included analysing the field of visualisation, and gradually got more specific, covering network visualisation, visualisation of massive networks and finally visualising massive networks in a browser. SAS also provide a strong argument for why this project is necessary. These are covered in Chapter 3. Additionally, research was conducted considering how software handles visualising massive networks, which is covered in Chapter 4.
- **A systematic review of existing network visualisation software.** This involved analysing, comparing a wide range of software. Components tested included: ease of use of the software, how the software performed under load, and what support it had for displaying visualisations of massive networks. This review can be found in Chapter 5.
- **A system was designed, implemented and evaluated with the purpose of proving how reducing the size of networks can lead to both an increase in performance and result in the information being visualised more easily.** The result was a web wrapper that encapsulated the Tulip Python API, which could be called from a JavaScript library. This would remove many nodes and edges deemed unnecessary to understanding the network as a whole by certain algorithms. This resulted in massive performance increases and let the information be visualised far more easily. For a network with one thousand nodes, loading times were decreased by 99.997% when both node pruning and node bundling based on number of edges was applied. This is covered in Chapter 6, 7 and 8.

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 have developed their own network visualisation software (for example, the SAS product 'Visual Investigator' [2]) which performs well for up to a few thousand nodes, but begins to operate more poorly when the number of nodes exceeds that - in relation to processing power, system memory and bandwidth. Thus, SAS, who explicitly support this project, are keen to find ways for larger networks to be visualised both with high performance (loading times are low and requirements on a user's machine are kept to a minimum) and high information throughput (the visualisation provides valuable information to the user).

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 data over 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 more intuitive 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]. Additionally, of what research there was, the majority of it focused on the user interface and how users interacted with the software system as opposed to improving the systems performance. Hence, it was decided that this project would focus how to increase the performance of visualisation systems.

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 metrics are met, then the visualisation software is unusable. If the system is fast but shows little useful information then the software will be of little benefit to end users in most cases. If the system is slow but displays useful information, at the end then the system is not very user friendly, but can still provide useful insights. Finally, if it is both fast and displays useful information then it is 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 gain 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 can greatly benefit those who need to understand their data, and make informed decisions (such as business decisions) based on the relationships present.

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]. This increase in quality of the visualisation is gained, however, at the cost of performance - with interactive visualisation generally requiring far more processing power.

3.2 Network Visualisation

Network visualisation is a branch of data visualisation involving the ability to display nodes and edges in a graphical and meaningful way to a user. A node can represent any object, from people to countries to laws, and edges are used to indicate 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]. (A subgroup is defined as a subset of the full graph that satisfies the four graph requirements [8] [9].) 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 term used to refer to a wide variety of network sizes, but is generally considered as anything above approximately a thousand nodes [10]. These

challenges generally present themselves in the form of computational challenges or visualisation challenges. Computational challenges are defined as when a system cannot quickly execute a given task due to lack of speed or performance from the CPU, RAM or backing storage, and is generally caused by either too much data being passed to the system, or algorithms necessary for the system to work running in exponential time [11]. If a machine has a slow processor, then the amount of time required 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" [12]. If the system RAM is not large enough to fit the network, then the performance of the network visualisation will degrade significantly due to external memory needing to be used, such as paging to disk.

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 (possibly many terabytes in size), it can be nearly impossible to store this data on one machine. Hence, storing data on the cloud is often chosen, whether internally hosted within a company or externally hosted. This has the benefit that "Cloud computing is a powerful technology to perform massive-scale and complex computing." [13], meaning that all required data for this challenge can be efficiently stored and processed.

Assuming that the machine is powerful enough in all 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 hundreds or thousands of 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, as it can be impossible to discern any relevant detail from what is displayed.

3.4 Visualisation of Massive Networks in a Browser

Visualising massive networks in a browser environment 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 [14] and hence performance is poor when working with large datasets, as it is not parallelisable. 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, consider how successful they were, and then make a system that allows for the visualisation of massive networks in a browser and across a network such as the internet.

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, due to the challenge of rendering so many nodes, interactively, using a browser (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. 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 near instant, but it soon becomes both difficult to understand and slow. The times shown in the below figures are from the initial release version of Visual Investigator, and since then performance has improved in many ways. However, 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 benefit the system greatly.

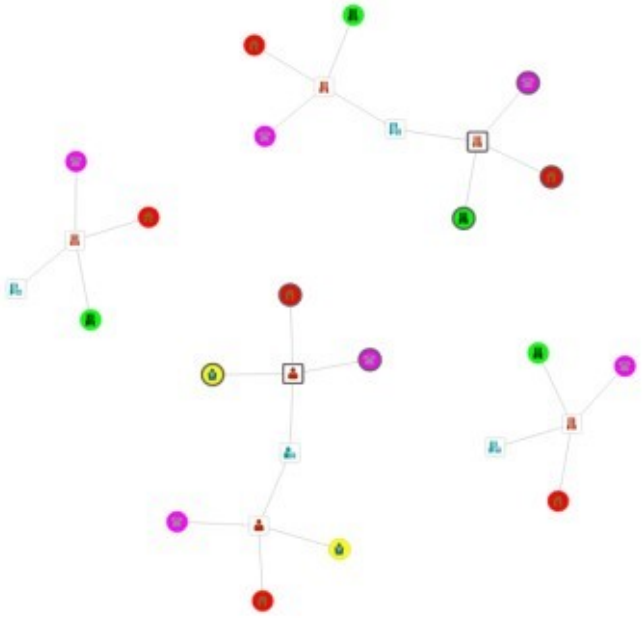

Figure 3.1: 30 nodes, Loading Time less than 1 second

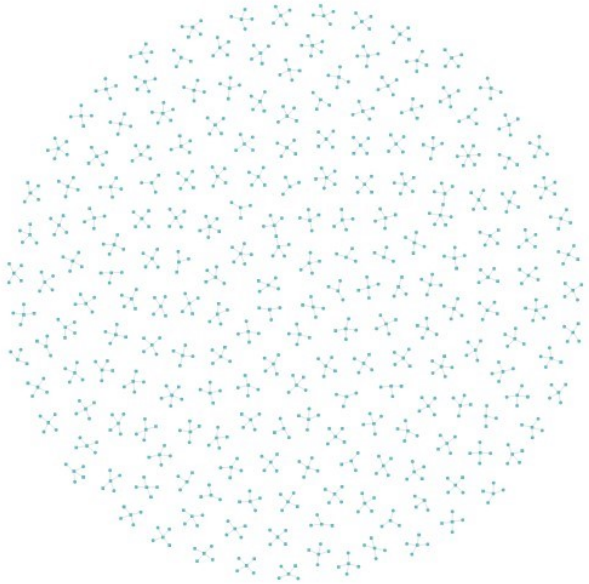

Figure 3.2: 1000 nodes, Loading Time 10 seconds

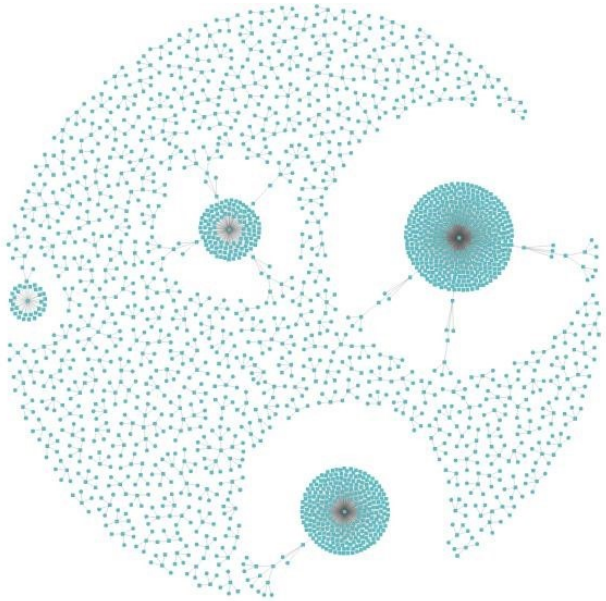

Figure 3.3: 2500 nodes, Loading Time 30 seconds

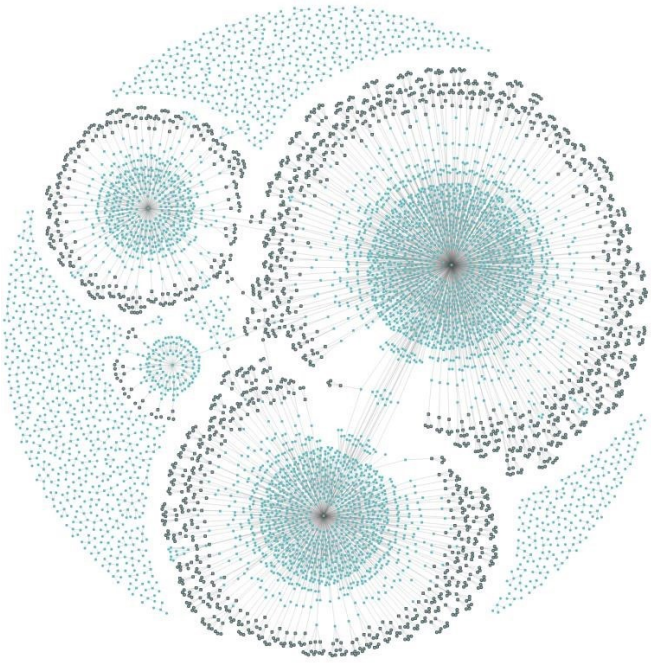

Figure 3.4: 7500 nodes, Loading Time 300 seconds

Chapter 4

Research

4.1 Background

Initial research and a systematic review of existing software (Chapter 5) 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

Node bundling is using algorithms to decide (with or without some form of user input) which nodes are similar to each other, and upon establishing this relationship, grouping them into a single 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" [15]. This allows for hundreds or thousands of nodes to be grouped into a single node in order to require less memory, processing power, bandwidth and screen space on the user's machine. A way to further improve the visualisation is to show how the nodes being bundled connected with themselves, for example if they are heavily connected, all connected to just a few nodes, or if they are in a ring formation. Figure 4.1 and 4.2 demonstrate both node and edge bundling.

Node Pruning

An example of a node bundling algorithm is 'Node Pruning'. This is a technique that takes each node in a network with only one edge and deletes it from the network. This can often reduce the number of nodes in a network by a significant amount, while retaining its overall shape. The algorithm is mentioned in literature as a step to creating a bundled network but is not named [16]. Hence, for the purpose of this dissertation, it was named 'Node Pruning'. More information about node pruning can be found in Section 7.1.4.

Node bundling based on cliques

Another example of node bundling is an algorithm based on cliques. This involves analysing each node and deciding which are closest to cliques, with a clique being defined as every node is adjacent to every other [17]. The result of this is highly interconnected groups of nodes becoming bundled into one node. More information about node bundling based on cliques can be found in Section 7.1.4.

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" [18], and "can be seen as sharpening the edge spatial density, by making it high along bundles and low elsewhere" [18]. Overdraw is defined as making edges longer than necessary [19]. If node bundling is done, then this process will have less of an impact, as all the edges will already be bundled together assuming the node bundling algorithms were optimal. This would imply all interconnected nodes have been bundled so there is only one edge traversing between subgroups as opposed to many. Like node bundling, it is often shown through the change of the size of the edge or the colour. See Figure 4.1 and Figure 4.2 for two examples of a combination of node and edge bundling.

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. This can be through hiding edges that are not connected to the highlighted nodes, or rendering 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" [22]. See Figure 4.3 for an example of system using a local edge lens.

4.2 Current Network Visualisation Software

In addition to finding out techniques used by software for large network visualisation, a list was 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 had been complimented as high quality network visualisation software - and was consequently further researched in Chapter 5.

- GUESS [23]
- SNAP [24]
- Gephi [25]
- GraphViz [26]
- Tulip [27]
- D3.js [28]
- Vis.js [29]

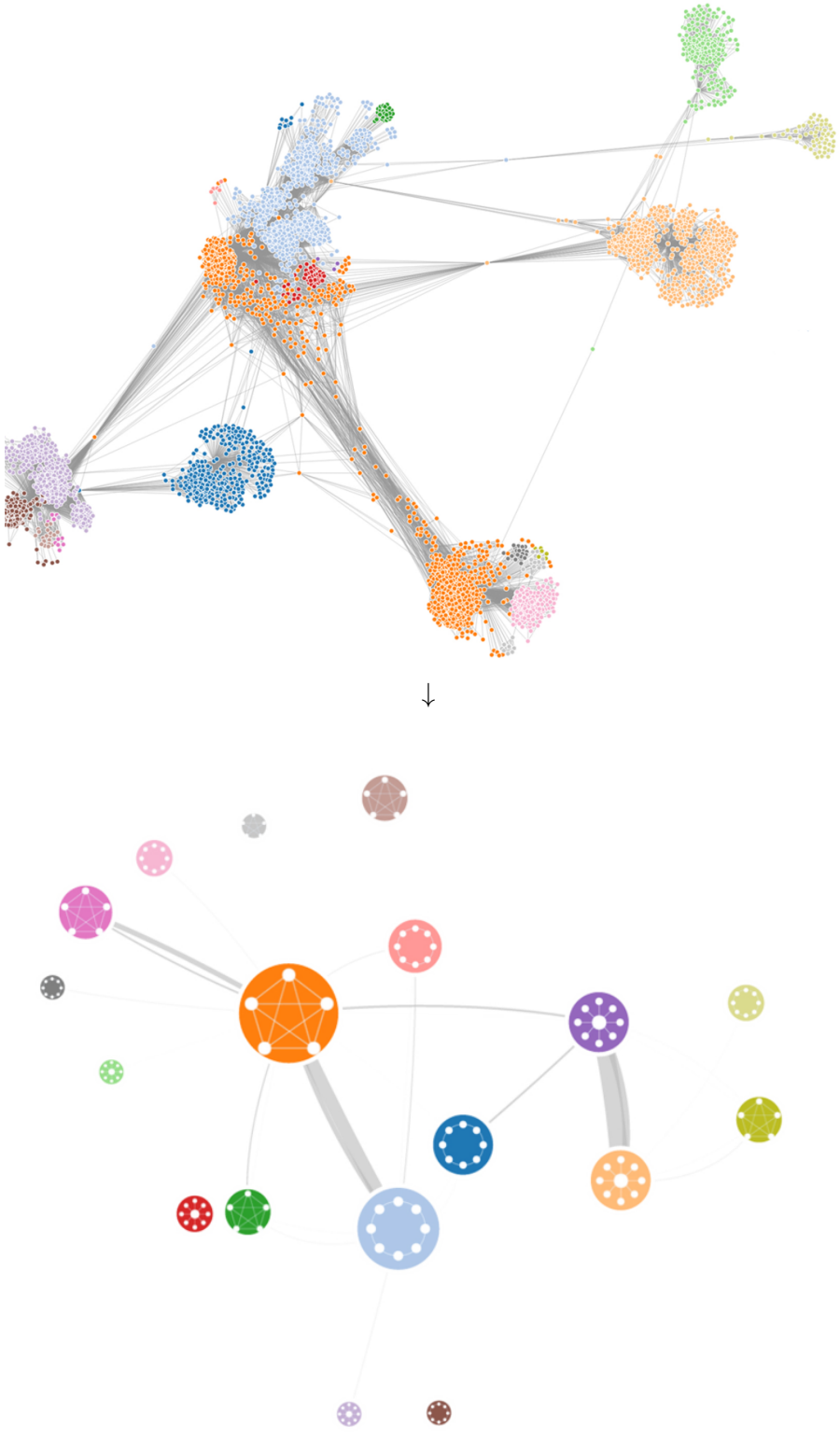

Figure 4.1: This demonstrates both node and edge bundling, where thickness of edge displays how many edges there are, and size of node indicates number of nodes group. Additionally, each node displays how nodes within it were connected. This particular example displays a comparison between a common network visualisation and a ModulGraph-based network visualisation. [20]

