Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

David Ferenc Bendiksen

A Way to Lay Out Argument Maps

Master's thesis in Computer Science
Supervisor: Srinivasa Rao Satti
June 2023

NTNU
Norwegian University of
Science and Technology

David Ferenc Bendiksen

A Way to Lay Out Argument Maps

Master’s thesis in Computer Science
Supervisor: Srinivasa Rao Satti
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Abstract

The author has undertaken a master's project in collaboration with Disputas AS, a Norwegian tech startup specialising in the graphing of informal logic. The goal of the project was to produce a graph layout algorithm specifically tailored to argument maps. The resulting product is Argumappr, which fulfils most of the specified requirements. It is a library for automatic graph layout generation that supports argument diagrams. It also provides the data structures necessary to represent them. The author presents the project, the data structure and the algorithm. The algorithm generates layouts by utilising the well-known Sugiyama framework for layered graph drawing. It borrows heavily from established graph drawing literature and implements various customised sub-algorithms for each of the four Sugiyama steps: cycle removal, node layering, crossing minimisation and edge straightening. The author's analysis suggests a total running time of $\mathrm{O}(|V|^4 + n^k)$, where n is some function of V and E, and $1 < k \in \mathbb{R}$. Additional testing and refinements are recommended. To the author's knowledge, the application of graph drawing theory to drawing argument maps is novel, and he hopes to inspire further inquiry into this field.

Sammendrag

Forfatteren har gjennomført et masterprosjekt i samarbeid med Disputas AS, en norsk oppstartsbedrift som spesialiserer seg innen visualisering av uformell logikk. Målet med prosjektet var å utvikle en graflayoutalgoritme spesifikt tilpasset argumentasjonsdiagrammer. Resultatet av prosjektet er Argumappr, som oppfyller de fleste av de spesifiserte kravene. Det er et bibliotek for automatisk generering av graflayouter som støtter argumentasjonsdiagrammer. Det tilbyr også datastrukturene som er nødvendige for å representere dem. Forfatteren presenterer prosjektet, datastrukturene og algoritmen. Algoritmen genererer layouter ved å benytte det velkjente Sugiyama-rammeverket for lagbasert graftegning. Den baserer seg på etablert graftegningslitteratur og implementerer forskjellige tilpassede delalgoritmer for hver av de fire Sugiyama-trinnene: syklusfjerning, laginndeling, krysningsminimering og kantutretting. Forfatterens analyse antyder en total kjøretid på $O(|V|^4 + n^k)$, der n er en funksjon av V og E, og $1 < k \in \mathbb{R}$. Det anbefales ytterligere testing og raffinering. Så vidt forfatteren vet, er anvendelsen av graftegningslære til å tegne argumentasjonsdiagrammer nytt, og han håper å inspirere til videre utforskning av dette feltet.

Preface

As of this writing, my studies at the Norwegian University of Science and Technology (NTNU) are drawing to an end. Since this project is my, for now at least, final piece of academic work, I would like this preface to be a tad bit informal. As might be expected, I feel like the Computer Science Master's Course has taken significant time to complete, but at the same time, these past five years have gone by in a flash. I owe many formative experiences and insights to my peers and the professors of NTNU. Suffice it to say a more learned man leaves the University than he who entered. For that, I am grateful. My period at NTNU now culminates in this master's thesis, which I hope may be of use to some of you.

I want to extend special thanks to the ones who made the master's project possible. First and foremost, thank you to Disputas AS for creating and proposing its problem statement; it became the grounds for my thesis. Paal Fredrik Skjørten Kvarberg and Andreas Odland Netteland were my contact persons from Disputas AS. I am indebted to them for their support and counsel during the project, but I am also obliged for the considerable freedom they gave me. Though they defined the goals, I always felt at liberty to choose my path. Last but not least, I am beholden to Srinivasa Rao Satti of NTNU for being my supervisor. For a spell, I doubted we would find a professor willing to back the project in time, but Professor Satti came through for us. I appreciate his allowing the project to naturally unfold.

At the end of this preface, I would like to thank the people whom I would not be here without. Asgeir, Halvor and Matias, I feel we did not always see each other often enough, especially during the Covid Pandemic. Please know that I will forever recall our time together at NTNU and all the fun we had. You were more than my classmates; you were my friends. Øivind, you may not know it, but I have always looked up to you as an intellectual. You have, in many ways, inspired me. Your help and friendship have been invaluable to me. Michael, our friendship has been a welcome constant in a sea of change. Few words are needed to convey how close we are. I hope to see you as often tomorrow as I do today. Sofie, my love, where would I be without you? Let us continue to take good care of each other in the coming chapters of our lives.

David Ferenc Bendiksen

Trondheim, 6th June 2023

This page was intentionally left blank.

Contents

List of Figures vi

List of Tables vii

List of Code Blocks vii

Glossary vii

Acronyms viii

1 Introduction 1

2 Background 2

3 Methods 3

4 Technologies 5

5 Theory 8

5.1 Graph Theory 8

5.2 Graph Drawing 10

5.3 Argumentation 11

6 Data Structure 12

7 Algorithm 16

7.1 Main Loop 16

7.2 Cycle Removal 19

7.3 Node Layering 21

7.4 Crossing Minimisation 25

7.4.1 Crossing Counting 31

7.5 Edge Straightening 33

8 Discussion **38**

8.1 Evolution of Methods 38

8.2 Requirement Satisfaction 38

8.3 Testing Limitations 39

8.4 Time Complexity 40

8.5 Further Work 42

9 Conclusion **43**

References **46**

List of Figures

1 The logos of the version control solution of the project 5

2 An unofficial but popular JavaScript logo 5

3 The TypeScript logo 6

4 The logos of the runtime environment and package manager of the project . 6

5 The logos of the testing technologies of the project 7

6 The logos of the linting and formatting tools of the project 7

7 The TypeDoc logo 8

8 A directed graph with five nodes and five edges 9

9 A tree rooted in *a* 9

10 The three basic argument structures 11

11 A directed graph laid out using the Sugiyama framework 17

12 An illustration of how edges are *drawn* 19

13 How conjunctions are handled during layering 24

14 How warrants are handled during layering 25

15 How constraints work 28

16 How bad merging order may halt progress 29

17 How warrants are handled before crossing minimisation 29

18 How conjunctions are handled before crossing minimisation 30
19 An illustration of how Brandes and Köpf's algorithm works 36

List of Tables

1 The given algorithm requirements 3
2 Some Graphlib functions 13
3 The adjustable properties 18
4 A summary of the time complexities of the steps 40

List of Code Blocks

1 How conjunctions and warrants are represented 14
2 How some Graphlib functions are overridden 15
3 The main loop of the algorithm 18
4 The cycle removal procedure 20
5 The greedy feedback set builder 20
6 The network simplex implementation 22
7 The feasible tree builder 23
8 The crossing minimisation function 26
9 The layer sorting function 27
10 The crossing counting procedure 32
11 The coordinate assignment algorithm 34

Glossary

Agile Software development methodologies that emphasise *agility* over rigid and heavy structures. Usually encompass iterative development, adaptive planning and self-organising. 4, 38

Argument diagram See *argument map*. i, 1, 2, 11, 12, 23, 43, 44

Argument map A diagram providing a visual representation of the structure of one or more arguments. Also *argument diagram*. i, 1, 2, 16, 27, 42, 44, 45

Backend The behind-the-scenes part of an application handling data access and logic. The server. 6

Bayesian reasoning The application of Bayesian probability theory to inductive reasoning [1]. 2

Boolean A data type that can only hold either *true* or *false* as its value. 4, 26

Class A sort of template for creating virtual objects with specific behaviour. 12, 13, 15

Constructor In this context, an object constructor. A function that instantiates a virtual object. 23

Docstring A piece of text in the source code intended to document and explain a code segment. 4, 8

Frontend The user-near part of an application. The graphical user interface. 6

Functional test A test of functional requirements. Within software quality assurance, *functional* refers to what the code does (as opposed to how it does it). 4

Lexicographic Describes a sequence, meaning it is in lexicographic order (a generalisation of alphabetical order). 30–32

Linter A program that *lints*, i.e., analyses a piece of code and flags or fixes any identified errors. vi, 7

Repository In this context, a software repository, which is a virtual storage location containing the data that constitutes a piece of software and metadata. 2, 6–8, 16, 27, 35, 39

Single-threaded Indicates that the software in question only loads a single logical processing core at a time. 5

String A data type for holding text characters. 12, 13

Type In this context, a data type. A definition of what values a variable can hold. 6, 12, 14, 15, 18

Unit test A test, usually an automated one, that checks whether a piece (a unit) of software behaves as intended. 4, 7, 38, 39

Acronyms

AI artificial intelligence. 2

AS *aksjeselskap* (the Norwegian term for 'joint-stock company'). i, 2

DAG directed acyclic graph. 10, 16, 19, 21, 38, 40, 42

FAS feedback arc set. 19

FS feedback set. vii, 19–21

JS JavaScript. 4–7, 12, 14, 16, 18, 39, 40, 44

NTNU Norwegian University of Science and Technology. iii, 2

TDD test-driven development. 4, 7, 38, 44

TS TypeScript. 1, 5–8, 15, 22, 28, 30, 31, 33, 40, 44

URL uniform resource locator (a web address). 16

VCS version control system. 5, 44

1 Introduction

Graphs are invaluable tools for visualisation and understanding. Among the numerous types of graphs and their many uses, there are so-called *argument maps* or *argument diagrams* that might assist us in analysing logic. These can be useful to academics wishing to comprehend a piece of argumentation, or decision-makers may use them to weigh points against each other and identify the best way forward. There are many use cases for argument maps, but the information era has yet to produce comprehensible tools for laying out such maps. Granted, there are several available general graphing tools, and some software is aimed at argument maps, but there are none offering automatic drawing of them. Graph drawing is well-researched by now, and it should be possible to create algorithms for generating argument diagrams. As the complexity of available information continues to increase, so does the need for accessible ways of modelling it. And a program offering automatic layout creation for argument diagrams of arbitrary size would help in this regard.

This article is a follow-up to a preparatory master's report concerning the same issues. In this paper, the author will introduce a Norwegian tech startup working with logic graphing. This firm has presented some of its problems to serve as the grounds for a master's project. It is these problems that the author has addressed in his master's project, and which will be examined in this thesis. After briefly reviewing the company and its needs, the author will describe his methods and his results, before discussing various aspects pertaining to the project. One of the results will be a graph data structure that is well-suited to represent argument diagrams. The main result will be a graph layout algorithm that is specifically developed for argument maps. As far as the author is aware, this algorithm will be the first application of graph drawing methods to automatically lay out argument diagrams.

The structure of this article is as follows: Section 2 will provide some background information. The said startup, its problems and the algorithm requirements will be presented there. In Section 3, all practised methods will be explained. To shine some light on the environment the software development occurred in, Section 4 will list and concisely describe the technologies applied. Section 5 will review some relevant theory. Namely, it will touch upon *graph theory*, *graph drawing* and *argumentation* (also known as *informal logic*). In sections 6 and 7, the resulting code will be explored. While Section 6 will present the data structure necessary to represent argument maps, Section 7 will present the custom layout algorithm — i.e., the main result of the master's project. Finally, Section 8 will contain the discussion, before Section 9 summarises and concludes the thesis.

Several means may help the reader comprehend the text. Technical terms are included in the Glossary. Acronyms will be spelt out when first used, and they are listed in the Acronyms section. Comments and supplementary information will be put into footnotes. Though the theory section (Section 5) will provide a basis, several terms and symbols will be defined underway. In general, the author has attempted to make this paper as accessible as possible. Nevertheless, some knowledge of mathematics and programming will be presupposed. Specifically, an understanding of mathematical notation and TypeScript (TS) code will be presumed. This is simply because of practical limitations; not everything can be explained down to the basics here.

2 Background

Disputas AS, hereafter referred to as just Disputas, is a Norwegian startup developing IT tools that assist users in their reasoning. The company is primarily concerned with education in critical thinking, Bayesian reasoning and natural language artificial intelligence (AI). Kvarberg [2], a co-founder of Disputas, explains that they aim to '[…] build technologies that might have a strong impact on the state of the public conversation' via their three-step plan:

> First, we develop Ponder, an interactive educational platform for text analysis and critical thinking. Then we develop Hylas, an AI-powered assistant for critical literacy and persuasive writing. And finally, we create the Web of Belief, which is a logical knowledge graph for organizing thoughts in terms of their logical interconnections.

Ponder [3], the current main product of Disputas, is an educational platform providing text annotation and argument map creation. The application supports actions such as logical analysis of texts, visualisation of one's own argumentation and essay writing. Educational exercises in Ponder leverage its graphical interface and automatic assessment of free-form natural language to give students a unique way to practice and learn in subjects involving critical thinking. One of the key selling points of Ponder is its ability to automatically structure and display argument maps. Essentially, it draws graphs that represent the argumentation in a targeted text or of a user.

Having attracted the attention of both mathematicians and computer scientists, graph drawing is a fairly well-researched area (Oria gives circa 16 000 hits on the subject), and there exist several software libraries for drawing graphs (GitHub hosts almost 1 800 related repositories). The developers of Disputas have tried to employ existing graph drawing frameworks in Ponder, and they have produced reasonable results. However, they are wrestling with a problem: Argument maps contain structures that are unorthodox from a graph-theoretical standpoint. These peculiarities are discussed in Section 5.3. Because of this, no available graph drawing code offers all of the required functionality. The Disputas developers have tried to remedy this issue by applying post-processing on top of the extant frameworks. These solutions have only been partially successful, and Ponder often produces odd graph layouts. Moreover, the attempted fixes have drastically increased the complexity of Ponder's code base.

Because of the aforementioned issues, Disputas proposed a master's project that would involve researching graph drawing and creating a graph layout algorithm custom-made for argument diagrams. Note that the firm wanted a *graph layout* algorithm, not a *graph drawing* algorithm. I.e., the custom-made algorithm was not required to generate any graphics; it merely had to assign positional data to elements. A set of more specific requirements is listed in Table 1. These requirements and the terms used in their definitions will be discussed later in this article.

As of this writing, the author is an employee of Disputas. Because of his affiliation with the company, he was familiar with the graphing problems of Ponder. After being made aware of Disputas's project proposition, he asked to have this project as the grounds for his master's thesis. The representatives of Disputas concurred. After being queried by the author, Professor Srinivasa Rao Satti of the Norwegian University of Science and

Technology (NTNU) agreed to be the project supervisor. Thus, everything was in place for the project to start.

Table 1: The given algorithm requirements.

Requirement	Description
Layered DAG layout	The output should be *optimal* positions of vertices and links in a layered directed acyclic graph.
Adequate speed	Computations should be fast enough (in absolute terms) to handle graphs of expected size without degrading user experience.
Minimise edge crossing	Though a fairly obvious and standard goal, edges should not cross each other or pass through nodes.
Large graph support	Graphs containing more than 80 nodes should be supported with minimal increase in time costs.
Edge–on–edge support	1–4 nodes should be able to point to an edge, and said nodes should be allowed to be otherwise connected.
Minimal layout changes	Limiting layout changes (to not confuse users) should be considered together with *normal* quality measures.
Conjoined edge support	Multiple nodes should be able to share a single link to another node.

Source: Based on [4, Table 2] .

The initial part of the project was dedicated to research and preparation. There will be more about the structure of the master's project in Section 3. Since the project was partitioned, a preliminary report [4] precedes this thesis. Naturally, the content of the report is highly relevant, and much of the work done in this project has been based on it. Any facts or concepts that have been borrowed directly from the report will, of course, be followed by a citation per standard rules.

3 Methods

As previously mentioned, the master's project was divided into two parts: a preparatory phase and a working phase. Each of them lasted one semester.

During the preparatory phase, the author's focus was to produce a literature review. The resulting report [4] was meant to provide insight into the problem statement of Disputas and the field of graph drawing. This knowledge was to allow the author to effectively create a new graph layout algorithm that would fulfil all specified requirements. It was during the preparatory period that the author and employees of Disputas discussed and produced the requirements listed in Table 1. Project stakeholders were also identified [4, Table 1], but the overview is not included here since it is of little direct relevance. The requirements were based on identified use cases [4, pp. 2–6]. This method of creating requirements is a common practice within software engineering [5, pp. 101–137]. As tends to be the case with literature reviews, the lion's share of the preparatory report was documentation of previous work [4, pp. 7–22].

The working phase was conducted as one might expect of a software project: The author developed new code based on the specified requirements and documented the project

in this thesis. As is commonplace within software engineering (at least when agile is applied, which today means *often*) [5, pp. 72–74], there was a recurring short meeting where the author reported on his progress and any challenges, and received feedback from the *customer* (representatives of Disputas). To provide ample time for development without too much interruption, this meeting occurred once a week throughout the project period. Apart from the aforementioned, there were few formal methods applied during development since the author worked alone on the project. However, some aspects still warrant the attention of this section.

Functional testing, both via automated unit tests and manual tests by the author, was used throughout the development process. Unit tests were written based on known requirements before the targeted code segment was complete. Afterwards, the segment was developed with the aim of having all tests pass. Such an interleaving of testing and development is known as test-driven development (TDD), which is an agile method of some renown [5, p. 221].

Ideally, the software produced in this project [6] should have provided a basis for further development by others. The code should have been understandable, easy enough to use and simple to modify. I.e., it ought to have had a high degree of maintainability [7, p. 195]. To this end, various conventions were used during development. Modularity was emphasised. The project was created as an npm module (npm is discussed in Section 4) and also uses npm modules. This is the standard decomposition pattern [7, p. 210] used in Node environments (Node is discussed in Section 4). Applying this pattern also increased the portability [7, p. 195] of the project. The map structure was designed to logically partition the project, give a good overview and inform developers of where different content might lie. The algorithm was divided into a series of steps, and all code segments performing a particular *action* were extracted into functions. Functions were named as imperative phrases, clarifying their use cases. Moreover, all functions were documented by docstrings. Variables were named using as simple and natural language as possible. E.g., any boolean values were named as simplified fact statements (like `isSoftwareProject`). The ordering and clustering of functions and variables were intended to make reading and navigation easier. Modern language syntax was preferred over legacy syntax when possible and sensible.[1]

As a last note for this section, it is worth noting that the author made specific efforts to increase the portability of the project too. Section 4 will make it clear that the algorithm was implemented in a Node environment. The official modern standard for packaging JavaScript (JS) code for reuse is ECMAScript modules [8]. However, many frameworks still rely on legacy code or have developers that prefer the older CommonJS standard [9]. Support for ECMAScript modules is growing [10] but still a long way from common. Therefore, the author decided to develop the project towards supporting both CommonJS and ECMAScript modules.

[1]As an example of a sensible exception: Though JavaScript's `forEach()` was preferred over older iteration methods, conventional for loops were used when conditional *breaks* or *continues* were necessary.

