Universiteit Leiden
The Netherlands

Informatica & Economie

A comparison of algorithms for visualizing large social networks

in web-based environments

Robert Blinde

Supervisors:
Frank Takes & Antonio Barata

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

31/07/2019

Abstract

Multiple JavaScript libraries render network graphs in the web browser using the latest standards, such as *WebGL*, *HTML5* Canvas, and *SVG*. This thesis will examine three libraries – *D3*, *Ogma*, and *Cytoscape* – and compare their performance using two metrics: initialisation time and frames per second (FPS). Using randomly generated networks of different sizes and a real-world network, both metrics will be computed. The random networks are created using the NetworkX package, the real world network comes from the *KONECT* project. This thesis then recommends which of the three libraries to use in a real-world application and suggests further research.

Acknowledgements

I would like to thank my supervisors Dr. F. W. Takes and A. Barata from Leiden Institute of Advanced Computer Science (LIACS), Leiden University, for their support and guidance.

Contents

Chapter 1

Introduction

This chapter describes the context and motivation behind this thesis, presents the research question, and details the structure of the document.

1.1 Context

Today, every company or organisation produces more than just products or services: they generate data. Examples of this data include customer information, inventory, financial information, products and services sold, and employee salaries. Most of this data remains unused in the company's data warehouses, if it is retained at all. Challenges lie in summarising, visualising, and describing the information contained in the data [13].

As an example consider *Orbis*, a database containing company data and financial information from 300 million companies worldwide. A portion of this data concerns relationships between companies. In this case, companies are considered related if one company owns more than 50 percent of another company or if they have board interlocks. Board interlocks are defined as the number of shared directors across companies [9]. To describe the data in Orbis, a network (or graph) is created from the database. This network can then be analysed using a variety of methods. Well-known methods in network analysis are community detection, link prediction, and the computation of centrality measures. Community detection identifies clusters that are highly connected to each other but sparsely connected to the rest of the network [29]. Link prediction tries to calculate the probability of an edge formation between two nodes [20]. Node centrality measures the importance of a node with respect to the other nodes based on the structure of the network and can be computed in multiple ways, such as degree centrality, closeness centrality, or betweenness centrality [24].

Visualising networks and exploring them helps users comprehend the results of the aforementioned methods. Gephi [2] is an example of an open-source network visualisation tool. Network data, such as edge lists, can be imported and will then automatically be visualised. Properties of a node can be mapped to its visual attributes.

For example, the number of employees in a company can be mapped to the size of the node, and its industry can be mapped to the colour of the node. The process of importing data and choosing the mapping of data properties to visual attributes – in particular, the positioning and rendering of the nodes in the plane – takes time, because the underlying algorithms are time- and resource-intensive.

The following example illustrates network visualisation. A dataset contains information on the 5,000 biggest companies worldwide based on revenue. After narrowing the selection only to the companies in The Netherlands whose main activity is not in the industrial sector, the resulting graph may look like the one in Figure 1.1. In this example, the company's main activity is mapped to the colour of the node. The number of relationships between the companies is mapped to the weight (thickness) of the edge. A noteworthy observation in the example is the fact that the two ING companies are not directly related but are still accessible to each other via the NN GROUP company.

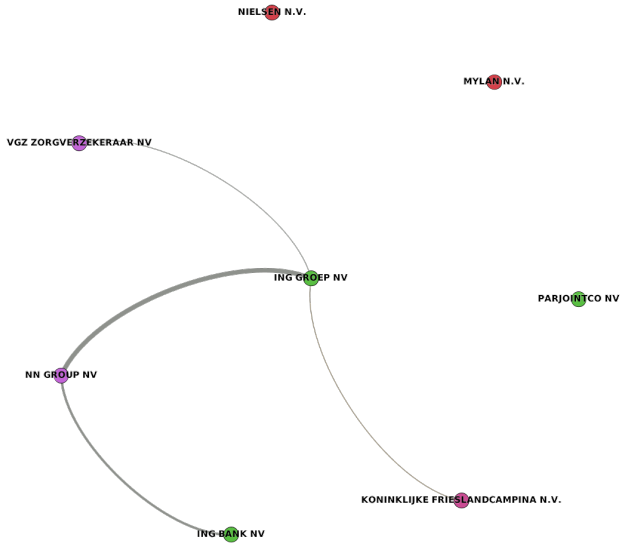

Figure 1.1: Graph of Dutch companies in non-industrial branch.

A web platform for network analysis has multiple benefits compared to offline software. The software can be used on any computer, independent of the operating system. Only a modern browser is required. Furthermore, no software or *plug-ins* must be downloaded and installed in order to use the platform. A possible disadvantage of using a web-based visualisation application is limited performance; not all web standards can access the full potential of the computer's graphics card.

1.2 Research Question

This thesis will describe the methodology and scalability questions related to a web application for network analysis and provide a recommendation for which JavaScript library to use when developing such an application. The best JavaScript library for graph visualisation will be determined experimentally. Therefore,

the research question can be formulated as follows:

> *How do we ensure that a web application for network analysis is interactive, fast, usable, and scalable?*

1.3 Thesis Overview

The structure of this thesis will now be outlined briefly. Chapter 2 will give background information and review related work in the areas of networks and network visualisation. In Chapter 3, the JavaScript libraries that are being compared will be described in detail. Chapter 4 describes the methodology used to conduct the experiments. Chapter 5 describes the setup of the experiments in detail and presents the results. Finally, Chapter 6 concludes the thesis with a discussion of the findings and suggestions of possible topics for future research.

Chapter 2

Background and Related Work

This chapter illustrates the most important concepts and definitions needed to understand the subsequent chapters. The first section describes networks in general. The next will explain network analysis and its related activities. The last concept that will be explained is network visualisation.

2.1 Networks and Network Creation

Many systems can be depicted as networks (or graphs): internet routers connect users, highways and public transportation connect cities, social platforms connect users and their online friends, and ownership and board interlocks connect organisations. Formally, a graph G is a structure that consists of vertices (V) and edges (E), or $G = (V, E)$ [4]. Vertices (also known as nodes or points) are connected through edges (also known as arcs or lines), implying a relationship between them. Two vertices are called neighbours when they are connected by an edge. Edges in a graph can either be directed or undirected. For example, if vertices represent users of a social media platform, and there is an edge when two users are "friends", then this graph is undirected, because user A can be friends with user B only if user B is also friends with user A. On the contrary, if the edge from user A to user B represents user A following news updates of user B, then this graph is directed.

An example of an undirected graph can be found in Figure 2.1. This graph depicts character co-occurrence in *Les Misérables*. Nodes are positioned by simulating forces so that edges are of approximately equal lengths and overlap as little as possible. A more detailed description of this method will be provided in Section 2.3.1. The characters are represented by the vertices. If two characters appeared in the same chapter of the book, an edge is drawn between their nodes. The thicker the edge, the more often the characters appeared together.

