
University of Glasgow | School of Computing Science

# Faster force-directed layout algorithms for the D3 visualisation toolkit

Pitchaya Boonsarngsuk

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

Level 4 Project — March 21, 2018

**Abstract**

In the past few years, data visualisation on the web is becoming more popular. D3, a JavaScript library, has a module that focuses on simulating physical forces on particles for creating force-directed layouts. However, the currently-available algorithms does not scale very well for multidimensional scaling. To solve this problem, the Hybrid Layout algorithm and its pivot-based near neighbour search enhancement was implemented and integrated into the D3 module. The existing D3's and Bartasius' implementation of Chalmers' 1996 algorithm were also optimised for the use case and compared against the Hybrid algorithm. Furthermore, experiments were also performed to evaluate the impact of each user-defined parameters. The results show that for larger data sets, the Hybrid Layout consistently produces fairly good layouts in a shorter amount of time. It is also capable of working on larger data sets, compared to the D3's algorithm.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

In the age of Web 2.0, new data are being generated at an overwhelming speed. Raw data made up of numbers, letters, and boolean values are hard for humans to comprehend and infer any relation from it. To make it easier and faster for us, humans, to understand a data set, various techniques were created to map raw data to a visual representation.

Many data sets have many features while humans perceive 2D illustrations best, leading to the challenge of dimensionality scaling. There are many approaches to this problem, each with its own pros and cons. One of the approaches is multidimensional scaling (MDS) which hi-light the similarity and clustering of data to the audiences. The idea is to map a data point to a particle in 2D space and place them in a way that the distance between each pair of particle in 2D space represents their distance in high-dimensional space.

With the recent trend of moving away from traditional native applications to easily-accessible cross-platform web applications, many data visualisation toolkit for JavaScript such as Google Charts, Chart.js and D3.js, are developed. With these libraries, it is easier for website designers and content creators to setup interactive attention-grabbing infographics, allowing more people to understand their works with less cognitive load.

One of the most popular free open-source data visualisation library is Data Driven Documents[13][12]. The premise is to bind an arbitrary raw data to a web page content and then apply data-driven transformations, breathing life into it, all while only using standard web technologies and avoiding any restrictions from proprietary software. This makes the library highly accessible, allowing applications to reach wider audience.

The D3-Force module, part of the D3 library, provides a framework for simulating physical forces on particles. Along with that, a spring-model algorithm was also implemented to allow for creation of a force-directed layout. While the implementation is fast for several thousands particles, it does not scale well with larger data sets, both in term of memory and time complexity. By solving these issues, the use cases covered by the module would expand to support more complicated data sets. The motivation of the project is to improve this scalability issues with better algorithms from the School of Computing Science.

## 1.2 Project Description

The University of Glasgow's School of Computing Science have some of the fastest force-directed layout drawing algorithms in the world. Some of these are Chalmers' 1996 Neighbour and Sampling technique[14], 2002 Hybrid

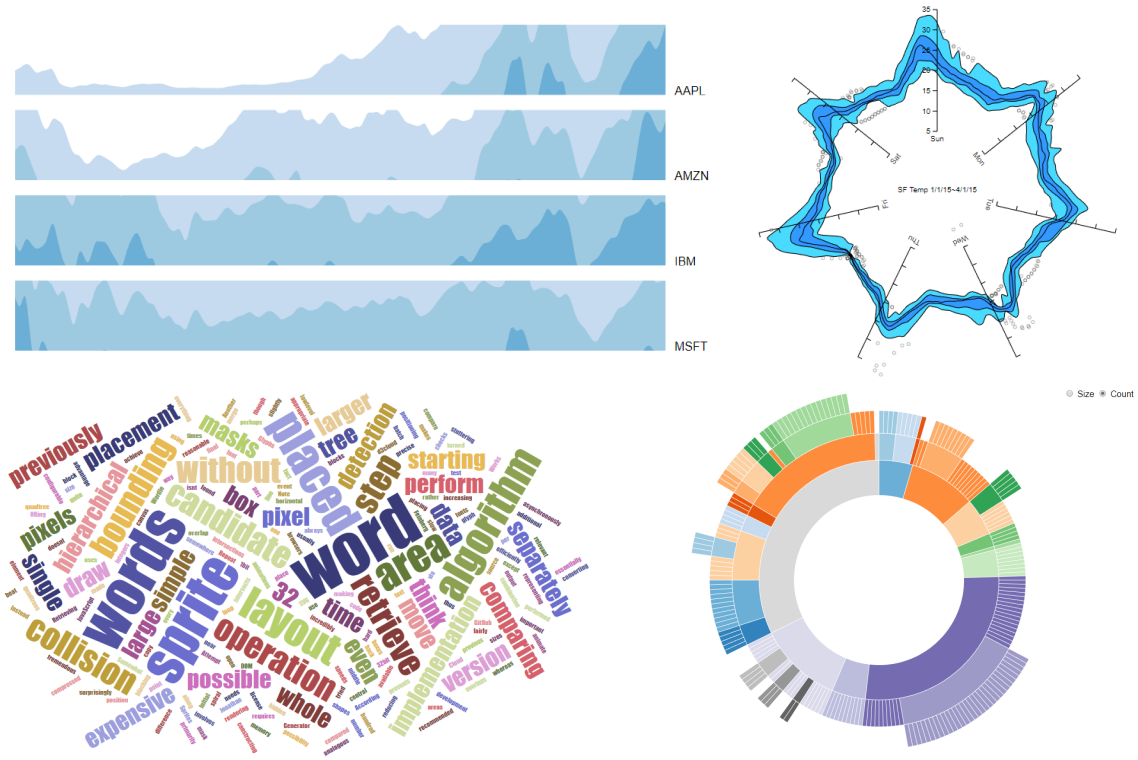



Figure 1.1: Several different visualisations based on the D3 framework: Horizon Chart (top left), Radial Boxplot (top right), Word Cloud (lower left) and Sunburst Partition (lower right).

Layout algorithm[21] and its 2003 enhanced variant[20]. These algorithms provide huge improvements, both in term of speed and memory complexity. However, these algorithms are only implemented in an older version of Java which limits its practical use. In 2017, Bartasius has implemented the 1996 algorithm along with several others and a visual interface in order to compare each algorithm against the other[9].

In short, the goal of the project is to

- implement Hybrid Layout algorithms from School of Computing Science in JavaScript
- integrate the implementation into the D3 library and Bartasius' tool set
- optimise existing implementations of basic spring model and Chalmers' algorithm
- evaluate and compare each algorithm

## 1.3 Outline

The remainder of the report will discuss the following:

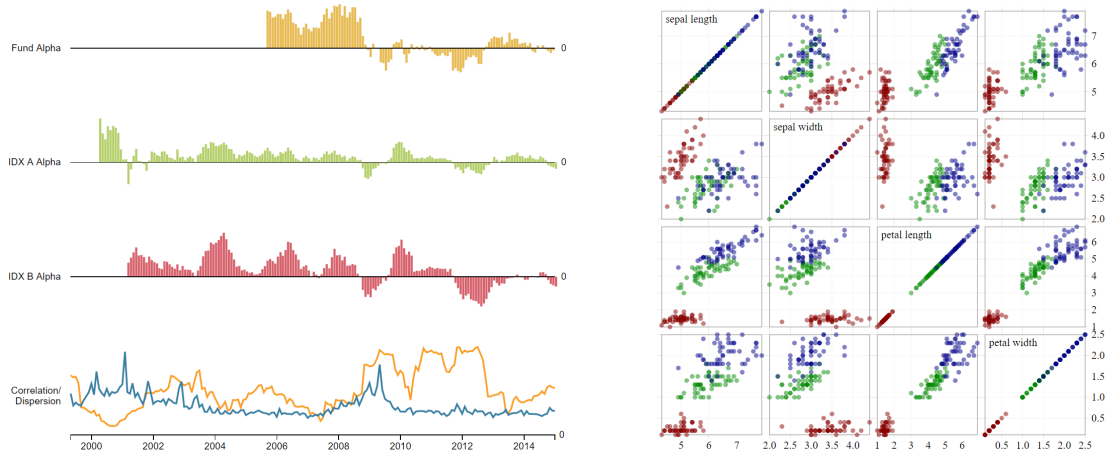
- **Background** This chapter discusses approaches to visualise high-dimensional data, and introduce the theory behind each implemented algorithm.
- **Design** This chapters discusses choices of technologies.
- **Implementation** This chapter will briefly show decisions and justifications made during the implementation process, along with several code snippets.

- **Evaluation** This chapter will detail the process used to compare the performance of each algorithm, starting from the experiment design to the final result.
- **Conclusion** This chapter gives a brief summary of the project, reflect on the process in general, and discusses possible future improvements.

# Chapter 2

# Background

With the emergence of more complex data, each having many features, the need of mapping the high-dimensional data down to 2D space is increasing. Figure 2.1 shows two approaches to the problem. One of the earliest method is to align graphs on the basis of one axis they all share. While it is still being used, the use cases are limited due to all graphs having to share an axis. On the other hand, scatterplot matrix performs scatterplot of every pair of dimensions, allowing users to see relations between many different dimensions. However, the screen space usage also rises quadratically, making it unsuitable for data with a very high number of dimensions.




(a) Single-axis composition

(b) Scatterplot Matrix

Figure 2.1: Different approaches to visualise high-dimensional data

Unlike the two previously mentioned techniques, multidimensional scaling (MDS) took another approach by aiming to reduce data dimension by preserving the level of similarity, rather than the values. Classical MDS (also known as Principal Coordinates Analysis or PCA)[15] achieves this goal by creating new dimensions for scatterplotting, each made up of a linear combination of the original dimensions, while minimising a loss function called strain. For simple cases, it can be thought of as finding the a camera angle to project the high-dimensional scatterplot onto a 2D image. Because strain assumes Euclidean distances, making it incompatible with other dissimilarity ratings. Metric MDS improves upon classical MDS by generalising the solution to support a variety of loss functions[11]. However, the disadvantage of $O(N^3)$ time complexity still remains and linear combination may not be enough for some data sets.

This project focuses on several non-linear MDS algorithms using force-directed layout. The idea is to attach each pair of data points with a spring whose equilibrium length is proportional to the high-dimensional distance between the two points, although the spring model we know today does not necessary use Hooke's law to cal-

culate the spring force[16]. Several improvements have been introduced to the idea over the past decade. For example, the concept of 'temperature' purposed by Fruchterman and Reingold[17] solves the problem where the system is unable to reach an equilibrium state and improves execution time. The project focuses on an iterative spring-model-based algorithm introduced by Chalmers[14] and the Hybrid approach which will be detailed in subsequent sections of this chapter.

There is a number of other non-linear MDS algorithms avilable. t-distributed Stochastic Neighbour Embedding (t-SNE)[25], for example, is very popular in the field of machine learning. It is based on SNE[10] where probability distributions are constructed over each data point in a way that the other similar objects have higher probability of being picked. The distributions derived from both high-dimensional and low-dimensional distances are compared using the Kullback-Leibler divergence, a metric for measuring the similarity between two probability distributions. Then, the 2D position of each data point is iteratively adjusted to maximise the similarity. The biggest downside is that it have both time and memory complexity of $O(N^2)$ per iteration. In 2017, Bartasius[9] implemented t-SNE in D3 and found that not only is it the slowest algorithm in his test, the produced layout is also the many times worse in term of Stress, a metric which will be introduced in section 2.5. However, comparing the Stress of a t-SNE layout is unfair as t-SNE is designed to optimise the distribution divergence and not Stress.

Other algorithms use different approaches. Kernel PCA tricks the classical MDS (PCA) into being non-linear by using the kernels[24]. Simply put, kernel functions are used to create new dimensions from the existing ones. These kernels can be non-linear. Hence, PCA can use these new dimensions to form a non-linear combination of the original dimensions. The limitation is that the kernels are user-defined, thus, it is up to the user to select appropriate kernels to produce a good layout. Local MDS[26] performs a different trick on MDS by only using MDS in local regions and stitching each region together, using convex optimisation. While it focuses on Trustworthiness and Continuity, the error metrics concerning each data point's neighbourhood, its overall layouts fail to form any meaningful clusters. Sammon's mapping[23], on the other hand, find a good position for each data point by using gradient descent to minimise Sammon's error, another function similar to Stress (section 2.5). However, gradient descent can only find a local minimum and the solution is not guaranteed to ever converge.

The rest of this chapter will describes each of the algorithm and performance metrics used in this project in detail.