4 Technologies

To implement the algorithm, the author had to decide which technologies to use. Note that, in this context, *technologies* refer to programming languages, software systems, frameworks and the like. The following will present the main technologies applied in the project and argue why they were chosen.

(a) The Git logo.

(b) The GitHub logo.

Figure 1: The logos of the version control solution of the project.

Source: Git Homepage and GitHub Homepage .

Git is the world's most popular version control system (VCS), and it is predominantly used together with **GitHub**, the world's largest Git hosting service [11, p. 165]. The Git logo and the GitHub logo are displayed in Figure 1. In software development, VCSs are essential [11, p. 10]. They help document progress, make developer collaboration easier and provide redundancy. With a VCS in place, project-crippling errors can quickly be rectified by reverting files to previous known-to-work states, and information regarding who made the breaking changes is readily available. Moreover, since several parallel versions of the project may exist at any given time, segmenting work is simple, and conflicting changes can be resolved at a later stage. The author struggled to imagine any programming projects that would not benefit from using Git. Though Disputas never had any explicit demands regardings VCSs, the firm already used Git and GitHub. Their popularity and the author's personal preferences also argued for using them.

Figure 2: An unofficial but popular JavaScript logo.

Source: Ramaksoud2000 via Chris Williams, Public domain, via Wikimedia Commons .

JS is a well-known programming language that is heavily used in web development [12]. One may see the JS logo in Figure 2 used on the internet, though it is not an official one. Strictly speaking, JS was not used in this project, but TypeScript (discussed next), which builds upon and compiles into JS, was. Therefore, JS is mentioned here. Though JS may be best known as the scripting language for web pages, several non-browser environments also use it. Among such JS-based non-browser environments, is Node.js, which is listed further down in this section. JS is both lightweight and powerful. It is multi-paradigm, meaning it supports several programming styles, and although it is single-threaded, it supports asynchronous actions by leveraging a so-called event loop.

However, a weakness of JS is that it is dynamically and weakly typed. Hence, developers have lacklustre control over how objects look and how variables are stored.

Figure 3: The TypeScript logo.

Source: TypeScript Homepage .

TS is, in simple terms, strongly typed JS [13]. The official TS logo is displayed in Figure 3. TS adds to the syntax of JS, allowing developers to specify the shape of objects and fix variables to a single type. TS compiles into JS, so it can run in any environment JS can. If type errors are encountered during compilation, the compilation halts and informs the developer of the problem. TS may also work with code editors to highlight problems in real time. Additionally, TS supports type inference, so even without the use of extra syntax, TS can provide helpful hints to developers. From the author's experience, the only times JS is preferred over TS are when projects are too small to gain a lot from TS and when projects are huge but implemented in JS, thus, making the transition to TS too expensive. Disputas already used TS extensively, so writing this project in TS made good sense. This was not only because it would make the new code a natural extension of the existing code base of Disputas, but also because it would fit well with the wish of creating an npm package. What *npm packages* are is explained in the following paragraph.

(a) The Node.js logo.

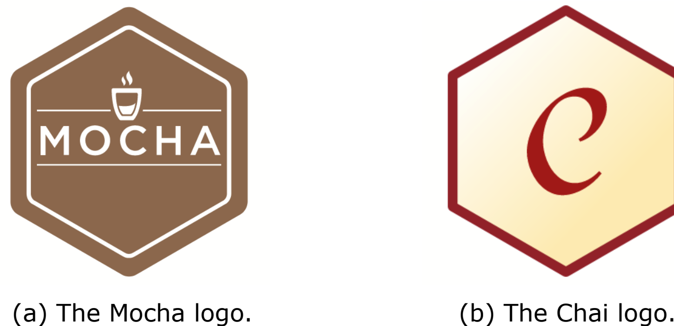
(b) The npm logo.

Figure 4: The logos of the runtime environment and package manager of the project.

Source: Node.js Brand Guide and npm Repository .

Node.js is a JavaScript runtime system that is designed to build scalable network applications [14]. Figure 4a contains the Node.js logo. Code running in a Node environment will behave just like it was running in a browser. This platform agnosticism can be practical for web developers for a number of reasons. Using Node.js, they do not have to use widely different technology stacks for frontend development and backend development. Instead, they can use the same syntax and familiar frameworks to develop all parts of an application. Moreover, Node applications, Node frameworks and smaller Node projects can all be packaged, managed and shared by **npm**, the package manager for Node.js [15]. It provides access to one of the largest developer ecosystems in the world, and one can easily manage one's own packages and implement others' packages

via the npm command line client. The official npm logo is shown in Figure 4b. Disputas, like many other companies, uses Node.js in many of their web development projects. Since Disputas wanted to use the graph drawing algorithm to dynamically calculate layouts on the client side, using Node.js and creating a new npm package for this project appeared best.



(a) The Mocha logo. (b) The Chai logo.

Figure 5: The logos of the testing technologies of the project.

Source: Mocha Homepage and Chai Repository .

Mocha and Chai were used together to form the testing solution for the project. Mocha is a JS test framework that supports Node.js [16]. Figure 5a displays the logo of Mocha. The Mocha framework is simple to use, yet powerful. It was well-suited to create unit tests for this project. The Mocha test suite reports which tests passed, which tests did not pass and in what way they failed, providing invaluable information to the developer. Chai is an assertion library meant to be paired up with some JS testing framework [17]. It provides simple ways of checking various aspects of the targeted code. The Chai logo is in Figure 5b. By using Mocha and Chai together, the author was able to specify the requirements of different segments of code by writing tests. Afterwards, the author could continue developing whilst aiming to have all tests pass. This is the TDD method mentioned in Section 3. Having a solid testing framework is also very useful during debugging.

(a) The ESLint logo. (b) The Prettier logo.

Figure 6: The logos of the linting and formatting tools of the project.

Source: ESLint Homepage and Prettier Repository .

ESLint and Prettier were used to aid the author during development. ESLint is a configurable JS linter that supports TS [18]. Its purpose is to highlight potential problems in one's code, be it syntax errors, possible runtime bugs or breaches of best practices. Figure 6a displays the ESLint logo. Prettier is an opinionated code formatter that supports TS and other environments [19]. It refactors one's code to ensure a consistent style both within a file and across multiple files. Though it edits input source code to a degree, it will never make changes that would affect the behaviour of the code. I.e., code that is formatted by Prettier will always be functionally equivalent to the original code. The author had a lot of experience with these tools and viewed them as very helpful. In this project, ESLint was used together with TS to highlight mistakes and deviations from

conventions. Although ESLint may be configured to handle styling issues too, Prettier is created specifically for this purpose and was preferred by the author. Therefore, Prettier was used to automatically format files upon saving.

Figure 7: The TypeDoc logo.

Source: TypeDoc Repository .

TypeDoc is a relatively new documentation generator for TS [20]. Its logo is shown in Figure 7. As previously mentioned, all functions in the project were accompanied by docstrings. These follow the TSDoc specifications that TypeDoc is based on. TypeDoc could therefore be used read the source code files, find the docstrings and generate HTML, resulting in a page that documents the entire project. The value of such a tool should be self-evident.

5 Theory

To provide a foundation to build upon, this section will review a few theoretical concepts. In addition to the definitions contained here, many other terms and symbols will be defined underway during the presentation and discussion of the results, particularly in Section 7. This duality stems from certain theory only being of relevance to sub-components of the project and other theory having universal value (in the context of this project).

5.1 Graph Theory

Graphs are discrete mathematical structures that model pairwise relations between objects [21, pp. 641–642]. A graph $G = (V, E)$ consists of a non-empty set of *nodes* (also called *vertices*) $V \neq \emptyset$ and a set of *edges* (also called *links*) $E \subseteq V \times V$ that, respectively, represent elements and their connections. u, v and w are often used to denote arbitrary nodes. Let n be a positive integer. If there are many nodes to reference, say n nodes, the notation $v_0, v_1, \cdots, v_{n-2}, v_{n-1}$ is preferred. Note that, in this paper, the author will consistently use zero-based numbering since it is the numbering scheme used in most programming languages. If an edge connects the nodes u and v, the edge is written as (u, v), and it is *incident* with u and v [21, pp. 651–652]. Moreover, u and v are *neighbours*. The *degree* of a node is the number of edges incident to it.

Figure 8 shows an example of a *directed graph*. In this project, directed graphs are the most relevant of graph types. A graph is directed if its edges have *direction* — i.e., an edge (u, v) relates u to v but not v to u, and $(v, u) \neq (u, v)$ [21, p. 643]. Given a directed edge (u, v), we say u is the *tail node* of the edge and v is the *head node* of the edge. Furthermore, (u, v) is an *outedge* of u and an *inedge* of v. In a directed graph, the *indegree* of a vertex u is its number of inedges and is denoted $d^+(u)$ [21, p. 654].

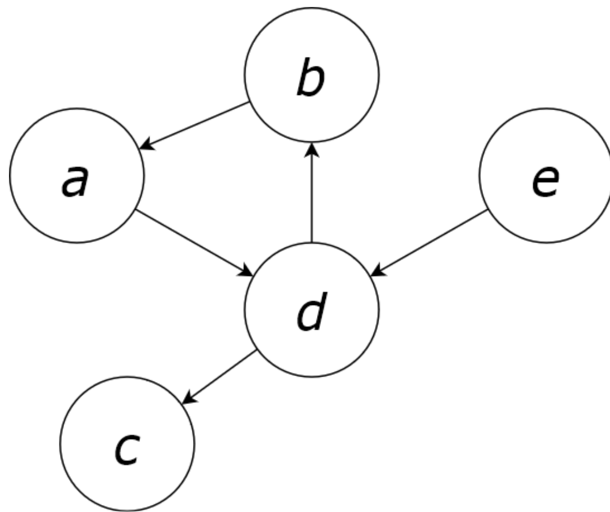


Figure 8: A directed graph with five nodes and five edges.

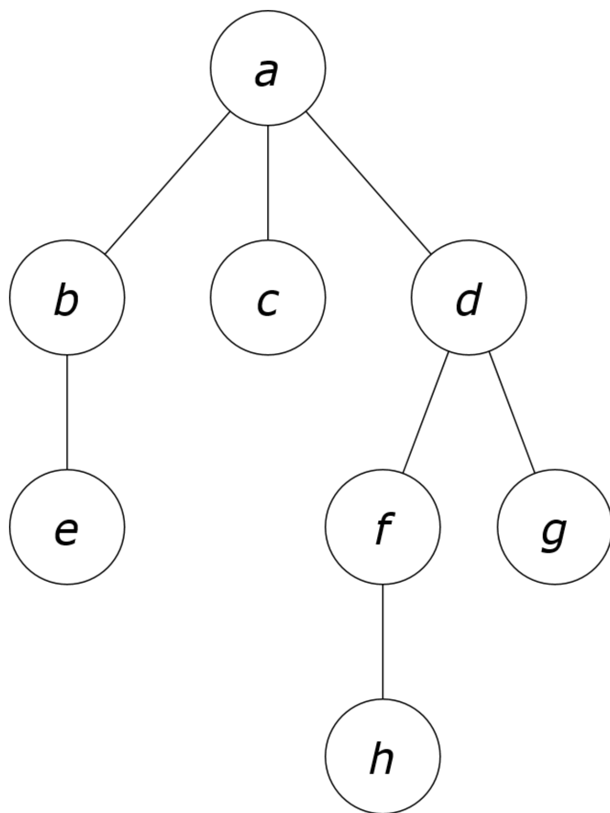


Figure 9: A tree rooted in *a*.

Likewise, its *outdegree* is its number of outedges and is denoted $d^-(u)$. The graph in Figure 8 contains one *cycle*, $\{(a,d),(d,b),(b,a)\}$, which is a path of edges that leads back to the starting node [21, p. 680]. However, it does not contain any *loops*, which are edges where the tail node and head node are the same [21, p. 643]. In other words, an edge (u,v) is a loop only if $u=v$. If a directed graph has neither cycles nor loops, it is a *directed acyclic graph (DAG)* [22, p. 224].

If an undirected graph has no simple circuits, it is a *tree* [21, pp. 746–748]. An example of a tree is shown in Figure 9. The displayed tree is a *rooted tree*. A tree is rooted if it has a node designated as its *root* and all edges are directed away from it. In the example tree, *a* is the root, and every edge is implicitly directed downwards. The three outedges of *a* lead to the three vertices *b*, *c* and *d*. These nodes are *children* of *a*, and *a* is their *parent*. A vertex that has no children is a *leaf*. *e*, *c*, *h* and *g* are the leaves in Figure 9. Trace a path from a given vertex *u* all the way to the root. All nodes encountered in this process, excluding *u* itself and including the root node, are the *ancestors* of *u*. E.g., in Figure 9, the ancestors of *g* are *d* and *a*. Construct a set of nodes *D* by adding all children of an arbitrary node *u*, adding the children of the children, the children of their children and so on until the leaves are reached. *D* is then the *descendants* of *u*. For instance, in the displayed tree, the descendants of *d* are *f*, *g* and *h*. This terminology is also used when discussing DAGs in general.

5.2 Graph Drawing

Although graphs are concepts from graph theory, graph drawing has come to be seen as a field in and of its own [22, p. vii]. There exist a myriad of different approaches to how to draw graphs, each with its strengths and weaknesses. The approaches applied in this project will be explained underway. In preparation for this, it is useful to define some terms and review what aesthetic properties are desired.

A *graph drawing* is a visual representation of a graph [22, p. 2]. Specifically, a drawing Γ of a graph G is a mapping that associates every node $u \in V$ to a particular point $\Gamma(u)$ of the plane and every edge $(u,v) \in E$ to a simple curve $\Gamma(u,v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$. Technically, Figure 8 and 9 are not graphs but graph drawings. In this project, so-called *layered graph drawings* are the most relevant. These are a type of *hierarchical graph drawings* [22, p. 409]. Put simply, a graph drawing is layered when it presents nodes in a series of discrete, parallel *layers*. Incidentally, Figure 9 is a layered drawing while Figure 8 is not. A formal definition of a hierarchy follows [22, p. 411]. Consider a graph G with a mapping $\lambda : V \to \{0, 1, \ldots, n-2, n-1\}$ where $0 \leq n < |V|$ such that V is partitioned $V = V_0 \cup V_1 \cup \ldots \cup V_{n-2} \cup V_{n-1}$, $V_i = \lambda^{-1}(i)$, $i \neq j \Rightarrow V_i \cap V_j = \emptyset$ and $\lambda(v) = \lambda(u) + 1$ for every edge $(u,v) \in E$. Such a graph $G = (V, E, \lambda)$ is a *level graph*, and a *hierarchy* is a level graph where $\forall v \in V_i,\ 0 < i : \exists (u,v) \in E$ such that $u \in V_{i-1}$.

Clearly, one would like to produce *good* drawings. To do that, a definition of what quality entails in a graph drawing is needed. The following aesthetic criteria provide a notion of how to ensure drawing quality [22, p. 411]:

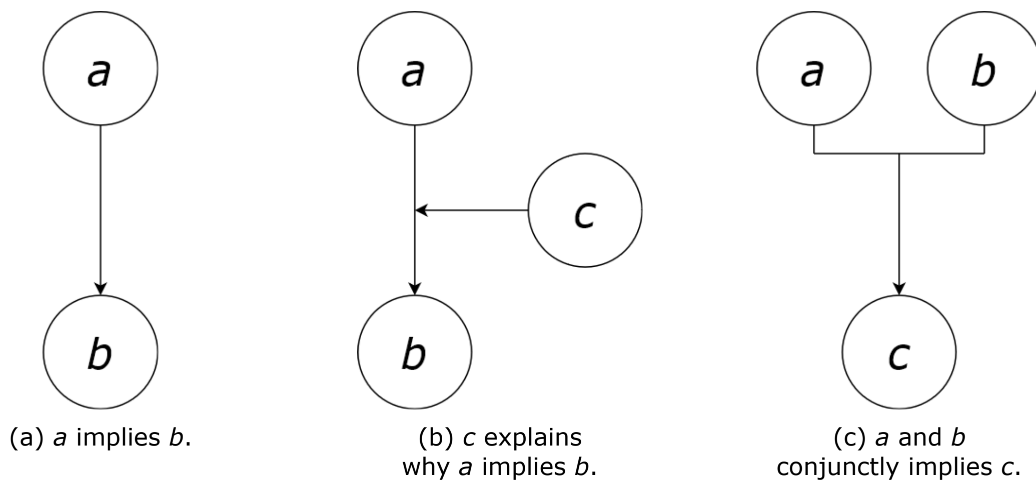
- Edges should generally point in the same direction.
- Edges should be short.
- Vertices should be uniformly distributed.

- Edge crossings should be minimised.
- Edges should be as straight as possible.

These criteria are subject to some common sense. For example, edges may very well be too short and reduce the overall quality of the drawing. Moreover, improving some measures of quality often worsens others. Therefore, one must balance the different criteria to produce good drawings.

5.3 Argumentation

The basics of logic are well-known, and everyone possesses some degree of logical intuition. But there are different ideas on how to formally structure arguments and what terms we should use. Moreover, ordinary people and experts alike can passionately disagree on what seems logical. To be clear, the sort of logic discussed here is what is called *informal logic* or *argumentation theory* [23, pp. 14–28]. This field is not orthogonal to formal logic, but it includes rhetoric and often deals with ambiguous real-life aspects — something mathematicians and certain types of logicians ignore in favour of an idealised and *pure* form of logic.



(a) *a* implies *b*.

(b) *c* explains why *a* implies *b*.

(c) *a* and *b* conjunctly implies *c*.

Figure 10: The three basic argument structures.

In the context of this project and argument diagrams, it is sufficient to understand three basic argument structures. The simplest structure, sometimes called a *simple argument*, is shown in Figure 10a. In this example, a is a premise that logically implies the conclusion b.[2] When presented with such an argument, a reasonable person might wonder why a implies b. Hence, a backing of the implication between a and b is necessary. This need gives rise to the structure in Figure 10b. Here, premise a still implies b as its conclusion, and c is a statement that provides reason to believe why the implication is valid. In the Toulmin Framework [24, pp. 89–95], a (the premise) is called the *data*, b (the conclusion) is referred to as the *claim*, and c (the implication backing) is named the *warrant*. Therefore, structures like the one in Figure 10b — a, b, c, (a, b) and $(c, (a, b))$ — will be called *warrant structures*. Links like the one going from c to (a, b) will be referred to as *warrant edges*. Because it is where the warrant edge comes from,

[2]This is basically modus ponens. I.e., $a \Rightarrow b, a \vdash b$.

c will at times be called the *warrant source*. In this article, c and $(c, (a, b))$ will be called a warrant and denoted ω. Sometimes, a single statement is not enough to deduce any conclusions. But it might be the case that one or more additional statements lead to a conclusion by logical conjunction. This leads to the structure of Figure 10c, which shows that a and b collectively form a premise that implies the conclusion c.[3] Let a premise set P consist of a and b from Figure 10c. Conjoined edges like (P, c) and its premise nodes P will be called conjunctions and denoted ς.