Assuming the existence of two tables, *entities* and *relationships*, the general process of transforming tabular data into network graphs is as follows. First, for each row in the *entities* table, a node is created. Then, visual attributes corresponding to one or more columns are added. For example, the colour of the node maps to a categorical datatype. Next, edges between the existing nodes are created using the *relationships* table. Weights

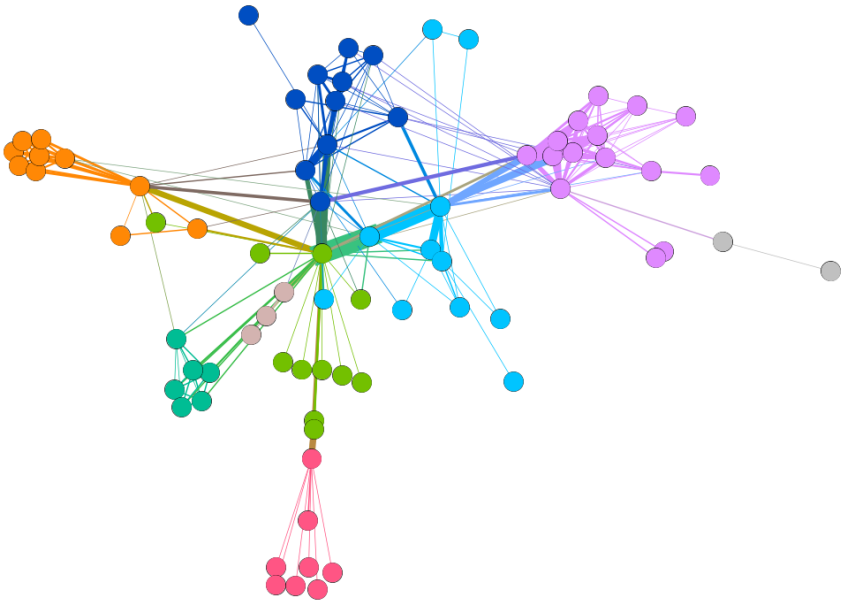

Figure 2.1: Co-occurrence of characters in *Les Misérables*.

can then be added to the generated edges, indicating the number of connections between the nodes in the data. Finally, the resulting network graph can be aggregated or divided into sub-networks. Aggregating nodes means grouping multiple nodes together and treating them as one. Dividing networks into sub-networks allows for creating and organising meaningful snapshots of a large network based on different perspectives.

2.2 Network Analysis

Interest in network analysis, more specifically social network analysis, has increased rapidly over the past two decades [3, 26]. Social network analysis can be defined as the process of studying relationships between social bodies [25]. Therefore, it is not a theory but a methodology for the examination of social networks, and it can be used in multiple fields of research. Scientists can specialise in a certain type of network, such as friendship networks or corporate networks. The analysis of networks originated in mathematics and graph theory, which allows for quantitative measurements such as centrality or density [25].

Analysis of networks does not focus solely on individual nodes, but also examines the relationships between nodes and tries to answer how these relationships are influenced by the network itself. However, by their nature, networks are difficult to understand. For example, networks can evolve over time as nodes are created or disappear. Edges between the nodes can either be directed or undirected and can be weighted or contain additional properties [27]. A real-world example of this is the World Wide Web: web pages are created, updated, and deleted continuously.

Networks can be analysed at the level of the individual node or the network as a whole. When investigating individual nodes in a network, the centrality metric is the most commonly used [24]. Centrality measures the importance of nodes within a network. Multiple measures, each with its own definition of “importance”,

have been developed. Examples include degree centrality, betweenness centrality, and closeness centrality. Popular metrics for analysing the network as a whole are average shortest path, clustering coefficient, and connectedness. The average shortest path denotes the average distance between two nodes in the network.

The undirected graph in Figure 2.2 illustrates a centrality metric. Vertices are coloured based on the betweenness centrality of each vertex, from smallest (red) to greatest (blue). To compute betweenness centrality, all shortest paths are first calculated for the entire graph. A shortest path is the path of minimal length between two nodes. The betweenness centrality of a vertex is the relative number of shortest paths that go through the vertex [17], which is depicted in Figure 2.2.

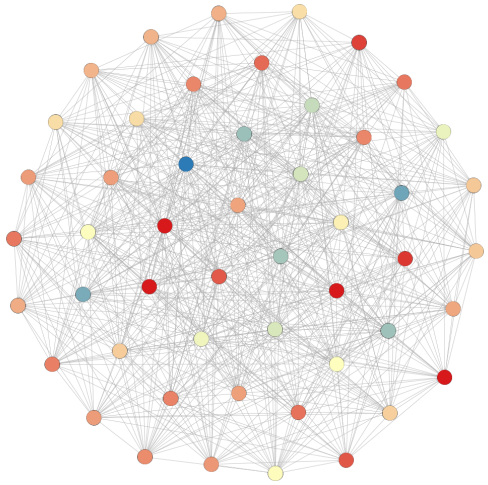

Figure 2.2: Betweenness centrality in an undirected graph, from smallest (red) to greatest (blue).

2.3 Network Visualisation

A visual representation of a network is an image containing the vertices and edges of the network. Such drawings are important instruments for understanding the network, communicating results of network analysis, and visually comparing the structures of different networks. Network data is explored by creating different layouts for the same network and mapping data properties to the visual attributes, such as colouring the nodes based on one of the aforementioned network analysis metrics or mapping the strength of the relationship to the width of the edges. The same network data can be represented with different layouts, which can lead to interpretation errors [21]. The only important factor is which vertices are connected and through which edges. That being said, the placement of vertices and edges within a visualisation can influence its understandability and usability.

A wide variety of software packages, such as Gephi [2] or Osprey [7], allow for visualisation and manipulation of networks. These tools offer different layout algorithms. Users can filter data based on node or edge properties and, as a result, create new networks. These node or edge properties can also be used to change the appearance of the vertices and edges. Different statistics and metrics regarding the graph can be computed. Finally, the network visualisations can be exported to different image formats.

2.3.1 Visualisation Algorithms

An algorithm for the visualisation of networks typically takes in network data (for example, in the form of an adjacency list) and, given certain parameters, outputs an image in which nodes are placed into a certain layout that satisfies those parameters [8]. Thus, for every network visualisation algorithm, many different graphical representations can be output. Parameters can be adjusted to create the desired outcome: a graph drawing with high readability. Readability is the quality that measures how well a diagram communicates its meaning; higher readability results in greater usefulness and clearer communication.

Different visualisation algorithms result in different graph drawings. These results can be assessed using multiple quality measures and performance metrics. The most popular quality measure is the number of edges in a drawing that overlap. When a drawing has no crossing edges, it is planar, which is ideal. To name a few other quality metrics, smaller drawings are preferred over larger ones, symmetry is desired, and short edges represented by straight lines are favoured [8]. Typically, the more criteria or quality measures are to be optimised, the longer it takes for the algorithm to complete. Algorithms can be made to run faster by easing certain criteria. Some algorithms try to automatically account for quality metrics; however, readability is not fully objective, and graph drawings should be adapted to the people viewing them. To illustrate, the graph derived from a cube is planar. However, it is commonly depicted with edges crossing, as shown in Figure 2.3.

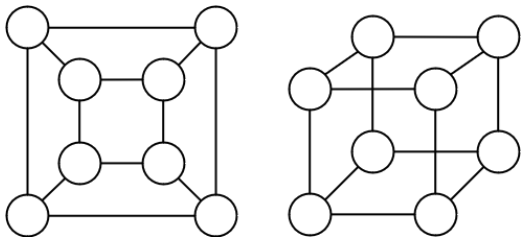

Figure 2.3: Different graph drawings of a cube.

Force-directed layout algorithms, introduced by Eades [12], aid in understanding the structure of network data by positioning nodes in such a way that edges connecting them are of approximately equal length and overlap as little as possible. Force-directed algorithms are often of high quality due to their built-in quality measures and therefore result in easy-to-read visualisations. These algorithms are based on a physical system in which nodes and edges are assigned different forces. Typically, nodes have an electric charge and will push other nodes away, while edges are represented by springs that pull connected nodes together. The main algorithm consists of a loop, and at every iteration, both forces are calculated and used to update the positions of the nodes. This process continues until the graph is "stable", meaning the forces require minimal energy. Force-directed layouts only take the network structure into account; node and edge properties are untouched. Figure 2.1 is an example of a graph drawn using a force-directed layout.