## 2.1 Link Force

D3 library, which will be described in section 3.2, have several different force models implemented for creating a force-directed graph. One of them is Link Force. In this brute-force method, a force is applied between the two nodes at the end of each link. The force pushes the nodes together or apart with varying strength, proportional to the error between the desired and current distance on the graph. Essentially, is the spring model with a custom spring-force calculation formula. An example of a graph produced by the D3 link force is shown in figure 2.2. In MDS where the high-dimensional distance between every pair of nodes can be calculated, a link will be created to represent each pair, resulting in a complete graph.

The Link Force algorithm is inefficient. In each time step (iteration), a calculation has to be done for each pair of nodes connected with a link. This means that for MDS with $N$ nodes, the algorithm will have to perform $N(N-1)$ force calculations per iteration, essentially $O(N^2)$. It is also believed that the number of iterations required to create a good layout is proportional to the size of the data set, hence the total time complexity of $O(N^3)$. The model also cache the desired distance of each link in memory to improve speed across multiple iterations. While this greatly reduces the number of calls to the distance-calculating function, the memory complexity also increases to $O(N^2)$. Because JavaScript memory heap is limited, it runs out of memory when trying to process a complete graph of more than around three thousands points, depending on the features of the data.

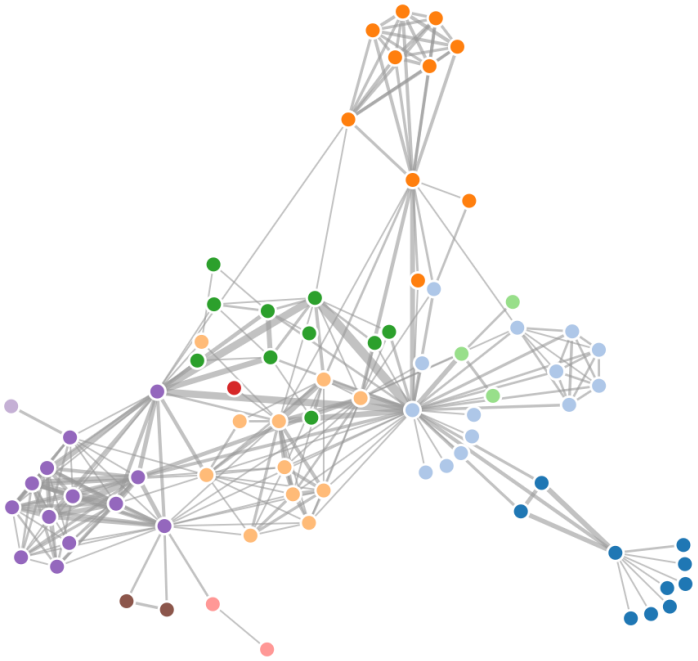


Figure 2.2: An example of a graph produced by D3 Link Force.

## 2.2 Chalmers' 1996 algorithm

In 1996, Matthew Chalmers proposed a technique to reduce the time complexity down to $O(N^2)$, which is a massive improvement over link force's $O(N^3)$, potentially at the cost of accuracy. This is done by reducing the number of spring force calculations per iterations, using random samples[14].

To begin, each object $i$, is assigned two distinct sets. $Neighbors$ set, referred to as $V$ in the original paper, stores a sorted list of other objects that are closest to $i$, i.e. have low high-dimensional distance. These objects are expected to be put near $i$ in 2D space. At the start, this set is empty. The second set is $Samples$ (referred to as $S$). This set contains a number of other random objects that not member of the $Neighbors$ set, and is regenerated at the start of every iteration.

In each iteration, each object $i$ only performs spring force calculations against every other objects in its $Neighbors$ and $Samples$ sets. Afterward, each random object is then compared against other objects in the $Neighbors$ set. If a random object is closer to $i$, then it is swapped into the $Neighbors$ set. As a result, the $Neighbors$ set becomes a better representative of the most similar objects to $i$.

The total number of spring calculations per iteration reduces from $N(N-1)$ to $N(Neighbours_{size} + Samples_{size})$ where $Neighbours_{size}$ and $Samples_{size}$ denotes the maximum number of objects each $Neighbours$ and $Samples$ set, respectively. Because these two numbers are predefined constants, the time complexity is $O(N)$.

Previous evaluations indicated that the quality of the produced layout improves as $Neighbours_{size}$ and $Samples_{size}$ grows larger. For larger data sets, setting too small values could cause the algorithm to miss some details. However, favourable results can be obtained from numbers as low as 5 and 10 for $Neighbours_{size}$ and $Samples_{size}$[21].

## 2.3 Hybrid Layout for Multidimensional Scaling

In 2002, Alistair Morrison, Greg Ross, and Matthew Chalmers introduced a multi-phase method, based on Chalmers' 1996 algorithm to reduce the run time down to $O(N\sqrt{N})$. This is achieved by calculating the spring forces over a subset of data, and interpolating the rest[21].

In this hybrid layout method, the $\sqrt{N}$ sample objects ($S$) are first placed on the 2D space using the 1996 algorithm. The complexity of this step is $O(\sqrt{N}\sqrt{N})$ or $O(N)$. After that, each of the other objects $i$ are then interpolated as described below.

1. Find the 'parent' object $x \in S$ with the least high-dimensional distance to $i$. This is essentially the nearest neighbour searching problem.
2. Define a circle around $x$ with radius $r$, proportional to the high-dimensional distance between $x$ and $i$.
3. Find the quadrant of the circle which is the most satisfactory to place $i$.
4. Perform a binary search on the quadrant to determine the best angle for $i$ and place it there.
5. Select random samples $s$ from $S$. $s \subset S$.
6. Calculate the sum of force vector between $i$ and each member of $s$.
7. Add the vector to $i$'s current position.
8. Repeat step 6 and 7 for a constant number of times to refine the placement.

In this process, step 1 has the highest time complexity of $O(S_{size}) = O(\sqrt{N})$. Because there are $N - \sqrt{N}$ objects to interpolate, the overall complexity of this step is $O(N\sqrt{N})$.

Finally, the Chalmers' spring model is applied to the full data set for a constant number of iterations. This operation have the time complexity of $O(N)$.

Previous evaluations show that this method is faster than the Chalmers' 1996 algorithm, and can create a layout with lower stress, thanks to the more accurate positioning in the interpolation process.

## 2.4 Hybrid MDS with Pivot-Based Searching algorithm

The bottleneck of the Hybrid Layout Algorithm is the nearest-neighbour searching process during the interpolation. The previous brute-force method results in the time complexity of $O(N\sqrt{N})$. This improvement introduces pivot-based searching to approximate a near-neighbour and reduces the time complexity to $O(N^{\frac{5}{4}})$[20].

The main improvements is gained by pre-processing the set $S$ ($\sqrt{N}$ samples) so that each of the $N - \sqrt{N}$ other points can find the parent is faster. To begin, $k$ points were selected from $S$ as 'parent'. Each pivot $p \in k$ have a number of buckets. Every other points in $S - \{p\}$ assigned a bucket number, based on the distance from to $p$ as illustrated in figure 2.3.

To find a parent of an object, a distance calculation is first performed against each pivot to determine which bucket of each pivot is the object in. From this, the content of each bucket is searched for the nearest neighbour.

Pre-processing:
**for all** pivot in $k$ **do**
  **for all** points in $(S - k)$ **do**
    Perform distance calculation
  **end for**
**end for**
Find parent for object $i$:
**for all** pivot $p$ in $k$ **do**
  Perform distance calculation.
  Determine the bucket for $i$ in $p$.
  **for all** point in the bucket **do**
    Perform distance calculation
  **end for**
**end for**

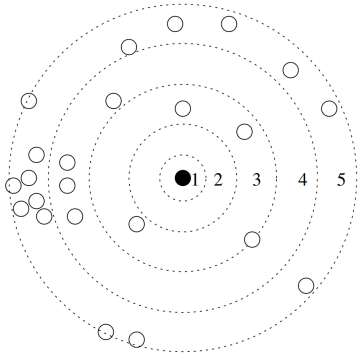



Figure 2.3: Diagram of a pivot (dark shaded point) with five buckets, illustrated as discs between dotted circle. Each of the other points in $S$ are classified into buckets by the distances to the pivot.

The complexity of the preprocessing stage is $O(\sqrt{N}k)$. For query, the average number of points in each bucket is $\frac{S_{size}}{number of buckets} = \frac{\sqrt{n}}{n^{\frac{1}{4}}}$. Since a query will be performed for each of the $N - \sqrt{N}$ points not in $S$, overall complexity is $O(\sqrt{N}k) + (N - \sqrt{N})N^{\frac{1}{4}} = O(N^{\frac{5}{4}})$.

With this method, the parent found is not guaranteed to be the closest point. Prior evaluations have concluded that the accuracy is high enough to produce good result.

## 2.5 Performance Metrics

To compare different algorithms, they have to be tested against the same set of performance metrices. During the development, a number of metrics were used to objectively judge the produced layout and computation requirements. The evaluation process in chapter 5 will focuses on the following metrics.

- **Execution time** is a broadly used metric to evaluate any algorithms that require any significant computational power. Some applications aim to be interactive and the algorithm have to finish the calculations within the time constraints for the program to stay responsive. This project, however, focuses on large data sets with minimal user interaction. Hence, the execution time in this project simply measures of the time an algorithm takes to produce its "final" result. The criteria to consider a layout "finished" will be discussed in details in section 5.2.1.

- **Stress** is one of the most popular metric for non-linear MDS algorithms and is modelled from the mechanical stress of a spring system. It is based on sum-of-squared errors of inter-object distance[14]. The function is defined as follow.

$$Stress = \frac{\sum_{i<j}(d_{ij} - g_{ij})^2}{\sum_{i<j} g_{ij}^2}$$

  $d_{ij}$ denotes the desired high-dimensional distance between object $i$ and $j$ while $g_{ij}$denotes the low-dimensional distance.

  While Stress is a good metric to evaluate a layout, its calculation is an expensive operation ($O(N^2)$). At the same time, it is not part of operation of any algorithm. Thus, by adding this optional measurement step between iterations, every algorithm will takes a lot longer that complete, invalidating the measured execution time of the run.

- **Memory usage:** With more interests in the field of machine learning, the number of data points in a data set is getting bigger. It is common to encounter data sets with millions instances, each with possibly hundreds of attributes. Therefore, memory usage shows how an algorithm scales to larger data sets and how many data points can a computer system handle.

## 2.6 Summary

In this chapter, several techniques of visualising multidimensional data have been explored. As the focus of the project is on three spring-model-based algorithms, the principles of each of the method have been discussed. Finally, in order to measure the performance of each algorithm, different metrics were introduced and will be used for the evaluation process.

# Chapter 3

# Design

This chapter discusses decisions for selecting technologies and libraries during the development process. It also briefly describes each technology, available alternatives, and Bartasius' application which this project is built on.

## 3.1 Technologies

With the goal of reaching as wide audience as possible, the project advisor set a requirement that the application must run on a modern web browser. This section briefly introduce web technologies used to develop the project.

### 3.1.1 HTML, CSS, and SVG

HTML and CSS are the two core technologies used to build web pages. Modern web applications can not avoid these standards, and this project is no exception. HTML (Hypertext Markup Language) describes the structure and content of a web page. CSS (Cascading Style Sheets) defines the visual layout of the page. The latest major version of the standards are HTML 5 and CSS3, both of which are currently supported by all major web browsers. Aside from user interface, this project rely heavily on Scalable Vector Graphics (SVG), an open XML-based vector image format, to be supported in the HTML standard to render the produced layout. HTML 5 allow SVG to be embedded directly in an `<svg>..</svg>` tag, rather than a seperate xhtml document in previous HTML versions. In this project, an SVG is used as a base canvas to display the produced graphics. Each data point is then drawn as a circle as shown in figure 3.1.

```
▼<svg id="svg" width="100%" height="600">
  ▼<g>
    ▶<g class="brush" fill="none" pointer-events="all" style="-webkit-tap-highlight-color: rgba(0, 0, 0, 0);">…</g>
    ▼<g class="nodes">
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="535.015676105761" cy="39.44409041839056"></circle>
       <circle r="10" transform="translate(921,300)" fill="#e377c2" cx="-740.1484822068517" cy="306.08734347930414"></circle>
       <circle r="10" transform="translate(921,300)" fill="#e377c2" cx="-359.9115017737762" cy="242.54520932727775"></circle>
       <circle r="10" transform="translate(921,300)" fill="#e377c2" cx="1.7886982735506973" cy="726.7407959202319"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="-132.30982364641036" cy="-435.08340155772174"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="127.53802944943486" cy="-493.25699332934823"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="128.36963586778472" cy="-233.40499001135652"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="193.47158208518152" cy="150.76152090406922"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="357.74299448153045" cy="-139.10514796202855"></circle>
       <circle r="10" transform="translate(921,300)" fill="#8c564b" cx="-111.55680863620579" cy="-164.7284271888174"></circle>
    </g>
  </g>
</svg>
```

Figure 3.1: An example of SVG document representing data points.

