Master's thesis

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

Trym Grande

Force-Directed Graph Drawing for Representation of Reasoning

Master's thesis in Informatics
Supervisor: Srinivasa Rao Satti
Co-supervisor: Paal Fredrik Skjørten Kvarberg
June 2023

NTNU
Norwegian University of
Science and Technology

Trym Grande

Force-Directed Graph Drawing for Representation of Reasoning

Master’s thesis in Informatics
Supervisor: Srinivasa Rao Satti
Co-supervisor: Paal Fredrik Skjørten Kvarberg
June 2023

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

Preface

This research project was solicited by the technology company Disputas and is incubated at the Norwegian University of Science and Technology (NTNU), under the Department of Computer Science.

I wish to thank my supervisor, Srinivasa Rao Satti, at the Department of Computer Science for his feedback on the thesis during the writing process. I would also like to thank my external supervisor, Paal Kvarberg at Disputas, for his continuous guidance. In addition, I am grateful for the developer, Andreas Netteland at Disputas, for helping and providing feedback on code implementations. Finally, I would like to thank my friends and family for their support throughout this project.

Abstract

Collaborative Bayesian reasoning is a process in which two or more people try to answer a research question by sharing their beliefs about the different reasons relevant to the given question. In order to facilitate collaborative Bayesian reasoning, it would be helpful to use a technology for representing beliefs and the reasons that bear on those beliefs graphically.

This research project explores the implementation of force-directed graph drawing for representation of reasoning. The goal of the project is to explore the potential application of this type of graph drawing for representing complex reasoning structures, known as Bayesian networks.

The research question is about how force-directed layout algorithms can provide a more intuitive and effective way to visualize and help with complex reasoning compared to traditional methods. The Design Science research method was used to develop a prototype solution to the problem by implementing features using an existing library. The resulting product is a functional prototype that covers most of the specified requirements.

Oppsummering

Samarbeidende bayesiansk resonnement er en prosess der to eller flere mennesker prøver å svare på et forskningsspørsmål ved å dele deres tro på de ulike årsakene som har betydning for spørsmålet. For å tilrettelegge for samarbeidende Bayesiansk resonnement, ville det vært nyttig å bruke en teknologi for å representere tro, og årsakene som ligger bak disse, grafisk.

Dette forskningsprosjektet utforsker implementeringen av kraftstyrt graftegning for representasjon av resonnement. Målet med prosjektet er å utforske den potensielle anvendelsen av denne typen graftegning for å representere komplekse resonnementstrukturer, kjent som Bayesianske nettverk.

Forskningsspørsmålet handler om hvordan kraftstyrte layoutalgoritmer kan gi en mer intuitiv og effektiv måte å visualisere og hjelpe med komplekse resonnementer sammenlignet med tradisjonelle metoder. Forskningsmetoden Design Science ble brukt til å utvikle en prototypeløsning på problemet gjennom å implementere funksjoner ved bruk av et eksisterende bibliotek. Det resulterende produktet er en funksjonell prototype som dekker de fleste spesifiserte kravene.

Table of Contents

List of Figures viii

List of Tables ix

Glossary xi

1 Introduction 1

1.1 Problem Description and Motivation 1

1.2 Expected Results 2

1.3 Outline of this Document 2

2 Background 3

2.1 Preparatory Project 3

2.2 Disputas 3

2.3 Ponder 3

2.3.1 Directed Acyclic Graph Layout Algorithm 4

2.4 Force-Directed Graph Drawing Algorithm 4

2.5 Collaborative Bayesian Networks 4

2.5.1 Superforecasting 7

2.5.2 Use Case 7

2.6 Graph Theory 7

2.6.1 Graph Data Structure 7

3 Relevant Technologies 9

3.1 PageRank 9

3.2 React 9
3.3 D3-Force 10
3.4 Springy 11
3.5 TypeScript 12
3.5.1 TypeDoc 13
3.6 Node.js 13

4 Research and Planning 14

4.1 Design Science as a Research Method 14
4.1.1 Artifact and Evaluation 15
4.2 Requirements 16
4.2.1 Requirement List 16
4.2.2 Requirement Details 18
4.3 Evaluation of Existing Technologies 20
4.3.1 Relevant Technologies 20
4.3.2 Reviewing Relevant Technologies 21
4.3.3 Performance Test and Analysis 21
4.3.4 Deciding on a Technology 26

5 Implementation 28

5.1 Feature Implementations 28
5.1.1 Starting Point 28
5.1.2 Edge Directionality 29
5.1.3 Node Dragging 30
5.1.4 Zoom and Pan 32
5.1.5 Variable Node Size 33
5.1.6 Edge Length Correction 37
5.1.7 Credence Value Visualization 37
5.1.8 Dynamic Text Visibility 38
5.1.9 Bayesian Inference 40