Force-directed layouts have some disadvantages. These algorithms often start with randomised positions and settle in a local minimum of the (various) objective(s). This minimum is typically worse than the best-case scenario, resulting in a decrease of the overall quality of the visualisation. Another disadvantage of force-directed algorithms is their runtime; at every iteration, all nodes must be visited in order to calculate the forces. This results in a complexity of $\mathcal{O}(N^2)$, where N is the number of nodes. Using the Barnes-Hut algorithm [1],

this can be reduced to $\mathcal{O}(N \log N)$.

2.3.2 Algorithm Comparison

There is no "one fits all" algorithm for creating high-quality and readable network visualisations. To assess the quality and performance of existing algorithms, large-scale experiments must be conducted, preferably with real-world graph data.

Brandenburg et al. [6] compared five well-known different graph drawing algorithms for the creation of undirected graphs. All algorithms used either a force-directed implementation or a minimising cost function to find a best-case solution. The researchers compared the algorithms using multiple metrics, including runtime on a *SPARCstation* 10, the average length of the edges, and how many edges overlapped. These tests were run against randomly generated graphs. All algorithms performed as expected, but no algorithm stood out. Algorithms producing higher-quality graph images had greater run times. The number of nodes and edges affected run times exponentially.

Di Battista et al. [10] presented a comparison of three algorithms in which edges bend at 90-degree angles. The most common domain for such drawings is database visualisation software. The three algorithms were tested against real-life networks, such as entity-relationship diagrams, and randomly generated variations thereof. Algorithms were analysed by defining a set of quality metrics, such as runtime, overlapping edges, and size of the resulting image. Again, no algorithm is a clear winner: each performs better in different categories. The algorithm with the fewest operations had the fastest runtimes but lacked in other quality measures.

Knowledge about the domain from which network data is derived is a key factor in choosing the right graph drawing algorithm. Another important factor is the readability of the resulting image; improving readability often increases run times due to a higher number of operations or built-in quality measures. Runtimes can be decreased at the cost of the overall quality of the visualisation.

Chapter 3

Libraries

To answer the research question, three JavaScript libraries – *D3*, *Ogma*, and *Cytoscape* – are assessed and compared. The first section describes the reasoning behind the choice of the three libraries, and the subsequent sections detail the workings of each library.

3.1 Rationale

These three JavaScript libraries were chosen because they promise to render large graphs in the browser, have good documentation or a large community of developers, and, most importantly, provide algorithms to create a graph with a force-directed layout that eventually comes to an equilibrium. Both the D3 and Cytoscape libraries are open-source, and the Ogma library is a commercial product. The *Sigma* library would be another suitable candidate, except that the algorithm implementing the force-directed layout is continuous and thus does not stop automatically. Therefore, no accurate performance measures can be determined for this library.

3.2 D3

Data-Driven Documents, or D3, is an open-source library for creating data visualisations, such as diagrams, matrices, or graphs, in web browsers. It is built upon standard technologies: *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS), and *Scalable Vector Graphics* (SVG). D3 focusses on ease of use rather than performance by directly updating the *Document Object Model* (DOM). The DOM contains the hierarchical structure of a webpage. By selecting an element in the DOM, the developer can bind input data to this element, constructing new elements in the document tree. Updating data thus results in the creation and removal of DOM elements. By default, this happens instantaneously, but it can be animated to run over time. Various helper modules exist to handle common visualisation tasks [5].

One of these helper modules, called d3-force, allows for the creation of force-directed layouts. A velocity Verlet [28] numerical integrator is implemented, simulating physical forces on the "particles" (the nodes). The Verlet integration is a method used to integrate Newton's equations of motion, and it describes the motion of a physical structure over time. The D3 simulation is a simplified version of the velocity Verlet. Both the mass of the particle and the time steps are considered constant and equal to one. So, $\Delta t = 1$ for each step, and mass $m = 1$ for all nodes. At each iteration (Δt), the positions of the nodes are updated using three primary forces: the force from all other nodes, the force between two connected nodes, and a force pulling nodes towards the middle of the visualisation.

Alongside the three forces, D3 uses "friction" and an *alpha* parameter. At every iteration friction decreases the distance a node has to move to its new position. The alpha parameter gradually decreases, reducing the strength of the forces between each iteration. When alpha is below a certain threshold, the force-directed layout will stop, resulting in a static visualisation. Without this approach, vertices would never stop moving.

3.3 Ogma

Ogma is a graph visualisation library created by the company Linkurious, based on *Web Graphics Library* (WebGL). Not all browsers support WebGL, so older browsers will fall back to HTML Canvas, which will result in decreased performance. The library is solely designed for the rendering of graphs where nodes, edges, and their properties can update in real time. A graph is an object consisting of entities (nodes) which are connected by edges. A constructor function binds this object to an HTML container. When initialising a graph, different options, such as the layout, can be passed.

Ogma provides a force-directed layout based on the ForceAtlas2 algorithm [19], originally developed as a standard solution as part of Gephi [2]. In this algorithm, nodes push nodes away and edges attract other nodes. Different options, such as the Barnes-Hut simulation, change the forces or how they are simulated. The computations are delegated to a so-called web worker to aid in optimisation. Web workers allow for running JavaScript code in background threads; these can perform certain tasks without interfering with the user interface and are thus more efficient. Because Ogma is a commercial product, details about the workings of the library have not been disclosed.

3.4 Cytoscape

Cytoscape [16] is an open-source graph theory library built for both the browser and the *Node.js* server. On the server, it uses another package to create image outputs; for browsers, it relies on HTML5 Canvas to render graphs. The architecture of Cytoscape consists of two main elements: the core and the collection. The core is an instance of a Cytoscape graph and contains functions to retrieve certain collections. Collections are used to filter nodes and edges based on their properties.

The library provides a Compound Spring Embedder (cose) layout which uses physics simulation to visualise the graph. It was implemented based on the article by Dogrusoz et al. [11], who eased constraints on previously known algorithms and based their algorithm on the traditional force-directed layout scheme with some added extensions regarding nested graphs. The algorithm consists of an initialisation phase followed by the three main phases. In the initialisation phase, threshold values and the initial positions of the nodes are calculated. The first main phase lays out a skeleton graph, and the second introduces the nested graphs. The final phase stabilises the graph, refining the positions of nodes.

Cytoscape has implemented the algorithm as follows. At each iteration, three forces are calculated: the forces pushing nodes away, the forces pulling connected nodes together, and the force pulling the nodes towards the centre of the viewport. Next, the forces are propagated from the parent nodes to their children, and their children, and so on. This is achieved by converting the graph into a *queue*, which allows for easier and more efficient tree traversal. This queue is stored in an "info" variable that all force-calculating functions can access, which prevents repeating the same calculations. After all forces are calculated and propagated, the positions of the nodes are updated. When the forces are in equilibrium or the maximum amount of iterations is reached, the layout is stopped, and the graph is drawn.

Chapter 4

Methodology

This chapter describes the methodology used to assess the JavaScript libraries. First, the creation of realistic graph data to generate the networks is explained. Then, the process by which the performance of each library was measured is outlined.