### 3.1.2 JavaScript

JavaScript is the most common high-level scripting language for web pages. It is dynamic, untyped and multi-paradigm, supporting event-driven, functional, prototype-based, and object-oriented programming styles. It mostly runs on client's browser interpreter, although platforms such as Node.js allow it to run on servers as well. Many APIs are designed for manipulating the HTML document, allowing programmers to create dynamic web pages by changing contents according to a variety of events. Alternative languages such as CoffeeScript and TypeScript are emerging, each adding more features, syntactic sugars, or syntax changes to improve code readability. However, in order to run these languages on browsers, they have to be compiled back to JavaScript. The learning resources availability is also leagues behind JavaScript. For these reasons, standard JavaScript was chosen for this project.

Being an interpreted high-level language, it is relatively slow. Due to limitations from standard APIs, it is also single-threaded. asm.js is an effort to optimize JavaScript by using only a restricted subset of features. It is intended to be a compilation target from other statically-typed languages such as C/C++ rather than a language to code in[27]. Existing JavaScript engines may gain performance from asm.js' restrictions such as preallocated heap, reducing load on the Garbage Collector. Firefox and Edge also recognize asm.js and compile the code to assembly ahead-of-time (AOT) to eliminate the need to run code through interpreter entirely, resulting in a significant speed increase[28]. However, the D3 library is still using standard JavaScript. A large chunk of the library have to be ported in order to be able to compare different algorithms fairly. Since a lot of effort is required to potentially improve performance significantly on 2 browsers and marginally on others, asm.js was not selected for this project.

WebAssembly (wasm) is another recent contender. Unlike JavaScript, it is a binary format designed to run with JavaScript on the same sandboxed stack machine[4]. Similar to asm.js, it is intended to be a compile target. With support for additional CPU instructions not available in JavaScript, it also perform predictably better than asm.js. Only recently exited the preview phase in March 2017, the support was not widespread and learning resources was hard to find. It also inherit a risk of not being widely adopted by browsers. As a result, WebAssembly was not considered as a viable option.

## 3.2 Data Driven Document

Data Driven Documents (D3 or D3.js)[13][12] is one of the most popular JavaScript library for interactive data visualisations in web browsers. The focus is to bind data to DOM (Document Object Model) elements in HTML or SVG documents and apply data-driven transformations to make the visualisation visually appealing. Its modular and free open-source nature also makes it flexible. Many visualisation algorithms can be easy integrated into it. In this project, aside from Force Link algorithm, the complicated process of translating velocities and location onto an SVG document are handled by the D3 library, allowing the project to just focus more on the algorithms.

There are several other data visualisation libraries such as Google Charts and Chart.js. However, most of them do not support force-directed layout and are not as flexible. In addition to being a requirement set by the project advisor, D3 seems like the best choice for this project.

## 3.3 ESLint

JavaScript has a very flexible syntax, which can lead to the problem of inconsistent coding style. ESLint[1] is a utility to check and, to a certain degree, fix JavaScript coding errors according to the user-specified set of rules. Unlike other popular alternatives such as js-beautify[2], ESLint also provide rules that check for possible syntax

errors such as referencing an undefined variable names. To assist the development, a custom ESLint rule set was created for this project and enforced for plug-in.

## 3.4 Bartasius' D3 Neighbour Sampling plug-in

In 2017, Bartasius implemented the Chalmers' 1996 algorithm and several others algorithms for his level 4 project at the School of Computing Science. All source files are released on GitHub under the MIT license. To reduce the amount of duplicated work, the project advisor recommended using the repository as a groundwork to implement other algorithms upon.

### 3.4.1 Input Data

The data is one of the most important element of the project. Without it, nothing can be visualised. Since the data may consist of many different features (attributes), each with a unique name, it makes sense to store each data point (node) as an JavaScript object, a collection of `key:value` pairs. To conform with D3 API, all nodes are stored in a list (array). Two example data structures are shown in figure 3.2


Figure 3.2: Examples of the data structure used to store the input data. On the left is nodes from the Poker Hand data set. Right shows nodes from the Antarctica data set. The two data sets will be explained in section 5.1.

### 3.4.2 Graphical User Interface

Due to the sheer amount of experiments to run, manually changing functions and file names between each run will be a tedious task. Bartasius developed a GUI for the plug-in to ease the testing process. For this project, modifications have been made to accommodate newly implemented algorithms.

Figure 3.3 shows the modified GUI used in this project. At the top is the canvas to draw the produced layout. The controls below are then divided into 3 columns. The left column controls data set input, rendering, and iterations limit. The middle column are a set radio and slider buttons for selecting the algorithm and parameters to use. The right contains a list of distance functions to choose from.

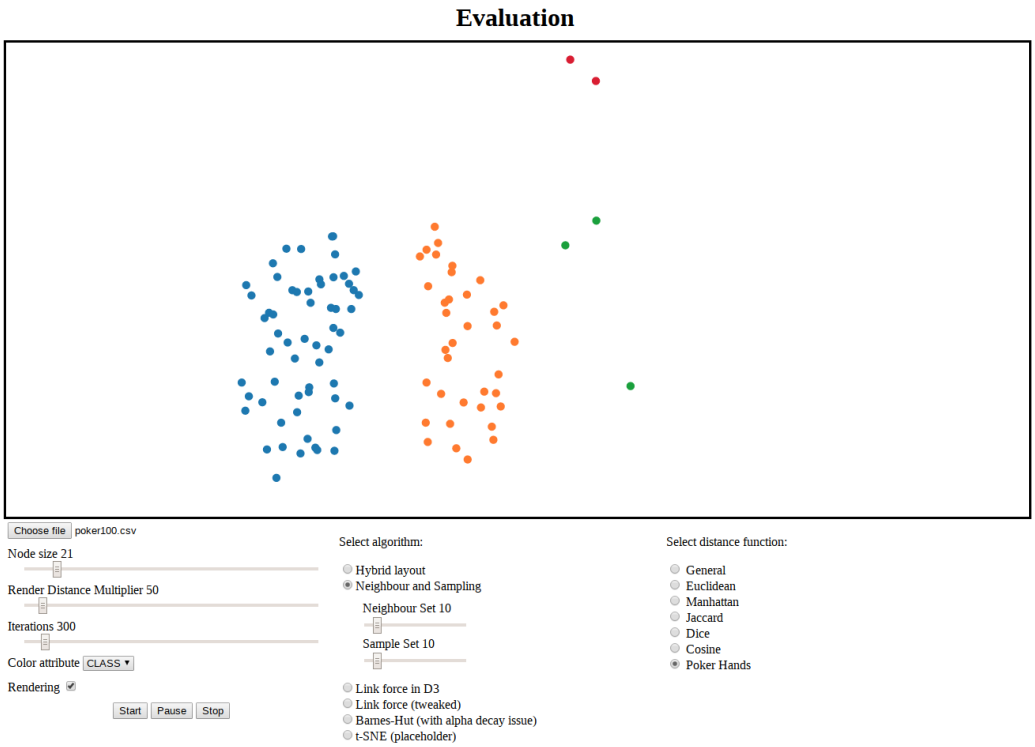



Figure 3.3: The graphical interface.

## 3.5 Summary

In this chapter, several technologies and alternatives were discussed. In the end, the project is set out to reuse Bartasius's repository, running on D3.js with standard JavaScript, HTML, CSS and SVG for their learning resources. The ESLint tool is also setup to format the JavaScript code and check for possible errors.

# Chapter 4

# Implementation

## 4.1 Outline

The D3 library is modular. One of the modules, D3-force, is found to be most relevant for MDS with spring model algorithms. Thus, this project aims to integrate the implementations into this module. D3-force provides a simplified Simulation object to simulate various physical force calculations. Each Simulation contain a list of nodes and Force objects. Interfaces were defined, allowing each Force to access the node list. To keep track of positions, each node will be assigned values representing its current location and velocity vector. These values can then be used by the application to draw a graph. In each constant unit time step (iteration), the Simulation will trigger a function in each Force object, allowing them to calculate and add values to each particle's velocity vector, which will then be added to the particle's position by the Simulation. For MDS, each data point is represented as a particle in the simulation.

Because D3 is a library to be built into other web applications, the algorithms implemented can not be used on their own. Fortunately, as part of Bartasius' level 4 project in 2017, a web application for testing and evaluation has already been created with graphical user interface designed to allow the user to easily select an algorithm, data set, and parameter values to use. Various distance functions, including one specifically created for the Poker Hands data set[18] which will be used for evaluation (section 5.1), are also in place and fully functional.

The csv-formatted data file can be loaded locally. Next, it is parsed by Papa Parse JavaScript library[3] and then loaded into the Simulation. Depending on the distance functions, per-dimension mean, variance, and other attributes may also be calculated as well. These values are used in several general distance functions to scale values of each feature. The D3 simulation layout is shown on an SVG canvas with zoom functionality to allow graph investigation. The distance function scaling was tweaked to only affect rendering and not the force calculation.

Several values used for evaluation such as execution time, total force applied per iteration, and stress may also be computed. However, these values are printed out to JavaScript console instead.

Due to the growing number of algorithms and variables, the main JavaScript code has been refactored. Functions for controlling each algorithm have been extracted to their own file, and some unused codes are removed or commented out.

Finally, a custom set of ESLint rules was created to format and check for possible JavaScript errors in the implementations made to the D3 library. The same rule set was also used to format the JavaScript code for the web application to a certain degree, but many more exceptions had to be made due to the difference in the code structure.

## 4.2 Algorithms

This section discusses implementation decisions for each algorithm, some of which are already implemented in D3 force module and the d3-neighbour-sampling plugin. Adjustments made to third-party-implemented algorithms are also discussed.

### 4.2.1 Link force

D3-force module has an algorithm implemented to produce a force-directed layout. The main idea is to change the velocity vector of each pair of node connected via a link at every time step, simulating force application. For example, if two nodes are further apart than the desired distance, a force is applied to both nodes to pull them together. The implementation also supports incomplete graphs, thus the links have to be specified. The force is also, by default, scaled on each node depending on how many springs it is attached to, in order to balance the force applied to heavily and lightly connected nodes, improving overall stability. Without such scaling, the graph would expands in every direction.

In the early stages of the project, when assessing the library, it is observed that many of the available features are unused for Multidimensional scaling. In order to reduce the computation time and memory usage, I created a modified version of Force Link as part of the plug-in. The following are the improved aspects.

Firstly, to accommodate an incomplete graph, the force scaling has to be calculated for each node and each link. The calculated values are then cached in a similar manner to the distances ($bias$ and $strengths$ in code 4.1). In a fully-connected graph, these values are the same for every links and nodes. To save on memory and startup time, the arrays were replaced by a single number value.

```
1  function force(alpha) {
2    for (var k = 0, n = links.length; k < iterations; ++k) {
3      for (var i = 0, link, source, target, x, y, l, b; i < n; ++i) {
4        link = links[i], source = link.source, target = link.target;
5        x = target.x + target.vx - source.x - source.vx || jiggle();
6        y = target.y + target.vy - source.y - source.vy || jiggle();
7        l = Math.sqrt(x * x + y * y);
8        l = (l - distances[i]) / l * alpha * strengths[i];
9        x *= l, y *= l;
10       target.vx -= x * (b = bias[i]);
11       target.vy -= y * b;
12       source.vx += x * (b = 1 - b);
13       source.vy += y * b;
14     }
15   }
16 }
```

Code 4.1: Force calculation function of Force Link as implemented in D3.

Secondly, D3's Force Link requires the user to specify an array of links to describe the graph. Each link is a string-indexed dictionary which is not the most memory-friendly data type. The cached distance values are stored in a separated array with index parallel to that of the links array. Since nodes are also stored in an array, I replaced the entire links array with a nested loop over the nodes array, reducing the memory footprint even further and eliminating time required to construct the array. The index for the cached distance is then adjusted accordingly.

```
1  function force(alpha) {
2    let n = nodes.length;
3    ...
4    for (var k = 0, source, target, i, j, x, y, l; k < iterations; ++k) {
```

```
5       for (i = 1; i < n; i++) for (j = 0; j < i; j++) { // For each link
6         // jiggle so x, y and l won't be zero and causes divide by zero error later on
7         source = nodes[i];  target = nodes[j];
8         x = target.x + target.vx - source.x - source.vx || jiggle();
9         y = target.y + target.vy - source.y - source.vy || jiggle();
10        l = Math.sqrt(x * x + y * y);
11        //dataSizeFactor = 0.5/(nodes.length-1), pre-calculated only once
12        l = (l - distances[i*(i-1)/2+j]) / l  * dataSizeFactor * alpha;
13        x *= l, y *= l;
14        target.vx -= x;  target.vy -= y;
15        source.vx += x;  source.vy += y;
16      }
17    }
18  ...
19  }
```

Code 4.2: Part of the customized force calculation function.

