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 — February 26, 2017

Abstract

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

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

Contents

1 Introduction **1**

2 Motivation **2**

3 Review of the Field **3**

3.1 Visualisation 3

3.2 Network Visualisation 3

3.3 Challenges of the Visualisation of Massive Networks in a Browser 3

3.4 Example within the Field 4

3.4.1 Visual Investigator 4

4 Research **7**

4.1 Background 7

4.2 Current Graphing Software 7

4.3 Systematic Review of Existing Software 9

4.3.1 Aim 9

4.3.2 Methodology 10

4.3.3 Data and Results 11

4.3.4 Conclusion 13

4.4 Next Steps 14

5 Design **15**

5.1 Scoping 15

5.2 Requirements 15

5.3 Design Decisions 16

5.3.1 Web Endpoint 16

5.3.2 A Basic Interface 17

5.3.3 Making the web wrapper flexible 17

5.3.4 JS Library 17

5.3.5 Return an image 17

5.3.6 Fully fledged interface 17

5.4 Technical Infrastructure 18

5.4.1 Sample Data Flow 18

6 Implementation **19**

6.1 Created Software 19

6.2 Problems Encountered 19

6.3 Evaluation 19

7 Conclusion **20**

7.1 Achievements 20

7.2 Recommendations? 20

Appendices **21**

A Appendices **22**

Chapter 1

Introduction

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

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

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

Chapter 2

Motivation

Towards the end of my summer internship with SAS, I started discussing some possible project ideas that interested both them and me. Visualisation of huge networks in a browser was one of the ideas that we discussed which was interesting for a number of reasons. SAS already has in-house network visualisation software which works fine for up to a few thousand nodes, but begins to struggle in several ways for many more nodes than that. On top of that, network visualisation is something that I was interested in and seemed like something that I could work on that would stimulate me and be of benefit to SAS as well.

There were several directions the project could be taken in, from improving the performance of the current software so that it could handle far more data and still load (relatively) quickly, to trying to do more processing server side so that the client had less to do, to joining nodes or edges together that are similar (bundling) in order to reduce loading times for the client. Additionally, an important choice to make was whether or not time is something that mattered. A possible way to view the project would be to come with a way to display the networks as efficiently as possible. However, for huge networks, there is no point in rendering all of the nodes if they don't show anything useful due to the sheer number of them. Hence, another possible take on the project would be, with or without time constraints, to come up with a way to allow users to visualise huge networks (based on) hundreds of thousands or even millions of nodes and gain valuable knowledge.

TODO: expand - make a paragraph on time vs visualisabilty and also why displaying all is not plausible.

During the first couple of weeks of the project, contact was made to both my adviser and SAS in regards to what direction the project should be taken in. The end result of this discussion was that, at least initially, research would be focused on finding effective ways of visualising large networks, with time not being a critical factor, i.e. focusing on the visualisabilty of the network as opposed to trying to make it render as fast as possible.

Chapter 3

Review of the Field

3.1 Visualisation

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

Visualisation of data can be static or interactive, with interactive data able to give a user more information by, for example, selecting parts of the visualisation and getting more information on it, changing parameters for the visualisation and observing changes, or the moving of nodes in a network and seeing how the rest of the network responds. This can greatly improve the usefulness of the visualisation for users, with new trends or vulnerabilities becoming clearer faster through interacting with the visualisation.

3.2 Network Visualisation

Network Visualisation is a branch of Data Visualisation involving the ability to display nodes and edges in a meaningful way to a user. This can be used for a large variety of purposes, such as to discover how information is grouped, how subsets of data interacts with other subsets, and how interconnected or isolated the data is. `https://flowingdata.com/2010/11/17/why-network-visualization-is-useful/`

3.3 Challenges of the Visualisation of Massive Networks in a Browser

Visualisation of massive networks can lead to many challenges. Massive is a very vague term, but is generally considered as anything above about ten thousand ¡¡source inserted here¿¿. These challenges generally present themselves in the form of computational challenges or visualisation challenges. Computational challenges are when the a system struggles to display a network visualisation due to lack of speed or performance from the CPU, RAM, backing storage or bandwidth. If a machine has a slow processor then the amount of time to create and then render the network will be increased greatly. Similarly, if the RAM installed is slow or if the network is big enough that it will not fit in the available amount of RAM, then the performance of the network visualisation will degrade greatly.