Note that, in argument diagrams, different argument structures may be chained and extended, resulting in complex graphs: Several statements may independently imply the same conclusion; these statements might be conclusions of yet other statements; warrants could function as parts of other arguments, and the number of statements in conjunctions is not limited. From here on out, argument diagrams will be denoted as G_{arg} in mathematical contexts.

When considering Figure 10 from a graph-theoretical perspective, one can make a couple of relevant observations. Firstly, edges between nodes and other edges (i.e., warrant structures) are unheard of. Secondly, the way conjunctions are represented means that an argument diagram is a form of hypergraph.[4] Regardless, since these diagrams were to be presented in a layered manner, they will be viewed as special cases of traditional directed graphs. The practical implications of this should be clear once the graph layout algorithm has been understood.

6 Data Structure

To represent nodes and edges, a fitting data structure was required. Instead of creating a new class from scratch, the author based himself on the graph structure of Graphlib [25]. Graphlib is a JS library for creating and modifying graphs. Its graph class is built to represent typical graph information and provides various helpful functions. In fact, it arguably offered more than what was needed for this project, which is a luxury problem.

Graphlib [25] graphs can associate any data with itself, any of its nodes or any of its edges. The associated data of an element is called a *label*, and it can be of any type. Labels are very handy for flagging elements or storing away information for later use, and they were used extensively in this project. For instance: In Section 7, various processing functions are described. The changes made in these processes must generally be undone at later stages. Information about original states is therefore held in appropriate labels.

To provide some context to the code blocks of this section and Section 7, Table 2 contains a list of all graph class functions shown. As a note on `removeNode()`, removing a node implicitly deletes all incident edges as well. Also note that when an edge class is the expected argument, one can generally provide either such a class or its constituents — a tail node (ID string), head node (ID string), label (any) and name (string, but it is only intended for undirected graphs). Table 2 is non-exhaustive in two ways. Firstly, the author utilised other functions too, but these are not used in any of the displayed code blocks. For conciseness, they were not listed. Secondly, there are many more Graphlib functions that are available but were not put to use in this project and are therefore omitted. Regardless, *all* graph class functions will be available to consumers of

[3] Mathematically, $(a \wedge b) \Rightarrow c$, $(a \wedge b) \vdash c$.

[4] This is just a technicality. Hypergraphs (graphs in which a single edge can connect several nodes at a time) are not relevant to this project.

the produced npm package [6] because, as is about to be explored, the author's graph class extends (i.e., inherits from) the Graphlib graph class. For the sake of clarity, the implementation of the modified class is split up into Code Block 1 and Code Block 2.

Table 2: Some Graphlib functions.

Name	Argument(s)	Returns
`parent()`	A node ID (string)	The meta-node containing the node
`setNode()`	A node ID (string) and a *label* (any)	N/A
`setParent()`	Two node IDs (strings)	N/A
`edge()`	An edge (edge class)	The *label* of the edge
`setEdge()`	An edge (edge class)	N/A
`removeEdge()`	An edge (edge class)	N/A
`hasEdge()`	Two node IDs (strings)	Whether the implied edge is in the graph
`hasNode()`	A node IDs (string)	Whether the node is in the graph
`removeNode()`	A node ID (string)	N/A
`node()`	A node ID (string)	The *label* of the node
`isDirected()`	N/A	Whether the graph is directed
`nodeCount()`	N/A	The number of nodes in the graph
`sinks()`	N/A	All sinks in the graph
`sources()`	N/A	All sources in the graph
`graph()`	N/A	The *label* of the graph
`edgeCount()`	N/A	The number of edges in the graph

Considering that Graphlib [25] was built for plain old discrete graphs, an obvious modification to its graph class was the addition of warrant and conjunction data structures. How this was done is shown in Code Block 1. The `setConjunctNode()` function contained in lines 4–22 takes a node u and an edge (v, w) and combines them. It presumes that a simple argument or a conjunct argument already exists and adds u as a premise. That is, either (v, w) is simple and becomes the conjunction $(\{u, v\}, w)$ or (v, w) is conjoined and u is added to its premise set. In Ponder [3], users must build arguments step-by-step. This is why an extant edge can safely be presumed. Note how a meta-node is used to contain all premises and the conjoined edge goes from this meta-node to the conclusion. Since this meta-node is just a representation of the premise set and a handy tail for the conjoined edge, it should be ignored during a rendering of the final layout. This can be done by, when drawing, skipping over any node with a label where `isConjunctNode` is set to `true`. Lines 24–43 display the implementation of `setWarrantEdge()`. It takes a warrant source–to-be v_0, a target edge (v_2, v_3) and an optional label and name for the warrant edge. Like `setConjunctNode()`, it presumes an existing argument, which it obviously has to. `setWarrantEdge()` creates a dummy node v_1 to serve as a head for the warrant edge (v_0, v_1) and then adds the edge. Because v_1 is merely a dummy target for (v_0, v_1) — which represents $(v_0, (v_2, v_3))$ — it should not be drawn during rendering. Achieving this may be done by either skipping any nodes with a `isWarrantSink` property of `true` or respecting the zero-valued `width` and `height`. A keen-eyed reader might see that lines 32–35 open for the possibility of calling `setWarrantEdge()` on an existing warrant. This is to allow modification of the warrant edge label. Lastly, note how warrants and conjunctions are not mutually exclusive. I.e., a conjoined edge may have a warrant.

```
class Graph extends graphlibGraph {
  // ...

  setConjunctNode(node: NodeId, edge: Edge) {
    let vParentNode = this.parent(edge.v);

    if (!vParentNode) {
      vParentNode = `-> ${edge.w}`;

      this.setNode(vParentNode, { isConjunctNode: true });
      this.setParent(edge.v, vParentNode);

      const edgeLabel = this.edge(edge) || {};

      this.setEdge(vParentNode, edge.w, edgeLabel, edge.name);
      this.removeEdge(edge);
    }

    this.setParent(node, vParentNode);

    return this;
  }

  setWarrantEdge(
    sourceNode: string,
    targetEdge: Edge,
    label?: any,
    name?: string
  ) {
    const dummyNodeId = `${targetEdge.v} -> ${targetEdge.w}`;

    if (this.hasEdge(sourceNode, dummyNodeId)) {
      if (label) this.edge(sourceNode, dummyNodeId, name).label = label;
      return this;
    }

    this.setNode(dummyNodeId, { isWarrantSink: true, width: 0, height: 0 });
    const edgeLabel =
      label || (this as any)._defaultEdgeLabelFn(sourceNode, dummyNodeId, name);
    this.setEdge(sourceNode, dummyNodeId, edgeLabel, name);

    return this;
  }

  // ...
}
```

Code Block 1: How conjunctions and warrants are represented.

Code Block 2 displays the rest of the graph class code (except for `setConjunctNode()` and `setWarrantEdge()`, which have already been reviewed). It shows off some further tweaks that were needed. A Graphlib [25] graph class takes a set of options when it is instantiated. These can be used to set the graph to be a directed graph, a multigraph[5] and a compound graph. The constructor in lines 2–4 simply changes the default compound setting to `true`. In this context, making a graph *compound* means that meta-nodes are allowed. These are necessary for the author's solutions. Note that Graphlib uses *parent* and *child* for *meta-node* and *sub-node*. In this article, the latter terms will be preferred, and the former terms will be used as defined in Section 5.1. Lines 6–8 have no functional effect, but they correct an erroneous return type — graph labels can be of any type. In lines 12–44, the default edge removal is overridden. The addition of warrants and conjunctions puts more requirements on it. Originally, `removeEdge()` was a JS

[5]*Multigraphs* are graphs in which several edges can connect the same pair of nodes. They are not relevant to this project.

```
class Graph extends graphlibGraph {
  constructor(options?: GraphOptions) {
    super({ compound: true, ...options });
  }

  override graph(): any {
    return super.graph();
  }

  // ...

  override removeEdge(v: NodeId | Edge, ...wAndName: string[]) {
    let _v: NodeId;
    let _w: NodeId;
    let _name: string | undefined;

    if (wAndName.length) {
      _v = v as NodeId;
      [_w, _name] = wAndName;
    } else {
      const edge = v as Edge;
      _v = edge.v;
      _w = edge.w;
      _name = edge.name;
    }

    if (this.node(_v)?.isConjunctNode) {
      this.removeNode(_v);
    } else if (this.node(_w)?.isWarrantSink) {
      this.removeNode(_w);
    }

    const possibleWarrantSink = `${_v} -> ${_w}`;

    if (this.hasNode(possibleWarrantSink)) {
      const warrantSource = this.predecessors(possibleWarrantSink)![0];
      this.removeNode(possibleWarrantSink);
      this.node(warrantSource).isWarrantSource = false;
    }

    super.removeEdge(_v, _w, _name);

    return this;
  }
}
```

Code Block 2: How some Graphlib functions are overridden.

function that counted the number of arguments and accordingly changed its behaviour. It is one of the functions that, as mentioned earlier, take an edge class or its constituents. Unfortunately, there is to the author's knowledge no elegant way of implementing two different behaviours for the same function in TS. Therefore, the trick of using a rest parameter, which allows an indefinite number of arguments, and subsequently asserting the argument types was used. This is shown in lines 12–25. The if-statement in lines 27–31 checks whether the edge to be removed is conjoined or a warrant edge. If a conjoined edge is deleted, this implicitly removes its conjunction. Hence, there is no more use for the conjunction meta-node, and it should be deleted. If a warrant edge is deleted, the dummy node (i.e., the head node of the edge) should be removed. It is also possible that an edge targeted by a warrant could be passed to `removeEdge()`. This case is handled in lines 33–39. Since the targeted edge is gone, the warrant is also deleted. No more custom logic is necessary, so the rest is delegated to the default function.

7 Algorithm

In the preliminary study [4, pp. 7–9], it was explained that Disputas currently uses the Dagre JS Library [26] for generating graph layouts. Apart from Dagre's lack of native support for argument maps, Disputas was generally happy with its performance. Because of this, the author explored Dagre and the literature cited in its documentation [4, pp. 12–14]. Though this effort proved valuable, extending Dagre was decided against, and an entirely new graph layout algorithm was produced. The new algorithm builds on previous work and established graph drawing theory. However, the application of this knowledge towards producing argument maps is novel. Additionally, the particular way of implementing methods and the special handling of argument structures is entirely new.

The customised graph layout algorithm is the primary result of this master's project, and it will be presented in this section. It is implemented in the open-source library called Argumappr [6], which contains both the algorithm and the data structure described in Section 6. Per the time of this writing, the project consists of approximately 8 000 lines of code. For brevity and clarity, only the most relevant code blocks will be included here. However, every step of the algorithm will be duly explained. The interested reader may find all written code in the Argumappr repository. Please note the visit date of the URL in this paper. Argumappr may continue to be developed after the conclusion of the master's project.

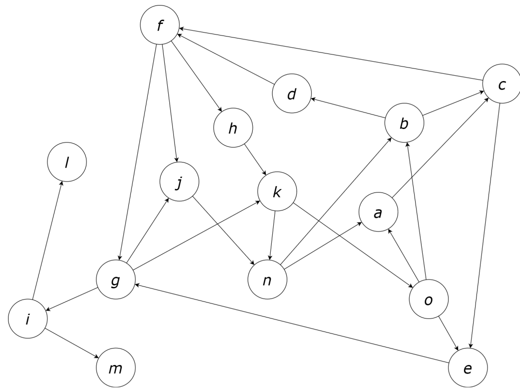
7.1 Main Loop

Firstly, it is useful to formally define the problem that needs solving. Given an argument map G_{arg}, there must be produced a layered graph drawing Γ. In addition to the regular mapping of traditional nodes V and edges E (i.e., simple arguments), warrants $\mathcal{W}$ and conjunctions $\mathcal{C}$ must be handled. Γ must map warrant edges E_ω to simple curves with one endpoint in the associated warrant source v_ω and the other in the centre of the targeted edge (u, v). This is done in Figure 10b. Further, Γ must map conjunct edges E_ς to a set of simple curves such that: Each sub-node $u \in V_\varsigma$ is an endpoint of a curve; all curves with an endpoint in a sub-node share their other endpoint A; the point A has an x-value equal to the average x-value of the sub-nodes and a y-value between the y-value of the sub-nodes and the y-value of the target node; one curve has an endpoint in A and the other in the target node. Figure 10c is drawn in this fashion.

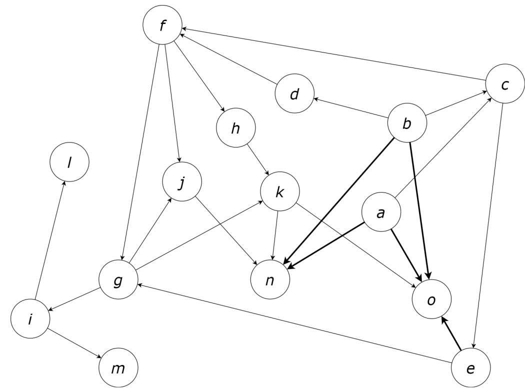
To generate layered graph layouts, the algorithm partitions the problem and solves it in four steps, not counting pre- and post-processing. These steps are:

1. Eliminate any cycles in the graph, producing a DAG.
2. Assign all nodes to discrete layers.
3. Reorder nodes within their layers to minimise the number of edge crossings.
4. Position nodes in a way that favours straight edges.

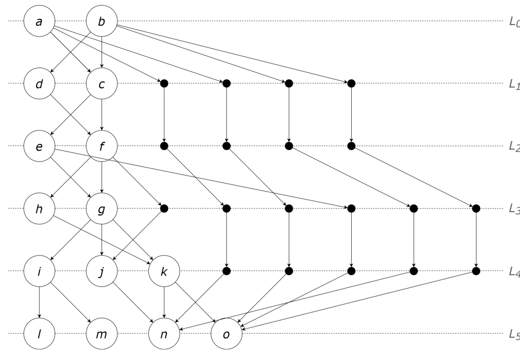
This is the Sugiyama method for layered graph drawing [27]. It is highly effective and by far the most popular approach to layered graph drawing [22, p. 410]. The process is illustrated in Figure 11. In Figure 11b, the bold edges have been reversed to make the example graph acyclic. The dark dots in Figure 11c and Figure 11d are dummy vertices.



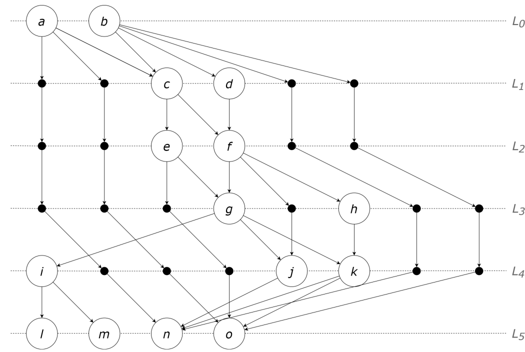
(a) A directed graph *G*.



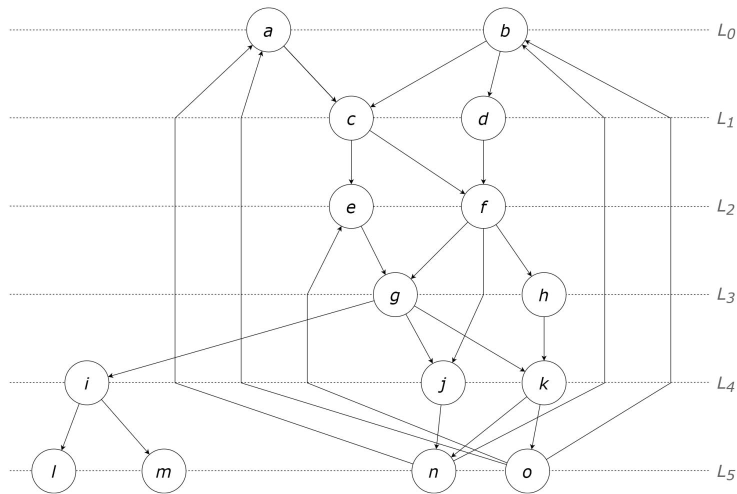
(b) *G* with its cycles removed.



(c) *G* after layering.



(d) *G* after edge crossing minimisation.



(e) *G* after edges are straightened (final drawing).

Figure 11: A directed graph laid out using the Sugiyama framework.

Source: Based on [22, Fig. 13.2] .

These are removed in the final drawing shown in Figure 11e. Each step and how they are handled will be detailed in this section.

```
function layOutGraph(graph: Graph) {
  if (!graph.isDirected()) {
    throw new Error("Graph must be directed for layered drawing");
  }

  const layoutGraph = buildLayoutGraph(graph);

  // Primary algorithm steps
  const originalEdges = removeCycles(layoutGraph); // Step 1
  const ranks = layerNodes(layoutGraph); // Step 2
  const graphMatrix = minimiseCrossings(layoutGraph, ranks); // Step 3
  straightenEdges(layoutGraph, graphMatrix); // Step 4

  restoreEdges(layoutGraph, originalEdges);
  finaliseWarrantPositions(layoutGraph);
  drawBezierCurves(layoutGraph);
  removeDummyNodes(layoutGraph);

  updateInputGraph(graph, layoutGraph);
}
```

Code Block 3: The main loop of the algorithm.

The main loop of the algorithm is displayed in Code Block 3, and lines 9–12 show the four steps of the Sugiyama framework. Each of the procedures has its own dedicated section (Section 7.2–7.5). The input is expected to be a directed graph of the type described in Section 6. To provide some protection against bad inputs, an error is thrown if a non-directed graph was provided (lines 2–4). After this assertion, the input graph is copied in line 6. During copying, default values are assigned to the graph, nodes and edges. These properties, their default value and what they control are listed in Table 3. Any properties explicitly set in the input graph will not be overwritten. This provides consumers of the function with a way of affecting how layouts are produced. The properties (except `maxrankingloops` and `maxcrossingloops`) and their use are borrowed from the Dagre JS library [26]. Once the copying is complete, the four primary steps of the algorithm are run. As previously mentioned, these will be explored further in their respective sections.

Table 3: The adjustable properties.

Name	Default	Element	Description
`ranksep`	225	Graph	Spacing between layers
`nodesep`	100	Graph	Horizontal node separation
`maxrankingloops`	100	Graph	Max iterations during layering
`maxcrossingloops`	100	Graph	Max iterations during crossing minimisation
`width`	300	Node	Node width
`minlen`	1	Edge	Minimum edge length

In Code Block 3, lines 14–17 perform post-processing that is necessary for the final layout. Since cycle removal might modify edges (see Section 7.2), any affected edges must be restored in line 14. Warrant arguments are transformed in the crossing minimisation step reviewed in Section 7.4. Therefore, they must receive some final positioning in line 15. All edges spanning more than one layer will have been split up by the crossing minimisation function. Line 17 removes produced dummy nodes and makes

the divided edges whole again. Finally, line 19 assigns all elements of the input graph their positional data.