After optimisation, the execution time decreases marginally while memory consumption decreases by a seventh, raising data size limit from 3,200 data points[9] to over 10,000 in the process. Details on the evaluation procedure and data size limitation will be discussed in section 5.3.1.


Figure 4.1: A comparison in memory usage and execution time between versions Force Link at 3,000 data points from Poker Hands data set for 300 iterations.

Next, `jiggle()` function was assessed. As shown in line 5-7 of code 4.1, in cases where two nodes are projected to be on the exact same location, `x`, `y` and, in-turn, `l`, could be 0. This would cause a divide-by-zero error in line 8. Rather than throwing an error, JavaScript would return the result as `Infinity` or `-Infinity`. Any subsequent arithmetic operations, except for modulus, with other numbers will also results in $\pm$`Infinity`, effectively deleting the coordinate and velocity values from the entire system. To prevent such error, when `x` or `y` is calculated to be zero, D3 will replace the values with a very small random number generated by `jiggle()`. While extremely unlikely, there is still a chance that `jiggle()` will random return a random value of 0. This case can rarely be observed when every nodes are initially placed at the exact same position. To counter this, I modified `jiggle()` to re-random a number until a non-zero value is found.

Finally, a feature is added to track the average force applied to the system in each iteration. A threshold value is set so that once average force falls below the threshold, a user-defined function is called. In this case, a handler is added to Bartasius' application to stop the simulation. This is feature will be heavily used in the evaluation process (section 5.2.1).

### 4.2.2 Chalmers' 1996

Bartasius' d3-neighbour-sampling plug-in has the main focus on Chalmers' 1996 algorithm. The idea is to use the exact same force calculation function as D3 Force Link for a fair comparison. The algorithm was also implemented as a Force object to be used by a Simulation. As part of the project, I refactored the code base to ease the development process and improved a shortcoming.

Aside from formatting the code, Bartasius' implementation does not have spring force scaling, making the graph explodes in every direction. Originally, the example implementation used decaying $alpha$, a variable controlled by the Simulation used for artificially scaling down the force applied to the system over time, to make the system contract back. A constant `dataSizeFactor`, similar to that in the custom Link Force, have been added to mitigate the requirement of decaying alpha.

Next, after seeing memory footprint of the optimized Link Force, an idea occurred to also cache all the distances between every pair of nodes. After the implementation, an experiment was ran to compare the performance. However, even with a data set of moderate size and higher number of iterations than typical requirement, the time spent on caching is higher than time saved, resulting in longer total execution time (figure 4.2). The JavaScript heap usage also raises to 128 MB with manually-invoked garbage collector (will be discussed in section 5.3.1) when originally, it has never used more than 50 MB. With all these drawbacks, this patch was withdrawn.

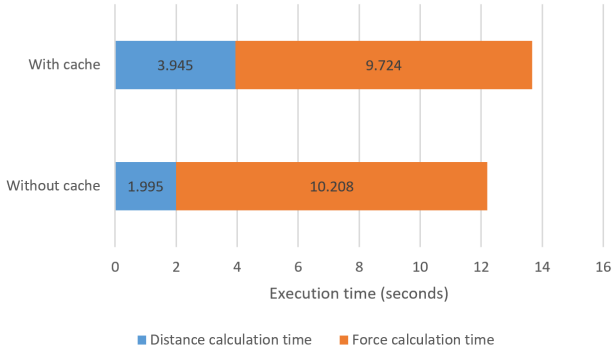



Figure 4.2: A comparison in execution time between force and distance calculation, with and without caching on 5,000 data points of Poker Hands data set for 300 iterations.

Lastly, the average force applied tracker, similar to that added to Force Link, is also added for the evaluation process. It should be noted that unlike Force Link, the system will not stabilise to a freezing point. Because the $Samples$ set keeps changing randomly, there is no single state where every spring forces cancel each other out completely. This can also be seen when the animation is drawn where every nodes keep wiggling about but the overall layout remains constant.

### 4.2.3 Hybrid Layout

Because Hybrid Layout is a multi-phase use of the Chalmers' algorithm, it does not fit well with the limited interfaces designed for Force objects. The approach taken is to implement the Hybrid algorithm as a new JavaScript object that takes control of the Simulation instead. To make it fit with other D3 APIs, the designs have to first be studied.

The D3 API extensively with the Method Chaining design pattern. The main idea is that by having each method returns the reference to itself rather than nothing, the method calls on the same object can be chained

together in a single statement[19]. In addition, the code readability is also improved to a certain degree. With the knowledge in mind, the same trend is followed for this Hybrid object.

```
let simulation = d3.forceSimulation() // Configured D3 Simulation object
  .nodes(allNodes);

let firstPhaseForce = d3.forceNeighbourSampling() // Configured Chalmers force object
  .neighbourSize(NEIGHBOUR_SIZE)
  .sampleSize(SAMPLE_SIZE)
  .distance(distanceCalculator);

let thirdPhaseForce = d3.forceNeighbourSampling() // Configured Chalmers force object
  .setParameter(Value); // Similar to above

let hybridSimulation = d3.hybridSimulation(simulation)
  .forceSample(firstPhaseForce)
  .forceFull(thirdPhaseForce)
  .numPivots(PIVOTS ? NUM_PIVOTS:0) // brute-force is used when < 1 pivot is specified
  .on("startInterp", function () {setNodesToDisplay(allNodes);})
  ...;

let firstPhaseSamples = hybridSimulation.subSet();
setNodesToDisplay(firstPhaseSamples);

hybridSimulation.restart(); // Start the hybrid simulation
}
```

Code 4.3: Simplified example of using the API.

As shown in code 4.3, the algorithm-specific parameters for each Chalmers' force objects are set in advance by the user. Since the Hybrid object interacts with the Simulation and force-calculation objects via general interfaces, other force calculators could potentially be used without having to modify the Hybrid object as well. In fact, D3's original implementation of Force Link also works with the Hybrid object. To terminate the force calculations in the first and last phase, the Hybrid object have an internal iteration counter to stop the calculations after predefined number of time steps. In addition, the applied force threshold events are also supported as an alternative termination criterion.

For interpolation, two separate functions were created for each method. After the parent is found, both functions call the same third function to handle the rest of the process (step 2 to 8 of in section 2.3).

Interpolation with brute-force parent finding:
Select random samples $s$ from $S$. $s \subset S$.
**for all** node $n$ to be interpolated **do**
  Create distance cache array, equal to size of $s$
  **for all** node $i$ in $S$ **do**
    Perform distance calculation.
    **if** $i \in s$ **then** cache distance
    **end if**
  **end for**
  PLACE N($n$, closest $i$, $s$, distance cache)
**end for**

Interpolation with pivot-based parent finding:
Preprocess pivots buckets
Select random samples $s$ from $S$. $s \subset S$.
**for all** node $n$ to be interpolated **do**
  Create distance cache array, equal to size of $s$
  **for all** pivot $p$ in $k$ **do**
    Perform distance calculation.
    **if** $p \in s$ **then** cache distance
    **end if**
    **for all** node $i$ in bucket **do**
      Perform distance calculation.
      **if** $i \in s$ **then** cache distance
      **end if**
    **end for**
  **end for**
  Fill in the rest of the cache
  PLACE N($n$, closest $i$ or $p$, $s$, distance cache)
**end for**

Since the original paper did not specify the "satisfactory" metric for placing a node $n$ on the circle around its parent (step 3 to 4), Matthew Chalmers, the project advisor who also took part in developing the algorithm, was contacted for clarification. Unfortunately, the knowledge was lost. Instead, sum of distance error between $n$ and every member of $s$ was proposed as an alternative. Preliminary testings shows that it works well and is used for this implementation.

With that decision, the high-dimensional distances between $n$ and each member of $s$ becomes used multiple times for binary searching and placement refinement (step 7 and 8). To reduce the distance function calls, a distance cache have been created. For brute-force parent finding, the cache can be filled while the parent is being selected as $s \subset S$. On the other hand, pivot-based searching might not cover every member of $s$. Thus, the missing caches are filled after parent searching.

## 4.3 Metrics

Many different metrics were introduced in section 2.5, some of them require extra code to be written. While memory usage measurement requires an external profiler, execution time can also be calculated by the application. For JavaScript, the recommended way is to take the high-resolution time-stamp before and after code execution. The method provides accuracy up-to 5 microseconds. However, it is important to note that with the level of precision, the measured value will vary from run-to-run, due to many factors both from software such as OS' process scheduler, and hardware such as Intel® Turbo Boost or cache prefetch.

```
p1 = performance.now();
// Execute algorithm
p2 = performance.now();
console.log("Execution time", p2-p1);
```

Code 4.4: Execution time measurement.

Stress calculation is done as defined by the formula in section 2.5. The calculation is independent of the algorithm. In fact, it does not depend on D3 at all. Only an array of node objects and a distance calculation is required. Due to its very long calculation time, this function is only called on-demand when the value has to be recorded. The exact implementation is shown in code 4.5.

```
export function getStress(nodes, distance) {
  let sumDiffSq = 0;  let sumLowDDistSq = 0;
  for (let j = nodes.length-1; j >= 1; j--) {
    for (let i = 0; i < j; i++) {
      let source = nodes[i], target = nodes[j];
      let lowDDist = Math.hypot(target.x - source.x, target.y - source.y);
      let highDDist = distance(source, target);
      sumDiffSq += Math.pow(highDDist - lowDDist, 2);
      sumLowDDistSq += lowDDist * lowDDist;
    }
  }
  return Math.sqrt(sumDiffSq / sumLowDDistSq);
}
```

Code 4.5: Stress calculation function.

# Chapter 5

# Evaluation

This chapter presents comparisons between each of the implemented algorithms. First, used data sets will be described. The experiment setup is then introduced, along with decision behind the each test design. Lastly, the results are shown and briefly interpreted.

## 5.1 Data Sets

The data sets utilized during the development are the Iris, Poker Hands[18], and Antarctic data set[22]. The Iris is one of the most popular data set to get started in Machine Learning. It contain 150 measurements from flowers of Iris Setosa, Iris Versicolour and Iris Virginica species, each with four parameters: petal and sepal width and height in centimeter. It was chosen as a starting point for development because it is a classification data set where the parameters can be used by the distance function and the label can be used to colour each instance. Each species is also clustered quite clearly, making it easier to see if the algorithm is working as intended.

The Poker Hands is another classification data set containing hands of 5 playing cards drawn from a standard deck, each is described in rank (Ace, 2, 3,...) and suit (Hearts, Spades, etc). Each hand is labelled as a poker hand (Nothing, Flush, Full house, etc). This data set is selected for the experiment because it contains over a million records. In each test, only subsets of the data is used due to size limitation.

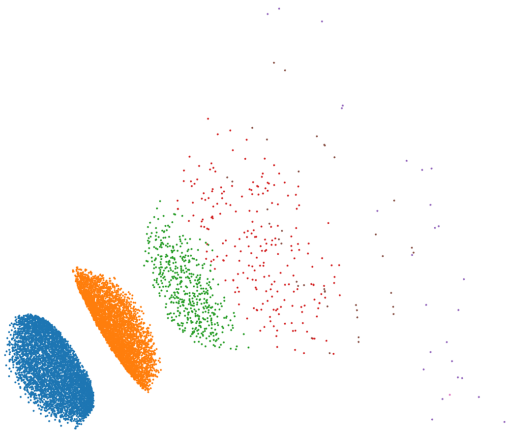


Figure 5.1: Visualisation of 10,000 data points from the Poker Hands data set, using Link Force.

The Antarctic data set contain 2,202 measurements by remote sensing probes over the period of 2 weeks at a frozen lake in the Antarctic. The 16 features includes water temperature, UV radiation levels, ice thickness, etc. The data is formatted into CSV by Greg Ross and is used to represent a data set with complex structure and high dimensionality. Due to the relatively small size of this data set, it is only used to compare the ability to show fine details.

## 5.2 Experimental Setup

Hardware and web browser can greatly impact the JavaScript performance. In addition to the code and data set, these variables have to be controlled as well. The computers used are all the same model of Dell All-in-One desktop computers with Intel® Core™ i5-3470S and 8GB of DDR3 memory, running CentOS 7 with Linux 3.10-x86-64. As for web browser, the official 64-bit build of Google Chrome 61.0.3163.79 is used to both run and analyse hardware usage with its performance profiling tool.