The other computational challenge is the ability to store the network. Whether it is stored locally on a machine or stored somewhere on a network, this relies on fast disk access and enough storage space (which, depending on the amount of data that is to be stored to be visualised, could be nearly impossible to store on one machine). Hence, when storing several massive networks, they will always be held on the cloud, which means a high quality internet connection is required.

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

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

3.4 Example within the Field

3.4.1 Visual Investigator

Currently, Visual Investigator (a SAS product ¡¡insert link for who SAS are¿¿) lets users view a network of entities in the system to see how they connect (for example: names, phone numbers and addresses). This works perfectly for a few entities, with useful information being shown clearly. However, with potentially millions of nodes, the network no longer shows any useful information - just a mass of nodes - and becomes unusably slow (Visual Investigator is a web app and both processing power and memory are hard limits which, depending on network size, will easily be hit).

As can be seen from the pictures below, the network starts out clear and load times are instant, but it soon becomes both meaningless and really slow. If there was a way to both speed up the loading times of huge networks, and make them convey meaningful information to the user, this would be very useful.

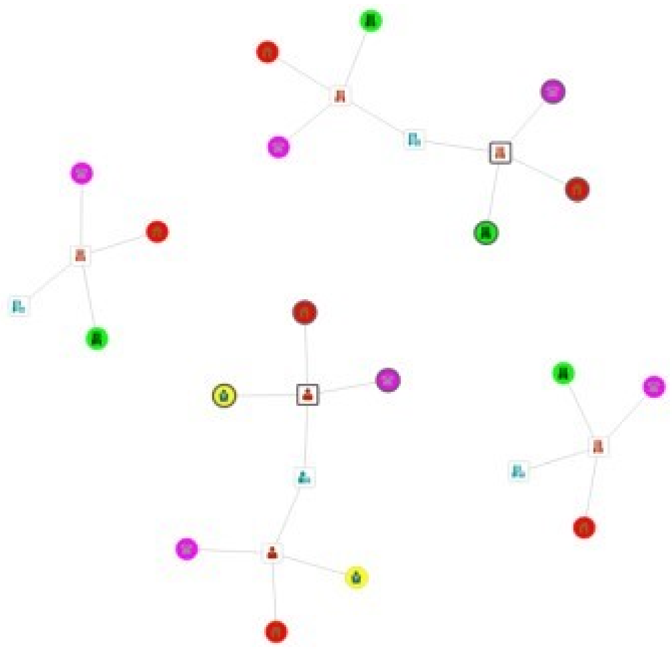


Figure 3.1: 30 nodes, Load Time less than 1 second

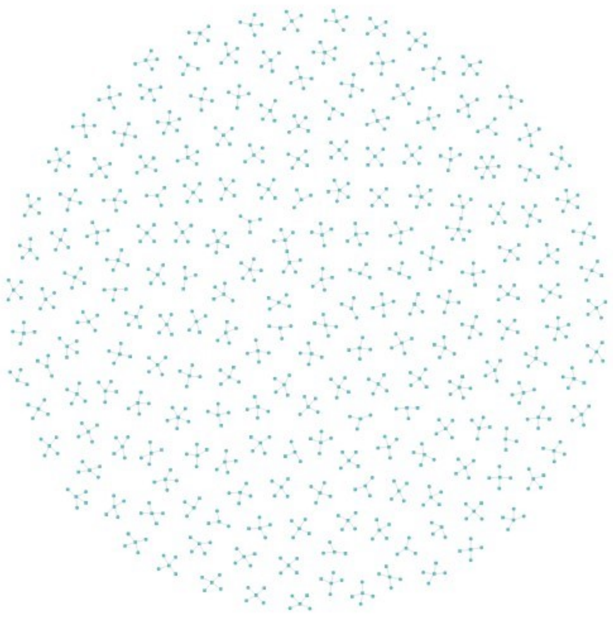


Figure 3.2: 1000 nodes, Load Time 10 seconds

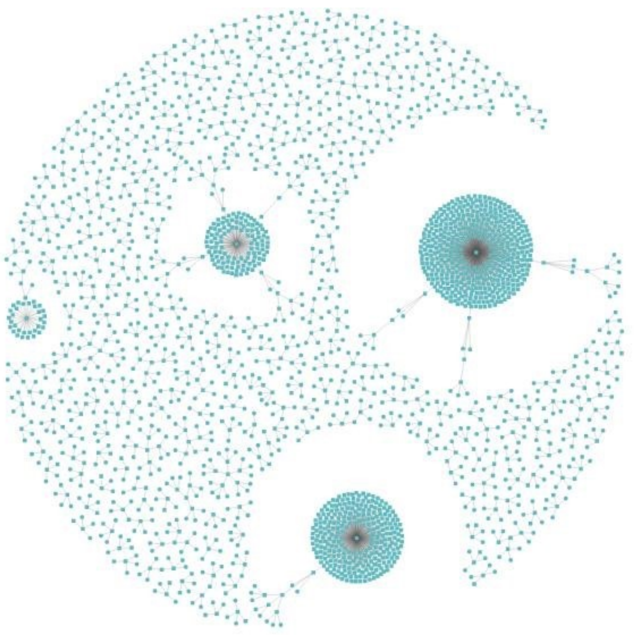


Figure 3.3: 2500 nodes, Load Time 30 seconds

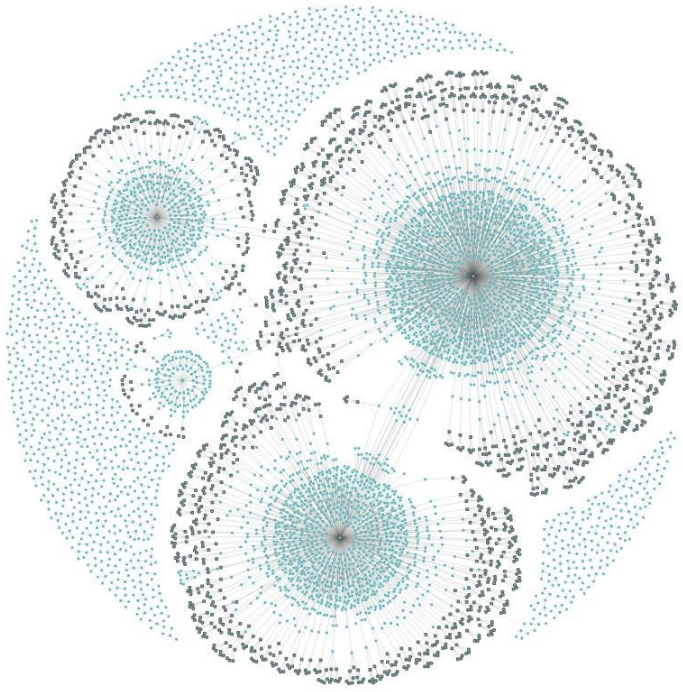


Figure 3.4: 7500 nodes, Load Time 300 seconds

Chapter 4

Research

4.1 Background

Initially, research done covered both network visualisation and specifically massive network visualisation, and as well as that, it looked into software that existed which visualised networks. In regards to the visualisation of massive networks, some important techniques learnt were:

- Node grouping/bundling: This is when software uses algorithms to decide (with or without some form of user input) which nodes are very similar to each other, and upon establishing this relationship, groups them in to one node, generally visually different in some way (often by size or colour). This allows for hundreds or thousands of nodes to be grouped into one node and hence take up for less memory, processing power, bandwidth and screen space on the clients machine. A way to make this even more helpful is to show how the nodes being bundled interact with themselves, for example if they are heavily connected or all connected to just a few nodes, or if they are in a ring formation.
- Edge bundling: This is similar to node grouping but applied to edges, when several edges take very similar paths from one node (or group of nodes) to another, then the edges can be bundled into one edge. If node grouping is done then this process will have less of an impact as all of the edges will already be bundled together assuming the node grouping algorithm was successful. Like node bundling, it is often shown through the change of the size of the edge or the colour. See Figure 4.1.
- Local Edge Lens: This involves rendering all or part of the graph, and upon focusing on a section of the graph, that part is zoomed in or clarified as if through a lens and that part of the graph is zoomed in, becoming more clear. Another way of implementing this can be that when the lens is applied, it can render more information that was not shown before in order to save computational power. See Figure 4.2.
- Deleting unnecessary content ”smartly”: This involves using an algorithm to remove certain nodes or edges that are deemed to be adding little or no useful information for the user. This algorithm can take no parameters, or work off a user’s input, in which they specify what they are interested in, so other information is partially or fully removed. See Figure 4.3.

4.2 Current Graphing Software

On top of finding out techniques used by software for large network visualisation, a list was also created of all software that was mentioned in papers or found during research that was related to large network visualisation.

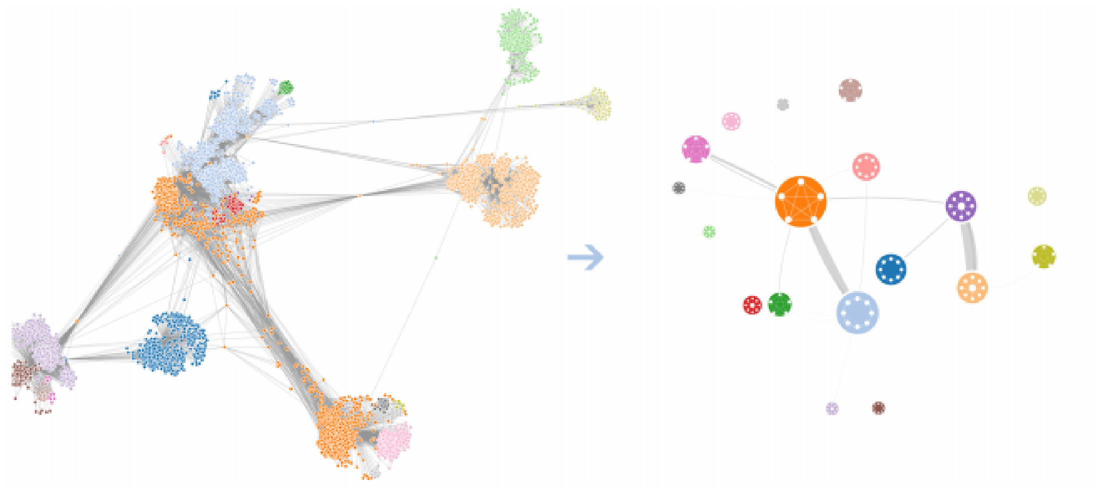


Figure 4.1: Demonstrates both node and edge bundling, where thickness of edge displays how many edges there are, and size of node indicates number of nodes group. On top of that, each node displays how nodes within it were connected together.