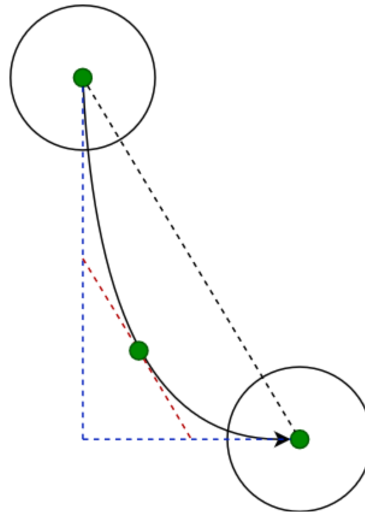


Figure 12: An illustration of how edges are *drawn*.

The astute reader might have noticed that line 16 has not yet been discussed. At that point in the algorithm, all nodes have their final positions, and edges can be *drawn*. This is done by `drawBezierCurves()`. It simply associates each edge with three points (start, middle and end) on a standard *quadratic Bézier curve* [28, pp. 303–307] from its tail node to its head node. Bézier curves are well-known and heavily used in computer graphics. Though they are derived from less than trivial mathematics, the recursive definition [28, pp. 310–311] allows one to easily use Bézier curves in programming (the so-called De Casteljau's Algorithm) and may also provide a more intuitive geometric interpretation of them. Figure 12 illustrates the concept with two non-aligned nodes. The corners of the biggest triangle are the control points. Each endpoint of the red line is in the middle of each blue line. The middle of the red line touches the Bézier curve, and this point is the centre green point. Together with the other two green points, these form the collection of points to be associated with the edge.

7.2 Cycle Removal

The first step of the algorithm is to search for and eliminate any cycles or loops. This is necessary because the layering step (see Section 7.3) must be provided with a DAG. Of course, changes made during this step must be corrected before the completion of the main algorithm. Optimally, this step should modify as few edges as possible. Any deleted edges will not be taken into account during the rest of the algorithm. Additionally, any reversed edges will be going *against the flow* in the final drawing. For a directed graph G, a *feedback arc set (FAS)* is a (possibly empty) set of edges $F \subseteq E$ that if removed, leaves G acyclic [22, p. 413]. Similarly, a *feedback set (FS)* is a (again, possibly empty) set of edges $R \subseteq E$ that if *reversed*, leaves G acyclic. Using these definitions and recalling the goal of minimising edge mutations, an optimal solution would be to reverse all edges in the smallest cardinality FS R^*. Sadly, the minimum FS problem is as hard as the minimum FAS problem [22, p. 413], which is known to be NP-hard [29]. Thus, heuristics must be resorted to.

```
function removeCycles(graph: Graph) {
  const graphCopy = buildSimpleGraph(graph);
  const { nodes0, nodes1 } = greedilyGetFS(graphCopy);
  const modifiedEdges = handleEdges(graph, nodes0, nodes1);

  return modifiedEdges;
}
```

Code Block 4: The cycle removal procedure.

The main procedure is shown in Code Block 4. In principle, the function is rather simple. In line 2, a simple copy (a copy of only nodes and edges, not any associated data) of the graph is made. It is then provided to the sub-routine that does most of the interesting work in line 3. This sub-routine, presented in Code Block 5, generates an implicit FS. It will be discussed shortly. In line 4, the information is passed on to a handler function that does two things. It reverses all edges in the FS, and it deletes any loops. The original edges and any associated data are returned so that they can be recovered in the final layout.

```
function greedilyGetFS(graph: Graph) {
  const nodes0: NodeId[] = [];
  const nodes1: NodeId[] = [];

  while (graph.nodeCount() > 0) {
    let remainingSinks = graph.sinks();
    while (remainingSinks.length > 0) {
      const sink = remainingSinks[0];
      graph.removeNode(sink);
      nodes1.push(sink);
      remainingSinks = graph.sinks();
    }

    let remainingSources = graph.sources();
    while (remainingSources.length > 0) {
      const source = remainingSources[0];
      graph.removeNode(source);
      nodes0.push(source);
      remainingSources = graph.sources();
    }

    if (graph.nodeCount() > 0) {
      const maxNode = getMaxNode(graph);
      graph.removeNode(maxNode);
      nodes0.push(maxNode);
    }
  }

  return [...nodes0, ...nodes1];
}
```

Code Block 5: The greedy feedback set builder.

The FS builder in Code Block 5 is an implementation of Eades et al.'a greedy cycle removal [30]. As just mentioned, it builds an implicit FS. Consider a graph G with n nodes. Let all nodes $v_0, v_1, \cdots, v_{n-2}, v_{n-1}$ be arbitrarily ordered on a straight line. Clearly, the set of all edges going from a higher-numbered vertex v_i to a lower-numbered one v_j, where $i > j$, is an FS. This is how `greedilyGetFS()` provides an FS. It imposes a linear ordering on all nodes, and it attempts to do so in a clever way.

Suppose two vertice sets V_0 and V_1 where $V_0, V_1 \subseteq V$ and $V_0 \cap V_1 = \emptyset$. The ith node of V_0 is denoted v_i^0, and the ith node of V_1 is denoted v_i^1. Put all sources of G into V_0, and place all sinks in V_1. Notice how edges that are incident to a source or a sink cannot be part of any cycles. Hence, in an ordering where $v_i^0 \prec v_j^1$ for all $0 \leq i < |V_0|$ and $0 \leq j < |V_1|$, all edges will be going in the same *direction*. That is, no edge will go from a higher-numbered to a lower-numbered vertex. Let $\delta(u) = d^+(u) - d^-(u)$. If G has no sources nor sinks, it seems reasonable that appending the vertex with the highest δ value to V_0 would lead to few edges going *against the flow*. I.e., doing so should not needlessly increase the cardinality of the implied FS.

Based on these observations, a greedy strategy emerges. The FS builder starts by defining two node lists. Then, as long as there are nodes in the graph, it puts all sinks into `nodes1` and all sources into `nodes0`, removing the processed nodes (and all incident edges) as it goes along. At the end of each loop, if there are still nodes left, the node with maximum δ value is appended to `nodes0`. When the graph is out of vertices, the lists are concatenated, imposing a total order on the vertices. As shown, an implicit FS attains.

Eades et al. [30] show that the size of the resulting FS is always less than $|E|/2 - |V|/6$. Let $\Delta(G)$ be the maximum degree in a graph G. In directed graphs with $\Delta \leq 3$, the FS cardinality will be at most $2/3\,|E|$. Furthermore, they show that their algorithm has a performance bound of $O(|E|)$. In the author's opinion, compared to other ways of computing FSs [22, pp. 413–417], Eades et al.'s method seems to strike a superior balance between speed and accuracy.

7.3 Node Layering

Some further definitions are necessary to explore this step. Let G be a DAG. To produce a layering is to impose a hierarchy (as defined in Section 5.2) on G. A *layering* or *ranking* $\mathcal{L}$ divides the nodes V into *layers*, which are subsets $\{L_0, L_1, \ldots, L_{n-1}, L_n\}$ where $1 \leq n$ [22, p. 417]. It does so such that if $u \in L_i$ and $v \in L_j$ where $(u, v) \in E$, then $i < j$. Layers are also known as *levels* or *ranks*. Given a vertex $u \in V$ layered in $\mathcal{L}$, its layer number is denoted $l(u, \mathcal{L})$. That is, $u \in L_i \Leftrightarrow l(u, \mathcal{L}) = i$. Denote $e = (u, v) \in E$. The *span* or *length* of e is $s(e, \mathcal{L}) = l(v, \mathcal{L}) - l(u, \mathcal{L})$. An edge e is associated with a *minimum length*, written $\delta(e) = i$ where $0 < i$ [31]. The *slack* of an edge is defined as the difference between its span and minimum length. If $s(e, \mathcal{L}) = \delta(e)$, the slack of e is zero, and we say that e is *tight*. Non-tight edges are *long*. Lastly, a ranking $\mathcal{L}$ is *feasible* if $\forall e \in E : s(e, \mathcal{L}) \geq \delta(e)$, and it is *proper* if $\forall e \in E : s(e, \mathcal{L}) = \delta(e)$.

As a note on long edges, the next steps in the layout algorithm, crossing minimisation and edge straightening (reviewed in Section 7.4 and 7.5, respectively), assume that all edges are tight. Therefore, long edges must be split up into a series of tight edges by the insertion of dummy vertices. In the author's implementation of the Sugiyama framework, the actual splitting procedure happens as a pre-processing step during crossing minimisation. Regardless, it is important to understand the idea now since layering has a direct impact on edge lengths and, therefore, the number of dummy nodes.

The problem of generating a ranking is called *the layering problem* or *the layer assignment problem* [22, p. 417]. The optimal layering of vertices is one of the few problems (in the context of this project) that can be feasibly solved with exact methods.

In this case, an optimal layering of nodes is one which introduces the fewest dummy nodes. Granted, this view of optimality might lead to layouts that are unnecessarily wide, but the resulting layouts are generally compact and readable [22, p. 427].

In this project, the author decided to implement Gansner et al.'s network simplex algorithm [31]. Gansner et al. formulate the layering problem as the following integer linear program[6]:

$$\begin{aligned} \text{Minimise} \quad & \sum_{(u,v)\in E} l(v,\mathcal{L}) - l(u,\mathcal{L}) \\ \text{subject to} \quad & l(v,\mathcal{L}) - l(u,\mathcal{L}) \geq \delta(u,v), \quad \forall (u,v) \in E \\ & l(u,\mathcal{L}) \geq 0, \quad \forall u \in V \\ & \text{all } l(u,\mathcal{L}) \text{ are integer.} \end{aligned}$$

They note that the constraint matrix is totally unimodular and that the program can therefore be solved by applying the simplex method. Further, they state that: 'Although its time complexity has not been proven polynomial, in practice [our procedure] takes few iterations and runs quickly' [31, p. 217].

```
1   function layerNodes(graph: Graph) {
2     const conjunctNodes = mergeConjunctNodes(graph);
3     const metaWarrantNodes = mergeWarrantStructures(graph);
4     const treeAndRanks = getFeasibleTree(graph);
5     const tree = treeAndRanks.tree;
6     let ranks = treeAndRanks.ranks;
7     const edgeIterator = new NegativeCutValueEdgeIterator(tree);
8     let loopCount = 0;
9
10    while (edgeIterator.hasNext() && loopCount < graph.graph().maxrankingloops) {
11      loopCount++;
12
13      const treeEdge = edgeIterator.next()!;
14      const nontreeEdge = getNontreeMinSlackEdge(graph, tree, ranks, treeEdge);
15
16      if (!nontreeEdge) continue;
17
18      tree.removeEdge(treeEdge);
19      tree.setEdge(nontreeEdge, graph.edge(nontreeEdge));
20      ranks = updateTreeValues(graph, tree, ranks, nontreeEdge);
21    }
22
23    normalizeRanks(graph, ranks);
24    balanceLayering(graph, ranks);
25    splitWarrantStructures(graph, ranks, metaWarrantNodes);
26    splitConjunctNodes(graph, conjunctNodes, ranks);
27    setYCoordinates(graph, ranks);
28
29    return ranks;
30  }
```

Code Block 6: The network simplex implementation.

Code Block 6 is the author's TS implementation of Gansner et al.'s network simplex [31, Fig. 4], and Code Block 7 shows his TS implementation of their initial feasible tree procedure [31, Fig. 5]. The general approach remains the same, and the changes that were needed are only related to the special argument elements. Luckily, the original

[6] Integer linear programming might not be what can be regarded as common knowledge, but it is beyond the scope of this article to explore. The interested reader is referred to [32].

network simplex algorithm did not need much editing to support argument diagrams. As is about to be explained, only minor pre- and post-processing was needed to make it work for this use case. Firstly, the core of the algorithm will be detailed; then an explanation of the processing will follow.

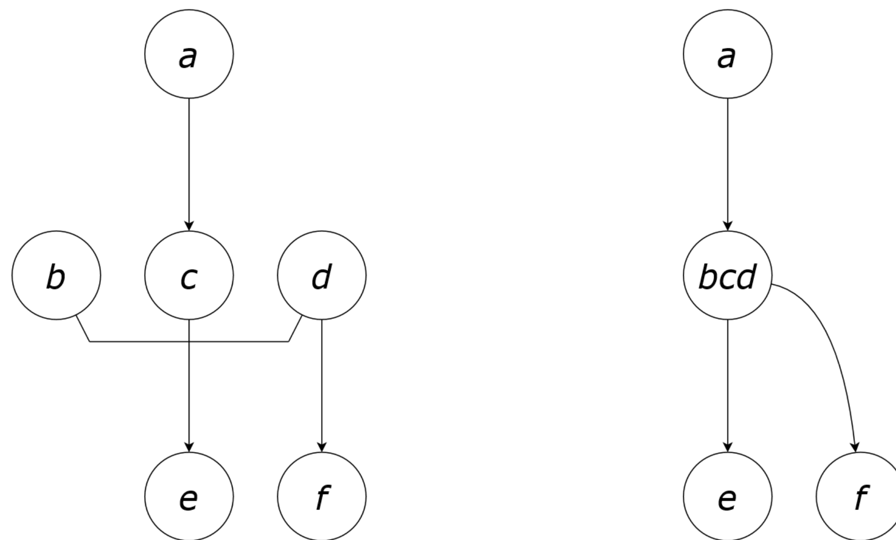
```
1   function getFeasibleTree(graph: Graph) {
2     const ranks = setRanks(graph);
3     const tree = getTightTree(graph, ranks);
4
5     if (graph.edgeCount() === 0) return { tree, ranks };
6
7     while (tree.nodeCount() < graph.nodeCount()) {
8       const { minSlack, minSlackEdge } = getMinSlack(graph, tree, ranks);
9       const { v, w } = minSlackEdge;
10      let rankDelta: number;
11      let newNode: NodeId;
12
13      if (tree.hasNode(v)) {
14        rankDelta = minSlack;
15        newNode = w;
16      } else {
17        rankDelta = -minSlack;
18        newNode = v;
19      }
20
21      tree.setNode(newNode, graph.node(newNode));
22      tree.setEdge(minSlackEdge, graph.edge(minSlackEdge));
23
24      for (const node of tree.nodes()) {
25        if (node === newNode) continue;
26        const newRank = ranks.getRank(node)! + rankDelta;
27        ranks.set(node, newRank);
28      }
29    }
30
31    setCutValues(graph, tree);
32
33    return { tree, ranks };
34  }
```

Code Block 7: The feasible tree builder.

In Code Block 6, line 4 is the start of the main part of the layering algorithm. The function it calls, `getFeasibleTree()`, will be explained afterwards. For now, consider that it somehow generates a feasible tree. In lines 5 and 6, the return value of `getFeasibleTree()` is destructured into the tree structure itself and a rank table. Line 7 hands the tree to a constructor to get an edge iterator. The iterator provides a tree edge that has a negative cut value if such an edge exists. It is designed to cyclically search through the edges of the tree since Gansner et al. claim that doing so can save many iterations [31, p. 220]. Now, a loop count is instantiated, and the optimisation loop starts. If there are no tree edges with negative cut values, or if the maximum number of allowed iterations is reached, the loop terminates. The while loop iteratively improves the layering by strategically replacing tree edges with non-tree edges and updating node ranks. In line 13, a negative tree edge is found. In line 14, a non-tree edge suitable to replace the tree edge and with as little slack as possible is found. A non-tree edge is suitable if it connects the tail and head components of the tree edge. If no such edge is found, line 16 skips to the next iteration. In lines 18–20, the tree edge is cut, the non-tree edge replaces it and the ranks of affected nodes (along with some additional data that helps make the procedure more efficient) are updated. After the loop terminates, the ranks are normalised. That is, all rank numbers are adjusted so that the smallest rank

number is 0. Line 24 marks the end of the main part of the algorithm. Here, the layering is balanced. This is done by moving nodes with $d^+ = d^-$ and several feasible ranks to the feasible layer with the fewest nodes.

The construction of an initial feasible tree happens in Code Block 7. This is an important procedure since it is often the biggest timesink during layering, and the initial solution tends to be close to optimal [31, p. 219]. In line 2, an initial ranking is computed. This is done by first assigning all sources to layer 0 and then iteratively assigning nodes whose parents have been ranked to the layer imposed by their parents. I.e., for each node u with only ranked parents V_u, assign u to layer number $\max(\{l(v, \mathcal{L}) + \delta(v, u) \mid v \in V_u\})$. A tight tree is produced in line 3 by picking an arbitrary node as the root node and adding all nodes that are reachable via tight edges to the tree. Line 5 merely handles graphs without edges. A loop aimed at adding all graph nodes to the previously constructed tree starts in line 7. Line 8 finds an edge that is incident to the tree and has the smallest amount of slack. Lines 21 and 22 add the edge and the incident node to the tree. Intuitively, when the goal is to minimise dummy nodes (and, therefore, also slack) this seems like a good way to grow the tree. The logic in lines 9–19 simply checks whether the new node is the tail or head of the incident edge and sets the appropriate sign for the slack. In lines 24–28, all tree nodes have their rank updated so that the incident edge is made tight. Lastly, after all graph nodes are part of the tree, all tree edges are assigned cut values in line 31.



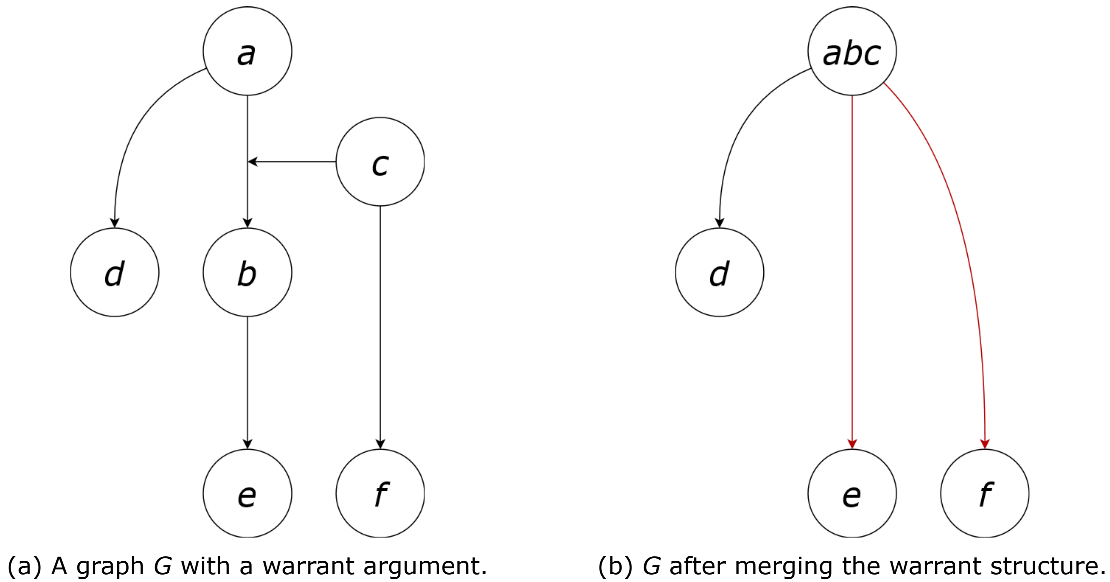
(a) A graph G with a conjunct argument.