5.2 Ponder Integration 46
5.2.1 Merging Graph Types 46
5.2.2 React Component 48
5.2.3 Rendering 49

6 Evaluation **52**
6.1 Initial Prototype Comparison 52
6.2 System Interaction 53
6.3 Cross-Browser Compatibility Test 55
6.4 System Performance Test 56
6.5 Requirement Review 57
6.5.1 Functional Requirements 57
6.5.2 Non-Functional Requirements 60
6.5.3 Requirement Evaluation Overview 60

7 Discussion and Conclusion **62**
7.1 Discussion 62
7.1.1 Advantages 62
7.1.2 Limitations 62
7.1.3 Further Work 63
7.2 Conclusion 63

Bibliography **64**

A Code **68**
A.1 D3-Force Performance Test 68
A.2 Springy Performance Test 70
A.3 Types 72
A.4 Index 72
A.5 Utility Functions 74
A.6 Graph Drawing 79

B Data from Performance Testing **84**

B.1 Data Collection for Initial Graph Layout Load Time 84

B.2 Data Collection for Graph Rendering Frame Rate 87

B.3 Data Collection for the Finalized System Performance Test 90

List of Figures

1.1 Proposed Design Idea 2

2.1 Small Graph Demo: Modeling of Immigration Considerations 6

2.2 Large Graph Demo: Map of EA Cause Areas 6

5.1 Initial Graph Design 29

5.2 Edge Directionality 30

5.3 Variable Node Size 36

5.4 Edge Length Correction 37

5.5 Credence Value Visualization 38

5.6 Text Visibility Improvements - Zoomed In 40

5.7 Text visibility Improvements - Zoomed Out 40

5.8 Interface Diagram 47

5.9 Ponder GUI - FDG View 50

5.10 Ponder GUI - DAG View 51

6.1 Initially Proposed Design Idea 53

6.2 FDG Drawing 53

6.3 System Interaction Diagram 54

6.4 Multiple Graphs Simultaneously 59

List of Tables

4.1 Requirements . . . 17
4.2 Comparison of Relevant Technologies Based on GitHub Star Rating . . . 20
4.3 Initial Graph Layout Load Times - Average . . . 22
4.4 Initial Graph Layout Load Times - Std.Dev. . . . 22
4.5 Graph Rendering Frame Rate - Average . . . 23
4.6 Graph Rendering Frame Rate - Std. Dev. . . . 23

6.1 Browser Test Results . . . 56
6.2 Initial Graph Layout Load Times . . . 56
6.3 Graph Rendering Frame Rate . . . 57
6.4 Requirement Evaluation . . . 61

B.1 Initial Graph Layout Load Time . . . 84
B.2 Graph Rendering Frame Rate . . . 87

Listings

4.1 Graph Building Springy . . . 24
4.2 Graph Building D3-Force . . . 25
4.3 measureFPS Function . . . 25
4.4 Initial Layout Load Time Measurement . . . 26
5.1 Arrow Head SVG Definition . . . 29
5.2 Drag Function . . . 31
5.3 D3 Zoom . . . 32
5.4 Set Node Radius Using PageRank Algorithm . . . 34
5.5 findDirectParents Function . . . 35
5.6 Set Node Degree . . . 36
5.7 Set Link Force Distance . . . 37
5.8 Set Node Opacity . . . 38
5.9 Dynamic Text Visibility . . . 39
5.10 Set Node Tooltip . . . 39
5.11 applyBayesianReasoning Function . . . 43
5.12 setBayesianValue Function . . . 43
5.13 calculateBayesianValue Function . . . 44
5.14 intersectAddCombinations Function . . . 45
5.15 nCombinations Function . . . 45
5.16 Merging Graph Types . . . 47
5.17 React Component . . . 49
A.1 d3-force-performance-test.js . . . 68
A.2 springy-performance-test.js . . . 70
A.3 types.ts . . . 72
A.4 index.ts . . . 73
A.5 utils.ts . . . 74
A.6 drawFDG.ts . . . 79

Glossary

.

.

Chapter 1

Introduction

This chapter introduces the reader to the topic of the thesis, by giving a short rundown of where the problem stems from, as well as the motivation for solving it. Following is an outline of the document that briefly describes all of its chapters.

1.1 Problem Description and Motivation

The problem itself comes from the forecasting and decision making domain. This area involves trying to accurately predict the probabilities of future events or outcomes. It has traditionally been attempted using field experts' opinions, either individually or in collaboration with each other. This form of reasoning involves elicitation and aggregation procedures that can be both time-consuming and difficult to conduct accurately without making mistakes. Traditional methods of communication are not ideal for this type of reasoning.