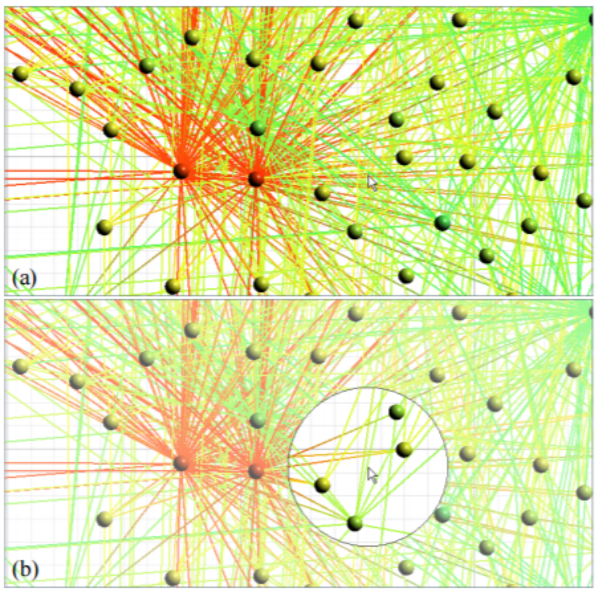


Figure 4.2: If a network is cluttered (a), then it can be hard to get useful information from it. A local edge lens (b) lets users easily identify which edges are connected to which nodes.

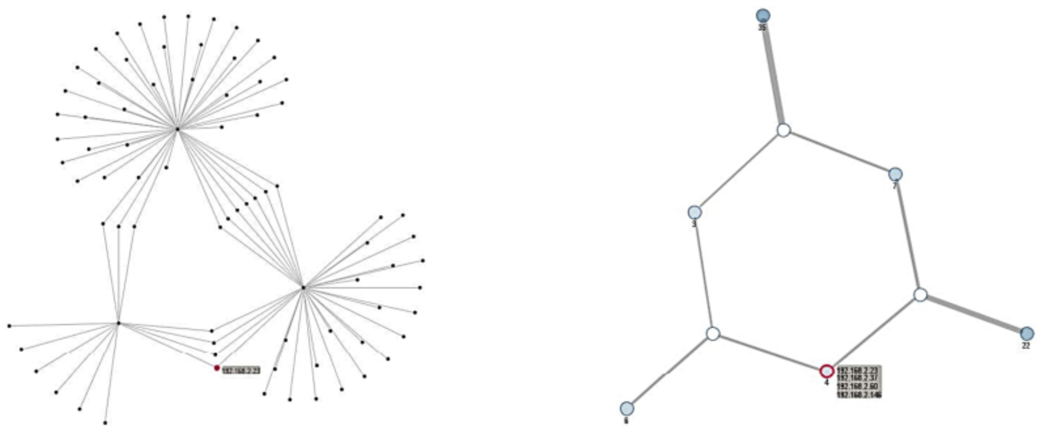


Figure 4.3: This figure clearly shoes how the deletion of many nodes does not always mean information is lost, and in some cases, information can become more clear as a result of the operation.

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 was reasonable to look further into, found by searching online and through mentions from papers:

- GUESS
- SNAP
- Gephi
- GraphViz
- Tulip
- D3.js
- Vis.js

This software is reviewed below.

4.3 Systematic Review of Existing Software

4.3.1 Aim

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

4.3.2 Methodology

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

The initial plan was to split the criteria into several categories and weigh each category as to its importance.

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

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

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

A major criteria would then be how well the software would scale up to massive networks - this would include running tests on increasingly large networks and seeing how long the software takes, and also if it includes node or edge bundling or other ways of increasing clarity of massive networks. Time taken to render is also an important statistic, but less important than clarity for massive amounts of data for this project. This is perhaps the most important category as no matter the above features, if the network is not clear for huge numbers of nodes / edges then it does not answer the problem at hand. A feature that would be desired to increase user clarity would be the ability to view the network in many different layouts - which would give users more flexibility and would often lead to better understanding.

This set of criteria was compiled into a list which was to act as the process for the experiment.