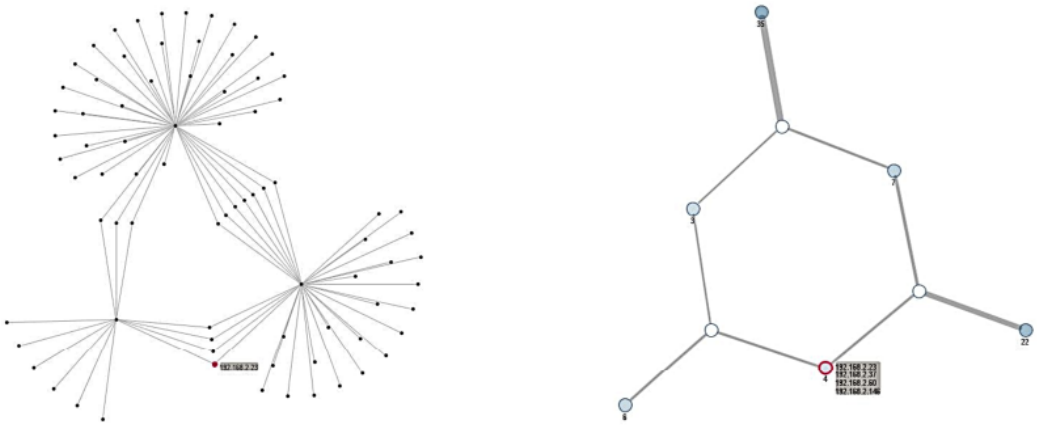

Figure 4.2: This figure shows how the bundling of many nodes mean that, although information is lost, the visualisation can become more clear as a result of the operation. [21]

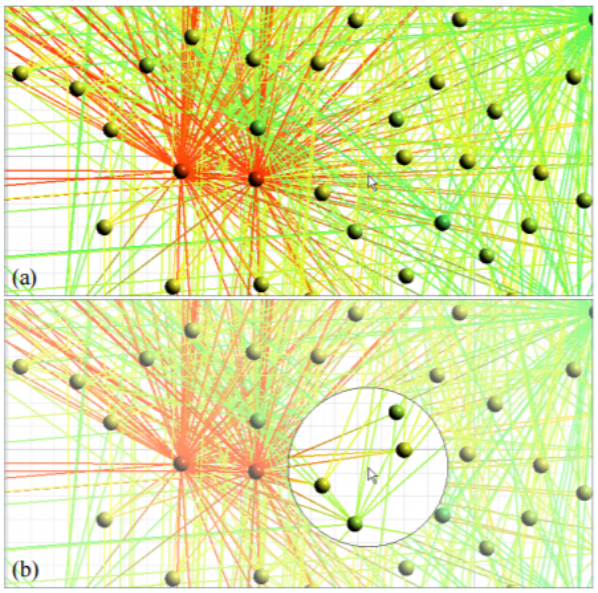

Figure 4.3: If a network is cluttered (a), then it can be hard to get useful information from it. A local edge lens (b) lets users easily identify which edges are connected to which nodes. [22]

Chapter 5

Systematic Review of Existing Software

5.1 Aim

The goal of the review was to consider several different types of network visualisation software, gathered in Chapter 4, 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 massive 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 criterion 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 feature to be analysed was 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 criterion 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 (such as Force Directed [30] - which is most common

layout algorithm - or lesser used algorithms such as spectral layout [31] or arc diagrams [32]), 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 more poorly 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 collected over a long time period over which relationships between the data evolve.

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

    (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 categories 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 aspect of the software of note that is not identified above?

Each piece of software was evaluated against this criterion, 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 stated in Section 5.2, 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 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.

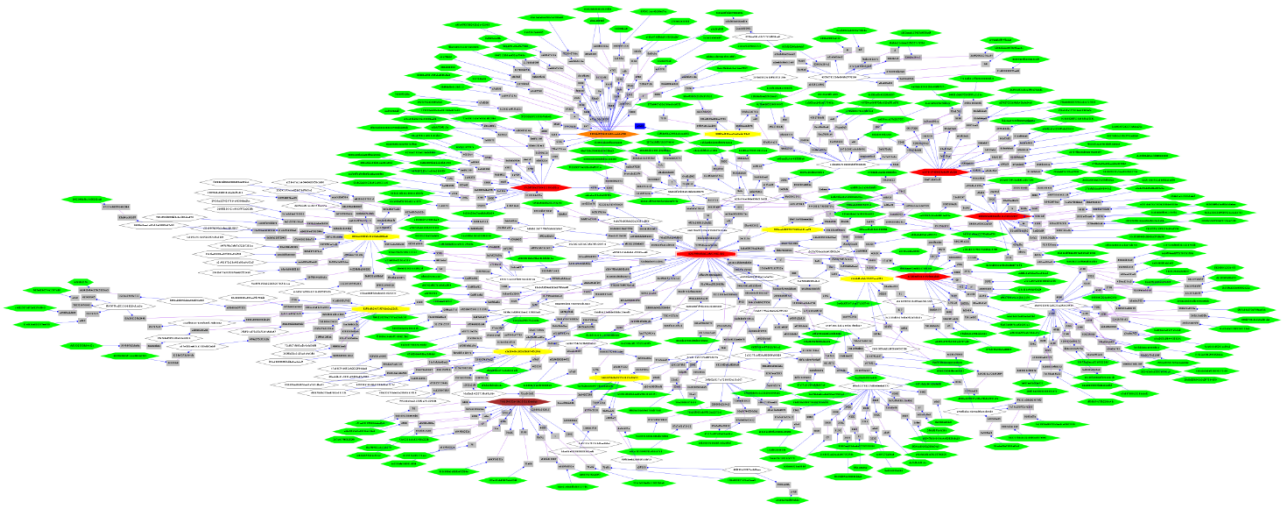

Figure 5.1: this shows how Graphviz's user interface is focused towards smaller networks of up to a thousand nodes or so, with a lot of detail being shown that becomes impossible to see quickly.

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 it was understood how the system worked and how to make simple networks. Additionally, networks could be loaded and saved 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. It is written in Java.

5.3.2 Graphviz

Graphviz, "collection of graph drawing tools" [33], was difficult to test as it is 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. It is predominantly written in C.

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. It is written in C++ and has a full Python wrapper.

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 as a result of D3.js's complexity, the documentation is more complicated than all 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. It does not perform as well as D3.js, with significantly lower rendering times as can be seen in Table 5.1.

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	<0.1	<0.1	0.2	0.6
Graphviz	<0.1	0.3	1.1	4.5
Tulip	<0.1	<0.1	<0.1	0.1
D3.js	<0.1	0.2	1.1	5.3
Vis.js	<0.1	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.

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 support several different massive network solutions.

Table 5.4 presents the data 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 in Section 5.2 was difficult as a collection of APIs was expected, which could have been easily compared by their distinguishing factors, 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 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 the other software, it allows for:

- Importing from a file (many file formats supported, including its 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, node and 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 [34]), which means that networks can easily be created by hand or converted to and from its 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 that alter massive networks by any or all of the techniques highlighted in Section 4.1. This collection of algorithms would then be analysed for what effect they had on improving visualisations and render times, and would be tested using D3.js or Vis.js.

Upon writing these algorithms, they would be tested:

- **In isolation.** What effect that function has on a network in regards to visualisation quality and render times
- **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 its 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 visualisation 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 its 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 bearing 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 to load and manipulate massive networks

A web application would be created that enable users to upload and visualise massive networks. This application would allow users to request to visualise a network and give them the choice to apply one or several bundling algorithms to the data. This would result in faster network transfers and significantly reduced rendering times, but these benefits would be at the cost of processing the network on the server.

5.7 Project Direction

It was decided that **creating a web application to load and manipulate massive networks** is how the project should progress. This would result in users being able to access visualisations of many massive networks without requiring a huge amount of computational power. This could benefit both employees of a business and business clients in many ways. Users who are required to analyse data would be able to use systems with lower specifications which would result in lower costs for companies using the service and more potential customers for companies selling the service. Most importantly, any user of the system would additionally be able to better visualise and understand massive networks.

Chapter 6

Design

6.1 Scope of the System

6.1.1 Overview

The first necessary step was to decide how the server would interact with networks. Given that Tulip was considered the best software under the criteria tested for in Section 5, a possibility would be to use the APIs that Tulip provides. Tulip provides both a C++ API [35] and Python API [36] (of which the Python API is a wrapper around the C++ API), so the web application could be made using either of the languages. As "Python is very flexible and makes experimentation easy" [37], and additionally Python has a popular web framework, Django [38], which is "a powerful Web application framework that lets you do everything rapidly from designing and developing the original application to updating its features" [39], this seemed a reasonable choice of stack to develop on. Additionally, this stack allows for the flexibility of Python and the power of C++, which makes it ideal for processing massive networks. It is worth noting that Tulip has covered web visualisations before, when its creators visualised the Firefox Dataset as part of a competition [40]. However, this was an isolated project and currently there is no supported web visualisation framework made by Tulip.

The application that was hence to be created was to use:

- 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 framework

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 the possible ability to connect to an external file host. This would enable it to be expandable to as many users as the host deemed necessary.

Django Code Structure

A typical Django project consists of a project and one or more applications. The core application contains all settings and configuration files. Each application is designed to contain an individual component of the system. This commonly consists of a set of urls, views, models and forms.

The web server for this project will consist of three applications. The core application will contain the settings and configuration, and a `urls` file that included both of the other applications views. The second application will include all functionality of the app relating to the Tulip Web Wrapper, and contain the model for a network, a form to upload a network, and code for bundling networks and converting networks to `json`. The final app would oversee the front-end of the system, containing templates for each web page and all JavaScript interfaces and libraries.

Django `urls`

A urls file routes requests from a user to the appropriate view, as well as allowing for reverse look-up to make writing templates neater [41]. The urls for the core applications will route all requests that begun with `api/` to the Tulip Web Wrapper, and all other requests to the front-end application.

Django `views`

A views file contains multiple views for a particular application. Each view gets called from a urls file after a web request is sent, and returns a web response [42]. They can return a HTML page to be rendered or a static JSON response, but also can contain logic for the application, interfacing with the database and user input. For this project, views would include showing all HTML pages, allowing users to upload and delete networks, and manipulate networks to be visualised.

Django `models` and `forms`

A models file defines the fields and behaviour of the data to be stored in the database. [43]. A forms file defines how the server accepts, processes and responds to data uploaded by a user [44]. The only model and form planned for this project would be for a network.

Web Wrapper

The Tulip Python API Web Wrapper (itself a REST API) would include several 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. A REST API is "the face of a web service, directly listening and responding to client requests" [45]. 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 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. It was decided that there would be three ways to optimise the information that was to be sent:

1. **Node pruning.** This is removing each node with one edge from the network.

2. **Node bundling based on cliques.** This is based on selecting nodes whose neighbours are highly interconnected and results on the selected node having those neighbours bundled into it.
3. **Node bundling by number of edges.** This is based on finding nodes with the most edges and bundling each node connected to it into it.

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 flexibility of the web wrapper would be prioritised over the number of features, given time constraints, although at a later date additional functions could be added with ease. Another advantage of creating a REST 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 clear 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 significant increase in performance benefits as the amount of data gets higher.

Hosting Data

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 [46], 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.

JavaScript Library

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 many 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.

Uploading Files

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).

Visualisation Framework

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.

Interface

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.

Wireframes were not created for the project as the process for development was completed in an iterative and agile style. This resulted in the system evolving throughout development as opposed to spending a block of time designing the interface at the start. Additionally, the user interface was not a primary focus of the project, thus major time was not spent on the design of the user interface.

After defining the scope of the project, MoSCoW requirements (a prioritisation technique based on selecting what a project Must Have, Should Have, Could Have and Won't Have) were created that can be read in Appendix A.1.

6.2 Design Decisions

Upon the scope of the system being decided, design decisions had to be made. This entailed deciding and justifying what features would be included as part of the system, and why.

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 terabytes 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, but mean that the visualisation was entirely static and there would be no way that the user could interact with it outside the server creating and returning another image. It was decided this was not critical functionality and hence it was not added to the system during this 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, since the JavaScript library would be kept entirely separate from the interface code.

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, based on the design decisions made in Section 6.2.

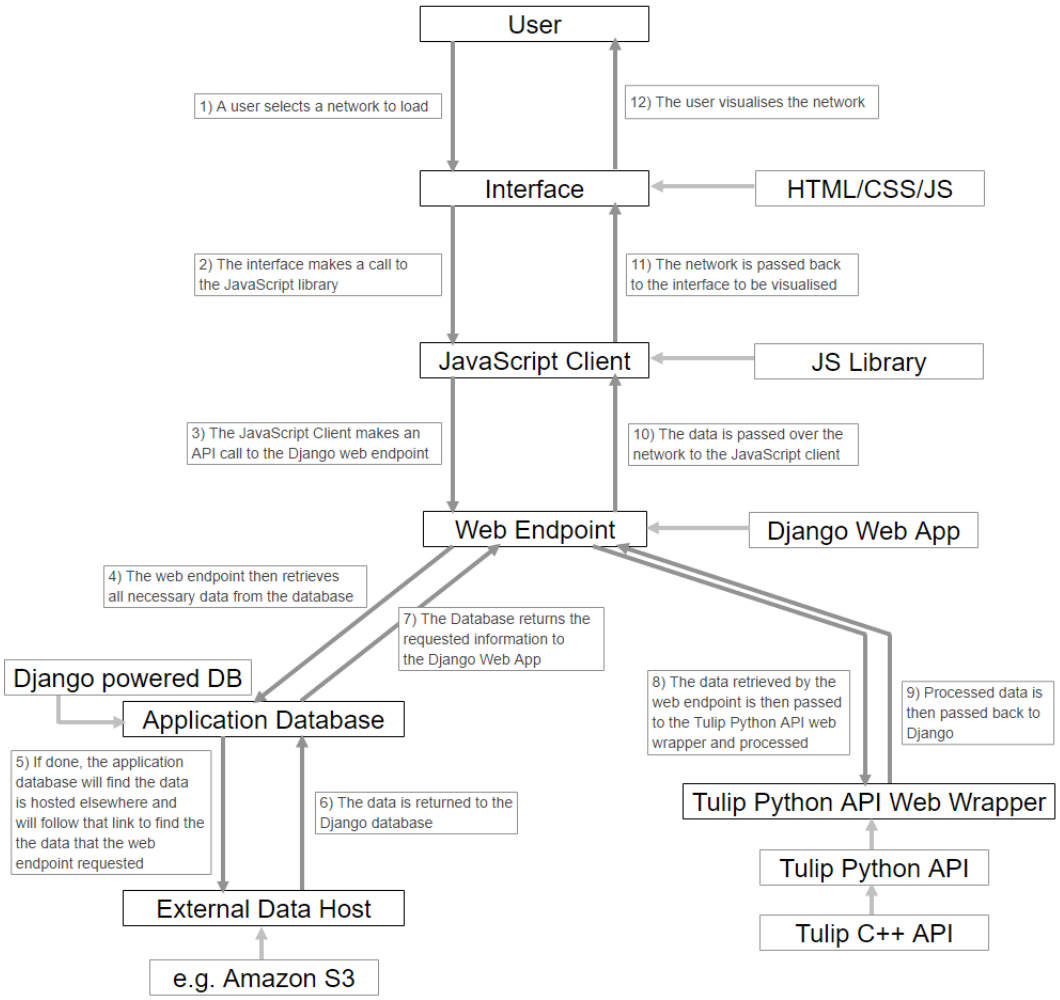

Figure 6.1: The Technical Infrastructure of the whole system, based off the sketch in Appendix A.2

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 how 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.
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 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, Django then calls a function from the Tulip Python API Web Wrapper and passes it the network. Tulip then loads in the network and then possibly reduces the network using any or all of: node pruning, node bundling based on cliques, or node bundling based on number of edges. Finally, it converts the network into a JsonResponse to be returned.
9. After the network has been processed and minimised, it is then passed back to the web endpoint 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 core components:

- 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 [47] that can be installed using pip [48]. 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 Handling errors on the server

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 opposed 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 Transferring networks 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 and recreating the network fully 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 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 (using computational power). Additionally, it would be possible to send the information over the network as a byte-array - which would be even more compact than JSON but sacrifice usability.