Disputas is an organization looking to extend the features of their Ponder platform to cover epistemic use cases, including collaborative probabilistic reasoning. Graphical visualization of reasoning is central to this use case.

Collaborative Bayesian reasoning is a process in which multiple people try to answer a research question by sharing their beliefs about the question. Research in the decision sciences has uncovered several effective means by which individuals and groups can improve the accuracy of judgments by using Bayesian reasoning.

The goal of this project is therefore to create a prototype force-directed graph drawing layout algorithm that covers the key aspects of graphical visualization of collaborative probabilistic reasoning.

A research question has therefore been formulated from this, as follows: "How can a force-directed graph drawing algorithm be implemented for a Bayesian network while maintaining visual clarity and user interaction?"

1.2 Expected Results

Disputas has provided an initial prototype, shown in Figure 1.1. This expresses how they envision a final prototype looking like.

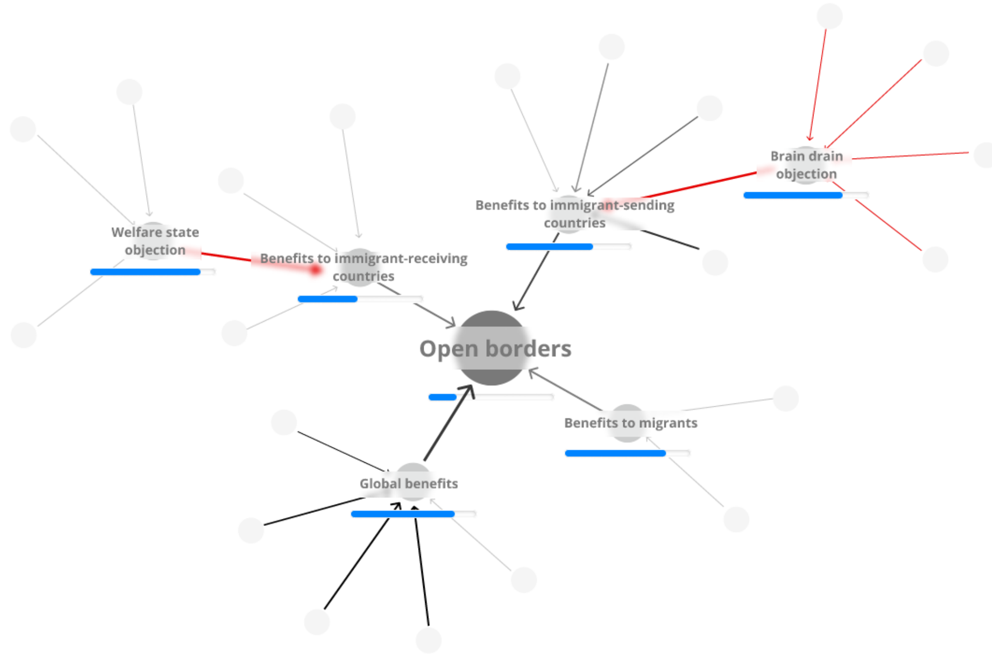


Figure 1.1: Proposed Design Idea

1.3 Outline of this Document

This document consists of seven chapters. Assuming you have already read this one (Chapter 1), which introduces the reader to the topic. Chapter 2 goes a little further in depth by exploring the background of the problem. Chapter 3 involves some of the relevant technologies used. Chapter 4 starts with defining the research method used, followed by the requirement definitions. It then goes on about preparing for implementation through exploring relevant technologies. Chapter 5 goes through the development process of implementing features according to the requirements, using an existing library. Chapter 6 evaluates the final product, by analyzing the fulfillment of each requirement. Chapter 7 contains a discussion and conclusion of the results by pointing out findings, as well as some further work needed. Finally, Appendix A lists all the source code used, while Appendix B contains the data collection used for performance testing.

Chapter 2

Background

This chapter goes a little further into the background of the problem than what was initially described. The problem is introduced and elicited by the solicitor. Then, the background and motivation for developing the product is elaborated on, followed by a description of its use case. In the end, graph theory, and its relevancy is described.

2.1 Preparatory Project

It should be noted that the preparatory project previously worked on by the author was not relevant to the thesis. The reason for this was that it was decided by the author to completely change to a different project topic, which led to the project's results being discarded. The thesis was therefore completed in just one semester.

2.2 Disputas

Disputas is a company founded in 2019 [19]. Their team consists of programmers, designers, and advisors. The idea for their key product came from a need at Norwegian universities, which is an educational platform for making and doing practice assignments, called Ponder.