1. Standalone or library
2. Documentation:
   (a) Is it easy to understand - explained clearly and concisely?
   (b) Is it Comprehensive - software fully documented?
   (c) Length of time until confident beginning to use system?
3. Time until a simple hard-coded graph could be displayed?
4. Can a file be loaded into the system?
5. Can a network be exported to a file?
6. How long it took to render the graph of:

(a) 30 nodes
(b) 200 nodes
(c) 1000 nodes
(d) 3000 nodes

7. If the software allowed for:
   (a) Zooming
   (b) Panning
   (c) Scrolling
8. Can a graph be dynamic - allowing data taken at multiple points over time to be viewed?
9. Does the software included features for displaying huge graphs:
   (a) Edge Bundling
   (b) Node bundling
   (c) Alternative layout options
10. Can a graph be multivariate?
11. Is there a possibility of changing graphical settings for the network?
12. Can a graph be displayed in 3D?
13. Any other idiosyncrasies?

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

4.3.3 Data and Results

Upon beginning to run the experiment, it became clear that several pieces of software would not be able to be assessed by the criteria created previously, for a number of reasons.

GUESS

The first piece of software, GUESS, was software that never left beta, and was last developed in 2007. This meant that the software had fairly little documentation and what it did have was poor. On top of this, the wiki no longer existed 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

This piece of software could not be compared to much of the criteria as the purpose of it is to analyse and manipulate graphs, as opposed to also visualise them. 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.

Due to the above limitation, at this point GUESS and SNAP were considered no longer appropriate. This left the remaining five software packages to be analysed and compared to the research criteria, stated above.

Gephi

This was the first software package that was as expected, and was fairly easy to test, use and was well documented. After about half an hour I was comfortable with how it worked, able to make simple graphs, and to import graphs from many formats (CSV, GDF, QML, GraphML, net, etc.) and export to all of them, along with as a PNG, PDF or SVG. It supported zooming, panning and scrolling, and having data being dynamic was a possibility. There was also multiple heavily customisable layout options, some of which would suit large networks more than others, although there was no additional support for large networks, other than the system being very efficient (bundling was not supported, along with partial viewing of the data). Additionally, there was the ability to change graphical settings, have the data graphed in 3D, and have a multivariate graph.

Graphviz

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

`http://www.graphviz.org/content/root` - this shows how the UI is clearly focused towards smaller networks of up to a thousand nodes or so.

Tulip

Tulip is an information visualisation framework that also lets you both visualise and analyse the data. It is very easy to use and is well documented, allows for easy importing and exporting of networks, and is really fast, allowing for 3000 nodes to be visualised instantly.

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

D3.js

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

Vis.js

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

Comparisons

For the majority of the criteria, all of the software packages performed very similarly.
TODO: Insert (more) tables and remove (more) data from above where appropriate.
TODO: Something like ”all of the above supported scrolling/zooming/panning.”

See Table 4.3.3 for render times.

	30	200	1000	3000
Gephi	Instant	Instant	Instant	0.5
Graphviz	Instant	0.3	1	3
Tulip	Instant	Instant	Instant	Instant
D3.js	todo	todo	todo	todo
Vis.js	todo	todo	todo	todo

Table 4.1: This shows how long it took each of the different pieces of software to render the specified number of nodes

4.3.4 Conclusion

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

However, given all of the above data and comparisons, Tulip was chosen as the most fitting software, performing the best under both light and heavy load, and also having a very fully-featured API in both C++ and Python (a C++ wrapper).

4.4 Next Steps

As a result of the outcome of research, there were several different directions that the project could go:

- Write several different programs in JavaScript (or Python etc.) that alter huge graphs by bundling etc. and compare how they become easier to visualise and/or render using D3.js This would include using several large networks of different types (sparse or very interconnected) and use different algorithms or techniques to see what is the most efficient way or best way to evaluate.
- Evaluate several different types of graphing software (Tulip/Gephi/GraphViz) and see where they begin to fail and what their bottlenecks are.
- Look into database systems that visualise data too. Many database packages allow for data visualisation.
- Build a web application for one of the pieces of desktop software in order to allow users to access a system and visualise a huge amount of data without the need for an extremely powerful machine.