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, along with ensuring the system could easily be used by further developers. 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 Storing 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.

7.1.4 Bundling a network

It had been decided that the three algorithms that were to be available to apply were node pruning, node bundling based on cliques and node bundling based on number of edges. Details of how these algorithms worked can be found below.

Node Pruning

Node pruning involves finding each node with only one edge and removing it from the network. Additionally, it is possible to include information about how many nodes have been bundled into each remaining node. This means less information is lost as a user can clearly see how many nodes with one edge were connected to each node that has been pruned. The algorithm for node pruning is outlined in Algorithm 1.

Algorithm 1 Defines the process for node pruning

```
G = (V,E) {a graph containing vertices and edges}
initialise list toDelete
for all v in V do
  if v has only 1 edge then
    add v to toDelete
  end if
end for
for all v in toDelete do
  delete v from V and related edges from E
end for
```

Node bundling based on cliques

This is calculated by using the `clusteringCoefficient` algorithm provided by Tulip [49]. Each node is assigned a value between 0 and 1, based on how close to a clique its neighbours are. For each node with a value near 1, an operation is done to bundle that node, shown below. It was decided that a `clusteringCoefficient` of greater than 0.8 was required for nodes to be bundled after several networks were tested against and less than that resulted in too much information being lost. The algorithm for node bundling based on cliques presented in Algorithm 2.

Algorithm 2 Defines the process for node bundling based on cliques

```
assign each node a clustering coefficient {using Tulip's tlp.clusteringCoefficient}
G = (V,E) {a graph containing vertices and edges}
initialise list toBundle
for all v in V do
  if clustering coefficient of v is greater than 0.8 then
    add v to toBundle
  end if
end for
for all v in toBundle do
  bundle v {The code to bundle can be seen in Algorithm 4}
end for
```

Node bundling based on number of edges

This is done by initially calculating the mean number of edges in the network. Then, any node that has greater than two times the mean number of edges is bundled, as explained below. Two times the mean was chosen after many sample networks were looked at based on data provided by SAS and this number seemed to be optimal in reducing the amount of data that needed to be passed to the client. Lowering the number resulted in too much data resulted in the visualisation loses meaning and increasing it meant there was little or no processing done in most cases. The algorithm for node bundling based on number of edges is outlined in Algorithm 3.

Algorithm 3 Defines the process for node bundling based on number of edges

```
G = (V,E) {a graph containing vertices and edges}
initialise integer edgeCounter
for all v in V do
  add the degree of v to edgeCounter
end for
meanNumberOfEdges = edgeCounter / size of V
initialise list toBundle
for all v in V do
  if degree of v greater than meanNumberOfEdges * 2 then
    add v to toBundle
  end if
end for
while toBundle is not empty do
  initialise vertex highestDegree
  for all v in toBundle do
    if degree of v is greater than degree of highestDegree then
      highestDegree = v
    end if
  end for
  bundle highestDegree {The code to bundle can be seen in Algorithm 4}
  remove highestDegree from toBundle
end while
```

Bundling a Node

Upon discovering what nodes need to be bundled, the node then needs to be actually bundled. This involves locating all of the nodes neighbours, getting all of their edges, redirecting all of those edges to the node that is being bundled, and finally deleting each neighbour node. Algorithm 4 shows this process.

Algorithm 4 Defines the process for bundling a node

```
toBeBundled {a node passed into this function to be bundled}
initialise list neighbours
for all neighbour of toBeBundled do
  add neighbour to neighbours
  for all edge of neighbour do
    if edge not connected to toBeBundled then
      disconnect edge from neighbour
      connect edge to toBeBundled
    else
      delete edge
    end if
  end for
end for
for all node in neighbours do
  delete node
end for
```

7.2 JavaScript Library

The JavaScript library involved creating a function to match each endpoint 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

```
1 // graphToLoad = the name of the network to be loaded
2 tulipWebApi.loadGraph(graphToLoad, function(result) {
3     networkCreator.drawSimpleGraph(result);
4 }, function(error) {
5     console.error(error)
6 });
7
```

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.

```
1  loadGraph: function(fileName, callback, errorCallback) {
2      $.ajax({
3          url: '/api/loadGraph',
4          data: {
5              'network_name': fileName,
6          },
7          cache: false,
8          type: 'GET',
9          success: function(result) {
10             if (result.success) {
11                 callback(result.data);
12             } else {
13                 if (typeof errorCallback === 'function') {
14                     errorCallback(result.message);
15                 }
16             }
17         },
18         error: function() {
19             if (typeof errorCallback === 'function') {
20                 errorCallback('Unknown server error.');
21             }
22         }
23     });
24 }
25
```

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 information passed to it by the JavaScript library, then converting all of the nodes and edges into two arrays, along with stylising them appropriately based on if they had been bundled or pruned, and then initialising the network. This involved specifying the container where the network would be rendered, the data it would contain and any optional parameters. A summary of this can be seen in Listing 7.3.

```
1  drawSimpleGraph: function(result) {
2
3      arrayOfNodes = [];
4      arrayOfEdges = [];
5
6      for (var i = 0; i < result.nodes.length; i++) {
```

```
7           arrayOfNodes.push({id: i, label: i});
8           result.nodes[i].outEdges.forEach(function(element) {
9               arrayOfEdges.push({from: i, to: element})
10          });
11      }
12
13      var nodes = new vis.DataSet(arrayOfNodes);
14      var edges = new vis.DataSet(arrayOfEdges);
15
16      var container = document.getElementById('main_network');
17
18      // provide the data in the required vis.js format
19      var data = {
20          nodes: nodes,
21          edges: edges
22      };
23
24      var options = { /* Setting many layout and physics parameters */ }
25
26      // initialise the network
27      var network = new vis.Network(container, data, options);
28  }
```

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

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 [50]. Several options were experimented with for this project, details of which can be seen below.

7.3.1 Layout

Within the `layout` [51] option, if `improvedLayout` was set to `false`, then Kamada Kawai Algorithm [52] is *not* applied, which reduces stability of the network but improved rendering times. This parameter was toggled multiple times throughout testing and was found to rarely be beneficial to the visualisation as a whole when disabled, despite decreased rendering times. Thus, it was decided to set it to `true` for this system.

7.3.2 Physics

In the `physics` [53] option, there were several different parameters which could be finely tuned. The first of these was `stabilisation`, which selected if the network should be stabilised or not, and if so, by what means. It was decided that it would be enabled, as disabling it caused the network to move too erratically on load - despite loading far sooner.

Additionally, the maximum and minimum speed could be set, the time between steps could be changed, and the model for the physics solver could be selected. This could be one of four preset solvers: `barnesHut` [54], `forceAtlas2Based` (based on the Force Atlas 2 algorithm [55]), `hierarchicalRepulsion`, or `repulsion`. Alternatively, a custom model could 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, the custom solvers tended to perform more poorly than the preset solvers, and the default option worked the best in most scenarios.

Additionally, it was possible to disable the Physics module entirely and the network would be displayed in a

ring, with nodes equally spaced and edges placed appropriately. This lead to near instant loading times but the visualisations becoming meaningless, so the module remained enabled.

7.3.3 Nodes

The `nodes` option allowed for each node to be customised. Options included the nodes: size, colour, shape, font, label, border and mass. These could be set for all nodes in the network or individually - and these options were used extensively to illustrate to a user what nodes had been bundled how much bundling had occurred.

7.3.4 Interaction

It was also possible to edit how users could interact with the visualisation through the `interaction` option. This allowed for: users to select multiple nodes, nodes to be locked into place or free for the user to move, nodes to show tooltips which could contain additional information, and the ability to zoom in or out. Zooming on the network and interaction with nodes was enabled so users could get more detailed information on the network and how it reacted to different nodes being moved. Future iterations of the project could include using tooltips to give more information about each node, depending on what was being visualised.

7.4 Front-end

7.4.1 Architecture

The front-end of the system was written using HTML, CSS and JavaScript. The HTML was written in Django Templates [56], and a base file was written which acted as a container for each of the other HTML files. A CSS framework, Bootstrap [57] was utilised in order to allow for a smart interface to be created very quickly, and ensure that development time was not spent on creating a user interface from scratch. The JavaScript for the system used jQuery [58], which made interacting with the DOM a very easy process. JavaScript was user to pick up events based on user interactions and make appropriate calls to the JavaScript library detailed in Section 7.2.

7.4.2 System Interaction

To use the finished system, a user would simply select a network to visualise, decide which algorithms would be applied to the network using check boxes, and then select ‘Load’. This can be seen in Figure 7.1.

Figure 7.1: This shows how a user would interact with the system to view a network

The two other key components of the front-end were uploading and deleting networks. Uploading a network involved: naming the network and selecting either a `.tlp` or a `.json` (in SAS’s format) file. A `.tlp` could be loaded directly into the system and would be saved directly. However, the `.json` file would have to be converted to a `.tlp` so Tulip could understand it. This involved uploading the `.json` file, creating a `.tlp` file the same name but a different file extension, parsing the `.json` file, and then copying the relevant information into the

blank `.tlp` file. Deleting a network was done by selecting the network to be deleted from a drop-down list, and clicking 'Delete'.

7.4.3 Finished Product Screenshots

Figure 7.2 and 7.3 show the final product rendering a network of 1000 nodes, and then rendering the same network after it had been Node Pruned and Node Bundled based on number of edges. These both show how Vis.js visualise networks and what the system looks like, and also show the benefit of processing networks on the server. Figure 7.2 took 59924ms to load and Figure 7.3 took 153ms to load, further highlighting the benefits of the system. All screenshots can be seen in Appendix C.

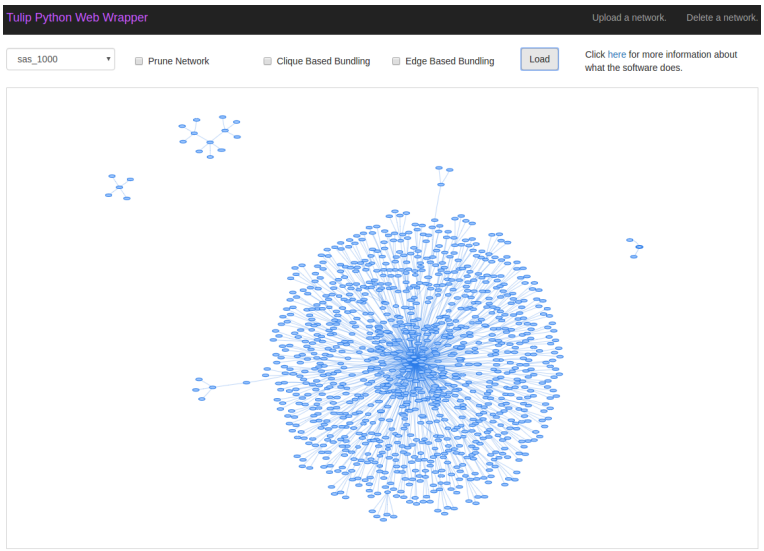

Figure 7.2: This shows the system visualising a network with 1000 nodes without any processing taking place.

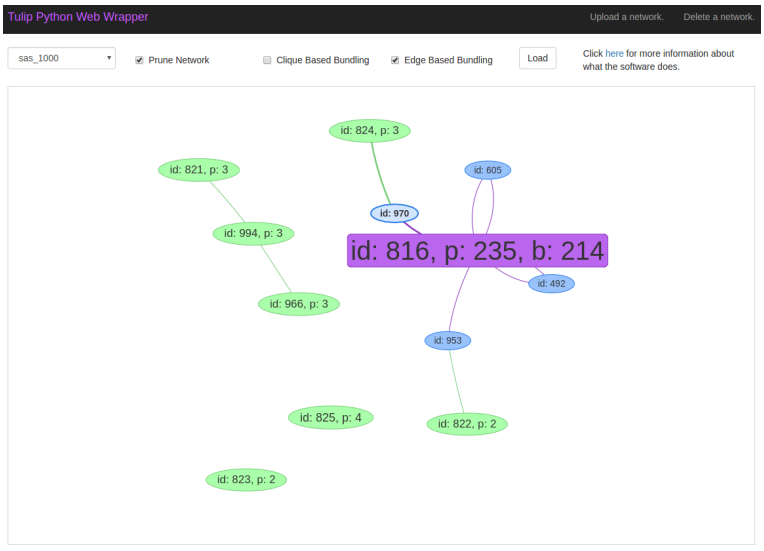

Figure 7.3: This shows the same network as seen in Figure 7.2, but with node pruning and then node bundling based on number of edges applied to it.

Chapter 8

Testing and Evaluation

8.1 Testing

8.1.1 Unit Testing

Unit tests have the purpose of testing each individual component of the application for expected behaviour [59]. Unit testing was conducted on the two main components of the Tulip Web Wrapper, along with on the Django `views` responsible for returning the HTML to the client.

Conversion from `.tlp` to `.json`

To confirm that the conversion from a Tulip network to a JSON object was successful, tests were created for several different networks. The test set up included loading three network from a file into memory using Tulip's `tlp.loadGraph()`. Then, on each of the networks, the conversion code was run and it was confirmed that the number of nodes and edges was correct, with the function `TlpJsonConverter.tlp_to_json()`.