4.1 Network Generation

Before the visualisations can be created, the randomised and realistic network data they will display must be generated. In theory, this could be achieved by creating a fixed number of nodes and randomly creating edges between the nodes until the desired amount is reached. This method of generating random graphs is called the Erdős–Rényi model [14]. In this model, each edge has an equal probability of being present or absent, independently of the other already-existing edges. This model of random graph generation is easy to implement, but it has one serious drawback: the resulting graph is inappropriate for modelling real-life phenomena. Because each new edge has an equal chance of existing, the graphs will have little to no clustering, unlike real-world networks.

To overcome this problem, a different technique for graph generation can be used: an algorithm using the preferential attachment model [23]. This type of algorithm works as follows. After the desired number of nodes are generated, the edges are created, much like in the Erdős–Rényi model. The key difference, however, is the likelihood of edges connecting to already-formed edges. In the example in Figure 4.1, the first edge is formed between the top left and bottom left nodes. When the node in the centre is selected, the probability of connecting to one of the left nodes is higher. In this case, an edge will be formed with the bottom left node. The bottom left node now has a degree of two, meaning the next node will be more likely to form an edge with that node. Generating graphs using this model creates more realistic networks than the Erdős–Rényi model, because clusters are more likely to exist. Therefore, the networks used in the experiments will have a greater resemblance to their real-life counterparts.

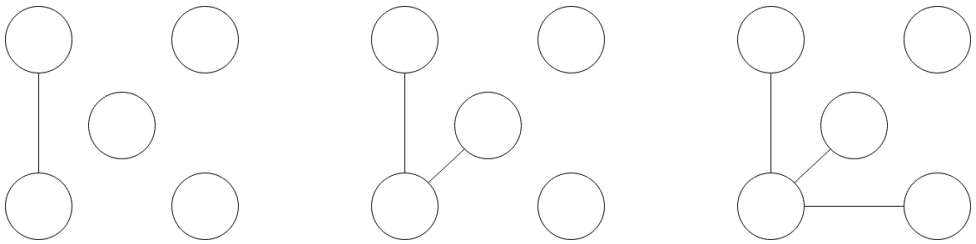

Figure 4.1: Generating edges based on the preferential attachment model.

4.2 Measuring Performance

Performance of the libraries is measured by comparing initialisation times and frame rates using the generated network data from Section 4.1. Initialisation times are used to assess the speed and scalability of the libraries; usability and interactivity are assessed by measuring frames per second (FPS). Both benchmark metrics are important for visualisations in web browsers, as longer page load times have been shown to increase the number of users who prematurely leave the page [22], while a sufficient frame rate is necessary for smooth interaction and animation. Each library will render graphs of different sizes. For each visualisation, the initialisation time, the time from page load to initial display of the visualisation, and average frame rate are measured. Initialisations are repeated ten times and averaged.

Interaction is simulated by randomly dragging a number of nodes using the mouse. Every time the browser requests a redrawing of the page, a timestamp is created – independent of the system clock – and stored. Frame rates can then be computed by measuring how many re-renders occur in a second. Analysis is repeated over an increasing number of data points ranging from 100 to $5,000$. The benchmarks are performed on an MSI laptop with a quad-core 2.5 GHz processor and 16GB RAM running Ubuntu 18.04. The visualisations are rendered in the Firefox browser, version 64.0.

Chapter 5

Experiments

This chapter starts by explaining how the experiments were set up, to facilitate replication. The next section explains the observations and results obtained from the conducted experiments. Finally, the results are discussed.

5.1 Experimental Setup

The first subsection will describe and explain the software used to conduct the experiments, followed by a general method to implement the JavaScript libraries.

5.1.1 Software

To clarify references to the software used in this chapter, the software will first be described and explained.

NetworkX

For the generation of realistic graph data, a Python package called *NetworkX* [18] was used. This package can be used for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. Its features include data structures for graphs, standard graph algorithms, network structure and analysis measures, and generators for random graphs. The latter feature was used during the data generation step. Using NetworkX, random graphs were created using the preferential attachment model as discussed in Section 4.1. More information can be found at `https://networkx.github.io/`.

5.1.2 Implementation

The random network graphs were generated using NetworkX. A simple *Command Line Interface* (CLI) program was created which takes in two arguments: the number of nodes (N), and the number of edges to attach from a new node to existing nodes (M). The NetworkX function returns a graph, which is then exported to a *JavaScript Object Notation* (JSON) file using the built-in node-link graph exporter. For each variation of N and M, a JSON file containing an undirected graph is created.

Aside from the randomly generated graphs, a real-life network named *Sister Cities*, from the KONECT project, was evaluated. Using a real-life network allows for better evaluation of the JavaScript libraries than only using random network data. The Sister Cities network, obtained from WikiData [15], is an undirected network containing cities around the world that have a relationship with one another. This relationship is an agreement between two cities to promote the culture and economy in both. A vertex represents a city and an edge represents the existence of a relationship between two vertices. The network contains $14,274$ vertices and $20,573$ edges, and its density is $2 \cdot 10^{-3}$. This network was chosen because its size is close to that of the largest random network tested (10,000 vertices).

Using Python's built-in *SimpleHTTPServer*, a simple web host was simulated locally. For each of the three JavasScript libraries, all files required for implementation were created. These were an HTML file, which is the entry point to the visualisation, a JavaScript file containing the library, and a second JavaScript file implementing the library. This latter file made use of JavaScript's built-in `fetch` function to load the JSON files containing the graphs.

The initialisation time was measured by writing a simple timer function. After the data was loaded, an `init` function set the timer and started visualising the graph. When the library finished rendering the network, an event was emitted, and the timer was stopped. The difference between the two times was calculated and logged to the browser's console. This was repeated ten times, and the results were averaged. The frame rate was measured similarly: after the "done rendering" event was emitted, a `tick` function was executed that ran every time the browser requested a re-render of the page using the `requestAnimationFrame` function in JavaScript. When this happened, the frame rate was calculated by counting how many re-renders occurred during a one-second interval and storing this value in a list. After 60 seconds, the average of all frame rates in the list was calculated and logged to the browser's console. The scores for each library can be found in Section 5.2.

5.2 Results

In Table 5.1, the initialisation times for the aforementioned Sister Cities graph are presented. The first row contains the three JavaScript libraries, and the second row shows the corresponding initialisation times in milliseconds along with the standard deviation. Table 5.2 lists the FPS for the Sister Cities graph. Again, the

first row contains the libraries, whereas the second row shows the FPS.

Below the tables, in Figures 5.1–5.4, the initialisation times for graphs of sizes $N = 100$, $N = 500$, $N = 1,000$, and $N = 5,000$ are displayed for each library. The vertical axis represents the time in milliseconds it takes to render a network graph, and the horizontal axis represents M, the number of edges to attach from a new node to already-existing nodes.

In Figures 5.5–5.8, the FPS for graphs with sizes $N = 100$, $N = 500$, $N = 1,000$, and $N = 5,000$ is shown. Here, the vertical axis represents the FPS and the horizontal axis, again, represents M. Ogma has the highest overall FPS which stays over 50 for all visualisations. D3 and Cytoscape gradually decrease when N and M increase.

Figures 5.9–5.14 are heatmaps representing the initialisation times for each value of M. Each (i, j) cell represents i number of nodes for algorithm j, and the label displays the initialisation time plus or minus the standard deviation. The lighter the colour of the cell, the higher the initialisation time. Cytoscape is the fastest, only matched by Ogma for size $N = 500$.

Figures 5.15–5.20 are heatmaps representing the FPS at fixed values of M. Each (i, j) cell represents i number of nodes for algorithm j. All three libraries also rendered a graph with the size of $N = 10,000$ and $M = 2$. On average, it took Ogma, D3, and Cytoscape 85.8, 130.3, and 8.5 seconds, respectively.