After much discussion and thought, it was decided that a web application would be built for the project, based on the Tulip Python API. This would result in a users being able to access a wealth of information without owning a huge amount of computational power, which could benefit both employees of a business, allowing them to work from home on less powerful devices and without the need for huge data connections, and also clients of a business, needing lower specification machines, which leads to more possible clients for a business.

Chapter 5

Design

5.1 Scoping

The goal of the project will be, primarily, to make an effective web wrapper for the Tulip Python API. This would take requests from a client, that would include a file ID (for a file stored locally or elsewhere), and any other parameters such as how much data to return (depending on machine power / network connection) or type of reducing to do.

There will also be, at least, a basic interface that will allow for interactions to be made with the web endpoint, and the requirement that the system has benefits over not using the web endpoint (i.e. sending the file directly from the storage to the interface).

Ideally, the web endpoint will be made flexible enough to allow for a large proportion of the useful parts of the Tulip API to be made available, allowing for other developers to make their own client to interface with it. On top of that, ideally the JavaScript client would not be hardcoded but be a library that the interface would use in order to communicate with the web endpoint. This would allow for other developers to use make their own interface to use with the JavaScript Client and Web Endpoint.

Some features that will be included if time allows are the ability for the user to ask for an image as opposed to a network of nodes to be returned, if, for example, the machine they are using is underpowered or has a bad network connection, along with the ability to have the web wrapper communicate with an external data host. This would be useful if the users machines have little file storage.

Finally, it would be nice to have a fully-fledged interface to allow for a pleasant and easy user experience with all options available to the user. However, given time constraints, this may not be possible.

5.2 Requirements

At this point, the MoSCoW method was used to create a set of requirements for the system, with differing priorities.

Must Have:

- A web endpoint that calls the Tulip Python API

- To be able to deal with large data (that is too large to just send of the network and be visualised be the client without manipulation)
- A basic demo interface
- To have render-able data returned from the endpoint

Should Have:

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

Could Have:

- The ability to ask for an image to be returned if client is low power
- The ability to interface with an external data host (like S3) (could look into using Hadoop)

Probably Won't Have:

- A fully fledged and pretty interface

5.3 Design Decisions

Justification for including or not including the following features:

5.3.1 Web Endpoint

This is the most core part of the project, creating a web wrapper for the Tulip Python API. The justification for making this is to allow low spec clients (low processing power / RAM / storage) to visualise data quickly. Currently alternatives include using spreadsheet software which takes a long time and isn't very meaningful, or using graphing software, but that requires installing the software locally. Making a web wrapper for Tulip would allow a client to log in to a system and visualise a potentially massive network quickly and easily due to all of the processing of the graph done on a powerful server. Also, assuming that the software will be dealing with a massive data (a user could have access to TBs of data) then it is not feasible that any client could store that locally.

5.3.2 A Basic Interface

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

Better than without - It is necessary that in the vast majority of (or all) cases that the system works better using the web wrapper than just passing the nodes directly to the client. There should be an large increase in performance as the amount of data gets higher.

5.3.3 Making the web wrapper flexible

This is not essential for allowing the creation of the application, but if I am to make a program for this project, it would be highly preferable to create an API that other developers can hook into and use easily. This involves making all the API calls as open as possible and covering as much as possible, and also making sure it is well documented.

5.3.4 JS Library

In a similar vein, 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 JS 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.