Other unrelated parameters were also controlled as much as possible. The simulation's velocity decay is set at default of $0.4$, mimicking air friction, and the starting position of all nodes are locked at $(0, 0)$. Although starting every nodes at the exact same position may seems to cause a very high initial spring force, the force scaling and the way D3 takes each node's velocity as part of spring force calculation prevent the system from spreading out too far. In practice, the graphs continue to expand for several more iterations before the overall layout reaches the correct size. Alpha, a decaying value used for artificially slowing down and freezing the system over time, is also kept at 1 to keep the springs' forces in full effect.

The web page is also refreshed after every run to make sure that everything, including uncontrollable aspects such as JavaScript heap, ahead-of-time compilation and the behavior of the browser's garbage collector, have been properly reset.

### 5.2.1 Termination criteria

Both Link force and the Chalmers' 1996 algorithm create a layout that stabilises over time. In D3, calculations are performed for a predefined number of iterations. This have a drawback of having to select an appropriate value. Choosing the number too high means that execution time is wasted calculating minute details with no visible change to the layout while the opposite can results in a bad layout. Determining the constant number can be problematic, considering that each algorithm may stabilise after different number of iterations, especially when considering that the interpolation result from the Hybrid algorithm can vary greatly from run-to-run(section 5.2.2).

An alternative method is to stop when a condition is met. One such condition proposed is the difference in velocity ($\Delta v$) of the system between iterations[21]. In other word, once the amount of force applied in that iteration is lower than a scalar threshold, the calculation may stop. Taking note of stress and average force applied over multiple iterations as illustrated in figure 5.2, it is clear that Link Force converges to a complete stillness while the Chalmers algorithm reaches and fluctuate around a constant as stated in section 4.2.2. It can also be seen that stress of each layout converges a value as the average force converges a constant, indicating that the best layout each algorithm can create can be obtained once the system stabilises.

Since stress takes too long to calculate at every iteration, the termination criteria selected is the average force applied per node. This criteria is used for all 3 algorithms for consistency. The cut-off constants are then manually selected for each algorithm for each subset used. Link force's threshold is a value low enough that there are no visible changes and stress have reached near minimum. The Chalmers' threshold is the lowest possible

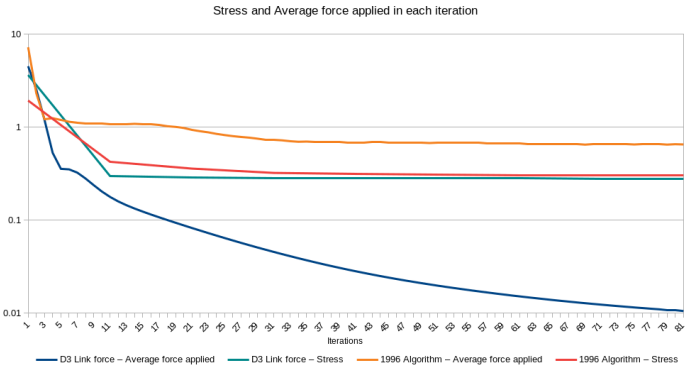



Figure 5.2: A log-scaled graph showing decreasing stress and forces applied per iteration over time converging to a constant number when running different algorithms over 10,000 data points from Poker Hands data set. Stress is calculated every $10^{\text{th}}$ iteration.

value that will be reached most of the time. It is interesting to note that with bigger subset of the Poker Hands data set, the threshold rises and converges to 0.66 from 3,000 data points onward.

By selecting this termination condition, the goal of the last phase of the Hybrid Layout algorithm is flipped. Rather than performing the Chalmers' algorithm over the whole data set to correct interpolation errors, the interpolation phase's role is to help the final phase reaches stability quicker. Thus, parameters of the interpolation phase can not be evaluated on their own. Taking more time to produce a better interpolation result may or may not effect the number of iterations in the final phase, creating the need to balance between time spent and saved by interpolation.

### 5.2.2 Selecting Parameters

Some of the algorithms have variables that are predefined constant numbers. Care have to be taken in choosing these values as bad choices could cause the algorithm to produce bad results or takes unnecessarily long computation time. To compare each algorithm fairly, a good set of parameters have to be chosen for each.

The Chalmers' algorithm have two adjustable parameters: $Neighbours_{size}$, $Samples_{size}$. According to previous evaluations[9][21], favorable layout could be achieved with values as low as 10 for both variables. Preliminary testings seems to confirm the findings and the values are selected for the experiments. In the other hand, Link force have no adjustable parameter whatsoever so no attention is required.

Hybrid layout have multiple parameters during the interpolation phase. For the parent-finding stage, there is a choice of whether to use brute-force or pivot-based searching method. In case of pivot-based, the number of pivots ($k$) have to also be chosen. Experiments have been run to find the accuracy of pivot-based searching, starting from 1 pivot to determine reasonable numbers to use in subsequent experiments. As shown in figure 5.4, the randomly selected $S$ set (the $\sqrt{N}$ samples used in the first stage) can greatly affect the interpolation result, especially with smaller data set with many small clusters. Therefore, each tests have to be run multiple times to generalise the result. From figure 5.3, it can be seen that the more pivots used, the higher accuracy and consistency. The diminishing returns can be observed at around 6 to 10 pivots, depending on the number of data points. Hence, higher number of pivots are not considered for the experiment.

Finally, the last step of interpolation is to refine the placement for a constant number of times. Preliminary testings shows that this step helps clean up a lot of interpolation artifacts. For example, a clear radial pattern and straight lines can be seen in figure 5.5a, especially in the lower right corner. While these artifacts are no longer visible in figure 5.5b, it is still impossible to obtain a desirable layout, even after more refinement steps

were added. Thus, running the Chalmers' algorithm over the entire data set after the interpolation phase is unavoidable. For the rest of the experiment, only two values, 0 and 20 were selected, representing with and without interpolation artifacts cleaning.

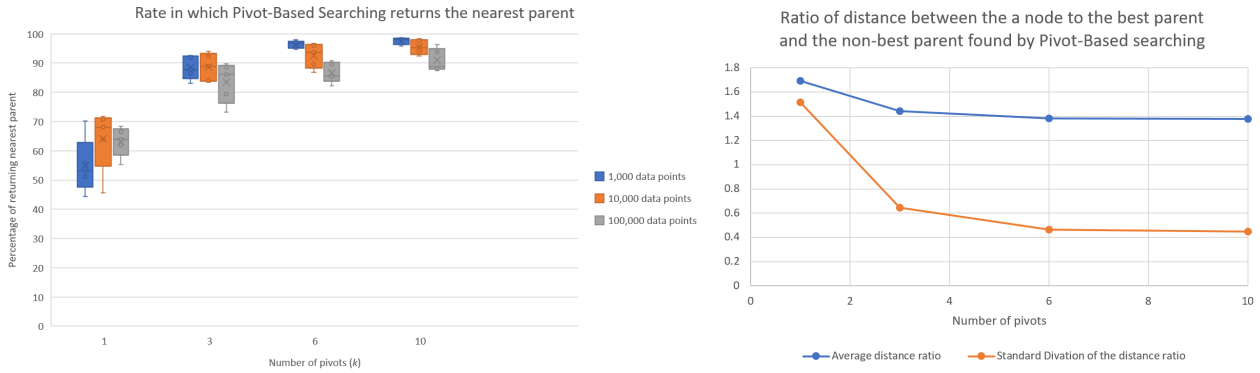
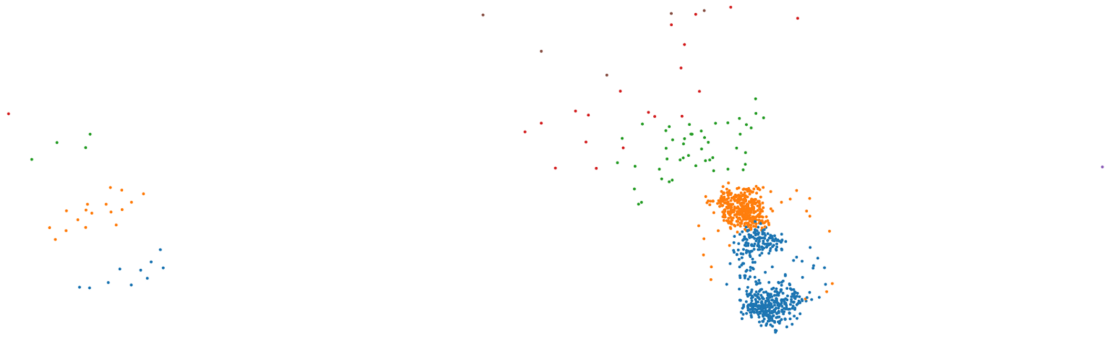
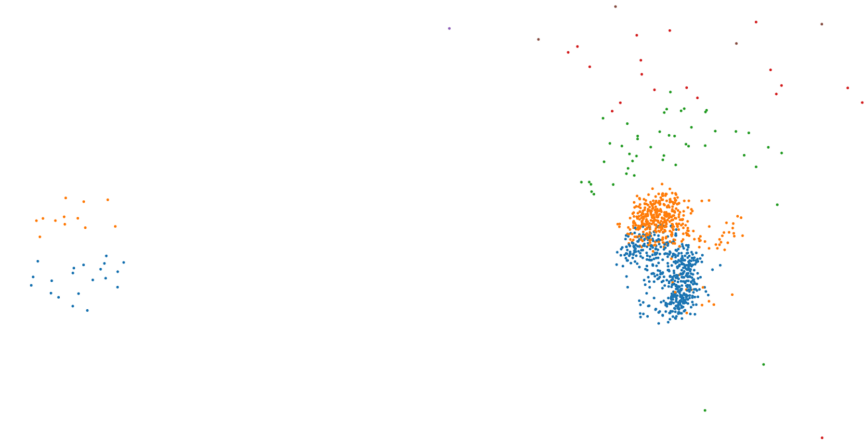



Figure 5.3: Graphs showing accuracy of pivot-based searching between $k = 1$, 3, 6 and 10. The left box-plot graph shows the percentage across 5 different runs (higher and more consistent is better). The right shows the high-dimensional distance ratio between the parent chosen by brute-force and pivot-based searching when they are not the same (closer to 1 is better). For instance, if the parent found by brute-force searching is 1 unit away from the querying node, a ratio of 1.3 means that the parent found by pivot-based searching parent is 1.3 unit away.

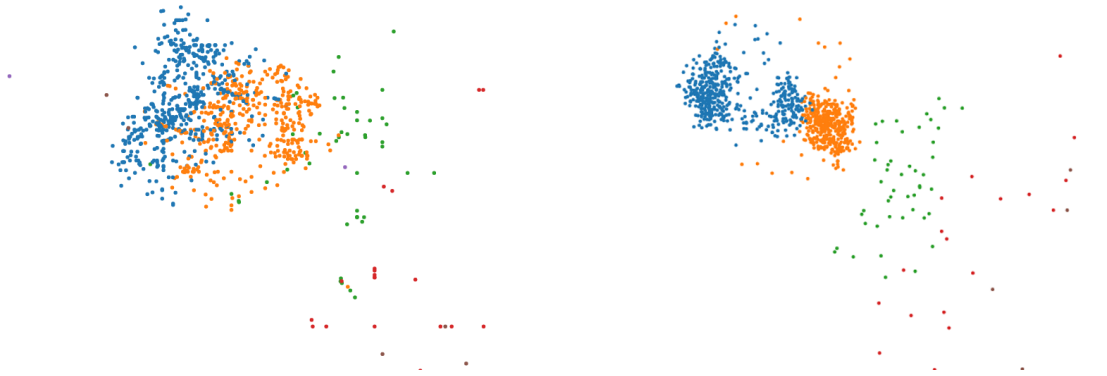


(a) An example of interpolation result with a more-balanced $S$



(b) An example of interpolation result with a less-balanced $S$

Figure 5.4: Difference in interpolation results of a subset with 1,000 data points. Left images show only data points in set $S$ and the right show the interpolation result.



(a) No interpolation refinement (b) 20 steps of Interpolation refinement

Figure 5.5: A comparison between the interpolation results

### 5.2.3 Performance metrics

As discussed in section 2.5, there are three main criteria to evaluate each algorithm: execution time, memory consumption, and the produced layout. Although stress is a good metric to judge the quality of a layout, it does not necessarily means that layouts of the same stress are equally as good for data exploration. Thus, the looks of the product itself have to also be compared. Since Bartasius have found that Link Force provide a layout with the least stress in all cases[9], its layout will be used as a baseline for comparison (recall figure 5.1).

It should also be noted that for ease of comparison, the visualisations shown in this chapter may be uniformly scaled and rotated. This manipulation should not effect the evaluation as the only concern of a spring model is relative distance between data points.