Ogma	**D3**	**Cytoscape**
$118,350.2 \pm 585$	$251,133.3 \pm 1104$	$8,899.6 \pm 523$

Table 5.1: Initialisation times for *Sister Cities* graph (ms).

Ogma	**D3**	**Cytoscape**
58.3	3.4	2.4

Table 5.2: Frames per second for *Sister Cities* graph.

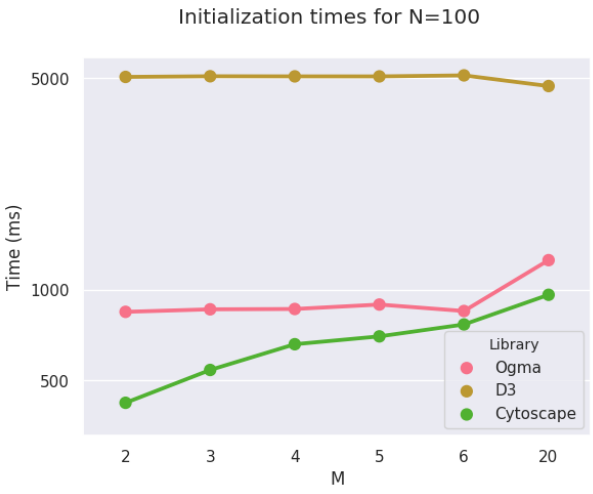

Figure 5.1: Initialisation times for $N = 100$.

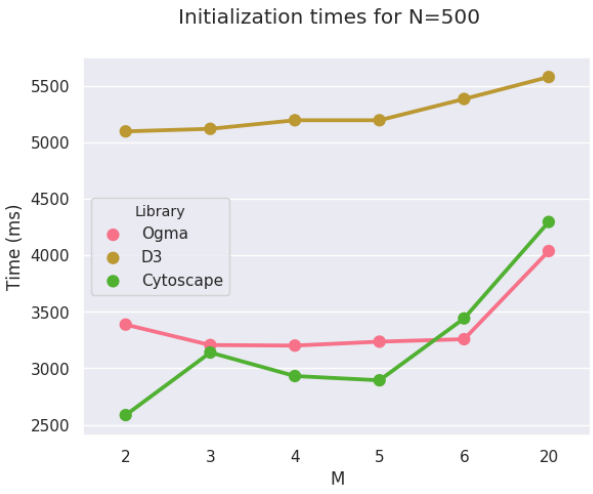

Figure 5.2: Initialisation times for $N = 500$.

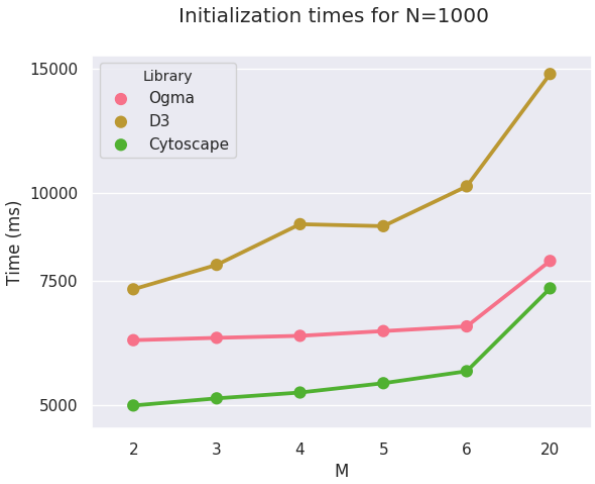

Figure 5.3: Initialisation times for $N = 1,000$.

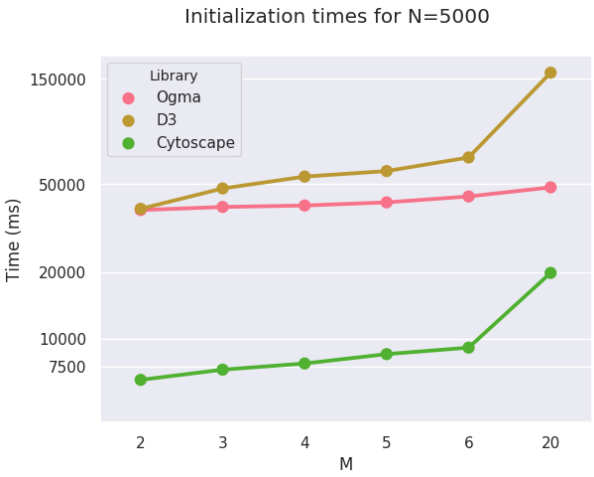

Figure 5.4: Initialisation times for $N = 5,000$.

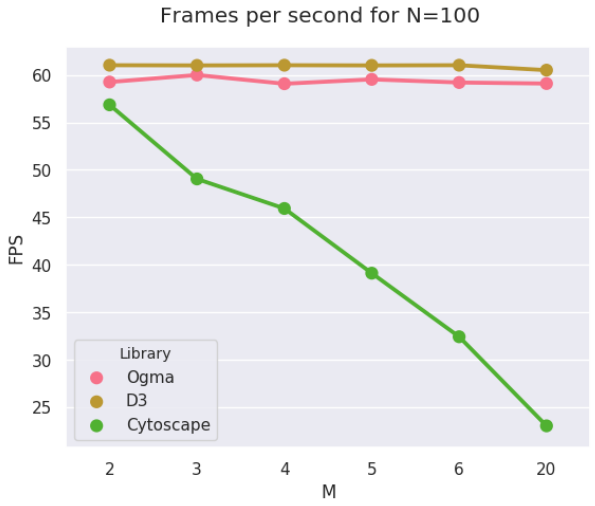

Figure 5.5: Frames per second for $N = 100$.

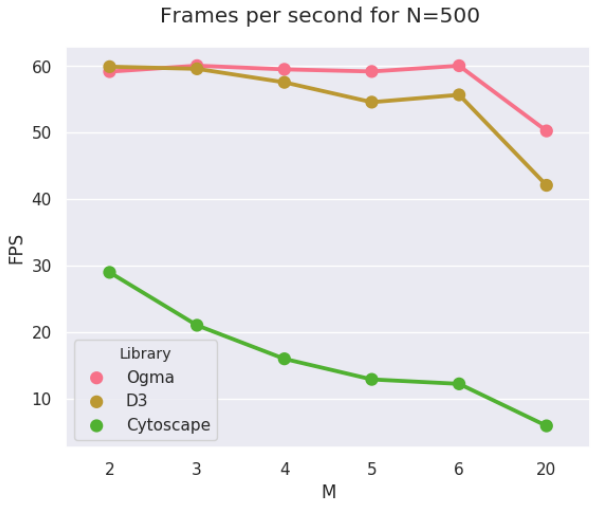

Figure 5.6: Frames per second for $N = 500$.

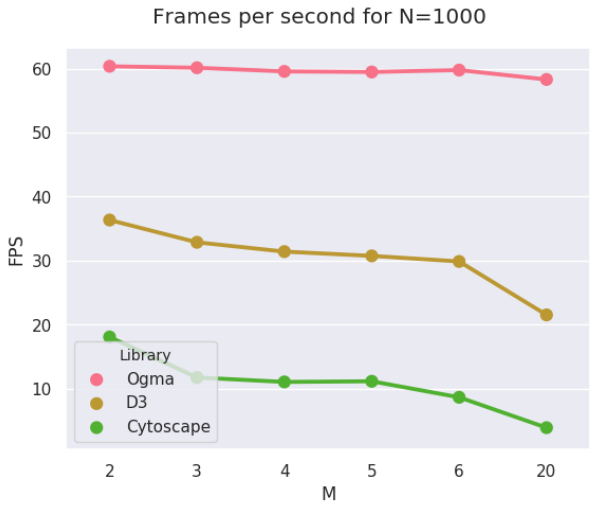

Figure 5.7: Frames per second for $N = 1,000$.