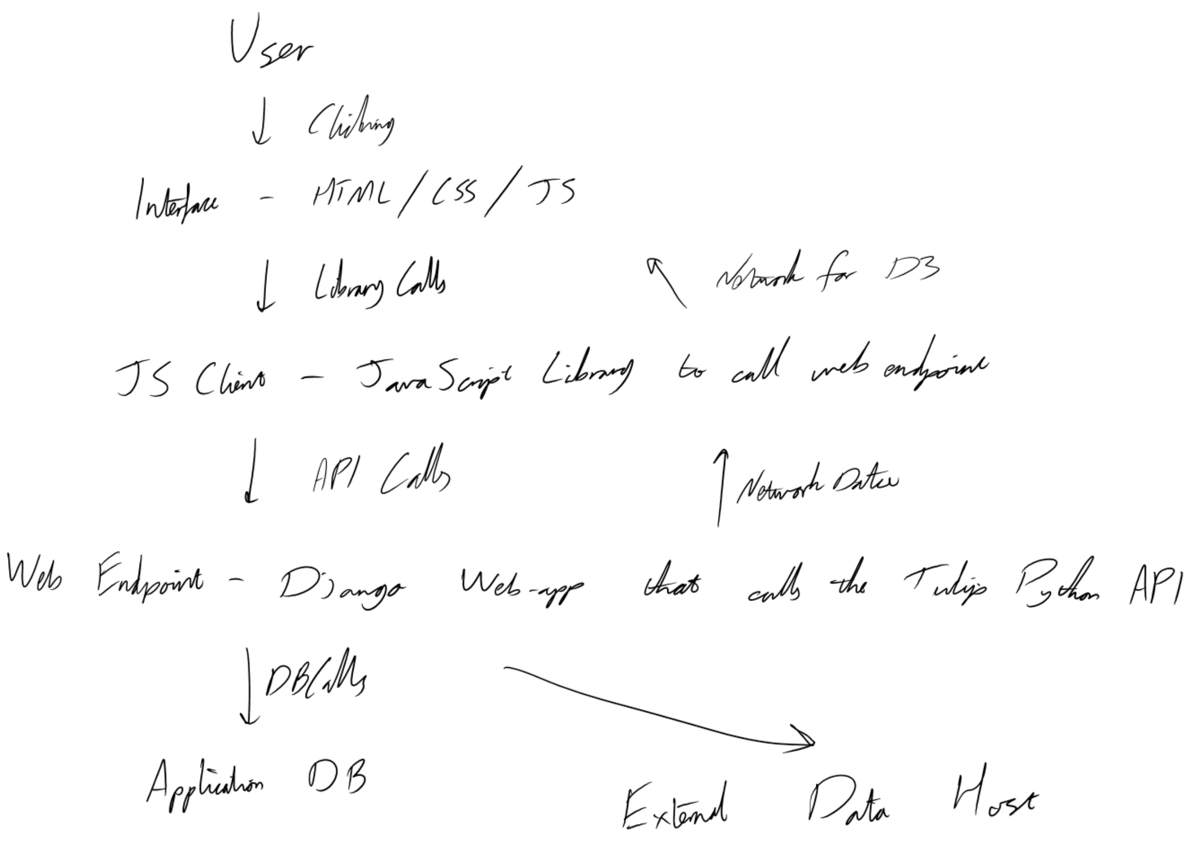
5.3.5 Return an image

As opposed to return a blob of data including nodes and edges, and ways to visualise them, it would be nice to allow for the user to select an option to just return an image of the graph. This would allow for the system to work on very low power machines or machines with a bad network connection.

5.3.6 Fully fledged interface

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

5.4 Technical Infrastructure



5.4.1 Sample Data Flow

Click Load Network
↓
Name Network
↓
Network details saved in system and network saved in external data host
↓
Click Open Network and select network to open
↓
Request sent to back end for network
↓
Database queried for network and location on external data host found
↓
External Data Host sends network data to back end
↓
Back end uses Tulip Python API to shrink the data and make it easier to send over a network
↓
New smaller network send over network to JS Client
↓
D3.js renders the network which is interactive

Chapter 6

Implementation

6.1 Created Software

What I actually made.

6.2 Problems Encountered

Some difficulties.

6.3 Evaluation

Some evaluation.

Chapter 7

Conclusion

7.1 Achievements

7.2 Recommendations?

Appendices

Appendix A

Appendices

Images and spreadsheets etc.

Bibliography