2.3 Ponder

Ponder leverages technologies for text annotation, graphical visualization, and semantic similarity in the creation of complex practice assignments with automatic assessment and feedback. Disputas is looking to extend features of the Ponder platform to cover epistemic use cases, including collaborative probabilistic reasoning. Graphical visualization of the logical structures of explanation and justification (reasoning) is central to this use case.

2.3.1 Directed Acyclic Graph Layout Algorithm

Disputas currently uses a Directed Acyclic Graph (DAG) layout algorithm for Ponder, based on Dagre [17]. The product produced from this project will function as an alternative graph presentation, coexisting with the DAG algorithm, inside of Ponder. This means that the user will be able to choose between two different graph presentations.

2.4 Force-Directed Graph Drawing Algorithm

A force-directed graph (FDG) drawing algorithms is a type of algorithm used to draw graphs in an aesthetically pleasing way. The purpose of this algorithm is to position the nodes in a graph, so that the edges have about the same length, and that they cross over each other as rarely as possible [43]. This technique is usually applied in either two-dimensional or even three-dimensional space. FDG drawing algorithms assign forces to every node and edge within the graph. Typically, these forces are used to attract directly connected nodes of the graph towards each other, while simultaneously using repulsive forces to keep other nodes apart.

The algorithms for doing this are based on a physical simulation, where nodes are treated as particles that repel each other, while edges are considered springs that make their connected nodes attract each other. The simulation runs until the system reaches a state of equilibrium, where all forces balance out, and the graph ends up being drawn in an aesthetically pleasing way [47]. FDG drawing algorithms are usually simple, making them very flexible for calculating layouts for simple graphs in a variety of different configurations. Overall, these graphs are useful for visualizing the connections between objects in a network, and can reveal more about the underlying structure. They minimize overlaps in the graph, evenly distribute nodes and edges, and organize them so that edges are of similar length to each other [18]. "Springy.js" provides a demo for an example of how such a force-directed layout algorithm works [24].

2.5 Collaborative Bayesian Networks

The key goal of this project is to create a layout algorithm that covers key demands of graphical visualization for the epistemic use case of Collaborative Bayesian reasoning. This is a collaborative reasoning process guided by the norms of Bayesian reasoning. Research in the decision sciences has uncovered several effective means by which individuals and groups can improve the accuracy of their judgments. A key finding is that "intuitive Bayesians", i.e., individuals who reason in a Bayesian manner, tend to make more accurate judgments. Another important finding is that groups outperform individuals in many decision-making tasks. See "Two directions for research on forecasting and decision making" for more on group deliberation and Bayesian reasoning [28].

Traditional methods of communication are not ideal for collaborative Bayesian reason-

ing. This form of reasoning involves elicitation and aggregation procedures that can be complicated and time-consuming to conduct without mistakes. Moreover, when the number of collaborators grows, the information consumption of their contributions to the collaborative reasoning process becomes unmanageable if their contributions are represented using traditional text-based methods. For these reasons, several actors have for some time been trying to develop innovative technologies to support the elicitation, aggregation, and representation of collaborative Bayesian reasoning.

An approach that has garnered considerable interest is the use of knowledge graphs to represent the reasoning of the collaborators involved in collaborative Bayesian reasoning. Such graphs are also called Bayesian networks, and although they have not been used much for collaborative Bayesian reasoning, they have been successfully used to model causal systems in a plethora of domains, including finance, clinical decision-support, and risk analysis [33].

The graphs that could support collaborative Bayesian reasoning can be called collaborative Bayesian networks. Such graphs can work just fine even with a very simple ontology where nodes are propositions and directed edges are inferences. People can report their credences (suggested truth values) regarding the likely truth of statements, and the perceived strength of inferences. Bayesian networks have several attractive features, including:

- An aggregation mechanism for group belief.
- Bayesian inference and belief propagation.
- The network/graph can grow quite large.

For graphs that are expected to grow large, it can be hard to determine in advance which direction the graph should grow, and in what manner. For this reason, collaborative Bayesian networks should not have a direction (like decision trees or argument diagrams). They should preferably be able to grow in all directions, in a way that space is used sparingly. See Figure 2.1 for an example of what this could look like.