Confirming that each bundling algorithms work

To test each algorithm initially three different networks were loaded into the system using the Tulip function `tlp.loadGraph()`. One of these networks would be used to test node bundling based on cliques, since it was a very interconnected network. The other two networks would be used to test node pruning and node bundling based on number of edges, being far more sparsely connected networks. Each test consisted of checking the number of nodes upon loading in the network, applying the appropriate operation that was being tested for, and then confirming that the number of nodes and edges returned from the function was as expected.

Successful page responses

The unit tests, which confirmed that both the part of the back-end that returned data to the front-end, worked by creating a fake call to the server using a GET or POST request as expected and that the Tulip Web Wrapper worked as appropriate. This was done by checking that the calls were successful and either returned a status code of 200 if a `HttpResponse` was returned, or included success = True if a `JsonResponse` was returned.

8.1.2 Integration Testing

Integration tests serve the purpose of testing that each component works predictably when combined with all other components it should work with [60]. For this project, they serve the purpose of confirming that the Tulip Web Wrapper worked as expected, and that it interacted with the rest of the back-end correctly.

The integration tests for the Tulip Web Wrapper involved creating a network in the database during the set-up phase of the test, and then loading in that network using different algorithms and ensuring the output at the end was correct. This entailed calling the `loadGraph` API endpoint using JavaScript AJAX [61] and passing in appropriate parameters such as the network name, and whether node pruning or node bundling using number of edges was enabled. The response of that call (a JSON object) was then decoded and it was checked if the call had been successful, and if it had contained the correct number of nodes and edges, along with the expected number of nodes that had been pruned of bundled. This ensured that the logic within the view was correct, that the bundling code worked as anticipated, and that the conversion worked correctly.

8.2 User Evaluation

As stated in Chapter 2, during background research for the project, it was found that a large proportion of literature existing on massive network visualisation focused on the user interface design and quality of interaction between a user and the system. It was hence decided that the aim of this project would be to create a system that utilised high performance network manipulation software to allow users to visualise massive networks within a browser, on a low powered client. This was completed by applying one or more algorithms to the network in order to: reduce its node and edge count, speed up render times and improve network transfer rates. An additional goal was to ensure that the Tulip Python API Web Endpoint and the JavaScript library were as flexible as possible, to ensure the easy development of future interfaces.

The interface made for this project was kept simple to allow for maximal development time to be spent optimising and testing the web wrapper (it can be seen in Figure 7.1). Additionally, the JavaScript network visualisation framework used (Vis.js) was chosen due to its simplicity, and it is known to be less flexible than other choices, such as D3.js - and hence if interface design had been a priority then another framework would have been chosen.

Given all the above facts, it was decided that a user evaluation was not appropriate for the project, as significant development time was not spent on the interface. The focus of the project was to make the server and client as high performing as possible. The performance is evaluated in Section 8.3.

8.3 Performance Evaluation

8.3.1 Aim

The goal of the performance evaluation was to thoroughly test the system that had been created. This would involve loading several different networks into the system, and then recording the performance of the system when different algorithms are used and comparing the results, providing a quantitative evaluation.

8.3.2 Methodology

As SAS had provided four different networks of varying sizes, these would be used evaluate the system. The networks had approximately thirty, two hundred, one thousand and three thousand nodes respectively, and each had approximately the same number of edges as nodes. Each of these networks would first be uploaded into the system and then rendered with the following parameters: no algorithm set, node pruning, node bundling based on number of edges and both node pruning then node bundling based on number of edges.

It is worth noting that node bundling based on cliques was not tested as it had no effect on the data provided by SAS due to the tree-like structure of the data.

The runs were conducted on the same machine, one by one, while no other application was running apart from the Django development server which hosted the site. Each run of the evaluation was done five times so a reliable average could be taken. For each run, the following information was collected:

- Client-Side:
  - **The time taken between the user clicking 'Load' and the client getting an AJAX response back.** This was recorded to discover how long the JavaScript took to respond, call the server using AJAX, and get a response - essentially recording the time spend processing the network and sending it across the network.
  - **The time spent processing the data got from the AJAX call and formatting it for Vis.js.** This was recorded in order to find out how long it took for JavaScript to process the data and format it appropriately for Vis.js, which included adding all of the nodes and edges to a list and then setting properties (such as size, shape and colour) appropriately.
  - **The time spent creating the Vis.js network.** How long it took to create the network in memory from the list of nodes and edges, and any `options` parameters set.
  - **The time spent rendering the network.** How long it took for the visualisation to display. This involves Vis.js creating all of the nodes, positioning them in a circle, and then applying the selected physics model and waiting for the network to settle before it is displayed.
- Server-Side:
  - **The time taken to load in the requested `.tlp` file from the database into Tulip.** This represents how long it took for the network to be located in the database, loaded in to memory and then loaded in to Tulip as a network object.
  - **How long pruning the network took.** If applicable, the length of time taken to loop over the network and prune it as appropriate, and return both the modified network and a list containing each node that has been pruned as well as noting how many nodes have been removed from it.
  - **How long node bundling based on the number of edges took.** If applicable, the time taken to decide which nodes should be bundled and to then bundle them, whilst tracking how many nodes have been removed from each bundled node.
- Network Information:
  - **The number of nodes in the network.**
  - **The number of edges in the network.**
  - **How many bytes the packet contains that is to be sent across the network.**
- Total Time:
  - **The total time taken from the user clicking 'Load' to the network rendering.**
  - **The standard deviation of the total time.**

8.3.3 Results and Discussion

The following four tables show the effect of different algorithms on different networks. Each table shows the total time taken to load, manipulate and render the network, and how much data was sent. Table 8.1 shows the results for a network with approximately thirty nodes, Table 8.2 shows the results for approximately two hundred nodes, Table 8.3 shows the results for approximately one thousand nodes and Table 8.4 shows the results for approximately three thousand nodes.

Each percentage in the following sections is created by comparing the time or packet size of the stated algorithm(s) - which are either node pruning, node bundling based on number of edges, or both - against not applying any algorithm.

30 Nodes

For a network with approximately 30 nodes, node pruning resulted in a 50% decrease in total time to visualise, and 36% decrease in packet size. node bundling based on the number of edges resulted in a 24% decrease in time to generate and render, and 36% decrease in packet size. Applying both node pruning and node bundling based on number of edges resulted in a 63% decrease in amount of time to visualise and a 40% decrease in packet size. Node pruning is clearly more effective than node bundling, providing lower packet sizes and faster network loading times.

	None	**Pruning**	**Edge-based Bundling**	**Both**
Total Time (ms)	170.8 ± 15.3	85.8 ± 20.5	129.6 ± 19.3	63.2 ± 15.2
Packet Size (bytes)	1300	827	1012	785

Table 8.1: Total Times to load a network of 31 nodes and 26 edges, and the size of the packet transferred over the network

200 Nodes

For approximately 200 nodes, node pruning provides a 91% decrease in total load time and a 49% decrease in packet size. For node bundling based on number of edges, the decrease in load time was 9% and the packet size decreased by 20%. Applying node pruning followed by node bundling based on number of nodes resulted in a 98% decrease in loading times and a 70% decrease in packet size. node pruning is again more effective at lowering the amount of data that needs to be transferred and lowering the amount of time to manipulate and render the network.

	None	**Pruning**	**Edge-based Bundling**	**Both**
Total Time (ms)	7382.8 ± 2010.2	663.0 ± 187.0	6749.0 ± 1075.0	146.6 ± 14.5
Packet Size (bytes)	7600	3900	6100	2300

Table 8.2: Total Times to load a network of 199 nodes and 199 edges, and the size of the packet transferred over the network

1000 Nodes

When the network with approximately 1000 nodes was used, similar results to above were seen. There was an 84% decrease in loading times and 51% decrease in packet size for node pruning, and a 36% decrease in loading times and a 29% decrease in packet size. When applying both node pruning and then node bundling based on the number of edges, the decrease in loading times was 99.997% and the packet size was reduced by 70%. Again, node pruning was shown to be more effective than node bundling based on number of edges, but additionally there was a massive decrease in loading times when applying both algorithms. It is hypothesised that the reason such an increase in performance was seen was due to the structure of the network lending itself well to being Node Pruned and then Node Bundled based on number of edges, and hence there was a massive reduction in nodes and edges to be rendered.

	None	**Pruning**	**Edge-based Bundling**	**Both**
Total Time (ms)	59924.4 ± 8343.5	9417.8 ± 2223.1	38139.8 ± 9383.0	152.6 ± 30.5
Packet Size (bytes)	41400	20300	29400	12400

Table 8.3: Total Times to load a network of 1089 nodes and 1087 edges, and the size of the packet transferred over the network

3000 Nodes

For the largest network, with approximately 3000 nodes, the following results were obtained. Loading times decreased by 81% and packet size decreased by 44% for node pruning. For node bundling based on number of edges, a decrease of 48% was seen for loading times and a 27% decrease was seen for packet size. When applying node pruning followed by node bundling based on number of edges the decrease in total time was 94%, and the decrease of packet size was 72%. Similar to above, a performance increase was most notable for node pruning.

	None	**Pruning**	**Edge-based Bundling**	**Both**
Total Time (ms)	252849.2 ± 32862.0	48956.8 ± 3937.0	131606.6 ± 6031.3	14532.8 ± 3341.0
Packet Size (bytes)	114000	63500	83500	42900

Table 8.4: Total Times to load a network of 2849 nodes and 2835 edges, and the size of the packet transferred over the network

Figure 8.1 shows the total load times for each of the different networks and each of the different algorithms. It can clearly be seen that node bundling based on number of edges is less effective than node pruning for all network sizes, and applying both algorithms is more effective. It is interesting to note that for different networks the amount of benefit varies. It is hypothesised that this is due to the structure of the network, with networks containing a lot of nodes with a single edge benefiting greatly from node pruning, and networks with single nodes connected to many other nodes benefiting when node bundling based on number of edges. It is worth noting that the scale on the y-axis is logarithmic.

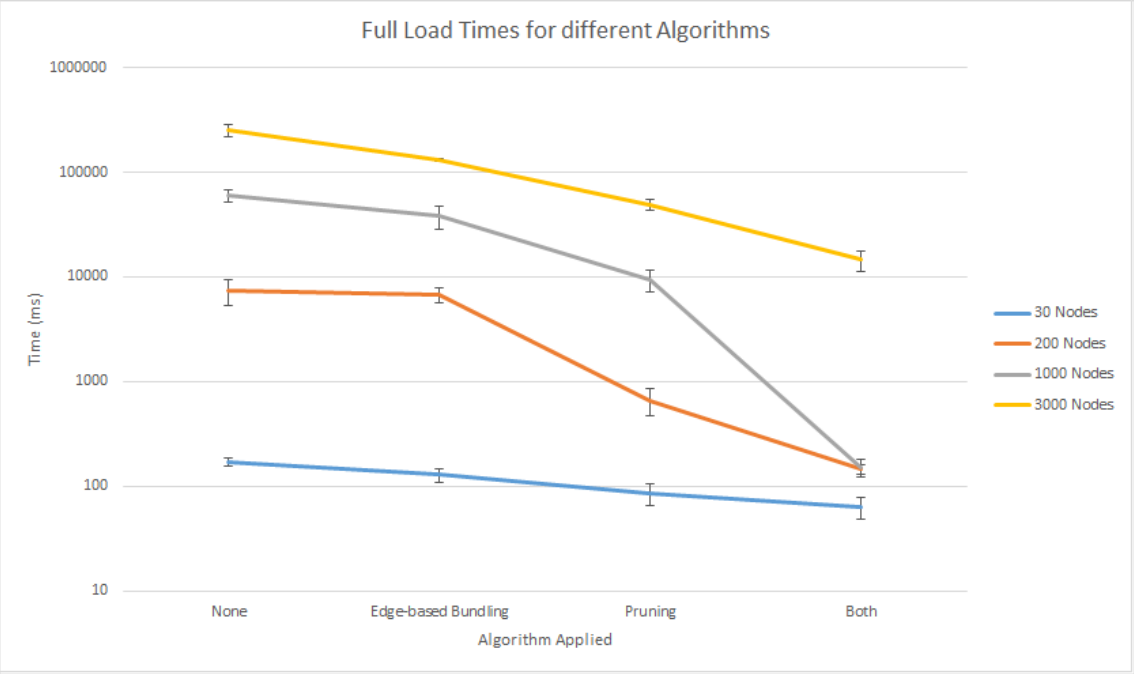

Figure 8.1: Load Times

Figure 8.2 shows a steady decrease in packet size from no algorithm being applied, to node bundling based on number of edges, to node pruning, to applying node pruning then node bundling based on number of edges. It is again worth noting that the y-axis has a logarithmic scale.

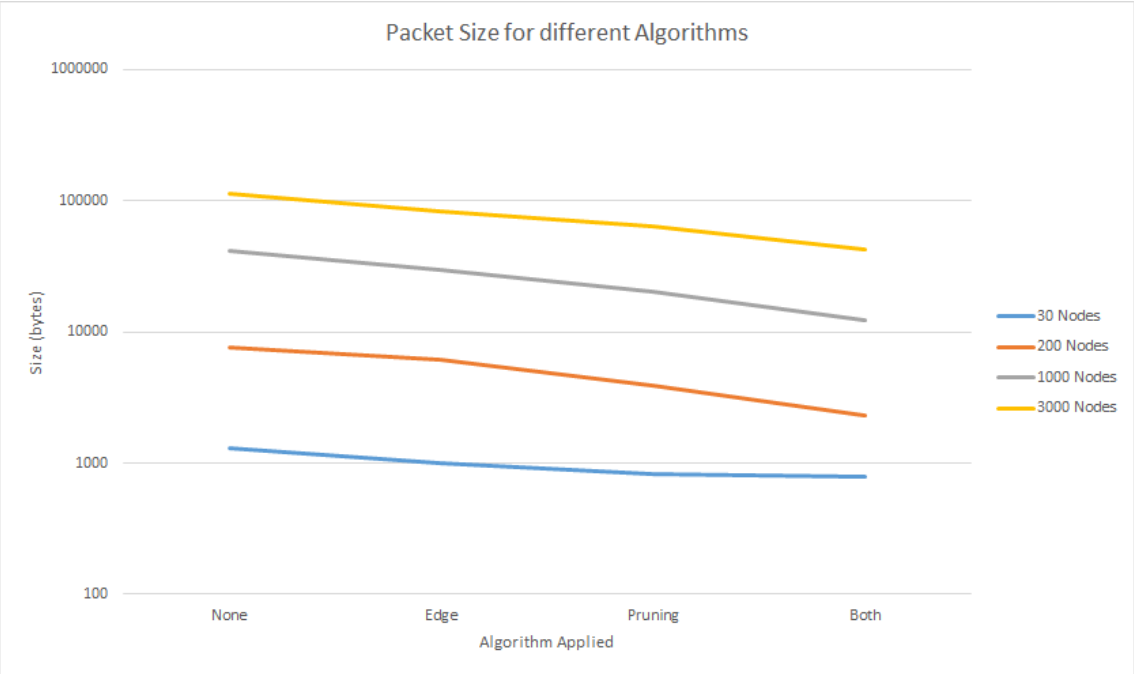

Figure 8.2: Packet Size

The above results show the benefits of applying different algorithms to reduce the amount of data that is to be visualised. This is in regards to both the amount of data that needs to be transferred, and how long it takes to load a visualisation - from a user clicking ‘Load’ to the network rendering.