## 5.3 Results

### 5.3.1 Memory usage

Google Chrome comes with a performance profiling tools, allowing users to measure JavaScript heap usage. While it is straightforward to measure the usage of Link Force, the garbage collector gets in the way of obtaining an accurate value for the 1996 algorithm. Because the $Samples$ sets and, to a certain degree, $Neighbours$ sets are reconstructed at every iterations, a lot of new memory spaces are allocated and the old ones are left unreachable, waiting to be reclaimed. As a result, the JS heap usage keeps increasing until the GC runs even though the actual usage is theoretically constant across multiple iterations (figure 5.6). Even though GC is designed to be only be run automatically by the JavaScript engine, Google Chrome allow it to be manually called in the profiling tool. For this experiment, GC will be manually called periodically during part of the run. The usage immediately after garbage collection is then be recorded and used for comparison. The peak usage before GC automatically gets invoked is also noted.

The hybrid layout has multiple phases, each with different theoretical memory complexity. As far as the interpolation phase is concerned, the buckets storage for pivot-based finding requires the most memory at $k(S_{Size}) = O(\sqrt{N})$. Compared to $N(Neighbours_{size} + Samples_{size}) = O(N)$ of the Chalmers' algorithm used in the final phase, the overall memory requirement should be equals to that of the 1996 algorithm.

The comparison has made between the 3 algorithms with hybrid layout running 10 pivots to represent the worst case scenario for interpolation. Rendering is also turned off to minimize the impact due to DOM elements

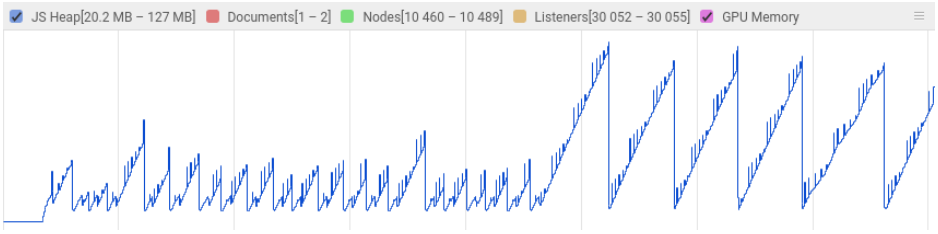



Figure 5.6: Fluctuating JavaScript heap usage due to frequent memory allocation of the Chalmers' algorithm. GC is manually invoked every second in the first half and automatically in the later.

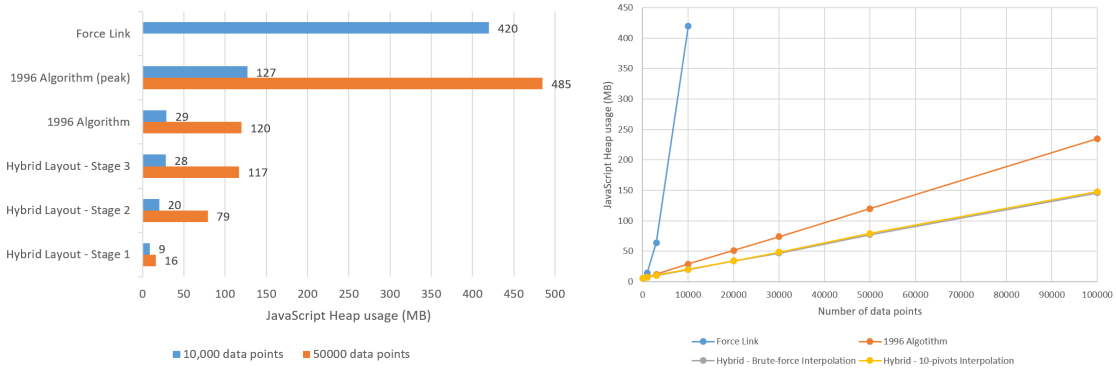



Figure 5.7: Comparison of memory usage of each algorithm

manipulation[9]. The results are displayed in figure 5.7. The modified Link Force, which use less memory compared to the D3's implementation (section 4.2.1), scales badly compared to all others, even with automatic garbage collection. The difference in the base memory usage between the 1996 algorithm and the final stage of Hybrid layout is also within the margin of error, confirming that they both have the same memory requirement. If the final phase of the Hybrid layout is skipped, memory requirement will grow at a slightly lower rate.

Although the original researchers, Chalmers, Morrison and Ross, have explored this memory aspect before, Bartasius experimented with the maximum data size the application handle before Out of Memory exception occurred[9]. A similar test is re-performed to find if there has been any changes. Due to JavaScript limitation, Link Force crashes the browser tab at 50,000 data points before any spring force is calculated, failing the test entirely. The similar behavior can also be observed with the D3's implementation. In contrast, the Chalmers' and hybrid algorithm can process as much as 470,000 data points. Interestingly, while the Chalmers' algorithm can also handle 600,000 data points with rendering, the 8GB memory is all used up, causing heavy thrashing and slowing down the entire machine. Considering that, paging does not occur when Link Force crashes the browser tab, memory requirement may not the only limiting factor in play.

All in all, since a desirable result can not be obtained from Hybrid algorithm if the final stage is skipped, there is no benefit in term of memory usage from using the Hybrid layout, compared to Chalmers' algorithm. Both of them have a lot smaller memory footprint compared to Link Force and can work with a lot more data points on the same hardware constraint.

### 5.3.2 Different Parameters for the Hybrid Layout

In section 5.2.1, it has been concluded that the value of the parameters can not be evaluated on their own. Based on findings discussed in section 5.2.2, 10 different combinations of interpolation parameters were chosen: Brute force and 1, 3, 6, and 10 pivots, each with and without refinement at the end. Due to possible variations from the sample set $S$, each experiment is also performed 5 times. Data sets used are Poker Hands with 10,000 data

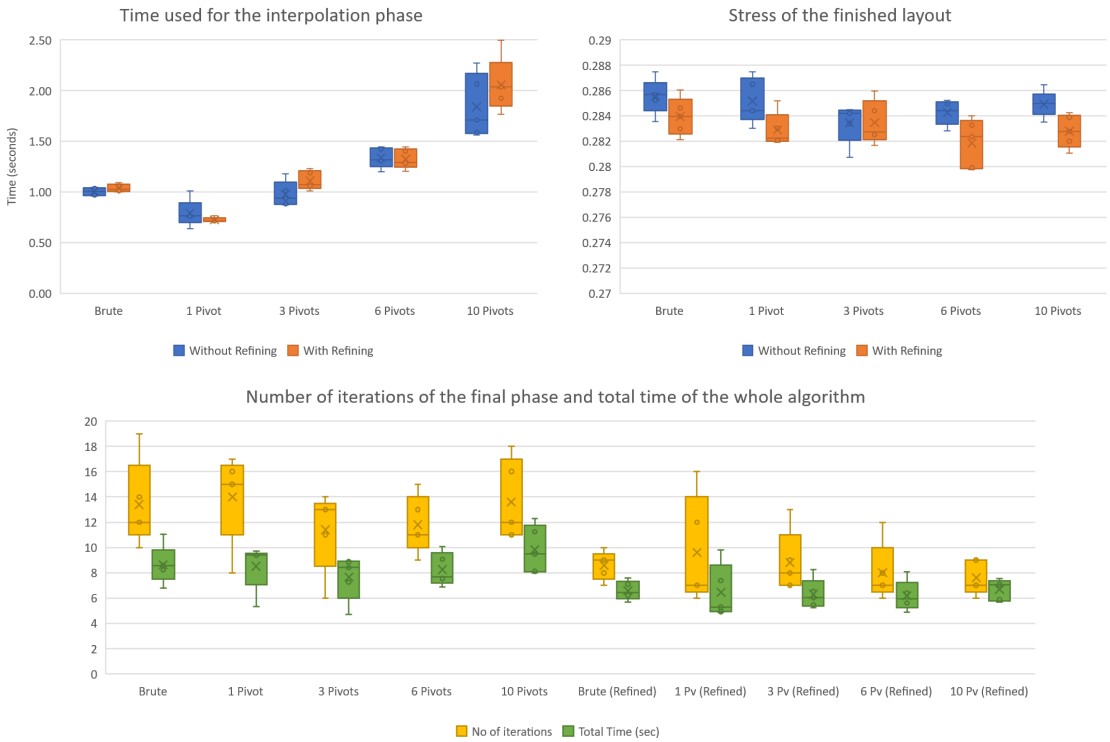



Figure 5.8: Comparison of different interpolation parameters of the hybrid layout at 10,000 data points.

points, which is the highest amount where stress can be calculated before crashing the web page, and 100,000 data points to hi-light the widen difference in interpolation time.

It should also be noted that while the original researchers had a similar experiment[20], it only explored the difference in execution time between random parent, brute-force, and pivot-based parent finders. Different number of pivots were not taken into consideration and the produced results were assumed to be equal across multiple different runs.

Figure 5.8 and 5.9 shows that most of the execution time is spent in the final phase, making the number of iterations very important. While refining the interpolation result takes an insignificant amount of time, it both reduces the stress of the final layout and help the last phase reach stability much faster across the board. Figure 5.10 also suggest that the produced layout is much more accurate. Without refining, it can be seen that a lot of "One pair" (orange) and "Two pair" (green) data points circles around "Nothing" (blue) when they should not. Thus, there is no compelling reason to disable this refinement step.

Surprisingly, despite lower time complexity, selecting higher number of pivots on the smaller data set can results in higher execution time than brute-force, negating any benefits of using it. At 10,000 data points, 3 pivots takes approximately as much time as brute-force, marking the highest sensible number of pivots to use. Looking at the lower number, the time saved from using 1 pivot does not reflect on the total time used by the layout. At 100,000 points, however, a significant speed up can be observed and is reflected in the total execution time. This suggests that pivot-based searching could shine with even larger data set and slower distance functions.

Between brute-force and 1 pivot, there is no visual difference aside from variation from run-to-run. The stress measurement seems to support this subjective opinion. On the other hand, brute-force seems to results in a more consistent total execution time. Considering that refinement is stronger with bigger data set as there are more points to compare against, it make sense that the effect of low accuracy is easily corrected in larger data set.


Figure 5.9: Comparison of different interpolation parameters of the hybrid layout at 100,000 data points. The two Box and Whisker plots are aligned for ease of comparison.

(a) No interpolation refinement

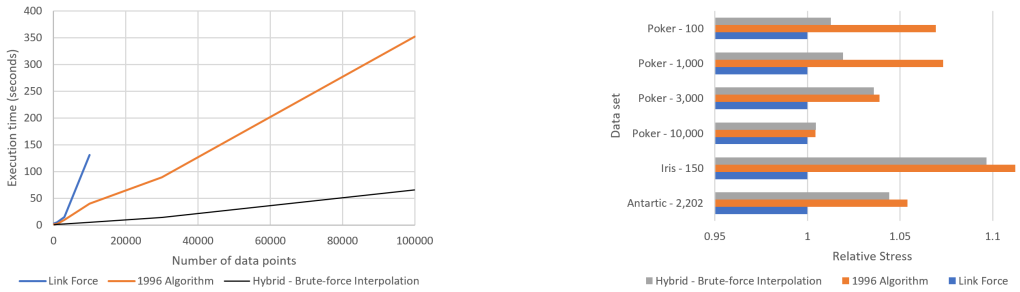
(b) 20 steps of Interpolation refinement

Figure 5.10: Comparison of the produced layout using 6 pivots, 100,000 data points

In summary, to obtain quality layout, the refining step of the interpolation phase can not be ignored. Pivot-based searching only provides a significant benefit with very large data set and/or slow distance function. Otherwise, brute-force method can consistently yield a better layout in less time.

### 5.3.3 The 3-way comparison

Figure 5.11a shows the execution time and stress of the produced layout of each algorithm with various data sets. The results reveal that the execution time of the Hybrid algorithm is superior to others across the board. The difference compared to Chalmers' algorithm is so large that the time differences from setting interpolation parameter seems insignificant. It should be noted that with smaller data sets, the processing time in each iteration can be faster than 17 milliseconds, the time between each frame on a typical monitor running at 60 frames per second. In D3-force, the processing is put on idle until the next screen refresh. As a result, the total execution time is limited to the number of iterations.



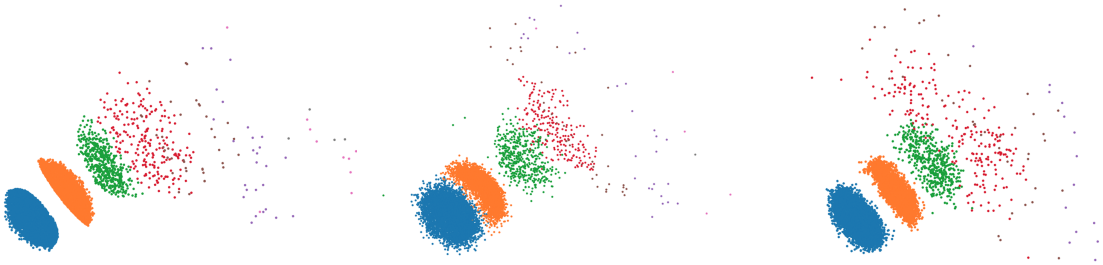

(a) Execution time for up to 100,000 data points of Poker Hands data set

(b) Relative stress of each finished layout compared to Link Force of different data sets

Figure 5.11: Comparison between different algorithms

As for the stress, a relative value is used for comparison. Figure 5.11b shows that the Hybrid algorithm results in a layout of lower stress overall. A trend also implies that the more data points available, the better the Chalmers' and Hybrid algorithm perform. In any cases, Link Force's always has the lowest stress.