(b) G after merging the conjunction.

Figure 13: How conjunctions are handled during layering.

Lines 2 and 3 from Code Block 6 contain the added pre-processing for conjunctions and warrants. In this case, conjunctions are simple to handle. All sub-nodes of a conjunct node are merely merged into one meta node. To be clear, any edges incident to sub-nodes are made to be incident to the meta node, and the conjoined edge is also made to be incident to it. The idea is illustrated in Figure 13. Warrant structures are a bit trickier. Not only is the source of a warrant edge positioned on a half layer (which violates the constraints of the integer linear program), but the targeted edge should also preferably not be long. This is solved by a trick similar to the one used on conjunctions. The warrant source, its edge, the target edge and the incident nodes are all merged into one meta node. Again, any edges incident to sub-nodes will be made to be incident to the meta node. Furthermore, any edges going from the warrant source or the head node

of the targeted edge are assigned a minimum length δ of 2. This process is illustrated in Figure 14, and the red edges in Figure 14b are the ones with $\delta = 2$. After the pre-processing, the layering algorithm can run its course without any further special considerations.



(a) A graph G with a warrant argument.

(b) G after merging the warrant structure.

Figure 14: How warrants are handled during layering.

The post-processing is done in lines 25–27 of Code Block 6. Line 25 restores warrant structures to their original shape. Let u be the tail node of the targeted edge, v the head node of the targeted edge and w the warrant source. Clearly, u can be assigned to the same layer as the meta node. Note how having set $\delta = 2$ for the out-edges of v and w now provides an easy way of incorporating them into the layering. The ranking process has been completed by this point, so all other nodes have optimal ranks. But children of v and w have been layered exactly one layer further down than they would have been if δ remained 1. Hence, v and w can be placed on layer $l(u, \mathcal{L}) + 1$ and $l(u, \mathcal{L}) + 0.5$, respectively. In line 26, meta conjunction nodes are split up again. All sub-nodes are simply assigned the same rank number as the meta nodes. Finally, line 27 assigns y-coordinates to all nodes. It gets the rank separation Δ_y (the `ranksep` property of the graph) and for all nodes $u \in V$, it assigns u to y-coordinate $l(u, \mathcal{L}) \cdot \Delta_y$.

7.4 Crossing Minimisation

The crossing minimisation step is also known as *the vertex ordering step* [22, p. 432]. This is because crossing minimisation is achieved through the reordering of nodes within layers. There are primarily three ways of approaching crossing minimisation: one-sided optimisation, multi-layer optimisation and planarization [22, pp. 434–441]. Based on the research of the preparatory project [4, pp. 19–20], considering the issue as a series of one-sided optimisation problems seemed to be the way to go. It has been proven that globally minimising edge crossings is NP-hard [33]. Sadly, reducing the problem to a one-sided one does not change this result [34]. Once again, good heuristics are needed.

In this step, it is assumed that the input graph G is properly layered. Remember from the definition in Section 7.3 that a layering $\mathcal{L}$ is proper if all edges are tight. Consider a pair of layers, L_{i-1} and L_i for some $0 < i < |\mathcal{L}|$. For now, let E be the set of edges between

L_{i-1} and L_i. That is, $E = \{(u, v) \mid u \in L_{i-1} \wedge v \in L_i\}$. A *permutation* π_j of a layer L_j is an ordering of the vertices in L_j. Let the nodes in L_{i-1} have some order along a horizontal line. Likewise, let the nodes in L_i be ordered along a parallel horizontal line. The *crossing count* $C(G, \pi_{i-1}, \pi_i)$ is the number of crossings among the edges in E. It is trivial to see that C is only dependent on the ordering of nodes, not their specific x-coordinates. Hence, the goal of this step is to generate the set of permutations $\Pi = \{\pi_j \mid 0 \leq j < |\mathcal{L}|\}$ that minimises $C(G, \pi_{n-1}, \pi_n)$ for all $0 < n < |\mathcal{L}|$. In one-sided approaches, this goal is achieved by pairwise considering adjacent layers and producing one permutation at a time. I.e., in each sub-problem, one layer is fixed and the other is permutated to reduce the number of edge crossings.

In the same paper [27] where Sugiyama et al. introduced the Sugiyama Framework, they propose the so-called Barycenter Heuristic for one-sided crossing minimisation. Despite its simplicity, it is one of the two most popular vertex ordering methods, and its popularity is warranted — not only is it fast, but it also produces excellent results [35, pp. 5–11]. Additionally, it will always find an ordering with no crossings if such an ordering exists. Because of these reasons, the author wanted to implement a version of the Barycenter Heuristic in this project.

```
1   function minimiseCrossings(graph: Graph, ranks: RankTable) {
2     const constraintGraph = preprocessDataStructures(graph, ranks);
3     const graphMatrix = readRankTable(ranks);
4
5     sortLayers(graph, constraintGraph, graphMatrix, true);
6
7     let orderHasChanged = true;
8     let loopCount = 1;
9
10    while (orderHasChanged && loopCount < graph.graph().maxcrossingloops) {
11      orderHasChanged = sortLayers(graph, constraintGraph, graphMatrix);
12      loopCount++;
13    }
14
15    return graphMatrix;
16  }
```

Code Block 8: The crossing minimisation function.

The main logic of the crossing minimisation procedure is shown in Code Block 8. Line 2 does some pre-processing that will be explained later. Line 3 converts the rank table to a two-dimensional array, which is better suited for tracking and changing the positions of nodes. After setting a boolean tracking whether any reordering has happened and instantiating a loop count, a while loop starts. For each iteration, it calls a barycentric sorting function and increments the loop count. The idea is that the total number of edge crossings should be reduced iteratively. If the maximum number of allowed iterations is reached or the sorting function makes no changes (meaning that the iterative improvements have *flattened out*), the loop is broken, and the final node ordering is returned. One might notice that the sorting function is called once in line 5 before the first iteration. The reason is that calling `sortLayers()` with `true` as its third argument ensures that any constraints are respected. This will be further explained in the following.

Code Block 9 displays the definition of `sortLayers()`. All this function does is *sweep* all layers, once from the top and once from the bottom, by calling the sweep layer function. It is `sweepLayer()` that implements a barycentric ordering algorithm. It will be explained next. The layer sorting function provides `sweepLayer()` with two layers at

a time. Remember that the Barycenter Heuristic is a one-sided optimisation approach. Thus, the *previous* layer in the iteration will be viewed as fixed, and only the *current* layer in the iteration will be mutated. Since the first iteration in lines 6–17 gets the previous layer by decrementing the current index, it starts at 1 instead of 0. Obviously, layer 0 has no layer previous to it. Lines 19–30 follow the same principle, but because it iterates *the other way*, it increments the current index to get the previous layer. Therefore, it starts at the final layer index minus one.

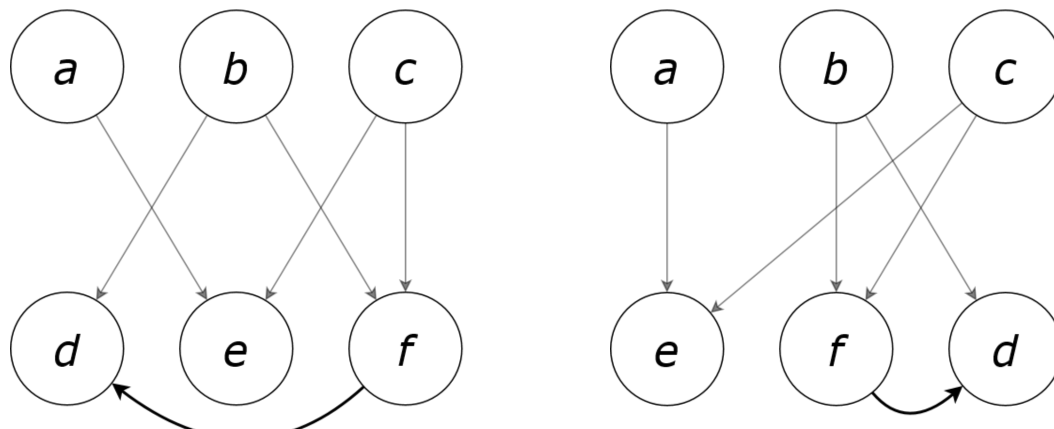
```
1   function sortLayers(
2     graph: Graph,
3     constraintGraph: Graph,
4     graphMatrix: NodeId[][]
5   ) {
6     for (let layerIndex = 1; layerIndex < graphMatrix.length; layerIndex++) {
7       const previousLayer = graphMatrix[layerIndex - 1];
8       const layer = graphMatrix[layerIndex];
9
10      graphMatrix[layerIndex] = sweepLayer(
11        graph,
12        constraintGraph,
13        previousLayer,
14        layer,
15        "down"
16      );
17    }
18
19    for (let layerIndex = graphMatrix.length - 2; layerIndex >= 0; layerIndex--) {
20      const layer = graphMatrix[layerIndex];
21      const nextLayer = graphMatrix[layerIndex + 1];
22
23      graphMatrix[layerIndex] = sweepLayer(
24        graph,
25        constraintGraph,
26        layer,
27        nextLayer,
28        "up"
29      );
30    }
31  }
```

Code Block 9: The layer sorting function.

Much of the interesting work happens in the sweep layer function. Unfortunately, even without including the definitions of sub-procedures, the function consists of almost 130 lines, so `sweepLayer()` will not have its code block included here. As always, the reader can see the full function definition in the project repository [6].

The Barycenter Heuristic of Sugiyama et al. [27] is designed around conventional and unconstrained discrete graphs. Therefore, the author based his implementation on Forster's constrained version of the Barycenter Heuristic [36]. It lent itself better for drawing argument maps. Here, a *constraint* is a limitation on how a pair of nodes, u and v, can be placed in relation to each other. In general, a constraint states that either $u \prec v$ or $v \prec u$ must hold in the linear order imposed by the ordering of nodes within a layer. Consider an arbitrary layer L_i, and let $u, v \in L_i$. Constraints are represented by a constraint graph $G_C = (L_i, C)$ where $C \subseteq L_i \times L_i$ is a set of constraint edges. A constraint edge $(u, v) \in C$ imposes $u \prec v$. Hence, an ordered layer will fulfil its constraints only if all edges in its constraint graph go *to the right*. Figure 15 displays a pair of layers with two different permutations of the lower layer. The bold edge represents a constraint. In Figure 15a, a constraint is violated, visible by its edge going *to the left*. The permutation in Figure 15b, however, does not violate the constraint. Note how the

problem of producing an ordering such that no constraints are violated only has a solution if the constraint graph is acyclic.



(a) Two layers with a violated constraint.

(b) Two layers with no violated constraints.

Figure 15: How constraints work.

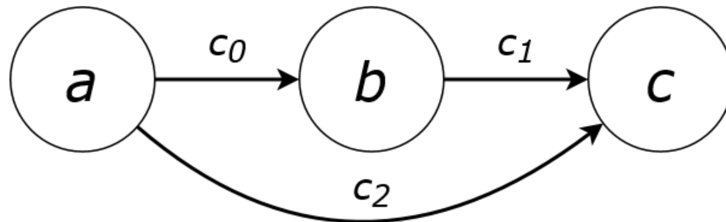
Source: Based on [36, Fig. 1] .

Now on to describing how `sweepLayer()` works. The underlying idea is the same as in the original Barycenter Heuristic [27]:

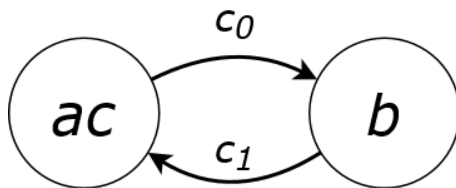
- Consider two and two layers at a time.
- Let one of the layers be fixed and the other one mutable.
- For each node in the mutable layer…
  - Find all neighbours of the node in the fixed layer.
  - Calculate the average neighbour index, the *barycenter* of the node.
- Sort the nodes of the mutable layer based on their barycenters.
- (Prioritise letting nodes keep their original positions if…)
  - (Any two nodes have equal barycenters.)
  - (Or any node has no parents and, thus, an undefined barycenter.)
- If the resulting permutation reduces the number of crossings…
  - Let it be the new order of the mutable layer.
  - Else, discard the results.

However, there is also the issue of respecting constraints. The sweep layer function is a nigh unmodified TS implementation of Forster's `CONSTRAINED-CROSSING-REDUCTION` procedure [36, Algorithm 1]. This procedure handles constraints by partitioning nodes into node lists that, initially, only contain themselves, and then concatenating them according to violated constraints. Doing this does not allow any nodes to be placed between any constrained pair of nodes. Forster argues that, although not generally optimal, doing this does not reduce result quality [36, p. 209]. The algorithm assigns all nodes in the mutable layer a barycenter per the normal approach and creates a list

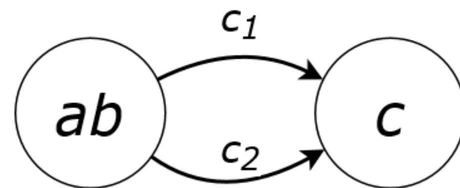
for each node. It then searches for a constraint that would be violated by the ordering implied by the barycenters. If such a constraint is found, the nodes incident to the constraint edge is merged by concatenating their node lists. This way, their position relative to each other is conserved. The new *node* resulting from the concatenation is assigned a barycenter by considering the neighbours of all sub-nodes. The search for violated constraints continues until none is found. Finally, the nodes implied by the node lists are sorted based on their barycenters.



(a) A pre-merging constraint graph G_C.



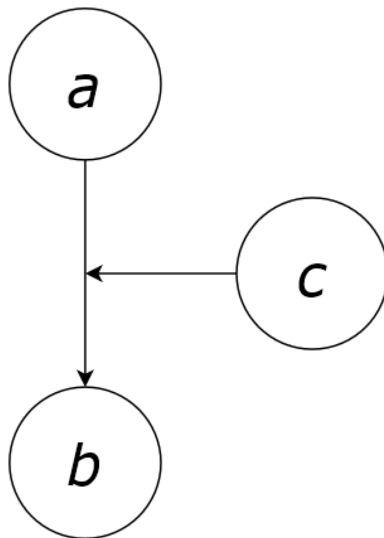
(b) G_C after merging *a* and *c* first.



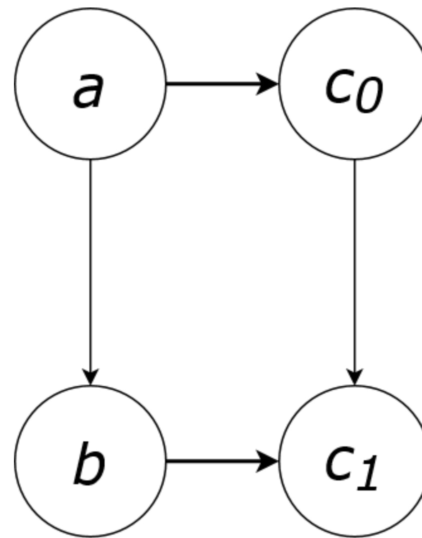
(c) G_C after merging *a* and *b* first.

Figure 16: How bad merging order may halt progress.

Source: Based on [36, Fig. 4] .



(a) A graph *G* with a warrant.



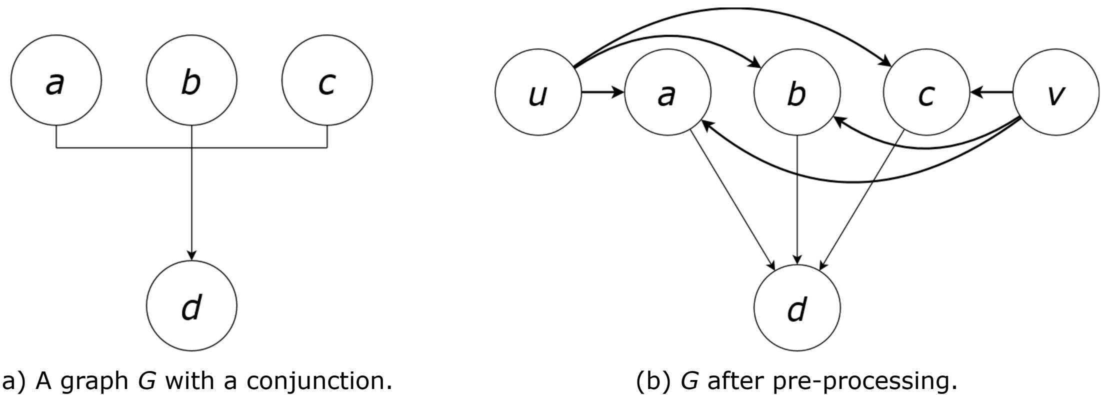
(b) *G* after pre-processing.

Figure 17: How warrants are handled before crossing minimisation.

Picking a violated constraint to handle is non-trivial. Haphazard choices may introduce cycles to the constraint graph. As stated earlier, a constrained crossing minimisation problem with a cyclic constraint graph has no solution. In Figure 16, an example is shown. Here, the nodes *a*, *b* and *c* are to be merged. If c_2 is found first, *a* is merged

with c, and a cycle is introduced to the constraint graph. This is shown in Figure 16b. On the other hand, if c_0 is found first, the result in Figure 16c attains, and progress remains unhindered. Because of these nuances, `sweepLayer()` uses a more or less unchanged TS implementation of Forster's `FIND-VIOLATED-CONSTRAINT` procedure [36, Algorithm 2]. It is a modified topological sorting algorithm (see, e.g., [37, p. 573] for a review of topological sorting), and it considers constraints in a lexicographic order based on the topological sorting numbers of head and tail nodes in ascending and descending order, respectively [36, pp. 210–211].

Obviously, the described algorithm adds a fair bit of complexity on top of the method described by Sugiyama et al [27]. The Barycenter Heuristic simply calculates a set of averages and sorts a layer. Consider two adjacent layers. Let V denote the nodes of the mutable layer, and allow E to be the edges between the layers. To calculate the barycenters, every node in V and the neighbours of each is considered once. Hence, it takes $O(|V| + |E|)$ time to find the barycenters. Afterwards, the nodes in V can be sorted in $O(|V| \log|V|)$ time [37, p. 207]. The total running time of the Barycenter Heuristic is therefore $O(|V| + |E| + |V| \log |V|)$ per sweep. Have C denote constraints. Forster shows that finding an appropriate constraint takes $O(|C|)$ time [36, p. 212]. He then goes on to prove that the running time of his modified barycenter algorithm is $O(|C|^2 + |E| + |V| \log |V|)$ [36, pp. 212–213]. This is notably worse than the bare Barycenter Heuristic, but the added functionality is clearly necessary for this project.