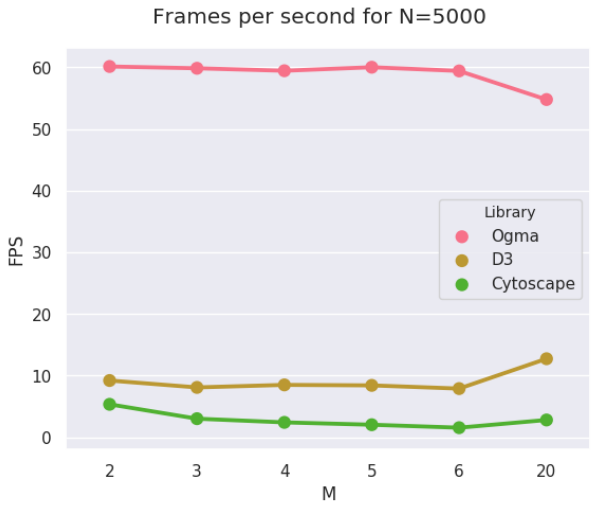

Figure 5.8: Frames per second for $N = 5,000$.

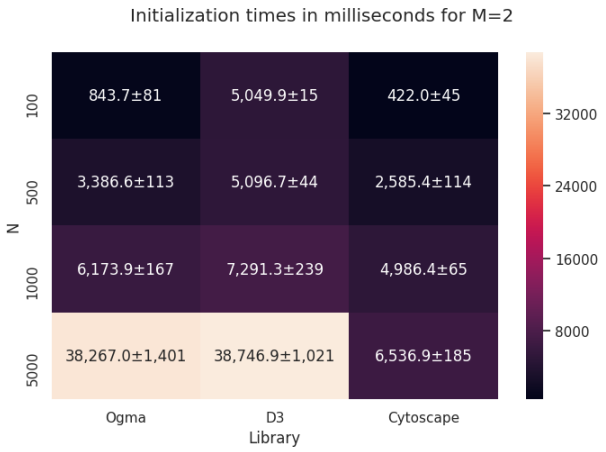

Figure 5.9: Heatmap of initialisation times for $M = 2$.

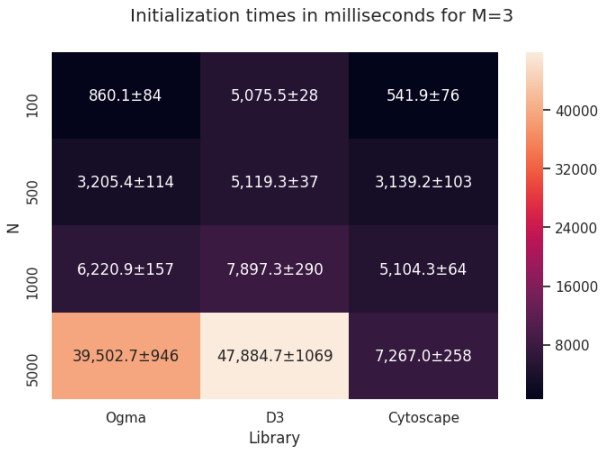

Figure 5.10: Heatmap of initialisation times for $M = 3$.

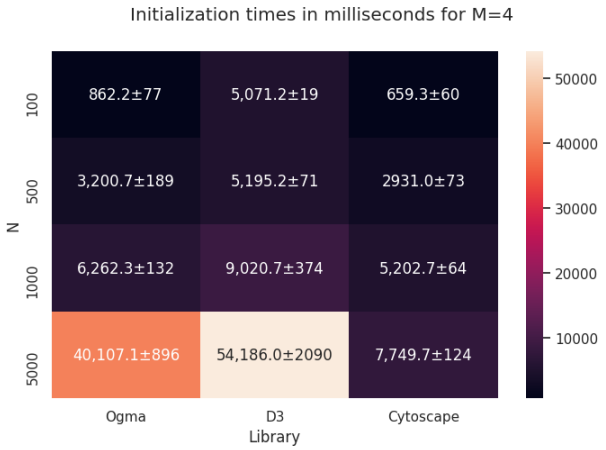

Figure 5.11: Heatmap of initialisation times for $M = 4$.

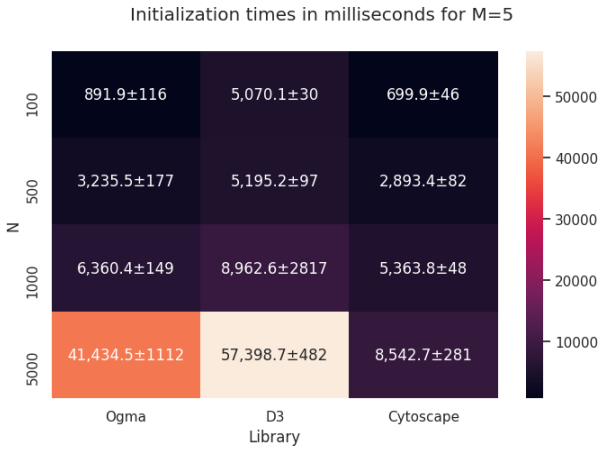

Figure 5.12: Heatmap of initialisation times for $M = 5$.

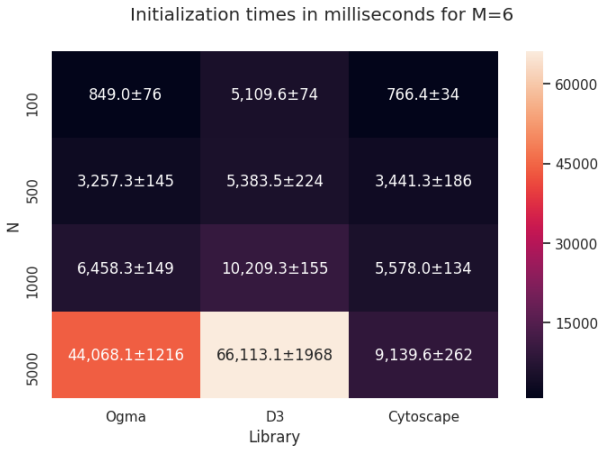

Figure 5.13: Heatmap of initialisation times for $M = 6$.

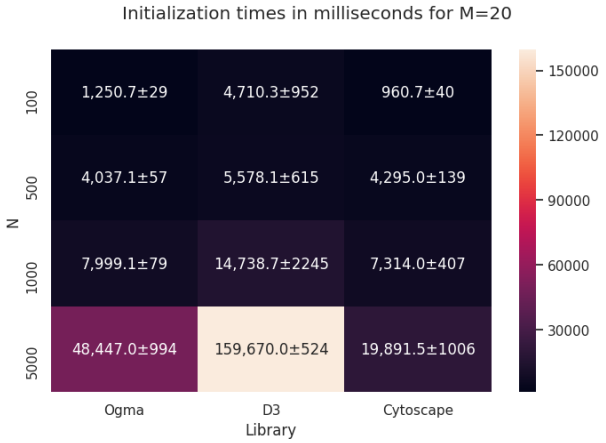

Figure 5.14: Heatmap of initialisation times for $M = 20$.