For approximately 30 nodes, the amount of time saved when node bundling based on edges was 41ms, and for approximately 3000 nodes the time saved was 120759ms (just over 2 minutes). The amount of time spent processing the networks on the server in order to get the stated time savings was 0.36ms for 30 nodes and 94ms for 3000 nodes. For node pruning, the time spent pruning on the server for 30 nodes was 0.38ms and resulted in an 84.8ms speed up, and for 3000 nodes the time spent pruning on the server was 169ms and the time saved in total was 203010ms (just over 3 minutes and 20 seconds). Finally, when applying both algorithms, there was 0.42ms spent processing the information on the server for 30 nodes, with a total time saving of 107ms, and 192ms applying the algorithms on 3000 nodes, with a time saving of 237177ms (just under 4 minutes). These results clearly show the benefit of applying the bundling algorithms.

It is worth noting that, despite the fact that performance increased greatly when applying the above algorithms, this is solely due to the fact that less nodes were shown. There are attempts to reduce the amount of data lost, both in removing the programmatically decided least critical nodes to the visualisation, and showing where nodes have been bundled and how many nodes have been bundled.

The full set of results can be seen in Appendix B.

Chapter 9

Conclusion

9.1 Summary

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 manipulate networks in a high-performance way and then visualise them within a browser.

The first half of the project included a review of the field, research on network visualisation and a systematic review. Initially, a review of the field of visualisation, covered in Chapter 3, and an exploration of existing solutions to visualisation massive networks, covered in Chapter 4, was done. This resulted in a solid foundation of knowledge to build on and uncovered techniques such as edge bundling, local edge lenses and node bundling - which was explored later on the project.

The next phase was conducting a systematic review of several network visualisation packages found throughout the research phase of the project, which is covered in Chapter 5. This involved creating a set of criteria and defining a strict process for how the software should be reviewed, and resulted in Tulip being declared as the most powerful network visualisation tool. Additionally, this review provided knowledge about how software systems visualise and handle massive networks.

The second half of the project involved designing, implementing, testing evaluating a web application which was to manipulate networks on a server and then send the reduced network to a client to be visualised. The design stage involved confirming the scope of the project, making several design decisions and then designing a technical infrastructure based on the design decisions. This can be read in Chapter 6.

Implementation mostly included actually creating the web application, which is covered in Chapter 7. Several difficult decisions had to be made throughout implementation which are discussed throughout the chapter, and several algorithms were written to optimise and reduce the size of networks in order to make them more easily visualised, and to make load times quicker. Both the Tulip Web Wrapper and JavaScript were designed as APIs so developers could easily interface with them. Finally, the user interface and network visualisation was created, which involved using the CSS framework Bootstrap, and network visualisation framework Vis.js.

The final part of the project, testing and evaluation, was covered in Chapter 8. It begun by writing unit tests and integrations tests for the back-end of the server. These confirmed that each individual component worked as expected, and that all components worked together as they should. Finally, a performance evaluation was conducted on the system. Several optimisation algorithms were evaluated, and using both node pruning and then node bundling based on number of edges resulted in a 99.997% reduction in load times for a network of approximately 1000 nodes.

9.2 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 the visualisation of a network is to use node 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 and evaluation of the visualisation system created for this project.

9.3 Personal Reflections

This project has given me great insight into the field of visualisation. Throughout it, I have developed several new skills, such as API design, and the importance and difficulties of conducting a systematic review. Additionally, my time management and task allocation skills have improved throughout the project, and I learnt a lot about self-directed projects.

9.4 Future Work

9.4.1 Supporting Big Data

A possible way that the application could be extended would 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 [62] (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.

9.4.2 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 [46] or Microsoft Azure Storage [63]) to the application. Hence, all of a company's data could be visualised with relative ease.

9.4.3 More Interactive Visualisation

A way to further limit the data lost when minimising the amount of data sent across the network would be to allow users to select a node that has had other nodes bundled into it, and then those nodes would be displayed. This would mean that users could identify parts of the network they are interested in and find out more information from those parts. This could be implemented in two ways. One way would be by continuing to load in the network in the background after the initial visualisation has been displayed, which would not slow initial load times but would put more stress on the system and use up more RAM. Alternatively, when a node is selected an

AJAX call would be made to the server and that upon receiving the data to be slotted inserted into the network, the visualisation would be updated.

9.4.4 Information Before Rendering

As the amount of time to process a network on the server is insignificant next to how long it takes to render, a possible feature would be, upon selecting parameters that the network is to be loaded using (what network and what bundling algorithms are to be applied) then a request is sent to the server and a number of nodes, edges, bytes and predicted time to render is returned. This would help users in understanding what algorithms are having a big impact on the dataset provided, and ensure that users are warned when loading times would be large.

9.4.5 JavaScript Testing

JavaScript logic is far more difficult to test, with JavaScript code being primarily based on DOM manipulation, reacting to user input and AJAX calls. However, should the project be developed further, and especially if it were to go into production for a system, then testing the JavaScript logic would be critical.

9.4.6 Tulip Plug-ins

Tulip supports many plug-ins created by third party developers. If users could upload their own plug-in and select how it would interact with the APIs then companies could customise each part of the Tulip Web Wrapper to suit their own needs. This would not be difficult to implement to due how modular the system created is, and would significantly increase the flexibility and capabilities of the system.

Appendices

Appendix A

Additional Information

A.1 MoSCoW Requirements

After defining the scope of the project in Section 6.1, the below MoSCoW requirements [64] were made.

Must Have:

- **A web endpoint that calls the Tulip Python API.**
- **To be able to deal with large data.** This data must be too large to easily send over the network and be visualised by the client without manipulation.
- **A basic interface to show how the product works.**
- **Data that can be rendered must be returned from the endpoint.**

Should Have:

- **A neat JavaScript client.** Neat is defined as a well-documented list of functions that the client can call that then make calls to the web end-point
- **A flexible web endpoint.** Flexible is defined as covering 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.

- **The ability for a user to link their hosted data to the application** This could be an in-house solution such as a company network storage, or an external data host such as Amazon's S3 [46]. Hence, big data (hundreds of gigabytes or terabytes) 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.

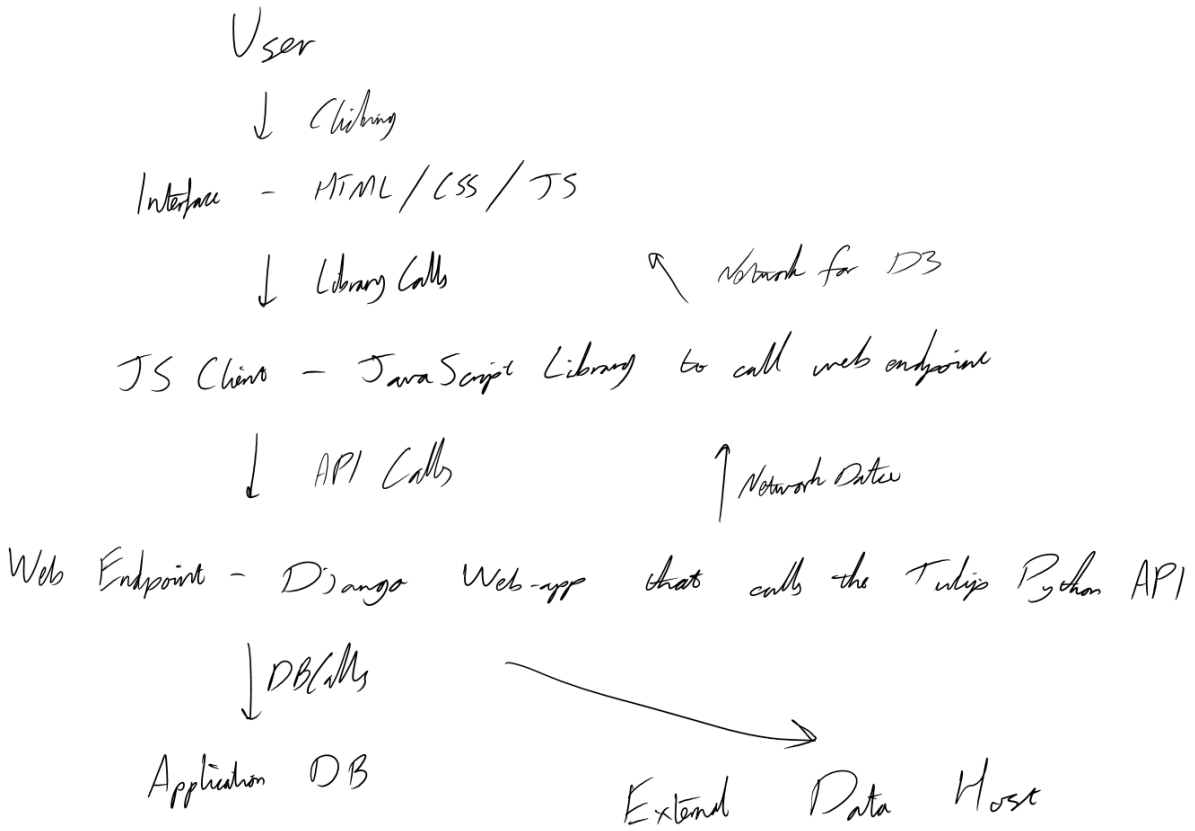

Figure A.1: Draft of Technical Infrastructure

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

Full Performance Evaluation Results

The below tables - Table B.1, B.2, B.3, B.4 - show the average of five runs collecting: how long different parts of the process took, the total time standard deviation, the number of edges and nodes, and the packet size.

	None	**Edge-based Bundling**	**Pruning**	**Both**
Time from click to AJAX response	12.4	17	15.6	15.8
Time spend processing in Vis.js	0.8	0.8	0.6	0.6
Time spend creating vis.js network	22.2	20.8	12.6	9
Time spend drawing	135.2	91	57	37.8
Sum of all JS times	170.6	129.6	85.8	63.2
Time from first to last Date()	170.8	129.6	85.8	63.2
Total Time Standard Deviation	15.2	19.3	20.5	15.1
Number of nodes	31	18	9	7
Number of edges	26	13	4	2
Time to load in the TLP	0.64	0.7	0.8	0.72
Time to prune the network			0.38	0.32
Time to bundle on edges		0.36		0.1
Packet Size	1300	1012	827	785

Table B.1: 31 Nodes

	None	Edge-based Bundling	Pruning	Both
Time from click to AJAX response	50	20.2	22	19
Time spend processing in Vis.js	4	3.6	2.2	1
Time spend creating vis.js network	97.2	90	44.8	23.6
Time spend drawing	7231.6	6635.2	593.2	102.8
Sum of all JS times	7382.8	6749	662.2	146.4
Time from first to last Date()	7382.8	6749	663	146.6
Total Time Standard Deviation	2010.2	1074.9	187	14.5
Number of nodes	199	155	60	17
Number of edges	199	155	60	17
Time to load in the TLP	1.24	1.2	1.18	1.16
Time to prune the network			2.86	2.52
Time to bundle on edges		1.4		0.6
Packet Size	7600	6100	3900	2300

Table B.2: 199 Nodes

	None	Edge-based Bundling	Pruning	Both
Time from click to AJAX response	177.4	126.6	58.8	57
Time spend processing in Vis.js	15	13.4	9.2	2.6
Time spend creating vis.js network	361.2	242.2	108.2	16.2
Time spend drawing	59370.8	37757.6	9241.6	76.8
Sum of all JS times	59924.4	38139.8	9417.8	152.6
Time from first to last Date()	59925	38140.2	9418.4	152.6
Total Time Standard Deviation	8343.5	9383	2223.1	30.5
Number of nodes	1089	627	226	12
Number of edges	1087	625	224	10
Time to load in the TLP	5.32	4.72	3.44	3.28
Time to prune the network			33.28	33.26
Time to bundle on edges		13.68		2.34
Packet Size	41400	29400	20300	12400

Table B.3: 1089 Nodes

	None	**Edge-based Bundling**	**Pruning**	**Both**
Time from click to AJAX response	646	423	314.2	241
Time spend processing in Vis.js	35.8	33	26.8	16.8
Time spend creating vis.js network	850.4	593	308.8	135.4
Time spend drawing	251316.6	130557.2	48306.4	14139.6
Sum of all JS times	252848.8	131606.2	48956.2	14532.8
Time from first to last Date()	252849.2	131606.6	48956.8	14532.8
Total Time Standard Deviation	32862	3937	6031.3	3341
Number of nodes	2849	1737	802	286
Number of edges	2835	1719	792	276
Time to load in the TLP	162.54	50.5	11.04	7.46
Time to prune the network			169.06	184.72
Time to bundle on edges		94.04		7.32
Packet Size	114000	83500	63500	42900

Table B.4: 2849 Nodes

Appendix C

Screenshots of the System

The below screenshots show the system visualising a network of approximately: 30 nodes, 200 nodes, 1000 nodes and 3000 nodes. Each network is shown with no algorithm being applied, node pruning being applied, node bundling based on number of edges being applied, and both node pruning and then node bundling based on number of edges being applied.

Additionally, screenshots containing a highly-connected network are shown before and after node bundling based on cliques.

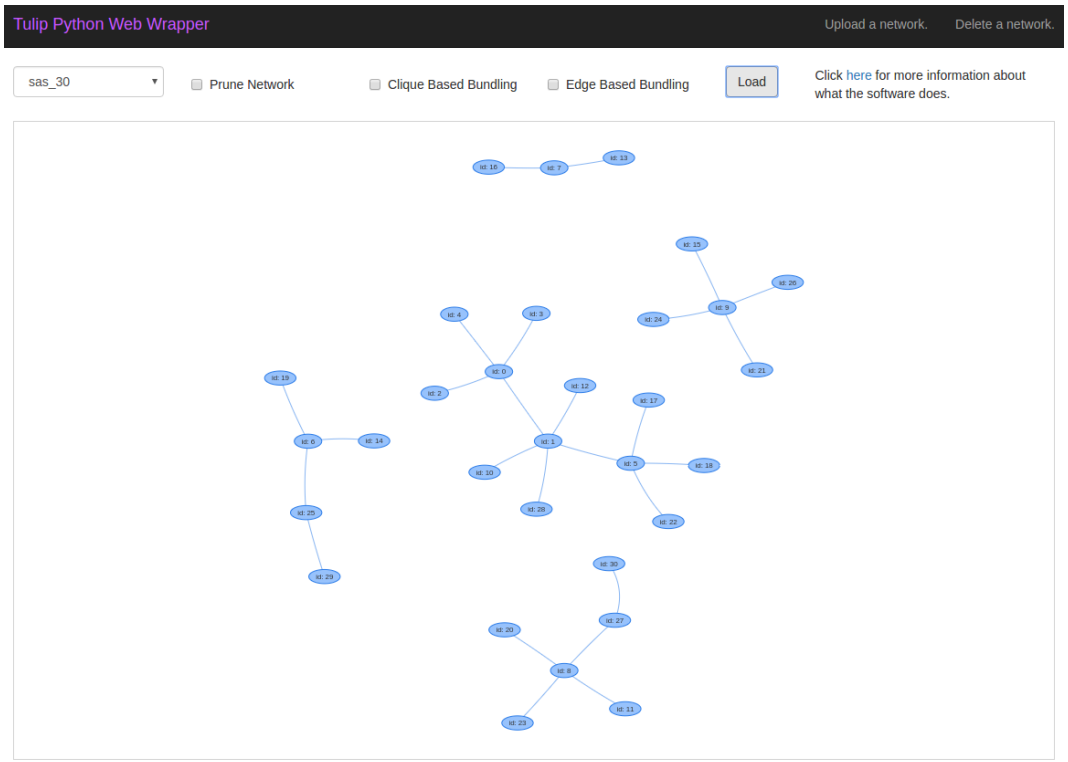

Figure C.1: This shows a network with 29 nodes being rendered without any algorithm having been applied on the server.