(a) A graph G with a conjunction.

(b) G after pre-processing.

Figure 18: How conjunctions are handled before crossing minimisation.

Returning to the pre-processing in line 2 of Code Block 8 will elucidate how warrants and conjunctions are translated into constraints. Three things happen in this line: Warrant structures are handled, conjunct nodes are handled and long edges are split into series of dummy vertices and edges. The idea underlying edge splitting was reviewed in Section 7.3. As a reminder, long edges are split into a series of tight edges to ensure that the layering $\mathcal{L}$ is proper. Warrants are handled by the introduction of two dummy nodes, an edge and two constraints. The concept is shown in Figure 17. The warrant source and edge are temporarily removed. Two dummy nodes are added. Let v_0 be the tail node of the targeted edge and v_1 be the head node. Denote the warrant source as v_2. Since the warrant source v_2 is layered by now, one dummy node v_3 can be placed at layer number $l(v_2, \mathcal{L}) - 0.5$ and the other one v_4 at $l(v_2, \mathcal{L}) + 0.5$. A dummy edge (v_3, v_4) is added. Then, the constraints (v_0, v_3) and (v_1, v_4) are added. This ensures that warrant structures remain clustered correctly. Note that any edges of v_2 will be rerouted so that all incoming edges (u, v_2) become inedges of v_3, and any outgoing edges (v_2, w) become outedges of v_4. Conjunctions are handled by introducing two dummy nodes to act as delimiters to the rest of the layer. Moreover, the conjoined edge is split into a series of simple edges going

from each sub-node to the target node. This is illustrated in Figure 18. Consider a set of conjunction sub-nodes P with a conjoined edge to a target node w. The conjoined edge is temporarily replaced by the set of edges $\{(p, w) \mid p \in P\}$. Two dummy nodes, u and v, are introduced as delimiters. All sub-nodes are constrained to keep to the right of u and to the left of v. This is sufficient handling for conjunct structures.

7.4.1 Crossing Counting

Another non-trivial procedure that is implicitly part of crossing minimisation is crossing counting. Before a permutation is accepted as the new order, the algorithm must assert that it results in fewer crossings than the old order. Thus, an efficient way of counting the number of edge crossings is needed. The naive approach would be to iterate over all edges and for each edge, count the number of crossing edges [22], [38]. Whether a pair of edges cross or not can be checked by comparing the relative ordering of their end nodes. The naive algorithm runs in $O(|E|^2)$ time. Since the number of crossings in a graph is $\Theta(|E|^2)$, this running time is optimal. I.e., there cannot be a more efficient algorithm if reporting the actual crossings is a requirement. However, only the count is of interest in this context. It is therefore possible to do better.

Based on the preliminary research [4], Barth et al.'s counting algorithm [38] seemed to be the best easily implemented fast method. Consider a pair of layers. Let E be the set of edges between them, and have V_{small} be the nodes of the smallest of the two layers. The impressive running time of Barth et al.'s algorithm can then be expressed as $O(|E| \log |V_{\text{small}}|)$. The author's TS implementation of this algorithm [38, Fig. 5] is contained in Code Block 10.

Because of the mathematical nature of the counting algorithm, it seems more useful to explain the underlying idea than to review the code step-by-step. Nevertheless, some code ought to be directly commented on. In Code Block 10, lines 6–15 were added to handle varying sizes of the input layers. The reasoning behind this stems from the fact that Barth et al. [38] assumed that the *north layer* would be bigger than the *south layer*. Such an assumption is a luxury the author could not afford. Furthermore, lines 26–31 get the edges between the layers in a seemingly over-complicated way. Doing it this way is in fact necessary to have the edges be appropriately ordered. Why the ordering is crucial will be explained shortly.

First, a couple of additional definitions are in order. An *inversion* is a pair of elements that is unordered [38]. Specifically, in a sequence π of pairwise comparable elements, a pair (a, b) where $a, b \in \pi$ is called an inversion if $a \prec b$ in π but $a > b$. The number of inversions in a given sequence is called the *inversion number*. Consider a graph G with two adjacent layers, L_{i-1} and L_i, each with an associated permutation, π_{i-1} and π_i, and the edges between them E. In the graph drawing being produced, the lower-indexed layer L_{i-1} will have its nodes spread on a horizontal line *above* the horizontal line containing the nodes in L_i. Therefore, when considering any pair of neighbouring layers, the lower-indexed one is called *the nothern layer*, and the higher-indexed one is named *the southern layer*. Write out $\pi_{i-1} = n_0, n_1, \ldots, n_{k-1}, n_k$ and $\pi_i = s_0, s_1, \ldots, s_{l-1}, s_l$. Let $\pi_E = e_0, e_1, \ldots, e_{m-1}, e_m$ be an ordering of E, and have it be lexicographically sorted such that $(n_o, s_p) \prec (n_q, s_r)$ only if either $o < q$ or $o = q \wedge p < r$. E.g., for Figure 15a, π_E would be $(a, e), (b, d), (b, f), (c, e), (c, f)$. Allow π_S to be a sequence of indices to nodes in the southern permutation π_i such that each entry is the index of the end node of each edge in π_E. Once again using Figure 15a to provide an example, π_S would be $1, 0, 2, 1, 2$. Notice

how every inversion in π_S corresponds to an edge crossing among the edges of E. Hence, the inversion number of π_S equals $C(G, \pi_{i-1}, \pi_i)$.

```
1   function countCrossings(
2     graph: Graph,
3     northLayer: NodeId[],
4     southLayer: NodeId[]
5   ) {
6     let layer0: NodeId[];
7     let layer1: NodeId[];
8
9     if (northLayer.length >= southLayer.length) {
10      layer0 = northLayer;
11      layer1 = southLayer;
12    } else {
13      layer0 = southLayer;
14      layer1 = northLayer;
15    }
16
17    let firstindex = 1;
18
19    while (firstindex < layer1.length) firstindex *= 2;
20
21    const treesize = 2 * firstindex - 1;
22    firstindex -= 1;
23    const tree = new Array(treesize).fill(0);
24    let crosscount = 0;
25
26    const edges = layer0.reduce<Edge[]>((accumulator, node0) => {
27      layer1.forEach((node1) => {
28        if (graph.hasEdge(node0, node1) || graph.hasEdge(node1, node0)) {
29          accumulator.push({ v: node0, w: node1 });
30        }
31      });
32
33      return accumulator;
34    }, []);
35
36    edges.forEach((edge) => {
37      const head = edge.w;
38      const headIndex = layer1.indexOf(head);
39      let index = headIndex + firstindex;
40      tree[index]++;
41
42      while (index > 0) {
43        if (index % 2) crosscount += tree[index + 1];
44        index = Math.floor((index - 1) / 2);
45        tree[index]++;
46      }
47    });
48
49    return crosscount;
50  }
```

Code Block 10: The crossing counting procedure.

Based on these observations, Barth et al. [38] propose leveraging insertion sort (a well-known algorithm, see [37, pp. 17–19]) as a possible method of calculating the crossing count. π_E can be lexicographically sorted according to π_{i-1} and π_i by radix sort (another widely known algorithm, see [37, pp. 211–214]). From there, π_S attains as explained above. While running an insertion sort function, one can track the number of positions each element moves forward, and the sum of these will be the crossing count. Without any modifications, such a procedure would yield a performance of $O(|E|^2)$, no better than the naive method. However, Barth et al. also suggest using Waddle and Malhotra's *accumulator tree* [39] to improve the running time to $O(|E| \log |V_{small}|)$.

Let c be a natural number such that $2^{c-1} < |L_i| \leq 2^c$, and have T be a balanced binary tree with 2^c leaves [38]. In Code Block 10, T is represented as an array of size $2^{c+1} - 1$. The tree root is in position 0. A node in position i has its parent in position $\lfloor (i-1)/2 \rfloor$. Thus, left children are at odd indices, and right children are at even indices. Initially, all entries are 0. Every southern vertice $s \in n_i$ corresponds to a leaf. The crossing count is initialised to 0. While traversing n_S, the algorithm stores the accumulated number of times the index of a southern vertice was encountered in its leaf node. Internal nodes contain the sum of the entries in their children. For each entry in n_S, the leaf of the indexed southern vertice has its value incremented. Then, all predecessors of the leaf also have their entry incremented by the algorithm working its way up the tree to the root. During this process, if a left child is visited, the value of its right sibling is added to the crossing count. After n_S is traversed, the crossing count will be correct. Using this method, a running time of $O(|E| \log |V_{\text{small}}|)$ is achieved.

7.5 Edge Straightening

After having layered the graph and permutated the layers, the next step is to assign explicit x-coordinates to each node. It appears that most coordinate assignment methods build on the assumption that straight edges are preferable [22, pp. 441–443]. This is probably not a bad supposition, and there is some evidence that we have a perceptual preference for straight edges [40]. Because it entails assigning coordinates so that edges are straightened, this step is known both as *the edge straightening step* and as *the x-coordinate assignment step* [22, p. 441]. Favouring straight edges during the coordinate assignment is likely to increase the width of the graph drawing. This is unfortunate since compact drawings are preferred (this follows from the criterion of short edges in Section 5.2). Nonetheless, the trade-off is viewed as agreeable.

It is particularly desirable for long edges to be straight [22, p 441]. Recall that any long edges have been partitioned into dummy vertices and edges. Hence, given a set of dummy vertices $\{v_0, v_1, \ldots, v_{n-1}, v_n\}$, a long edge (u, w) will have become the path $(u, v_0), (v_0, v_1), \ldots, (v_{n-1}, v_n), (v_n, w)$. It seems sensible that at least the sub-path $(v_0, v_1), \ldots, (v_{n-1}, v_n)$ can be drawn as a perfectly vertical line, and any necessary bends can appear at (u, v_0) or (v_n, w). In fact, most edge straightening algorithms prioritise this.

As with the other steps of the Sugiyama framework, coordinate assignment too has many different approaches [22, pp. 441–443]. But a particular algorithm has ended up being regarded as 'the algorithm of choice' for this case. Brandes and Köpf's algorithm for horizontal coordinate assignment [41] is preferred because of its linear running time of $O(|V| + |E|)$, good result quality and ease of implementation. The author saw little reason not to implement this acclaimed method.

The main loop of the author's coordinate assignment algorithm is shown in Code Block 11. As just alluded to, it is basically a TS implementation of Brandes and Köpf's main algorithm [41, Alg. 4]. The overall idea is to generate four *extreme layouts* that are skewed top-left, bottom-left, top-right and bottom-right. Then, combine the results to get a balanced final layout. Note that it is presupposed that edge crossings have been minimised before this algorithm is run. Lines 4, 18, 19, 27 and 28 are directly related to the algorithm as described by Brandes and Köpf. The functions `alignVertically()` and `compactHorizontally()` contain the meat of the functionality, but `markConflicts()` is also important. For the sake of clarity, the overall approach will be described first. Then,

the three mentioned functions will be explored in more detail.

```
straightenEdges(graph: Graph, graphMatrix: NodeId[][]) {
  restoreConjunctNodes(graph, graphMatrix);
  const conjunctNodes = mergeConjunctNodes(graph);
  markConflicts(graph, graphMatrix);

  const biasedGraphs: BiasedGraphTuple = [
    buildSimpleGraph(graph, REQUIRED_PROPERTIES),
    buildSimpleGraph(graph, REQUIRED_PROPERTIES),
    buildSimpleGraph(graph, REQUIRED_PROPERTIES),
    buildSimpleGraph(graph, REQUIRED_PROPERTIES),
  ];

  biasedGraphs.forEach((biasedGraph, graphIndex) => {
    const horizontalDirection = ITERATION_ORDERS[graphIndex].split(" ")[0] as
      | "right"
      | "left";

    alignVertically(biasedGraph, graphMatrix, ITERATION_ORDERS[graphIndex]);
    compactHorizontally(
      biasedGraph,
      graphMatrix,
      horizontalDirection,
      graph.graph().nodesep
    );
  });

  alignToMinWidthGraph(biasedGraphs);
  balanceAndAssignValues(graph, biasedGraphs, conjunctNodes);
}
```

Code Block 11: The coordinate assignment algorithm.

Remember from Section 5.2 that short edges are preferable. Sugiyama et al. [27] propose a mathematical programming solution that seeks to minimise total edge length among other things. Let $x(u)$ be the x-coordinate of some node u. Since y-values are already fixed (from the layering process), only x-coordinates matter. Sugiyama et al. use $\Sigma_{(u,v)\in E}\,(x(u) - x(v))^2$ to describe the total edge length. Gansner et al. [31], however, simplify the issue by using $|x(u) - x(v)|$ as a surrogate for edge length. In the same vein, Brandes and Köpf [41] base themselves on the $|x(u) - x(v)|$ expression. Clearly, any solution that minimises $|x(u) - x(v)|$ also minimises $(x(u) - x(v))^2$. Notice how for any set of real numbers $X = \{x_0, x_1, \ldots, x_{n-1}, x_n\}$, $\Sigma_{i=0}^{n} |x - x_i|$ is minimised when x is equal to the median of X. Therefore, Brandes and Köpf's algorithm focuses on aligning nodes with their *median neighbour*. For instance, if a node u has three neighbours — v_0, v_1 and v_2 — that are ordered $v_0 \prec v_1 \prec v_2$ in their layer, u should be aligned with v_1. A more interesting case arises when the number of neighbours is even. Say, if a node u has two neighbours, v_0 and v_1, u must be aligned with either v_0 or v_1. In Brandes and Köpf's algorithm, the viable neighbour first encountered is chosen. Another point is that either upper neighbours or lower neighbours must be considered first. Thus, the results of an alignment process will depend on the iteration order. Either the neighbours above or the ones below any given node must be considered first, and among the neighbours, either the left or the right neighbour must be prioritised. This is why there are four layouts with different biases, and these must be combined to produce the final layout. Luckily, combining the four layouts are comparatively simple. Consider a sequence of real numbers $x_0, x_1, \ldots, x_{n-1}, x_n$ in increasing order. The *average median* is defined as $(x_{\lfloor n/2 \rfloor} + x_{\lceil n/2 \rceil})/2$. Brandes and Köpf [41, p. 40] show that for every node, combining the four calculated x-coordinates using the average median is order- and separation-conserving, and leads to balanced layouts of high quality.

Refer to Code Block 11. First to address the pre-processing in lines 2–4. In line 2, the processing of conjunct nodes from the crossing minimisation step is undone. Delimiter nodes and temporary edges are deleted, and conjoined edges are readded. In addition, the width of all conjunctions is calculated. In line 3, the same trick employed during layering is used. Conjunctions are once again merged into a single node incident to all edges that were incident to the conjunction sub-nodes. This is done to make the alignment of conjunctions and their target easier. Line 4 does the pre-processing described by Brandes and Köpf [41]. It will be discussed afterwards. In lines 6–11, four copies of the input graph are made to track the four different layouts. `buildSimpleGraph()` is intended for fast copying of nodes and edges while ignoring associated data. In this case, there are a couple of properties the algorithm presupposes. Therefore, these properties are listed in the constant `REQUIRED_PROPERTIES` and passed as a second argument, instructing the function not to ignore them. In lines 13–25 each of the four constructed graphs is assigned x-coordinates. A list of four iteration directions is stored in `ITERATION_ORDERS`: right and down, right and up, left and down, and left and up. Traversing nodes in this fashion is what skews the layouts. Lines 14–16 simply get the horizontal direction of the current iteration order. This is because `compactHorizontally()` only takes the horizontal iteration order as an argument. On the other hand, `alignVertically()` takes both iteration directions. `alignVertically()` and `compactHorizontally()` will be detailed in a bit. Lines 27 and 28 implement the post-processing as described by Brandes and Köpf. Line 27 identifies which of the four layouts has the smallest width. It then aligns the left-skewed graphs such that their minimum x-coordinate coincides with the minimum x-coordinate of the smallest width graph. Likewise, the right-skewed graphs are adjusted to make their maximum x-coordinate coincide with the maximum x-coordinate of the smallest width graph. Then, line 28 combines the four layouts to attain the final one.

As a short preamble to the following explanations: Although Brandes and Köpf's [41] blocks of pseudocode are relatively compact, the author's implementations of their procedures are fairly lengthy. For brevity, they will not be included in this thesis. However, they will be duly described. Moreover, both the referenced code blocks of Brandes and Köpf and the full implementation in the project repository [6] should provide the interested reader with the specifics.

The pre-processing function `markConflicts()` is a modified version of Brandes and Köpf's pre-processing procedure [41, Alg. 1]. It is meant to flag edges that are *conflicted* so that they may be resolved in a predictable and productive way. Recall that long edges have been subdivided. An *inner segment* is an edge going between two dummy nodes. That is, it is part of a sub-path that ought to be straight in the final drawing. A non-inner segment (i.e., an ordinary edge between two nodes) crossing an inner segment is what Brandes and Köpf call a *type 1 conflict*. The original algorithm of Brandes and Köpf only flags type 1 conflicts. This is to ensure that they are resolved in favour of straight inner segments. The author has modified this pre-processing procedure to also flag conflicts related to warrants and conjunctions. For warrants, keeping the target edge straight is essential for aesthetic reasons. Consider an edge (u, v) targeted by a warrant. Marking any out-edges of u other than (u, v) as conflicted, guarantees that u will be aligned with v, which in turn ensures that $\Gamma(u, v)$ will be vertical. Remember that conjunctions have once again been merged into meta nodes. It is desirable to keep the conjoined edge straight. To achieve this, the trick just described is put to use again. Contemplate a conjunction ς in which its premise nodes P have been merged into a single meta node p, and let c denote its conclusion (target) node. Marking any out-edges of p other than

the conjoined edge (p, c) as conflicted ensures that the final line segment of $\Gamma(p, c)$ (the curve said to go from A to c in the main problem formulation of Section 7.1) is vertical.

`alignVertically()` is based on Brandes and Köpf's vertical alignment procedure [41, Alg. 2]. Prior to any iterations, each node is assigned a *root node* and a *next node* property, and both of these are set to be the node itself. This allows the algorithm to view each node as a linked list (a known data structure, see [37, pp. 258–259]) that initially contains one element. The idea is that the linked lists can represent vertical sequences of aligned nodes. Now all layers are iterated through once. Depending on the specified iteration direction, layers will be considered top-down or down-up, and the nodes within layers will be considered either left-right or right-left. For each node, its median neighbour is found. If the iteration direction is downwards, upper neighbours are considered. Elsewise, lower neighbours are considered. In the case of an even number of neighbours, two candidates for median neighbour arise. If the direction iterated is rightwards, the left candidate will be considered first, while the right candidate is considered first if the iteration direction is leftwards. An identified median neighbour v_0 will be aligned with the current node v_1 only if the edge (v_0, v_1) is non-conflicted and the alignment is possible. That is, v_0 cannot already have been aligned with some other node. Nor can a node v_2 in the neighbour layer (the layer of v_0) have been previously aligned with a node v_3 in the current layer (the layer of v_1) such that $v_0 \prec v_2$ and $v_1 \succ v_3$. If a node does not have neighbours or it cannot be aligned with its median neighbour(s), the node remains unaligned. The final result of `alignVertically()` is a biased alignment of the nodes. In the example illustrated in Figure 19, the produced links between nodes are shown as thin edges.