Comparing the produced layout, at 10,000 data points (figure 5.12), Hybrid can better reproduce the space between large clusters as seen in the Link Force's layout. For example, "Nothing" (blue) and "One pair" (orange) have a clearer gap; "Two pairs" (green) and "Three of a kind" (red) overlap less; "Three of a kind" and "Straight" (brown) mixes together in Chalmers' layout but more separated in the Hybrid layout. However, for other classes with less data points (colored brown, purple, pink, ...), the hybrid layout fail to form a cluster, causing them to spread out even more. The same phenomenon can be observed at 100,000 data points (figure 5.13).



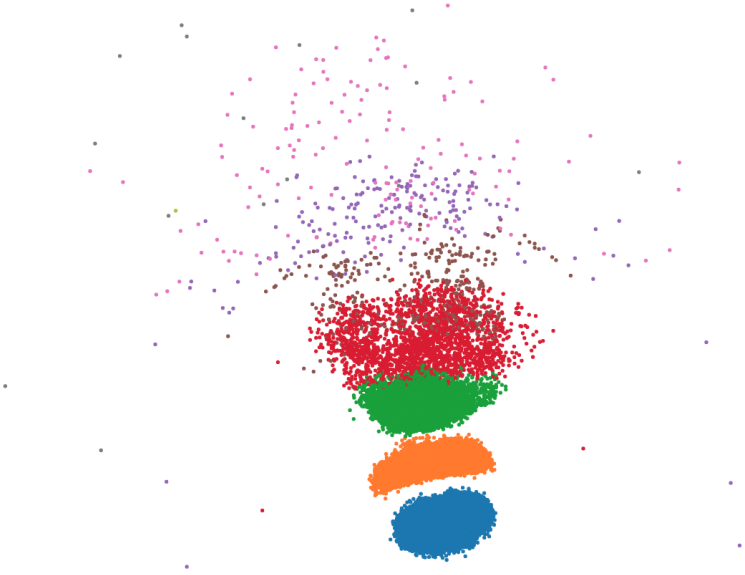
(a) Link Force

(b) Chalmers' 1996

(c) Hybrid Layout

Figure 5.12: Visualisations of 10,000 data points from the Poker Hands data set.

(a) Chalmers' 1996

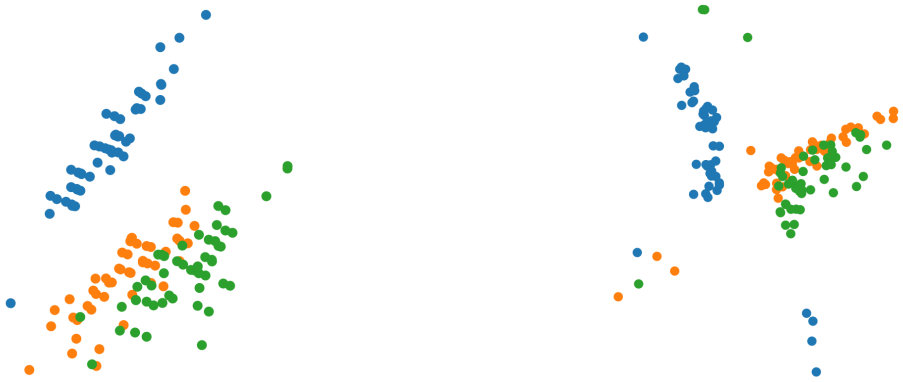


(b) Hybrid Layout

Figure 5.13: Visualisations of 100,000 data points from the Poker Hands data set.

The area where the 1996 and Hybrid algorithm fall short is the consistency in the layout quality with smaller data points. Sometime, both algorithms stop at the local minimum stress, instead of than global, resulting in an inaccurate result. Figure 5.14 and 5.15 shows examples of such occurrence. If the 1996 algorithms were allowed to continue the calculation, the layout will eventually reach the true stable position, depending on when a right combination of $Samples$ set is randomized to trip the system off its local stable position.

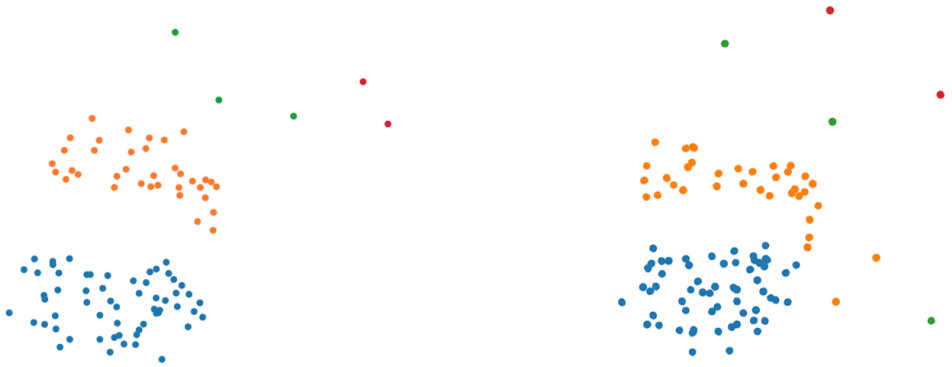
Moving to the Antartica data set with a more complicated pattern, all three algorithms produces a very similar results (figure 5.16). The big clustering difference is located around top center of the image. In Link Force, day 17 (brown) and 18 (lime) are lined up clearly, compared to others that fail to replicate the fine detail. Hybrid layout also fail to distinguish day 17, 18, 19 (pink) and 20 (grey) from each other in that corner. Aside from that, Hybrid form a layout slightly more similar to that of Force Link. Considering that the time used by Link Force, 1996 and Hybrid are approximately 14, 8, and 3.2 seconds respectively, it is hard to argue against using the Hybrid layout.



(a) A typical result from the 1996 Algorithm

(b) A result from the 1996 Algorithm when the layout stabilise at a local minimum.

Figure 5.14: Results from the Chalmers' 1996 algorithm on the Iris data set with the exact same parameters.



(a) A result with less stress.

(b) A result with higher stress.

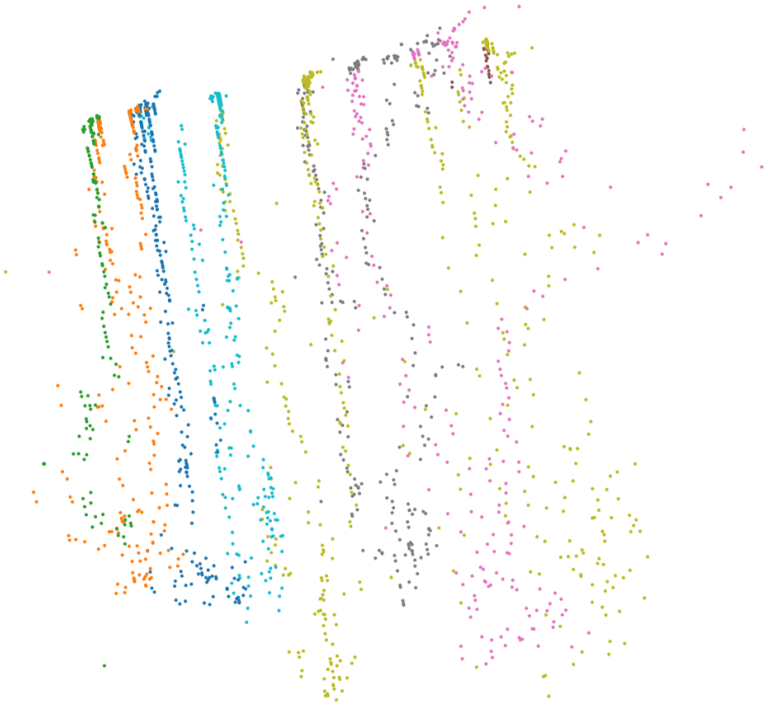
Figure 5.15: Variations of the results from the Hybrid on 100 data points from the Poker Hands data set with the same parameters.

## 5.4 Summary

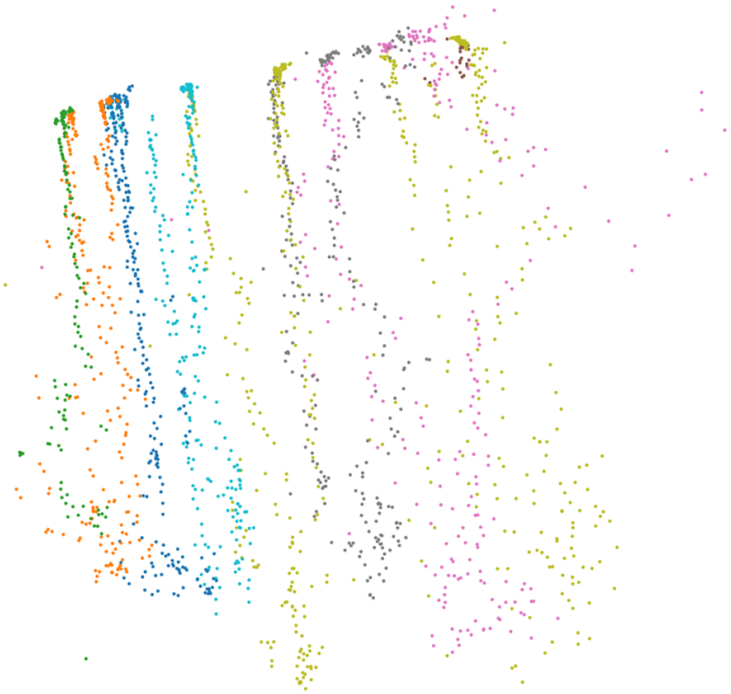
Each algorithm demonstrates its own strengths and weaknesses in different tests. For smaller data sets with a few thousand data points, Link Force works great and perform consistently. Most information visualisations on a web page will not hit the limitation of the algorithm. In addition, it allows the real-time object interactions and produces smooth animations which might be more important to most users. However, for a fully-connected spring model with over 1,000 data points, the startup time spent on distance caching starts to become noticeable and the each iteration can takes longer than 17ms time limit, dropping the animation below 60fps, causing visible stuttering and slowdown. Its memory-hungry nature also limits the ability to run on lower-end computers that a significant margin of the Internet users possess.

When bigger data sets are loaded and interactivity is not a concern, performing the Hybrid layout's interpolation strategy before running the 1996 algorithm results in a better layout in a shorter amount of time. It should be noted that, this method does not work consistently with smaller data set, making Link Force a better option. As for interpolation, simple brute-force method is the better choice in general. Pivot-based searching does not significantly decrease the computation time, unless a very large data set is concerned, and the result is less predictable.

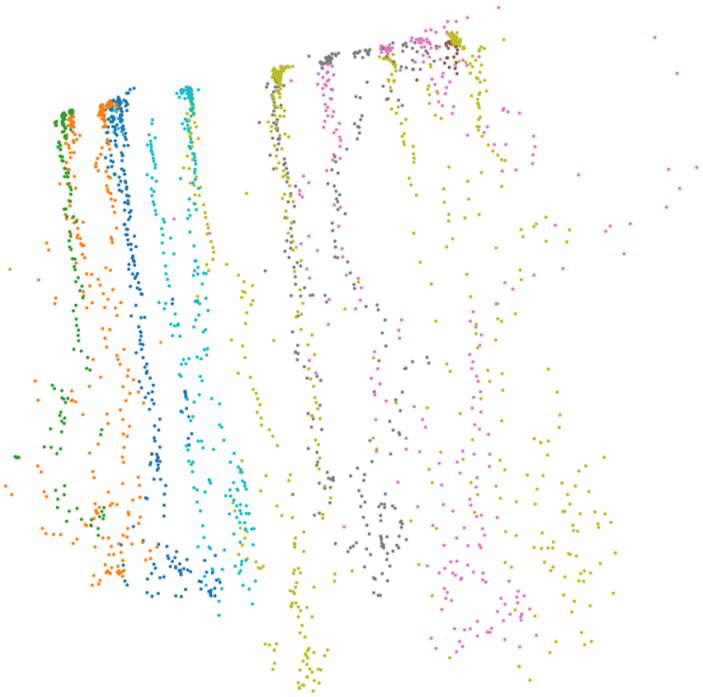
Looking back at the older Java implementation from 2002 running on Intel® Pentium III[21], it used to be



(a) Link Force



(b) Chalmers' 1996



(c) Hybrid Layout

Figure 5.16: Visualisations of the Antartica data set, color-keyed to Day.

that a 3-dimensional data set with 30,000 data points requires over 10 minutes to run, using Chalmers' algorithm and approximately 3 minutes using the Hybrid algorithm[20]. Comparing to now where 30,000 data point of Poker Hands, even with parameters stored as text-keyed dictionary object rather than an memory-offset-based class, can be visualised in 1.5 minutes with Chalmers' and 14 seconds with the Hybrid algorithm, it is clear that performance of general consumer devices have improved greatly.