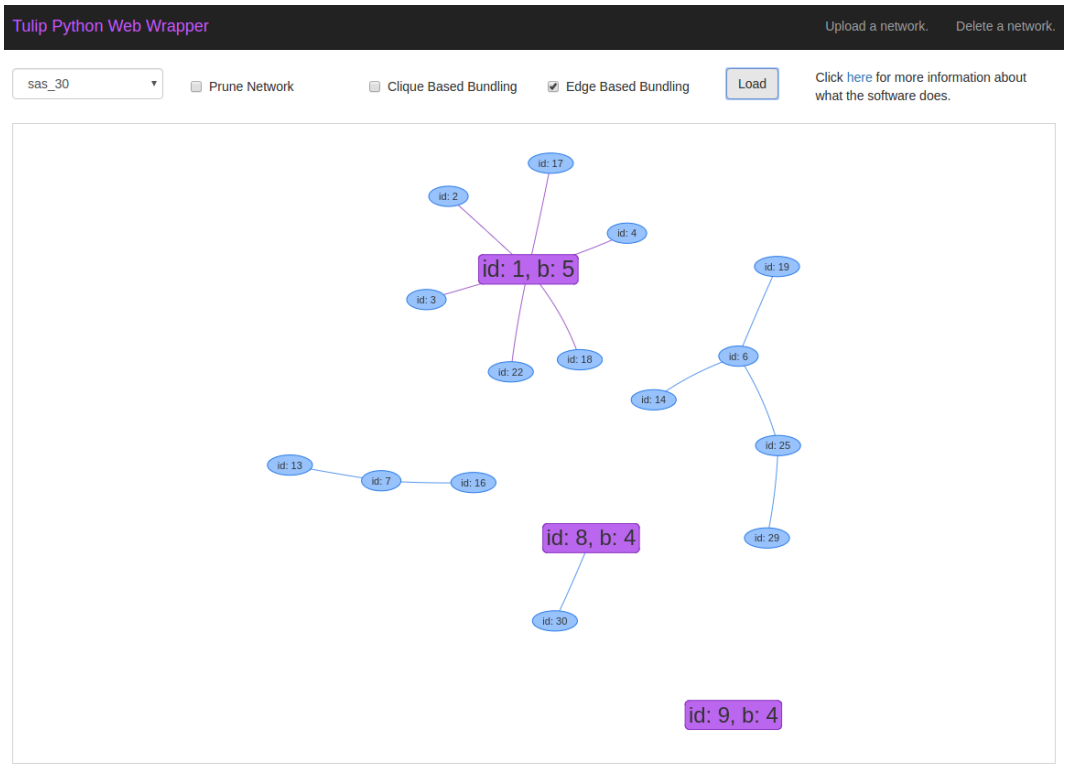

Figure C.2: This shows a network with originally 29 nodes being rendered with node bundling based on number of edges.

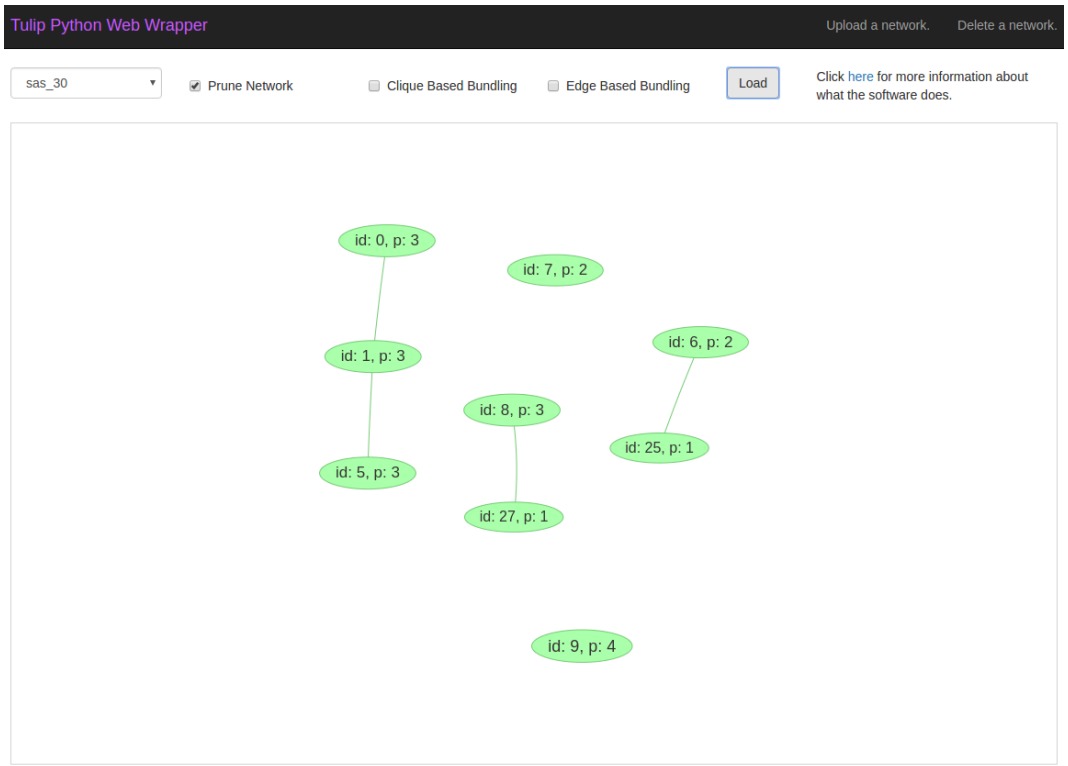

Figure C.3: This shows a network with originally 29 nodes being rendered with node pruning.

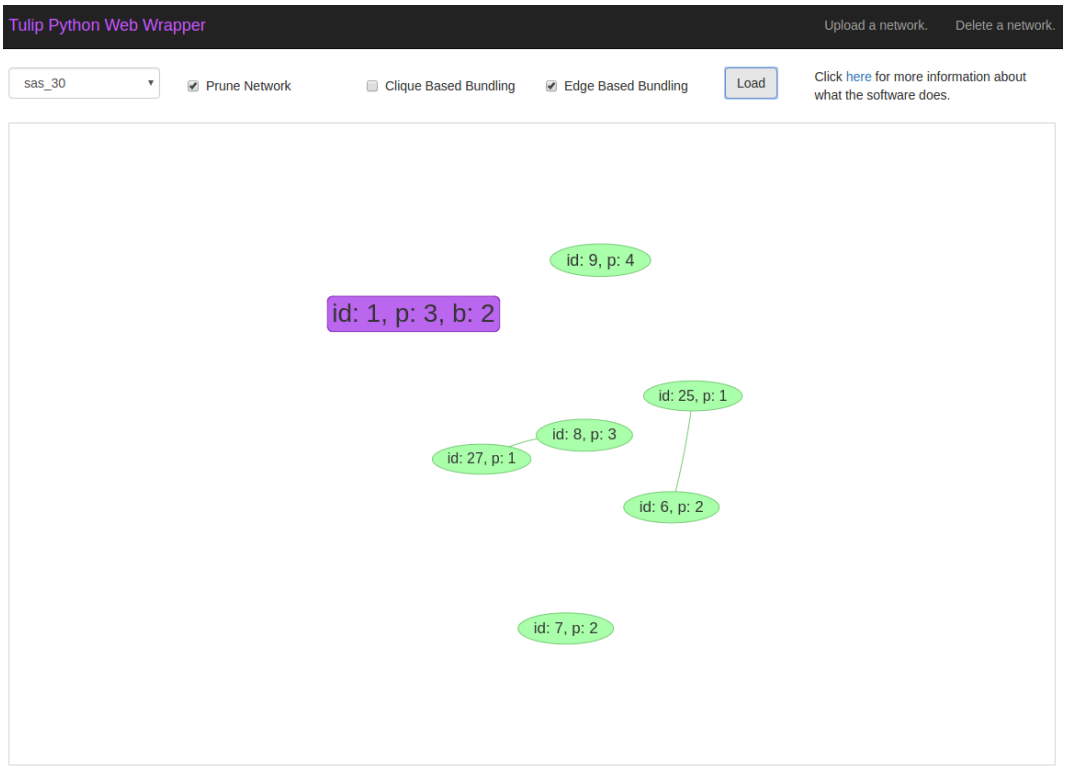

Figure C.4: This shows a network with originally 29 nodes being rendered with both node pruning and then node bundling based on number of edges.

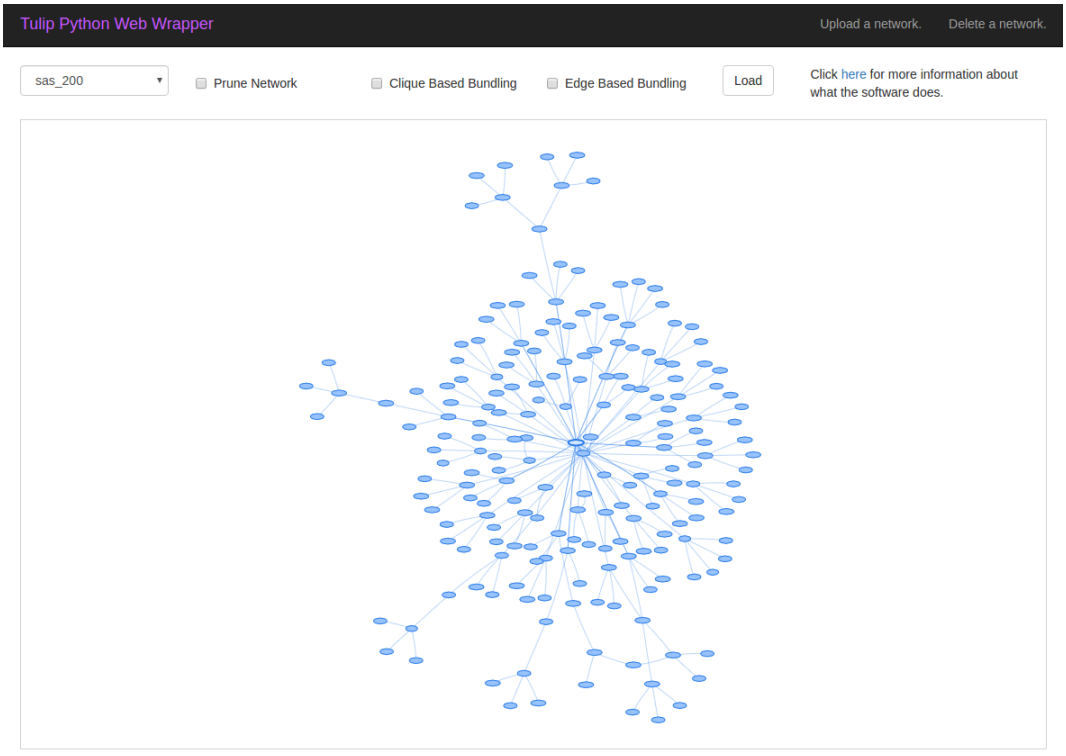

Figure C.5: This shows a network with 199 nodes being rendered without any algorithm having been applied on the server.

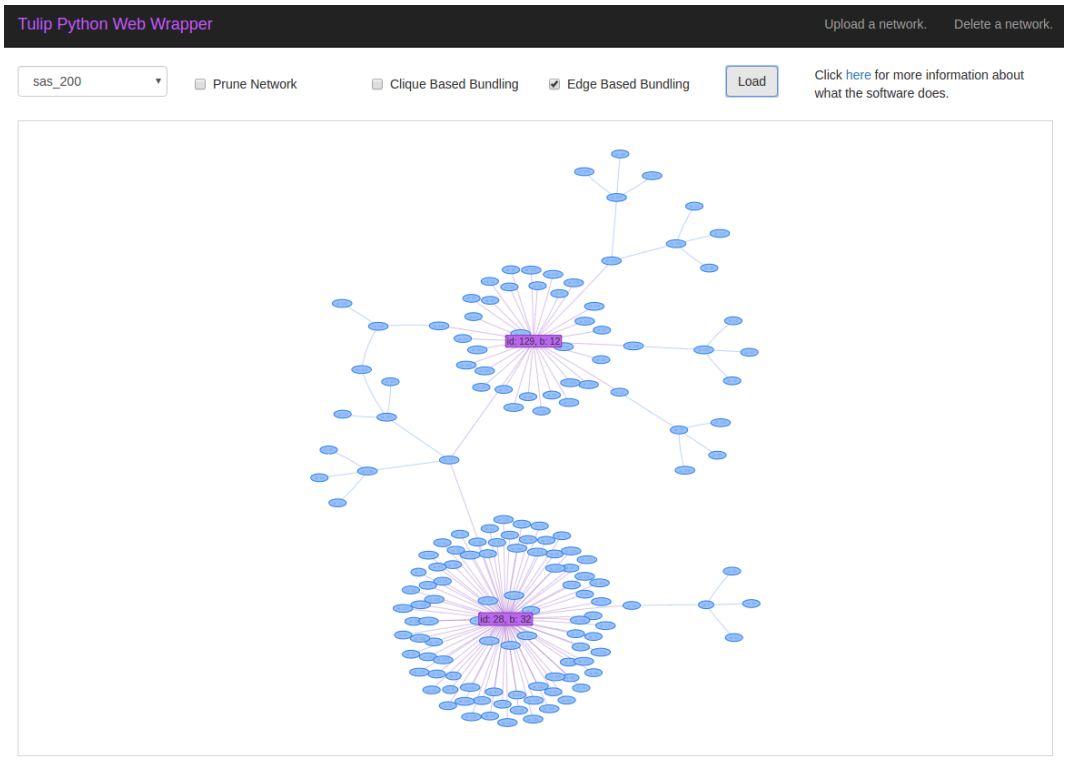

Figure C.6: This shows a network with originally 199 nodes being rendered after applying node bundling based on number of edges.