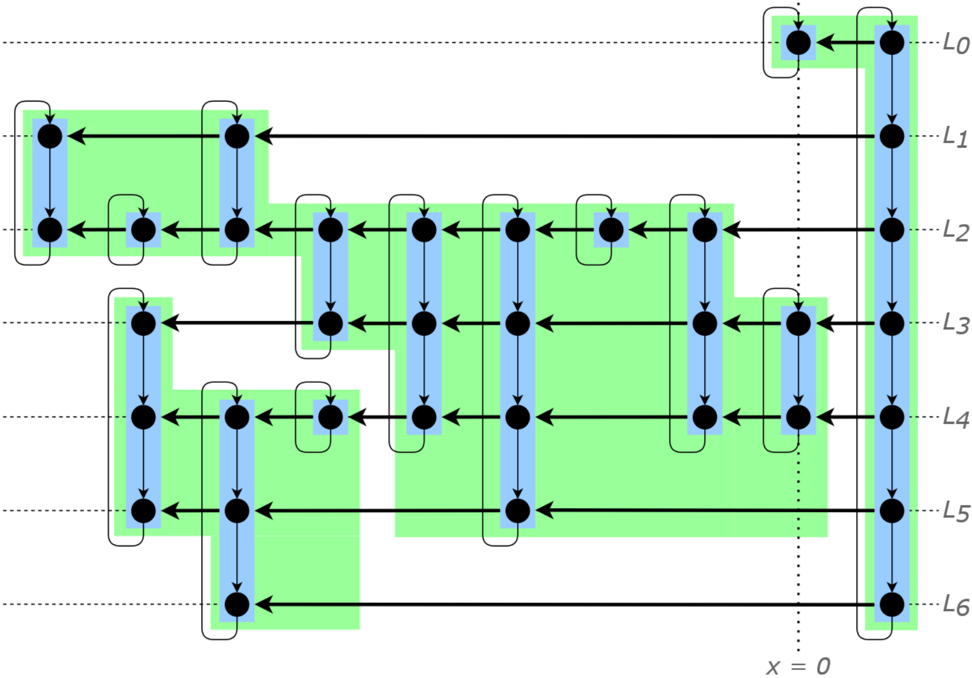


Figure 19: An illustration of how Brandes and Köpf's algorithm works.

Source: Based on [42, Fig. 1] .

The horizontal compaction procedure of Brandes and Köpf [41, Alg. 3] is implemented in the author's `compactHorizontally()`. Based on the alignment generated by `alignVertically()`, it assigns x-coordinates to the nodes. It is imperative to understand a few key points before moving on. View Figure 19. The blue rectangles are *blocks*,

which contain all nodes within a linked list (and, hence, a set of aligned nodes). A *block graph* is created by introducing directed edges from every node to the node preceding it within its layer (if any). These newly introduced edges are represented by the bold edges of Figure 19. With basis in the constructed block graph, the blocks can be subdivided into *classes*, which are displayed in green. A class is defined by the topmost reachable root of the block graph. Coordinates can now be easily determined within the classes. Amusingly, a layering method is applied here as a sub-algorithm: The Longest Path Algorithm [22, pp. 420–421]. This is not as computationally expensive as one might intuitively suspect. The Longest Path Algorithm is a simple process that has linear running time [43]. It works by recursively determining the coordinates of each block *relative to the defining sink of its class*. It starts at the defining sink, then works its way rightwards. Let β be the block with the currently highest relative coordinate within a class. Have $x(\beta)$ denote its relative x-coordinate. The next block is assigned a relative x-coordinate of $x(\beta) + \Delta_x$, where Δ_x is the minimum seperation (i.e., the value of `nodesep`). This process should feel familiar since it is, in fact, the same principle applied to produce an initial ranking in Section 7.3.

With these explanations out of the way, `compactHorizontally()` can be reviewed. It performs three sequential iterations over all nodes. In the first one, it assigns each node a `classSink` property that is initialised to be the node itself and an `xShift` property that is initialised to an infinite value. In the second iteration, it looks for nodes that are sinks defining a class in the block graph and runs the longest path algorithm. I.e., for each defining sink, the class it defines have its blocks assigned relative coordinates. The relative coordinate of a block is stored in each of its nodes — specifically, in the `xShift` property. In the third and final iteration, all nodes are assigned absolute x-coordinates. Each class is placed with minimum separation from previously placed classes, and the final coordinate of each node then attains since all coordinates relative to the class are known. Once again, notice how horizontal *direction* or preference will skew the resulting layout. Because of this, `compactHorizontally()` takes the preference as its third argument, `iterationOrder`. If `iterationOrder` is `"right"`, `xShift` is set to ∞ and the algorithm runs as explained by using maximum values. This gives results like the example in Figure 19. On the other hand, if `iterationOrder` is `"left"`, `xShift` is set to $-\infty$ and the algorithm runs by using minimum values, giving an opposite bias.

Returning to Code Block 11, line 28 deserves a final note. Recall how conjunction nodes have been merged. Thus, when x-coordinates were assigned, only the meta nodes were assigned coordinates. Line 28 therefore iterates through the sub-nodes and give them explicit absolute x-coordinates too. All vertices have now been assigned x- and y-coordinates. Only the warrants require some minor tweaking because of the processing done during crossing minimisation. But this has already been explained back in Section 7.1.

8 Discussion

The previous sections have reviewed the project and the resulting product. In this section, various aspects pertaining to the project will be discussed.

8.1 Evolution of Methods

The methods of the author developed over the course of the project. Section 3 mentions that TDD was employed during development. This practice was only followed to a certain point. Many if not all of the algorithms implemented in this project were of a highly mathematical nature. Of course, this is nothing unusual. All algorithms are mathematical in nature. But the literature covering the algorithms generally focused more on theoretical correctness and theoretical performance. Because the author invested much time into understanding the concepts and then meticulously implementing them in an actual programming language (i.e., not pseudo-code), he often got ahead of himself and built an algorithm before adequate testing had been prepared. That is not to say that testing was not extensively used, but rather that the development method oscillated between actual TDD and plain post hoc testing. Moreover, as the project matured, unit tests were deprioritised in favour of less formal, manual testing by the author. When actual layouts could be produced, it seemed in many cases as simply throwing various inputs at the algorithm and correcting any observed errors was an effective strategy. It is doubtful that either of the said deviations from the planned TDD approach adversely affected the author's efficiency or the quality of the code. The deviations were not made thoughtlessly, and one could argue that changing methods to suit current needs is quite agile [5, pp. 58–62].

Another note on the project methods is that the weekly check-ins with Disputas became shorter and less frequent towards the end of the project. This is viewed by the author as a natural progression. As the final state of a project becomes easier to predict, it is simply less to talk about. What the author ought to do with his remaining time was already evident, and any further improvements that were desired would have to be done by the company itself after the end of the master's project. Additionally, since most of this thesis was written in the final stage of the project period, less development occurred and, hence, there were even fewer relevant matters to discuss.

Excluding the two above paragraphs, the author kept to the methods described in Section 3 throughout the project. Granted, there were not many particular methods to adhere to or deviate from, but it is still worth stating that the few described development principles were indeed followed.

8.2 Requirement Satisfaction

See the requirements in Table 1, and for more details, the requirements section of the preliminary report [4, pp. 2–7]. The Argumappr library [6] produced in this project clearly meets most of the stated requirements, but it does not perfectly satisfy all of them. As far as the tests performed by the author show, Argumappr does produce layered DAG layouts with optimal positionings per the definitions used in this thesis. The goal of minimising edge crossings was achieved by using the modified Barycenter Heuristic. Conjoined edge support, or conjunction support, was implemented without any size restrictions. Edge-on-edge support, or warrant support, was also implemented.

However, at the time of writing, Argumappr does not support more than one warrant per edge. Furthermore, Disputas's wish of constraining layout changes was not fulfilled. These shortcomings will be discussed in Section 8.5.

The requirement of adequate speed is thought to have been met. Large graph support, which is tied to the speed requirement, is also believed to have been achieved. The expressed uncertainty stems from the ambiguous requirement definition, limited testing and the nature of the implementation language. *Adequate speed* was defined as an absolute running time low enough not to impact user experience. Ignoring the obvious subjective aspect (in the preparatory project, a *low enough running time* was estimated to be below 2.5 seconds [4, p. 23]), judging performance in absolute terms is problematic. This is because hardware plays too big a role. Although the author had no performance issues while performing tests on the hardware at his disposal, Ponder users with lower-end PCs might. Moreover, many of the built-in JS functions used in the project may have different implementations based on the platform that compiles the code. This greatly diminished the author's control over code efficiency, but, more importantly, it implies that different users with comparably powerful hardware may experience different running times (both asymptotically and absolutely). These are merely speculated issues; it is not known whether performance will be problematic. Regardless, further and more rigorous testing is warranted. Testing will be further discussed in Section 8.3, and theoretical performance will be reviewed in Section 8.4.

8.3 Testing Limitations

Primarily, two types of testing methods were used in the project: automated unit tests and manual white-box testing. Most of the applied unit tests may be found in the project repository [6]. It is perhaps obvious, but most of the available tests have gone through a series of refinement stages. Some have also been completely rewritten several times over. Moreover, some written tests were used in a white-box manner to probe the algorithm during development. Though useful, these tests were informal and intended to assist the author with specific issues before being summarily discarded. It is the author's opinion that such methods are both common and helpful. Nonetheless, further documenting such an amorphous process would likely serve little purpose. It is therefore only mentioned here in passing. After the algorithm was developed to the point of producing layouts, manual testing gained ground on automated testing. The manual tests were performed in an ad hoc fashion without much planning. One of Disputas's proprietary test repositories were put to use to see how the algorithm performed. It was a flexible way of debugging and visually presenting problems.

The author attempted to write unit tests with wide coverage. I.e., the tests were written to provide many test cases and use diverse inputs that were realistic. Nevertheless, it is of course infeasible to test every case and all inputs. This implies the possibility of certain inputs breaking the algorithm. Checking various expected inputs lessens the odds of bugs affecting most users, but there are always edge cases, and these are notoriously easy to miss. It is therefore hard to say how robust the software is with certainty. Manual white-box testing has the advantage of letting the developer *play around*. A plethora of inputs can be thrown at the program on a whim, and the behaviour of the algorithm is clearly visible by the quality and speed of outputs. Additionally, if the developer feels uncertain about the robustness or correctness of some particular piece of code, they can try to produce inputs targeting the associated functionality. Even with a lot of testing,

both automated and manual, there are still no guarantees. It is often observed that users, despite having no knowledge of the inner workings of the software, simply stumble upon errors during regular use. At least some minor bugs are likely to be in the code at the end of the project period. Performing several user tests over time may uncover any mistakes missed by the author.

8.4 Time Complexity

No formal running time tests were made for this project. This is partly because of the aforementioned vagueness of the performance requirements, and partly because there were no performance issues discovered during manual testing. Knowing full well that Disputas intended to use the algorithm to dynamically generate layouts user-side, the author kept performance in mind during development. Sub-algorithms were picked to balance speed and accuracy, and their implementations should be efficient. That is, the TS code was written to make limited use of costly built-ins, avoid unnecessary computations, etc. But recall that the performance of JS (which TS compiles into) is never guaranteed. To provide some semblance of an objective performance measure, this section will perform a theoretical running time analysis. Firstly, Table 4 summarises the time complexities of the algorithms employed in the different steps of the Sugiyama method. However, it lists the *default* algorithms, and the author customised these to handle warrants and conjunctions. They must therefore be evaluated again.

Table 4: A summary of the time complexities of the steps.

Step	Performance
Cycle removal	$O(\lvert E\rvert)$ in total
Node layering	Unproven polynomial but fast in practice
Crossing minimisation	$O(\lvert C\rvert^2 + \lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$ per *sweep*
(Crossing counting)	($O(\lvert E\rvert \log \lvert V_{\text{small}}\rvert)$ per layer pair)
Edge straightening	$O(\lvert V\rvert + \lvert E\rvert)$ in total

Before analysing the steps as they were implemented, an important fact must be kept in mind. Consider an arbitrary DAG G. See how the number of vertices and edges are connected. Clearly, there cannot be more edges than what is allowed in a DAG. Take one node at a time. The first one can be linked to all other nodes except itself; the next one can be linked to all nodes except itself and the first one; the next must exclude itself, the second and the first; and so on. Hence, the maximum number of edges is $\Sigma_{i=0}^{|V|}(|V| - i) = (|V|^2 - |V|)/2$, so $0 \leq |E| \leq (|V|^2 - |V|)/2$. From this, many of the below statements follow.

Looking at the cycle removal step, there were no real changes made to the original algorithm. Both copying the graph and handling edges (iterating over them, deleting loops and reversing *against-the-flow* edges) can be done in linear time. However, the use of lists to impose a node ordering means that for each edge (u, v), the indices of u and v must be found, which is generally done as a linear search. Thus, two $O(|V|)$ operations must be done for every edge. The resulting product asymptotically dominates the linear terms, giving the cycle removal step a total running time of $O(|V| \cdot |E|)$.

The node layering algorithm has an unproven but presumed polynomial running time. Consider a polynomial expression of arbitrary degree that expresses the running time of

the network simplex algorithm. Let n be a dominating term of degree $k > 0$ that depends on V and E. Instead of $n(V, E)$, the term is simply denoted n. The running time can then be written as $O(n^k)$. As far as the author knows, there are no implementation details that ought to impact running time. The pre- and post-processing is done in linear time. Have a and b be positive real numbers. Since $a|V|+b|E| = O(n^k)$, the processing obviously does not impact the time complexity.

Recall that during crossing minimisation, the layers are iterated from top to bottom and bottom to top. When a single pair of layers is considered, the Constrained Barycenter Heuristic *sweeps* the mutable layer before the new number of crossings is found. This means that each sweep takes $O(|C|^2 + |E| + |V| \log |V| + |E| \log |V_{\text{small}}|)$ time. Remember that here, C is the constraints in the swept layer, E is the edges between the fixed and mutable layer, V is the vertices in the mutable layer and V_{small} is the vertices of the smallest of the two layers. The per sweep bound is precise, but it is difficult to gauge the total running time based on it. Therefore, a few relaxations will be made. From now, denote the full node set V. Assume that after layering, the nodes are uniformly distributed such that if there are l layers, there are precisely m nodes in each. There is a maximum of $\Sigma_{i=1}^{m-1} i = (m^2 - m)/2$ constraints per layer. This results in a simpler but admittedly less tight bound per layer of $O(m^4 + m^2 + m^2 \log m) = O(m^4)$. By the assumptions made, there will be made l sweeps with this running time. Hence, a single full pass takes $O(l m^4)$ time. See how there are three natural *extreme* distributions: $l = 2 \wedge m = |V|/2$, $l = m$ and $l = |V| \wedge m = 1$. The worst out of these three is clearly the first since it gives a running time of $O(|V|^4)$ for a single full pass. The second and third imply $O(|V|^{5/2})$ and $O(|V|)$, respectively. Because m^4 dominates l (and because of the assumed uniformity), one would expect any increase in the number of layers to decrease the overall running time. Thus, a worst-case running time is $O(|V|^4)$. This expression is still just a measure of a single pass. There will be made several passes until a plateau is reached. However, by assuming the default maximum number of allowed iterations (the default of `maxcrossingloops`), a constant upper bound attains. Therefore, $O(|V|^4)$ is the total asymptotic running time. Of course, these results presuppose an even distribution of nodes. It is possible that some other combination of uneven layers could imply worse running times. At least intuitively, one would expect the results to be a fair representation of the algorithm's run time.

Straightening edges can be done in linear time. The *core procedure* must run four times to generate the four extreme layouts. Additionally, the pre-processing must visit all nodes once. However, this does not affect the time complexity. Any positive linear combination of $|V|$ and $|E|$ is asymptotically comparable to $|V| + |E|$. I.e., for all $a, b \in \mathbb{R}$ greater than 0, $a|V| + b|E| = \Theta(|V| + |E|)$.

In the main loop, the graph is copied twice in linear time. Restoring reversed or deleted edges, finalising warrant positions, drawing Bézier curves and removing dummy nodes all take linear time too. As was just mentioned, positive linear combinations of the number of nodes and edges are asymptotically comparable. Since the edge straightening step takes linear time, there is no need to consider the linear time functions of the main loop. The total running time of the main loop can thus be expressed $O(|V| \cdot |E|) + O(n^k) + O(|V|^4) + O(|V| + |E|) = O(|V|^4 + n^k)$.

8.5 Further Work

It is the hope of the author that the developed data structure and algorithm in the form of the Argumappr package [6] will continue to be maintained and built upon. There are several improvements that could easily be made to Argumappr given enough time. Even if this specific library does not see further development, the methods described in this paper can be adopted and refined by others. There might be flaws in the described procedures or they might be sub-optimal in different ways. Correcting things the author may have overlooked and generally improving the graph layout algorithm would provide a better way to draw argument maps.

As previously mentioned, Argumappr does not support more than one warrant per edge at the time of this writing. Additionally, warrant sources will always be placed to the right of the target edge. The limited number of warrants is merely caused by the time constraints of the master project, but the placement of warrant sources is because of Disputas's convention of putting pro warrants to the right and con warrants to the left. However, this implies that Argumappr does not support con warrants either. Having an option for changing side placements and supporting placement on both sides would be beneficial. It should be trivial to extend the functionality, allowing for arbitrary numbers of warrants per edge and adjustable side placement. The handling applied by the author is not limited by these parameters, so customising it to add the said functionality should not be problematic.

Since Ponder performs on-the-fly calculations of layouts for interactive graphs, one of the requirements was to limit the amount of change between each iteration of the graph. This would help users keep track of the elements and lessen any confusion caused by the layout shuffling around. The problem of constraining layout changes is a non-trivial one. When ranking the importance of the different requirements put forward by Disputas, the stability requirement was deemed the least important one. Unfortunately, there was not enough time to explore it. Thus, it remains unknown to the author how difficult implementing such a constraint would be. North [44] presents a promising idea for producing incremental DAGs layouts. Granted, North's work is highly relevant to the stability requirement, but it is not clear how hard it would be to merge his solution with the author's. In addition, it is unclear what impact this would have on the total running time. Future projects might shed light on this.