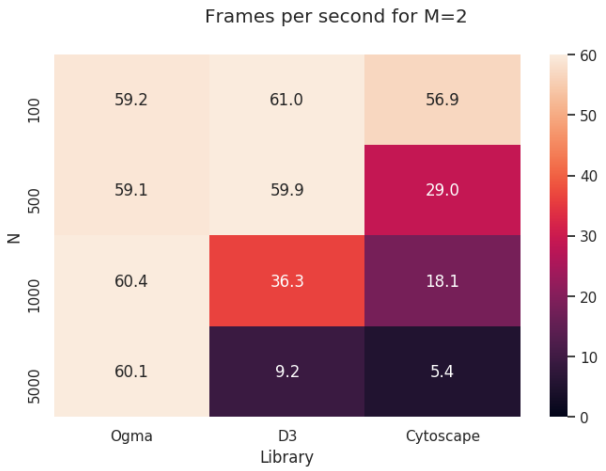

Figure 5.15: Heatmap of frames per second for $M = 2$.

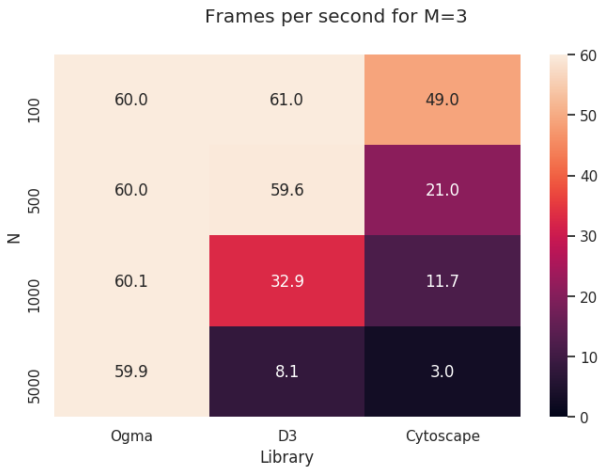

Figure 5.16: Heatmap of frames per second for $M = 3$.

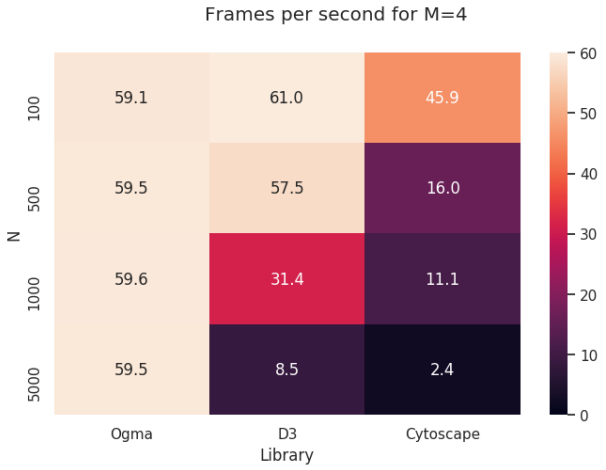

Figure 5.17: Heatmap of frames per second for $M = 4$.

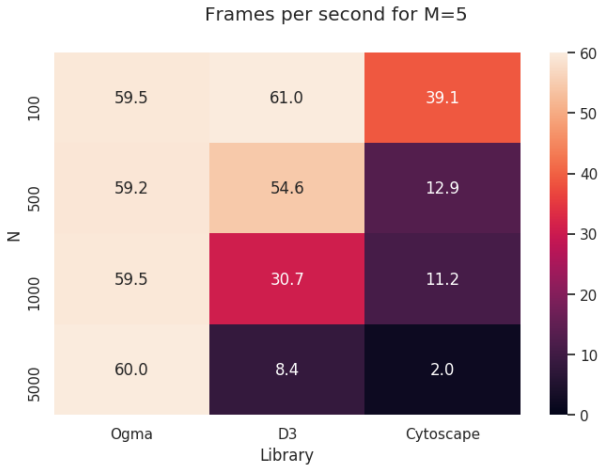

Figure 5.18: Heatmap of frames per second for $M = 5$.

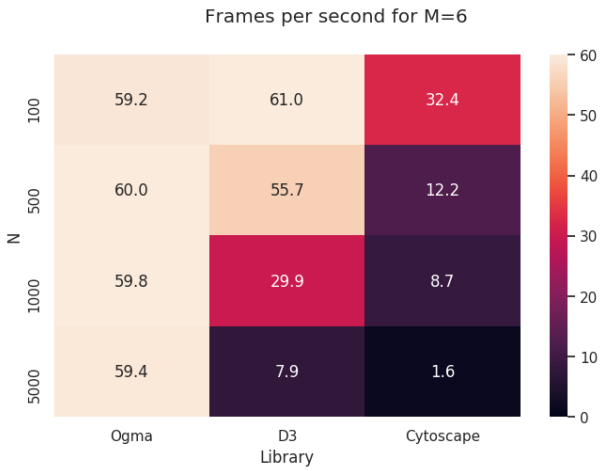

Figure 5.19: Heatmap of frames per second for $M = 6$.

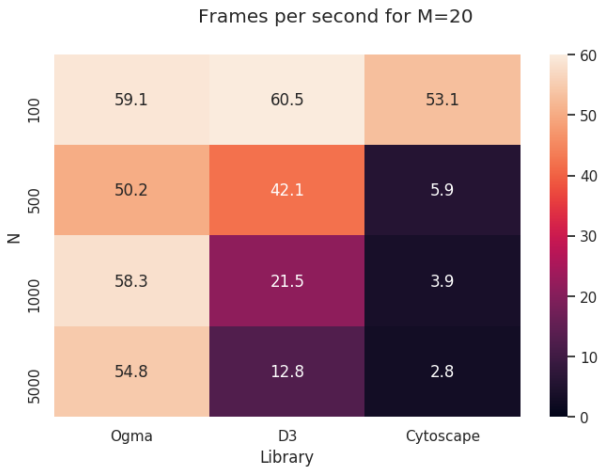

Figure 5.20: Heatmap of frames per second for $M = 20$.

5.3 Discussion

For all visualisations, the initialisation time from page load to a rendered graph is slowest for the D3 library. Its overall performance, or lack thereof, can be attributed to the use of SVG and the immediate drawing of the graph at the start. HTML5 Canvas is generally faster and uses less memory than the older SVG standard. Rendering nodes and edges at the outset decreases performance: at each iteration of the algorithm, the forces changing the nodes' positions are calculated and the visualisation is updated. Rendering only when the final positions are computed, as Ogma and Cytoscape do, would increase D3's performance. In addition, D3 is not solely focussed on graph drawing, and optimisation for large graphs is not a priority.

The Ogma and Cytoscape libraries yield similar results for graphs up to size $N = 1,000$. Larger graphs are rendered more rapidly with Cytoscape. Both libraries do not render anything until the final positions of all nodes are calculated, improving load times. The difference between them thus lies in the implementation of the force-directed layout. While Cytoscape is faster, Ogma's visualisations are more readable. For every library, there exists a positive linear relationship between the number of edges and initialisation times: the more edges, the longer it takes for a network visualisation to complete.