Overall, these algorithms are all valuable tools. It depends on the developer to use the right tool for the application.

# Chapter 6

# Conclusion

## 6.1 Summary on the project achievements

The following is the summarized list of work from the beginning to the end, over the course of two semesters.

- **Studied algorithms:** Each algorithm and relevant researches were studied and understood.
- **Researched and assessed libraries:** Open-source JavaScript libraries were looked into. D3-force module was inspected and assessed for potential faults.
- **Modified D3 Link Force:** The D3 Link Force implementation was forked and optimized for use with a complete graph such as multidimensional scaling using spring model. The applied force tracker was also added, allowing the user to stop the simulation once the system stabilised.
- **Modified d3-neighbour-sampling plug-in:** Chalmers' implementation was tweaked to scale the applied force against a constant, rather than relying on a decreasing value. The applied force tracker was added and the evaluation application interface was updated to include newer algorithms.
- **Implemented interpolation algorithms for hybrid layout:** Interpolation functions were implemented with support for both pivot-based and brute-force pivot finding.
- **Implemented Hybrid simulation controller:** A JavaScript object was created as part of the plug-in to control a D3 Simulation object through the 3 phases of Hybrid layout algorithm.
- **Evaluated interpolation parameters:** Since the interpolation process have many parameters, several values were tested and the impacts were evaluated, both independently and as a whole system. A good combination of parameters was found after experiments.
- **Compared the three algorithms:** Each algorithm's strength and weakness was identified and compared against each other. Link Force was found to only work well in small data set but does not scales while Hybrid Layout only perform nicely on larger ones.

## 6.2 Learning Experience

The project had many challenges that helped me learn of both software engineering and research practices. Working with older research papers, I have met with a lot of ambiguity in an otherwise thorough-looking description.

In terms of Software Engineering, both D3 force and d3-neighbour-sampling do not have a documentation on interfaces between each component. A lot of time were spent figuring out how each object interacts with each other and what the flow of the system is. At the same time, the free and open-source license of D3 allowed me to easily access the source code to learn and customise components such as Link Force. This project also helped me expand my knowledge of client-side web application technologies and its fast development.

As a result of evaluating this project, I believe that I have a better understanding of designing and conducting experiments on software performance. Furthermore, I also gained a valuable knowledge in JavaScript behaviour and the limitation of each performance profiling tool in different browsers.

## 6.3 Future Work

There are several areas of the project that was not thoroughly explored or could be improved. This section shows several directions that can enhance the application.

- **Incorporating Chalmers' 1996 and Hybrid interpolation algorithms into D3 framework:** Currently, all the implementations are published on a publicly-accessible self-hosted Git Server as a D3 plug-in. While the hybrid algorithm as a whole seems to make more sense as a user application implementation, the improved Chalmers' algorithm and the interpolation functions could be integrated to the core functionality of the D3 library.

- **Data Exploration Test:** The project focuses on overall layouts produced by each algorithm and a single Stress metric. One of the goal of MDS is to explore data, which has not been assessed. A good tool and layout should help users identify patterns and meanings behind small clusters with less effort. The project could be extended to include data investigation tools.

- **Data Sets:** The evaluation focuses mainly on 1 data set. It is possible that the algorithms could behave differently on different data set with different dimensionality, data types and distance functions. Hence, findings in chapter 5 may not apply to all.

- **Optimal parameters generalisation:** So far, only good combinations of parameters were determined for a specific data set. These values may not be universally optimal and can vary from data set to data set. Even the threshold value to stop Chalmers' algorithm also varies for different size of subset of the same Poker Hands data set. Future researches could be conducted to find the relation between these parameters to other information about the data set.

- **GPU rendering:** The use of GPU for general-purpose computing (GPGPU) is gaining popularity because GPU can perform simple calculations in parallel much faster than CPU. In 2017, Khronos group have introduced WebCL[5], OpenCL-like standard for web browsers. However, it have never gained any popularity and was not adopted by any browser.

  Other efforts such as gpu.js[6] turns to use OpenGL Shading Language (GLSL) on WebGL instead. While the latest WebGL 2.0 does not support Compute Shader due to the limiting feature set of OpenGL ES 3.0[7], all of the mathematical operations used in the algorithms in this project are supported. Following the approach, Chalmers' and the interpolation algorithms could be ported to GLSL in the future.

- **asm.js and WebAssembly:** As discussed in section 3.1.2, coding in lower-level languages such as C and C++ and compiling them to asm.js or WebAssembly could speed up the execution time. During the period of the project, support for WebAssembly has been growing with more learning resources available online. It is now supported on many major web browsers such as Firefox, Chrome, Safari, and Edge. The project could be ported to these languages to potentially reduce the execution time even further.

- **More-efficient hashing algorithms for parent finding:** Over the decade, the field of machine learning and data mining have gained a lot of interest. Many improvements were made to solving related problems, including high-dimensional near neighbour searching. Newer algorithms such as data-dependent Locality-Sensitive Hashing[8], could provide a better execution time or more accurate result. Future researches can be carried out to incorporate these newer algorithms into the interpolation process of the Hybrid layout and evaluate any difference they make.
- **Multi-threading with HTML5 Web Workers:** By nature, JavaScript is designed to be single-threaded. HTML5 allow new processes to be created and ran concurrently. These workers have isolated memory space and are not attached to the HTML document. The only way to communicate between each other is message passing. JSON objects passed are serialized by the sender and de-serialized on the other end, creating even more overhead. Due to the size of the object the program have to work with, it is estimated that the overhead will out weight the benefit and support was not implemented. In the future, the support can be added to verify the hypothesis.

## 6.4 Acknowledgements

I would like to thank Matthew Chalmers for his guidance and feedback throughout the entire development process.

# Appendices

# Appendix A

# Running the evaluation application

The web application can run locally by loading a single HTML file. It is located at

```
examples/example-papaparsing.html
```

The data sets used can also be found at `examples/data`.

Please note that a modern browser is required to run the application. The project focuses on Chrome 61 but was also tested on Firefox 57. Older versions and other modern browsers should also work.

Most of the settings are available on the web interface. However, the cut-off value to stop the simulation when the system stabilises is not. To change the values, navigate to

```
examples/js/algos/[algorithmName].js
```

and edit the parameter of `stableVelocity()` method.

Aside from the Poker Hands data set, all tests uses the "General" distance function, which is similar to the Euclidean distance but scales distances per feature and supports strings and dates. It also ignores `index` and `type` field in the CSV file so that the label of the Iris data set is not taken into account.

Euclidean distance function ignores `class`, `app`, `user`, `weekday` and `type` fields. This is for other data sets not used in this project. It will also crash when other fields contain non-number values. The error can be seen in the JavaScript console.

To calculate the Stress of the current layout, open the JavaScript console and run

```
d3.calculateStress(nodes, function (s, t) {
  return distanceFunction(s, t, props, norm);})
```

The web page will freeze until the calculation is finished and returned to the JavaScript console. It should be noted that calculating Stress of the layout with more than 5,000 data points can take a long time and may crash the browser tab.

# Appendix B

# Setting up development environment

The API references and instruction for building the plug-in is available in README.md file. Please note that the build scripts are written for Ubuntu and works on the lab's CentOS environment. However, it may have to be adapted for other distributions or operating systems. A built JavaScript file for the plug-in is already included with the submission, hence re-building is unnecessary.

For the plug-in, dependencies have to first been fetched and installed. Assuming that a recent version of Node.js and node package manager is already installed, run

```
npm install
```

To compile and pack the plug-in, run

```
npm run build
npm run minify
npm run zip
```

The output files will be located in the `build` directory.

To check the coding style, run

```
npm run lintcheck
```

The evaluation web page is self-contained and can be edited with any text editor without Node.js. It will load the plug-in from the `build` directory. When the new build of the plug-in is compiled, simply refresh the web page will load up the new build.

The code is currently hosted on a personal publicly-accessible Git service at `https://git.win32exe.tech/brian/d3-spring-model`. Since this is on a personal bare-metal server, it will be maintained with best-effort without guarantee.

# Bibliography


[1] Eslint - pluggable javascript linter. URL: https://eslint.org/.

[2] js-beautify. URL: https://www.npmjs.com/package/js-beautify.

[3] Papa parse. URL: https://www.papaparse.com/.

[4] Webassembly. URL: http://webassembly.org/.

[5] Webcl - heterogeneous parallel computing in html5 web browsers, Jul 2011. URL: https://www.khronos.org/webcl/.

[6] gpu.js, Jan 2016. URL: https://github.com/gpujs/gpu.js.

[7] Webgl 2.0 arrives, Feb 2017. URL: https://www.khronos.org/blog/webgl-2.0-arrives.

[8] A. Andoni and I. Razenshteyn. Optimal Data-Dependent Hashing for Approximate Near Neighbors. *ArXiv e-prints*, January 2015. arXiv:1501.01062.

[9] Remigijus Bartasius. Fast force-directed layout algorithms for the d3 visualisation toolkit. Bachelor's dissertation, School of Computing Science, 3 2017.

[10] S. Becker, S. Thrun, and K. Obermayer. *Advances in Neural Information Processing Systems 15: Proceedings of the 2002 Conference*. Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference. MIT Press, 2003. URL: https://books.google.co.uk/books?id=AAVSDw4Rw9UC.

[11] Ingwer Borg and Patrick J. F. Groenen. *Modern multidimensional scaling: theory and applications*. Springer, 1997.

[12] Michael Bostock. Data-driven documents. URL: https://d3js.org/.

[13] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011. URL: http://dx.doi.org/10.1109/TVCG.2011.185, doi:10.1109/TVCG.2011.185.

[14] Matthew Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 127–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press. URL: http://dl.acm.org/citation.cfm?id=244979.245035.

[15] Christopher Chatfield and Alexander J Collins. *Introduction to multivariate analysis / Christopher Chatfield*. 01 1980.

[16] P. EADES. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984. URL: https://ci.nii.ac.jp/naid/10024861912/en/.

[17] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. URL: http://dx.doi.org/10.1002/spe.4380211102, doi:10.1002/spe.4380211102.

[18] M. Lichman. UCI machine learning repository, 2013. URL: http://archive.ics.uci.edu/ml.

[19] Robert C. Martin, James O. Coplien, Kevin Wampler, James W. Grenning, Brett L. Schuchert, Jeff Langr, Timothy R. Ottinger, and Michael C. Feathers. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, 2016.

[20] Alistair Morrison and Matthew Chalmers. Improving hybrid mds with pivot-based searching. In *Proceedings of the Ninth Annual IEEE Conference on Information Visualization*, INFOVIS'03, pages 85–90, Washington, DC, USA, 2003. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=1947368.1947387.

[21] Alistair Morrison, Greg Ross, and Matthew Chalmers. A hybrid layout algorithm for sub-quadratic multidimensional scaling. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, INFOVIS '02, pages 152–, Washington, DC, USA, 2002. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=857191.857738.

[22] Greg Ross. *An Algorithmic Framework for Visualising and Exploring Multidimensional Data*. PhD thesis, 2005.

[23] J. W. Sammon. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, C-18(5):401–409, May 1969. doi:10.1109/T-C.1969.222678.

[24] Bernhard Schlkopf, Alexander Smola, and Klaus-Robert Mller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998. URL: https://doi.org/10.1162/089976698300017467, arXiv:https://doi.org/10.1162/089976698300017467, doi:10.1162/089976698300017467.

[25] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[26] Jarkko Venna and Samuel Kaski. Local multidimensional scaling. *Neural Networks*, 19(6):889 – 899, 2006. Advances in Self Organising Maps - WSOM05. URL: http://www.sciencedirect.com/science/article/pii/S0893608006000724, doi:https://doi.org/10.1016/j.neunet.2006.05.014.

[27] Alon Zakai. Big web app? compile it! URL: http://kripken.github.io/mloc_emscripten_talk.

[28] Alon Zakai and Luke Wagner. asm.js speedups everywhere, Mar 2015. URL: https://hacks.mozilla.org/2015/03/asm-speedups-everywhere/.