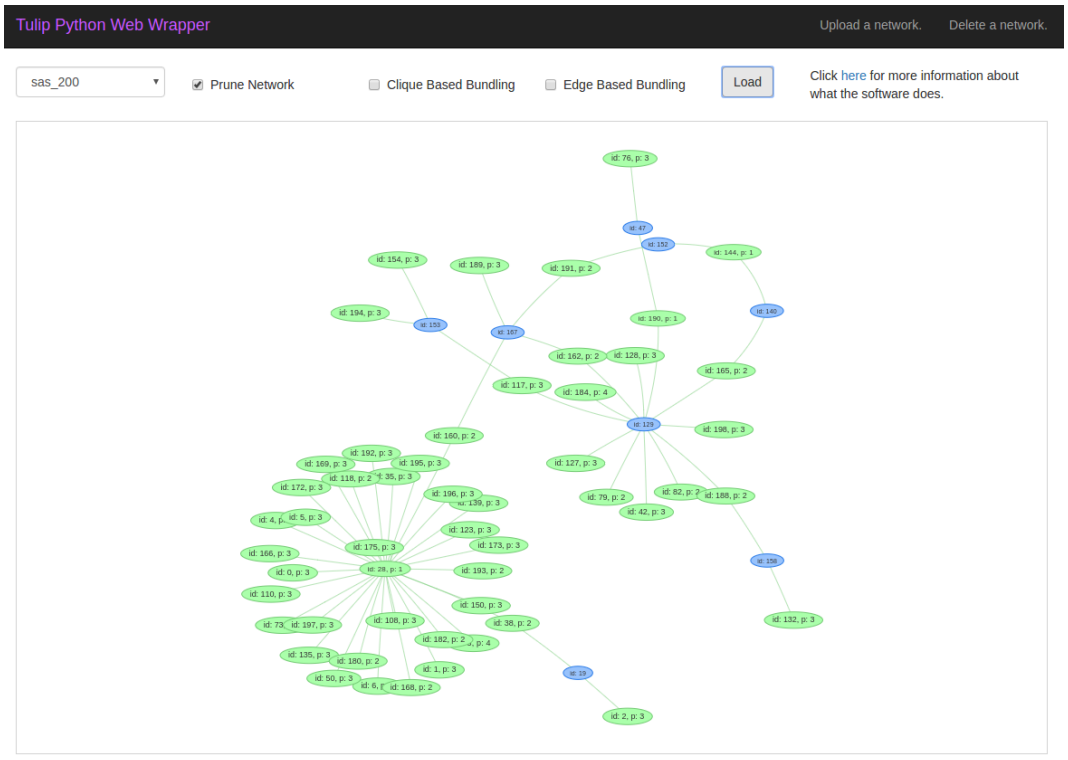

Figure C.7: This shows a network with originally 199 nodes being rendered after applying node pruning.

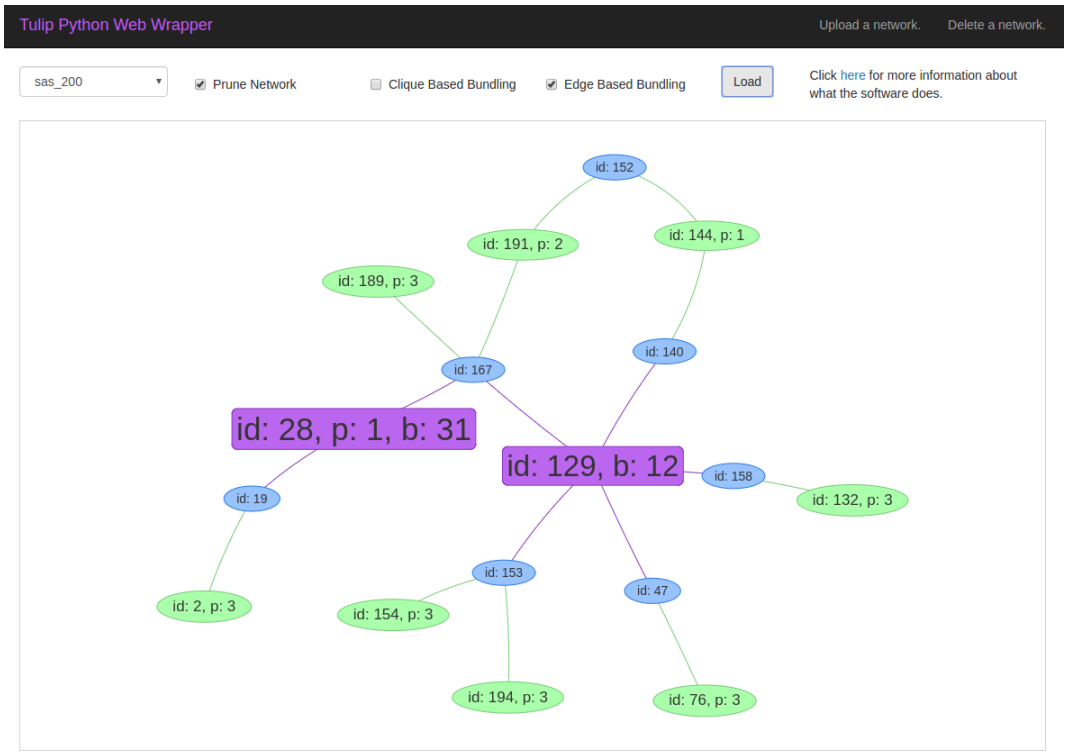

Figure C.8: This shows a network with originally 199 nodes being rendered after applying both node pruning and then node bundling based on number of edges.

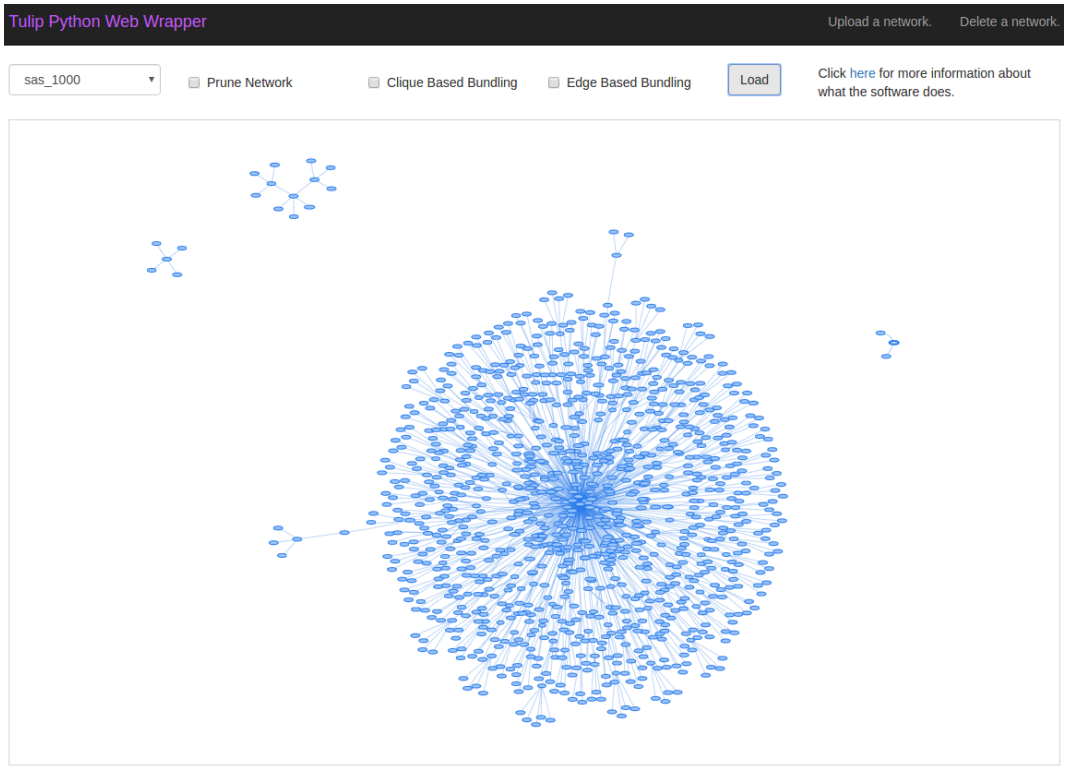

Figure C.9: This shows a network with 1089 nodes being rendered without any algorithm having been applied on the server.

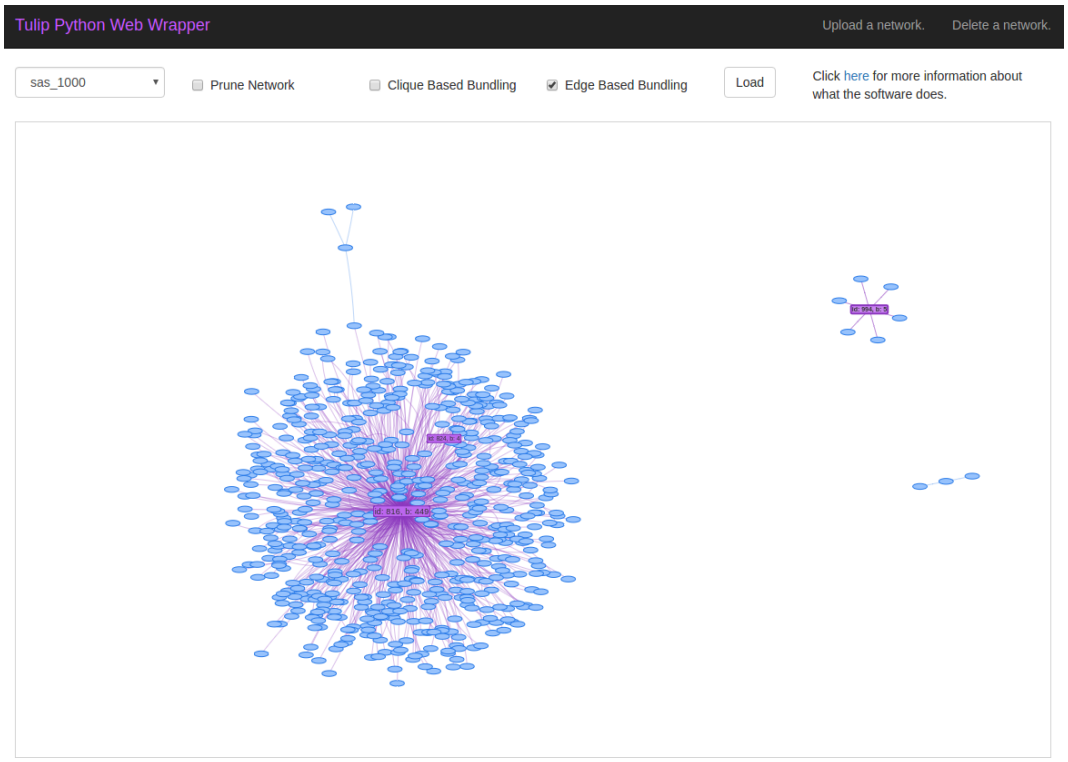

Figure C.10: This shows a network with originally 1089 nodes being rendered after applying node bundling based on number of edges.

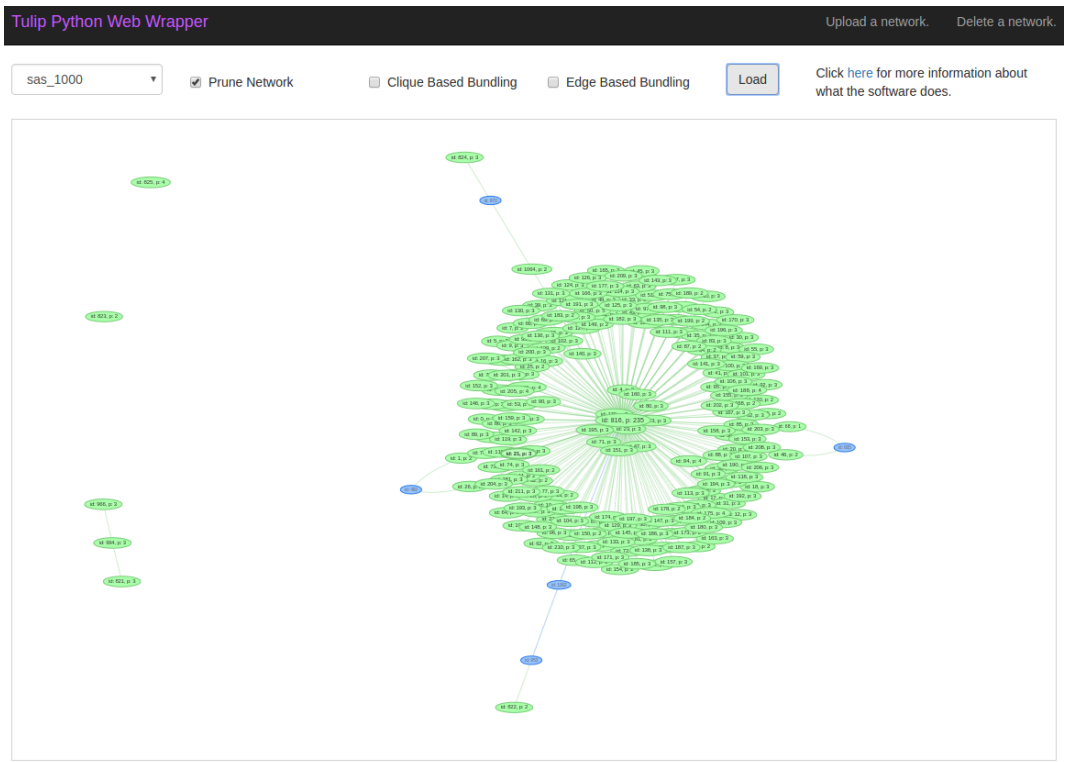

Figure C.11: This shows a network with originally 1089 nodes being rendered after applying node pruning.

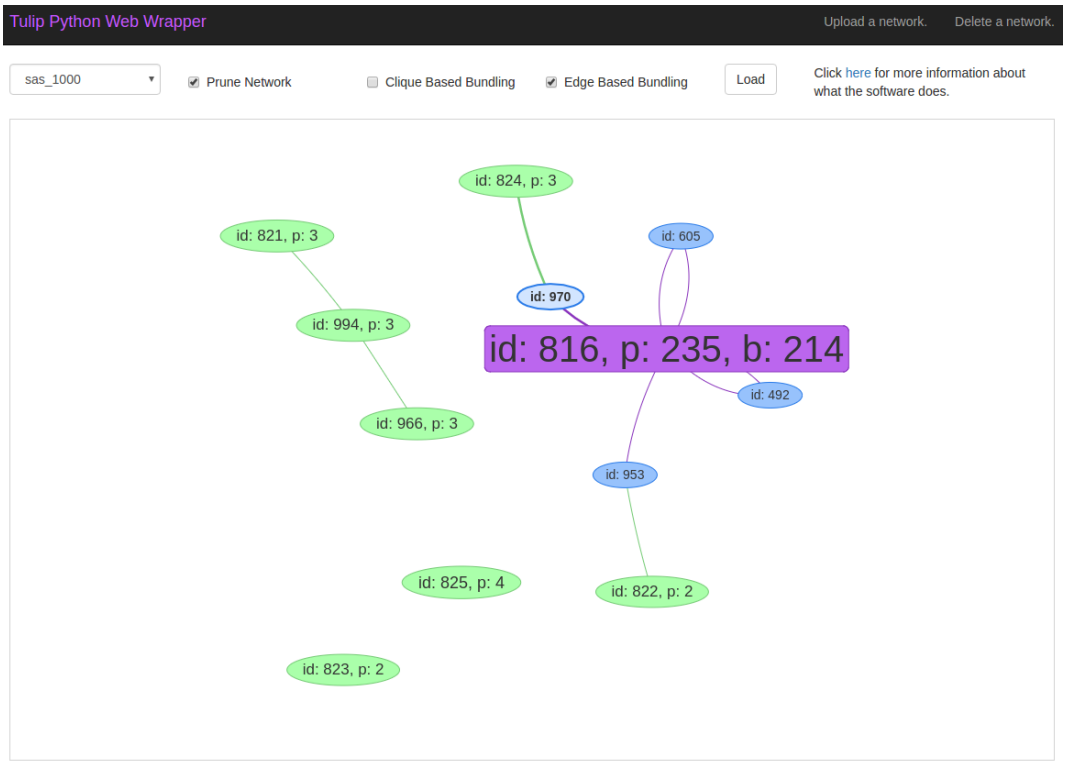

Figure C.12: This shows a network with originally 1089 nodes being rendered after applying both node pruning and then node bundling based on number of edges.