The FPS for all graph sizes is highest for the Ogma library. D3's FPS for the graphs with sizes of $N = 100$ and $N = 500$ is close to Ogma's results but drops to less than 20% of Ogma's performance in larger graphs. Ogma's high frame rates can be credited to its of the latest WebGL standard. Based on OpenGL, it provides an Application Programming Interface (API) that allows for rendering of 2D and 3D elements using JavaScript without the need for plug-ins. The code is run on the computer's graphics card (GPU) as opposed to the processor (CPU). This results in faster execution times and thus higher frame rates. The reason D3 is faster than Cytoscape lies in their different implementations: SVG versus HTML5 Canvas. While Canvas is faster for the initial drawing, redraws are slower because no information about previous drawings is stored. SVG uses a tree-like structure to store elements (nodes and edges) inside the webpage itself. Hence, updating positions does not require an entire redraw; rather, only the affected nodes are targeted, increasing FPS.

Chapter 6

Conclusion and Future Work

All three libraries provide an "ending" force-directed layout algorithm to render graphs. They also include functions to watch for user input (mouse movement) and execute internal functions, such as zooming and moving nodes around. These algorithms draw graphs in such a way that edge overlapping is minimal, and edges are approximately equal in length. This increases readability and thus the overall usability of the rendered visualisation. To test the libraries for their interactivity and usability, FPS was measured during an interval of 60 seconds in which nodes were dragged across the screen. Whereas the FPS of the D3 and Cytoscape libraries gradually decreased as the number of nodes and edges increased, the Ogma library rendered every graph at 50 FPS or more because of its WebGL implementation. From these results, it can be concluded that the Ogma library produces the most interactive and usable graphs.

To examine the speed and scalability of the three libraries, initialisation times for randomly generated graphs and a real-world graph from the KONECT project were measured ten times and the results averaged. The random graphs had sizes of $N = 100$, $N = 500$, $N = 1,000$, $N = 5,000$, and $N = 10,000$. The real-world Sister Cities graph contained $14,274$ vertices and $20,573$ edges. For all libraries, there was a positive linear relationship between the number of edges and initialisation times. The more edges there were, the longer it took to render a graph using the force-directed layout. For all graphs, the D3 library had the worst performance due to the immediate display of nodes and edges and the use of SVG rather than HTML5 Canvas. For smaller graphs up to size $N = 1,000$, the Ogma and Cytoscape libraries yielded similar results. When graphs got larger, Cytoscape significantly outperformed the other libraries.

In Section 1.2 the research question was defined as: *How do we ensure that a web application for network analysis is interactive, fast, usable, and scalable?*. In order to answer the question, three JavaScript libraries were examined, and their performance was compared using two metrics: initialisation time and FPS. This thesis found that for graphs up to size $N = 1,000$, the Ogma library performed best in both usability (FPS) and scalability (initialisation times). When the graph size increases, Cytoscape starts to significantly outperform Ogma in terms of initialisation times, but its relatively low FPS makes it less usable and interactive. Therefore, Ogma is judged to be the best library of those examined in this thesis due to its high readability, interactivity, and

usability, even though it is not the fastest library tested. The answer to the research question is thus to use Ogma for the implementation of the web application.

Future work could include the examination of more JavaScript libraries capable of rendering graphs in the web browser. Sigma has been mentioned in this thesis but was left out because its force-directed layout algorithm was continuous, so initialisation times could not be measured. A possible solution for this could be to write a plug-in, which would evaluate the position of the nodes each iteration and check if they differ by a given pre-defined number. If so, the layout algorithm could be stopped, and the initialisation time could be computed.

Other potentially interesting research would be to create a new library that makes use of the newer WebGL standard. A minimal library could contain a single force-directed layout algorithm with limited parameters and event handlers for mouse events such as dragging and zooming. Using WebGL instead of HTML5 Canvas or SVG would significantly increase usability and interactivity, so the main focus would be to implement an efficient and high-performance layout algorithm. This minimal base could later be extended with, for example, the ability to filter nodes and edges, allow easy import and export of data, and export the resulting graph as an image.

Bibliography

[1] Barnes, J., and Hut, P. A hierarchical o (n log n) force-calculation algorithm. *Nature 324*, 6096 (1986), 446.

[2] Bastian, M., Heymann, S., and Jacomy, M. Gephi: an open source software for exploring and manipulating networks. In *Third International AAAI Conference on Weblogs and Social Media* (2009).

[3] Borgatti, S. P., and Halgin, D. S. On network theory. *Organization Science 22*, 5 (2011), 1168–1181.

[4] Börner, K., Sanyal, S., and Vespignani, A. Network science. *Annual Review of Information Science and Technology 41*, 1 (2007), 537–607.

[5] Bostock, M., Ogievetsky, V., and Heer, J. D^3 data-driven documents. *IEEE Transactions on Visualization & Computer Graphics*, 12 (2011), 2301–2309.

[6] Brandenburg, F. J., Himsolt, M., and Rohrer, C. An experimental comparison of force-directed and randomized graph drawing algorithms. In *International Symposium on Graph Drawing* (1995), Springer, pp. 76–87.

[7] Breitkreutz, B.-J., Stark, C., and Tyers, M. Osprey: a network visualization system. *Genome Biology 4*, 3 (2003), R22.

[8] Davidson, R., and Harel, D. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics 15*, 4 (1996), 301–331.

[9] Davis, G. F. Agents without principles? the spread of the poison pill through the intercorporate network. *Administrative Science Quarterly* (1991), 583–613.

[10] Di Battista, G., Garg, A., and Liotta, G. An experimental comparison of three graph drawing algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry* (1995), Citeseer, pp. 306–315.

[11] Dogrusoz, U., Giral, E., Cetintas, A., Civril, A., and Demir, E. A layout algorithm for undirected compound graphs. *Information Sciences 179*, 7 (2009), 980–994.

[12] Eades, P. A heuristic for graph drawing. *Congressus Numerantium 42* (1984), 149–160.

[13] Einav, L., and Levin, J. The data revolution and economic analysis. *Innovation Policy and the Economy 14*,

1 (2014), 1–24.

[14] Erdős, P., and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci 5*, 1 (1960), 17–60.

[15] Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., and Vrandečić, D. Introducing wikidata to the linked data web. In *International Semantic Web Conference* (2014), Springer, pp. 50–65.

[16] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., and Bader, G. D. Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics 32*, 2 (2015), 309–311.

[17] Freeman, L. C. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[18] Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using NetworkX. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[19] Jacomy, M., Venturini, T., Heymann, S., and Bastian, M. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PloS one 9*, 6 (2014), e98679.

[20] Liben-Nowell, D., and Kleinberg, J. The link-prediction problem for social networks. *Journal of the American Society for Information Science and Technology 58*, 7 (2007), 1019–1031.

[21] McGrath, C., Blythe, J., and Krackhardt, D. The effect of spatial arrangement on judgments and errors in interpreting graphs. *Social Networks 19*, 3 (1997), 223–242.

[22] Nah, F. F.-H. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology 23*, 3 (2004), 153–163.

[23] Newman, M. E. Clustering and preferential attachment in growing networks. *Physical Review E 64*, 2 (2001), 025102.

[24] Opsahl, T., Agneessens, F., and Skvoretz, J. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks 32*, 3 (2010), 245–251.

[25] Otte, E., and Rousseau, R. Social network analysis: a powerful strategy, also for the information sciences. *Journal of Information Science 28*, 6 (2002), 441–453.

[26] Scott, J. Social network analysis. *Sociology 22*, 1 (1988), 109–127.

[27] Strogatz, S. H. Exploring complex networks. *Nature 410*, 6825 (2001), 268.

[28] Verlet, L. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review 159*, 1 (1967), 98.

[29] Yang, B., Liu, D., and Liu, J. Discovering communities from Social Networks: Methodologies and Applications. In *Handbook of Social Network Technologies and Applications*. Springer, 2010, pp. 331–346.