Though the asymptotic time complexity gives a good pointer to what performance to expect, it is best not to forget that overhead and linear-time operations do in fact matter. There are many feasible optimisations that the author is aware of and, possibly, several more that author might be unaware of. Two points of concern struck the author as the most obvious. The first one was mentioned in passing during the analysis of Section 8.4. Lists were used to impose the linear ordering of nodes during cycle removal. This worsened the running time from its theoretical $O(|E|)$ to the actual $O(|V| \cdot |E|)$. This could easily be rectified by assigning each node an index (using their label) instead of using lists. This would turn the repeated linear searches into a series of constant time operations, and the $O(|E|)$ running time would attain. The second point is a bit broader. Recall that during development, the author emphasised modularity. He also first implemented a fully functional layout algorithm for normal graphs, before extending this algorithm to support argument maps. This resulted in somewhat of a *pipe-and-filter pattern* [7, pp. 215–217] where the input and output of each step were statically defined. One may argue that this is positive for the software architecture, but it is

certainly not positive for the performance. Warrants and conjunctions are often processed and restored to their original states, before being processed in a similar fashion yet again. By relaxing the standards of how inputs and outputs should be and holistically viewing the algorithm, one could eliminate several redundant processing steps. How much of an impact this would have is not known, but it would however not affect the time complexity.

Perhaps this project might also inspire entirely new methods of laying out argument diagrams. Though the author focused on utilising the Sugiyama framework, there might be other suitable hierarchical drawing procedures out there. In the preliminary report [4, pp. 22–23], numerous other graph drawing methods are mentioned. Maybe *force-directed algorithms* or *genetic algorithms* could be applied to laying out argument diagrams. Other views of hierarchies (i.e., not horizontal layers) might be interesting to research — e.g., *radial level drawings*, in which the layers are arranged as concentric circles [22, p. 444].

It has been mentioned before that the Sugiyama method is the go-to method for drawing hierarchical graphs. Since this is a popular and highly researched framework, there are likely many new details, expansions and refinements available. Some of these could be useful for this use case but might have escaped the author's attention. For instance, a recent discovery by the author was the erratum of Brandes et al. [42]. There, they point out two flaws in the original algorithm made by Brandes and Köpf [41] and suggest how to fix them. One flaw is *double shifting* since '[…] offsets are actually added twice for vertices whose root has already been shifted to its final x-coordinate' [42, p. 4]. The other flaw is *shift accumulation*, which is serious but might not lead to many visible problems in practice, and it is explained by Brandes et al. [42, p. 5]:

> During compaction, shift values are determined for preceding classes relative to the current class. The implicit assumption in line (A) is, however, that the current class is not shifted itself. As a consequence, shift values are not accumulated along critical paths in the DAG of classes. It is not sufficient to consider the shift value of sink[v] in line (A) because it may not be in its final state, yet.

When Brandes et al. use 'line (A)' in the quote, they refer to the line of the horizontal compaction procedure that assigns the shift of a node to be equal to the minimum of its own value and the last positioned class plus minimum separation. Sadly, this erratum was discovered after the project had been finalised. Thus, implementing Brandes et al.'s fixes is another target for future work. But this goes to show that there might be several relevant papers that have yet to be identified. Moreover, the recency of Brandes et al.'s article (2020, which is only three years prior to this thesis) points out the need to keep track of newly published graph drawing literature.

9 Conclusion

Disputas, a Norwegian tech startup working with logic graphing, was introduced. Their problems and the subsequent project suggestion became the grounds for the author's master's project, which was documented in this thesis. The project was to initially research graph drawing literature, and then produce a graph layout algorithm specifically

tailored to argument maps. In collaboration, Disputas and the author formulated a set of requirements for the aforementioned algorithm. The project was divided into two parts: a preparatory phase, resulting in a literature review serving as the theoretical grounds for the project, and a working phase, in which the actual development of the customised algorithm took place. Few formal methods were employed by the author. There were weekly meetings with the *customer* (Disputas), TDD was utilised, various common development *best practices* were used and some extra focus was put on modularity. The technologies applied in this project were listed and their use was justified. Git and GitHub were used for VCS. TS was the programming language, and development took place in a Node.js environment to produce an npm package. Mocha and Chair formed the testing solution for the project. During development, ESLint and Prettier helped the author via formatting assistance. TypeDoc was used to generate documentation so the code was *self-documenting*. Various definitions from graph theory, graph drawing and argumentation were reviewed.

The project resulted in Argumappr, a library for automatic graph layout generation. It contains the data structure explored in Section 6 and the algorithm presented in Section 7. The data structure serves as a representation of argument diagrams. It is built upon the extant Graphlib JS library, which provides a data structure for discrete graphs and an assortment of relevant algorithms. The custom graph layout algorithm was the main result of this thesis. Based on the literature uncovered in the preliminary report, a framework consisting of several discrete steps and sub-algorithms for each of these steps were chosen. The algorithm leverages the Sugiyama framework, which partitions the problem of creating a layered layout into four steps: cycle removal, node layering, crossing minimisation and edge straightening. During cycle removal, a version of Eades et al.'s greedy cycle removal is used. It eliminates cycles by reversing as few edges as feasibly possible. The node layering is based on Gansner et al.'s network simplex algorithm, and it layers nodes optimally, meaning it produces a layering that minimises the number of long edges. Crossing minimisation is achieved by using a modified version of the barycenter heuristic, originally proposed by Sugiyama et al. The author's implementation has its basis in Forster's constrained barycenter heuristic, allowing the necessary limitations of node positions to be applied. As long as it does not violate any constraints, the constrained barycenter heuristic places nodes at the average position of their neighbours. It iteratively improves the number of crossings by repeatedly sweeping all layers from top to bottom and visa versa. The barycenter heuristic must count the number of crossings between pairs of layers. A hitherto optimal counting algorithm by Barth et al. was used for this purpose. If a permutation of a layer does not reduce the number of crossings, the permutation is discarded. To straighten edges, the author based himself on Brandes and Köpf's renowned algorithm. It tries to minimise edge length by aligning nodes with their median neighbours. Then, it creates a block graph and applies a longest path layering algorithm to assign explicit x-coordinates. This results in a layout that is skewed depending on the iteration directions. This is resolved by calculating four extreme layouts, and assigning each node the median average of its four coordinate candidates.

The author's methods changed somewhat during the project. TDD was not always used, and different testing methods were used at different times. Towards the end of the project, the weekly meetings became less frequent. Generally, the conventions described in Section 3 were followed. Argumappr sufficiently fulfilled most requirements but not all. However, only one warrant per edge is currently supported, and there are no constraints on how much the layout may change. The algorithm is believed to have

adequate speed, but it is ambiguous. Only limited testing has taken place, and there are likely undiscovered issues with the produced code. More testing is warranted. The asymptotic running time of the algorithm is $O(|V|^4 + n^k)$. Future experiments have a lot to look at. Both the Argumappr package and the concepts it was built upon can likely be refined and improved. It is probably simple to add more complex warrant functionality to Argumappr, and there are several optimisations that may be done. Though the author did not have time to look at layout stability, North's incremental layout generation seems like a good place to start. Hopefully, others will find argument map drawing interesting and produce more work related to it. Maybe entirely new vectors of approach may lead to superior performance. Existing and new papers on graph drawing will probably contain information that is useful for drawing argument maps. As of this writing, the author notes that Brandes et al.'s erratum points out flaws in the edge straightening method that may be fixed by implementing their suggestions.

References

[1] U. Schreiber, D. Corfield and T. Bartels, 'Bayesian reasoning', *nLab*, 7th Mar. 2023. [Online]. Available: https://ncatlab.org/nlab/show/Bayesian+reasoning (visited on 9th May 2023).

[2] P. F. S. Kvarberg, 'Technologies to bolster public debate', *Teknovatøren*, vol. 20, 4th Aug. 2021. [Online]. Available: https://www.teknovatoren.no/2021/08/technologies-to-bolster-public-debate/?fbclid=IwAR1R3i4RZpYOROxkw8zx97QfHATX1xAEi73E5Bgh4MniiezQMYx21wZze-Y (visited on 13th Apr. 2023).

[3] Disputas AS, *Ponder*, 20th Mar. 2023. [Online]. Available: https://ponder.disputas.no/ (visited on 9th May 2023).

[4] D. F. Bendiksen, 'Preparatory Master's Project: Laying Out Logic Graphs', *Norwegian University of Science and Technology (NTNU)*, 14th Dec. 2022.

[5] I. Sommerville, *Software Engineering*, eng, 10th ed., global ed. Boston Mass.: Pearson, 2016, ISBN: 978-1-292-09613-1.

[6] D. F. Bendiksen, *Argumappr*, version 1, 11th Jun. 2023. [Online]. Available: https://github.com/davider90/argumappr (visited on 11th Jun. 2023).

[7] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, 3rd. Addison-Wesley Professional, 2012.

[8] S.-y. Guo, M. Ficarra and K. Gibbons, '16.2 Modules', *ECMAScript® 2024 Language Specification*, 11th May 2023. [Online]. Available: https://tc39.es/ecma262/#sec-modules (visited on 11th May 2023).

[9] OpenJS Foundation, 'Modules: CommonJS modules', 5th Mar. 2023. [Online]. Available: https://nodejs.org/api/modules.html#modules-commonjs-modules (visited on 11th May 2023).

[10] Mozilla Corporation, 'JavaScript modules', 5th Oct. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 11th May 2023).

[11] S. Chacon and B. Straub, *Pro Git*, version 2, 12th Apr. 2023. [Online]. Available: https://github.com/progit/progit2 (visited on 13th Apr. 2023).

[12] Mozilla Corporation, 'JavaScript', 5th Apr. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 13th Apr. 2023).

[13] Microsoft Corporation, 'TypeScript', 12th Apr. 2023. [Online]. Available: https://www.typescriptlang.org/ (visited on 13th Apr. 2023).

[14] OpenJS Foundation, 'About Node.js®', 13th Apr. 2023. [Online]. Available: https://nodejs.org/en/about (visited on 13th Apr. 2023).

[15] npm, Inc., 'About npm', 13th Apr. 2023. [Online]. Available: https://www.npmjs.com/about (visited on 13th Apr. 2023).

[16] OpenJS Foundation, 'Mocha', 13th Apr. 2023. [Online]. Available: https://mochajs.org/ (visited on 13th Apr. 2023).

[17] Open source, 'Chai', 18th Jan. 2023. [Online]. Available: https://www.chaijs.com/ (visited on 13th Apr. 2023).

[18] OpenJS Foundation, 'Core Concepts', 27th Apr. 2023. [Online]. Available: https://eslint.org/docs/latest/use/core-concepts (visited on 27th Apr. 2023).

[19] Open source, 'What is Prettier?', 27th Apr. 2023. [Online]. Available: https://prettier.io/docs/en/index.html (visited on 27th Apr. 2023).

[20] Open source, ‘What is TypeDoc?’, 27th Apr. 2023. [Online]. Available: https://typedoc.org/guides/overview/ (visited on 27th Apr. 2023).

[21] K. H. Rosen, *Discrete Mathematics and Its Applications*, eng, 7th ed., global ed. New York: McGraw-Hill, 2013, ISBN: 0071315012.

[22] R. Tamassia *et al.*, *Handbook of Graph Drawing and Visualization*. CRC press, 2013.

[23] T. Stephen, *Return to Reason*. Harvard University Press, 2001, ISBN: 9780674012356.

[24] S. E. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003, vol. Updated ed. ISBN: 9780521827485.

[25] C. Pettitt, *Graphlib*, version 2.1.8, 17th Apr. 2023. [Online]. Available: https://github.com/dagrejs/graphlib (visited on 14th May 2023).

[26] C. Pettitt, *Dagre*, version 0.8.5, 10th May 2023. [Online]. Available: https://github.com/dagrejs/dagre (visited on 15th May 2023).

[27] K. Sugiyama, S. Tagawa and M. Toda, ‘Methods for visual understanding of hierarchical system structures’, *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[28] J. Vince, *Mathematics for Computer Graphics*. London: Springer, 2022, ISBN: 978-1-4471-7520-9. DOI: 10.1007/978-1-4471-7520-9_1.

[29] R. Karp, ‘Reducibility among combinatorial problems’, vol. 40, Jan. 1972, pp. 85–103, ISBN: 978-3-540-68274-5. DOI: 10.1007/978-3-540-68279-0_8.

[30] P. Eades, X. Lin and W. F. Smyth, ‘A fast and effective heuristic for the feedback arc set problem’, *Information Processing Letters*, vol. 47, no. 6, pp. 319–323, 1993.

[31] E. R. Gansner, E. Koutsofios, S. C. North and K.-P. Vo, ‘A technique for drawing directed graphs’, *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.

[32] L. A. Wolsey and G. L. Nemhauser, *Integer and Combinatorial Optimization*. John Wiley & Sons Ltd., 1988, ISBN: 9781118627372.

[33] M. R. Garey and D. S. Johnson, ‘Crossing number is np-complete’, *SIAM Journal on Algebraic Discrete Methods*, vol. 4, no. 3, pp. 312–316, 1983.

[34] P. Eades and N. C. Wormald, ‘Edge crossings in drawings of bipartite graphs’, *Algorithmica*, vol. 11, pp. 379–403, 1994.

[35] M. Jünger and P. Mutzel, ‘2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms’, eng, *Journal of Graph Algorithms and Applications*, vol. 1, pp. 1–25, 1997.

[36] M. Forster, ‘A Fast and Simple Heuristic for Constrained Two-Level Crossing Reduction’, in *Graph Drawing*, J. Pach, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 206–216, ISBN: 978-3-540-31843-9.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms*, 4th ed. MIT press, 2022.

[38] W. Barth, M. Jünger and P. Mutzel, ‘Simple and efficient bilayer cross counting’, in *International Symposium on Graph Drawing*, Springer, 2002, pp. 130–141.

[39] V. E. Waddle and A. Malhotra, ‘An E log E Line Crossing Algorithm for Levelled Graphs’, in *International Symposium Graph Drawing and Network Visualization*, Springer, 1999, pp. 59–71.

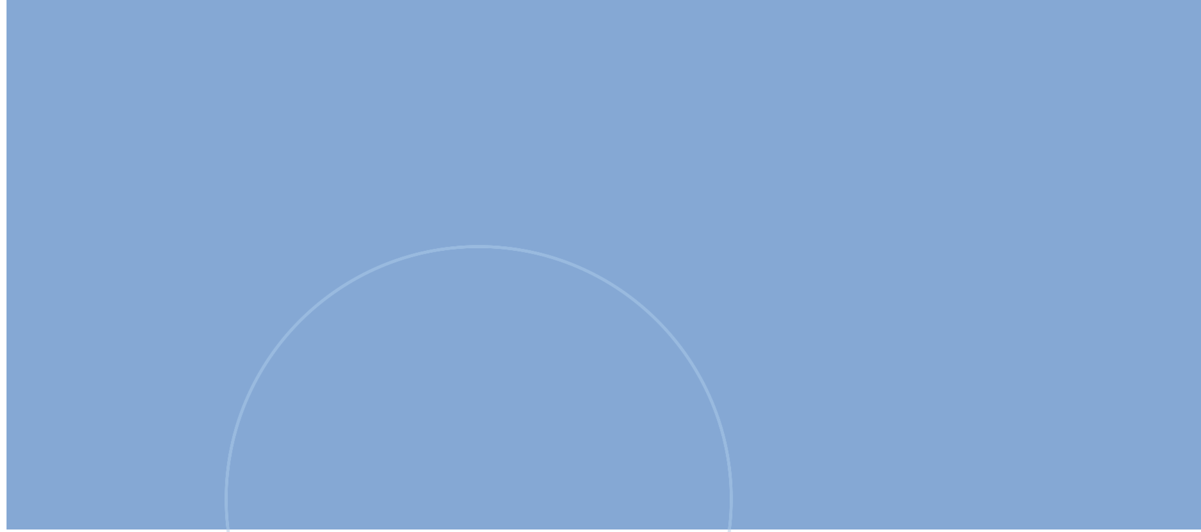
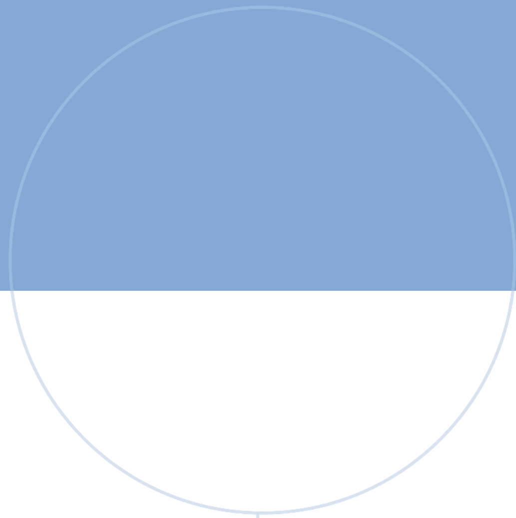
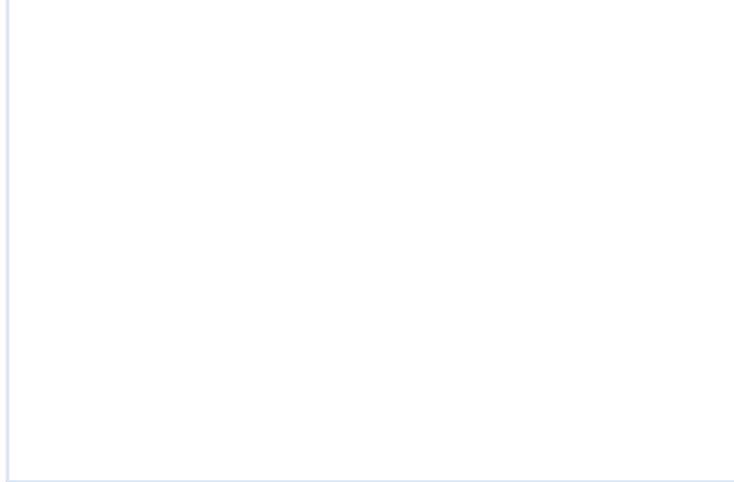
[40] W. Huang, P. Eades and S.-H. Hong, ‘A graph reading behavior: Geodesic-path tendency’, in *2009 IEEE Pacific Visualization Symposium*, IEEE, 2009, pp. 137–144.

[41] U. Brandes and B. Köpf, ‘Fast and simple horizontal coordinate assignment’, in *International Symposium on Graph Drawing*, Springer, 2001, pp. 31–44.

[42] U. Brandes, J. Walter and J. Zink, *Erratum: Fast and simple horizontal coordinate assignment*, 2020. arXiv: 2008.01252 [cs.DS].

[43] K. Mehlhorn, *Data structures and algorithms 2: graph algorithms and NP-completeness*. Springer Science & Business Media, 2012, vol. 2.

[44] S. C. North, ‘Incremental layout in dynadag’, in *Graph Drawing*, F. J. Brandenburg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 409–418, ISBN: 978-3-540-49351-8.

NTNU

Norwegian University of
Science and Technology