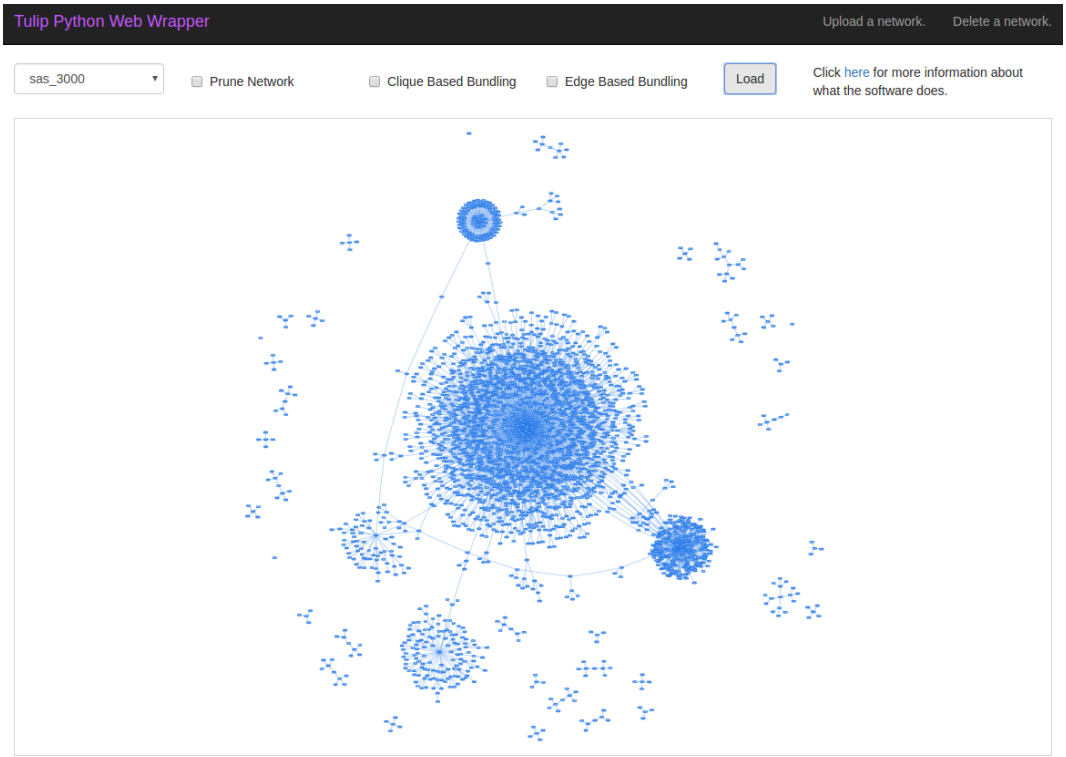

Figure C.13: This shows a network with 2849 nodes being rendered without any algorithm having been applied on the server.

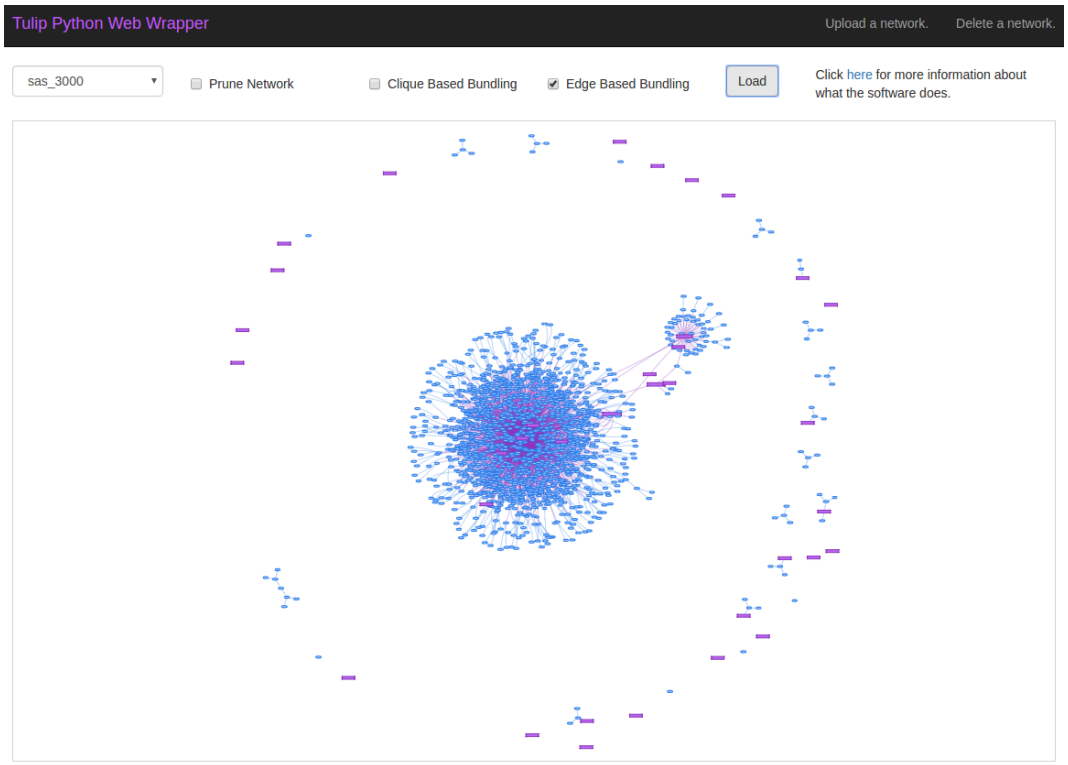

Figure C.14: This shows a network with originally 2849 nodes being rendered after applying node bundling based on number of edges.

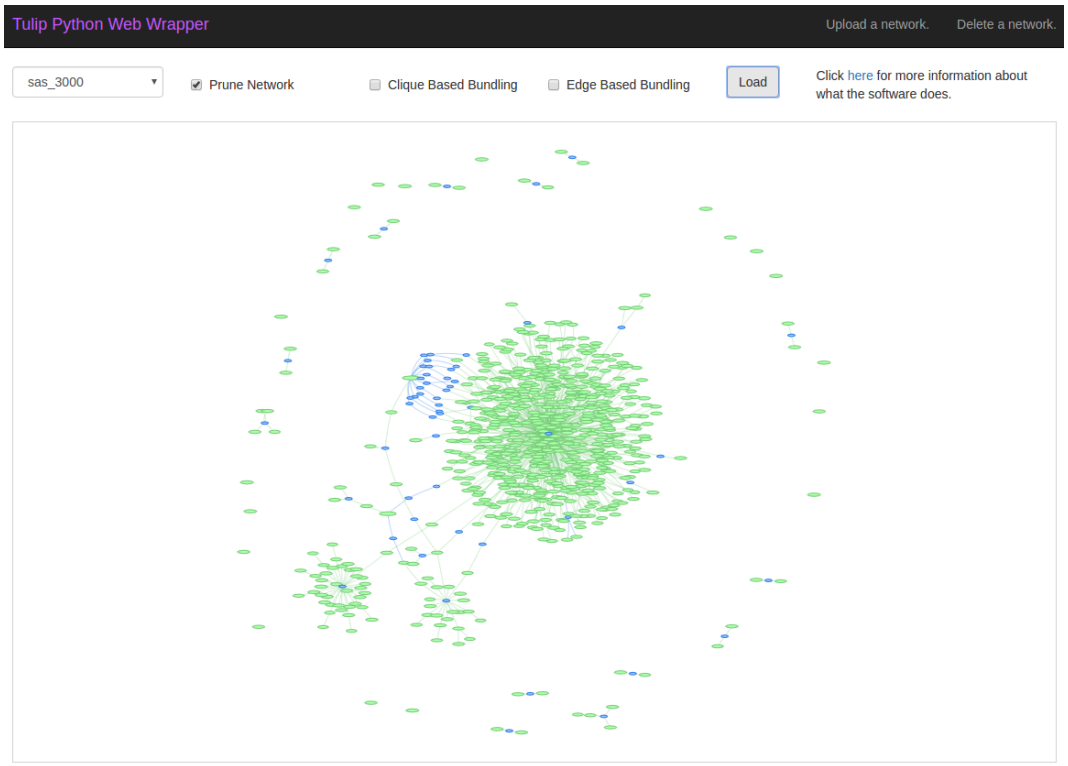

Figure C.15: This shows a network with originally 2849 nodes being rendered after applying node pruning.

Figure C.16: This shows a network with originally 2849 nodes being rendered after applying both node pruning and then node bundling based on number of edges.

Appendix D

How to run and use the System

D.1 Setting up the Django Server

The server was developed and tested on Linux. Python and Django are both fully supported on Windows and, although virtualenv works slightly differently, Windows compatibility should exist, although it is untested.

In order to run the project, the following steps must be taken:

1. **Ensure Python 3 is installed.** This can be downloaded from https://www.python.org/downloads/. Make sure that, when installing Python, pip (Python Package Index) is also installed. If pip is not installed, then it can be by entering: `sudo apt install python-pip`.
2. **Create a virtual environment.** If virtualenv is not installed, then open a terminal and enter: `pip install virtualenv`. Once this has finished installing, a virtual environment is created by navigating to the directory where the virtual environment should be stored and then entering: `virtualenv tulipwebwrapper --python=python3`. The flag ensures that Python 3 is used if multiple versions of Python exist on the system being used.
3. **Activate the virtual environment.** This is done by entering: `source /path/to/ve/tulipwebwrapper/bin/activate`.
4. **Installing the project requirements.** Navigate to the root of the project and then enter the folder named 'dev' then 'webtulip'. From here, open a terminal and enter: `pip install -r requirements.txt`.
5. **Ensure the database is up to date.** Enter: `python manage.py makemigrations` to check for and create database migrations (this will have no effect if no models have been changed), and then: `python manage.py migrate` which will run the migrations and ensure the database schema is up-to-date.
6. **Run the server.** Finally, to run the server, just enter: `python manage.py runserver` (or ./run.sh which contains that exact command.).

In short, once Python 3 and pip are installed, the commands are:

1. `pip install virtualenv`

2. `cd path/to/where/ve/should/be/created`
3. `virtualenv tulipwebwrapper --python=python3`
4. `source /path/to/ve/tulipwebwrapper/bin/activate`
5. `cd path/to/project/dev/webtulip`
6. `pip install -r requirements.txt`
7. `python manage.py makemigrations`
8. `python manage.py migrate`
9. `python manage.py runserver`

D.2 Accessing and using the interface

Once the server is running, navigating to `http://127.0.0.1:8000` in any browser will direct users to the home page of the system.

From here, users can navigate to the upload page (`http://127.0.0.1:8000/upload`) and upload networks into the system. The system accepts `.tlp` networks and `.json` networks in the format that SAS use. Examples of these can be found in /path/to/project/data/sas_data, which contains two folders, one containing the `.tlp`'s (previously converted) and one containing the original `.json` files. Additionally, in the project/data folder, there is g1.tlp and g2.tlp, two very simple networks which were used for testing. To upload a `.tlp`, set the network type to 'Tulip TLP' (default) and to upload a `.json`, set the network type to 'Industry Standard'.

Once network(s) have been uploaded, navigate back to the home page and from there, select the network to visualise and then select any algorithms to be applied to the network.

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] William Raymond Scott. *Group theory*. Courier Corporation, 2012.

[9] David F Anderson, Jodi Fasteen, and John D LaGrange. The subgroup graph of a group. *Arabian Journal of Mathematics*, 1(1):17–27, 2012.

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

[11] Brenda S Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994.

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

[13] 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.

[14] Stephan Herhut, Richard L Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. Parallel programming for the web. In *HotPar*, 2012.

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

[16] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Experiments on graph clustering algorithms. In *European Symposium on Algorithms*, pages 568–579. Springer, 2003.

[17] Fred S Roberts and Joel H Spencer. A characterization of clique graphs. *Journal of Combinatorial Theory, Series B*, 10(2):102–108, 1971.

[18] 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.

[19] Emden R Gansner, Yifan Hu, Stephen North, and Carlos Scheidegger. Multilevel agglomerative edge bundling for visualizing large graphs. In *Pacific Visualization Symposium (PacificVis), 2011 IEEE*, pages 187–194. IEEE, 2011.

[20] 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.

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

[22] 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.

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

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

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

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

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

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

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

[30] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[31] Brian Beckman. Theory of spectral graph layout. Technical report, Technical Report MSR-TR-94-04, Microsoft Research, 1994.

[32] Walter Didimo and Maurizio Patrignani. *Graph Drawing: 20th International Symposium, GD 2012, Redmond, WA, USA, September 19-21, 2012, Revised Selected Papers*, volume 7704. Springer, 2013.

[33] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphvizopen source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

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

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

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

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

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

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

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

[41] Django. URL Dispatcher — Django Documentation. https://docs.djangoproject.com/en/1.10/topics/http/urls/. Last accessed: 19th March 2017.

[42] Django. Writing views — Django Documentation. https://docs.djangoproject.com/en/1.10/topics/http/views/. Last accessed: 19th March 2017.

[43] Django. Models — Django Documentation. https://docs.djangoproject.com/en/1.10/topics/db/models/. Last accessed: 19th March 2017.

[44] Django. Working with forms — Django Documentation. https://docs.djangoproject.com/en/1.10/topics/forms/. Last accessed: 19th March 2017.

[45] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

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

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

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

[49] Tulip. tulip module API - Tulip Python 4.10.0 docs. http://tulip.labri.fr/Documentation/current/tulip-python/html/tulipreference.html#tulip.tlp.clusteringCoefficient. Last accessed: 17th March 2017.

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

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

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

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

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

[55] 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.

[56] Django. Templates — Django Documentation. https://docs.djangoproject.com/en/1.10/topics/templates/. Last accessed: 19th March 2017.

[57] Bootstrap. Bootstrap - The world's most popular mobile-first and responsive front-end framework. https://getbootstrap.com/. Last accessed: 19th March 2017.

[58] jQuery. jQuery. https://jquery.com/. Last accessed: 19th March 2017.

[59] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[60] Francesca Basanieri and Antonia Bertolino. A practical approach to uml-based derivation of integration tests. *4th International Quality Week Europe*, 2000.

[61] jQuery. jQuery.ajax(). https://api.jquery.com/jquery.ajax/. Last accessed: 19th March 2017.

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

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

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