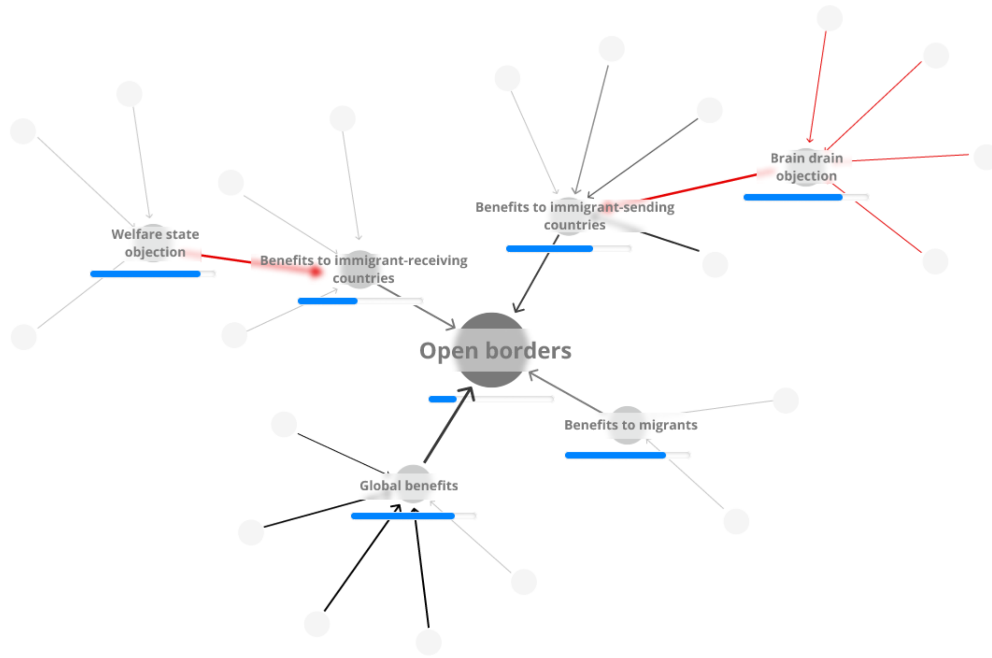


Figure 2.1: Small Graph Demo: Modeling of Immigration Considerations

In Figure 2.1, Disputas has modeled key considerations that are relevant to the issue of immigration. However, as mentioned, a key attraction of collaborative graphs is that they can grow really large. An exciting idea is that of graphs morphing into each other, propagating beliefs through Bayesian inference relations over a vast intellectual terrain. See Figure 2.2 for an illustration of what this could look like.

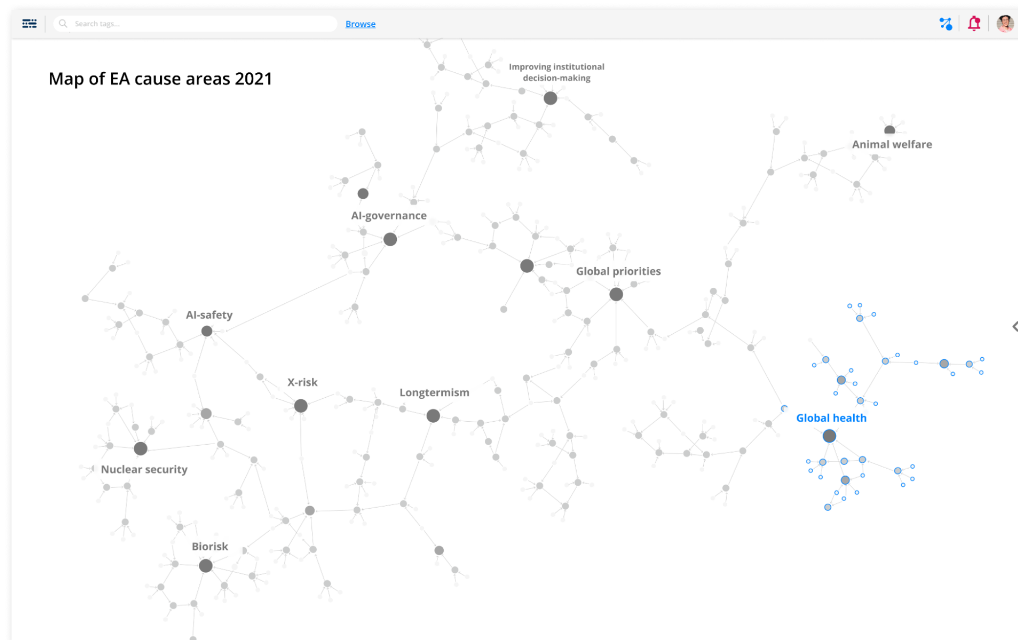


Figure 2.2: Large Graph Demo: Map of EA Cause Areas

2.5.1 Superforecasting

"Superforecasting" serves as an evidence-based, systematic approach for predicting future outcomes or events accurately. Its popularity stems from research described in the book titled "Superforecasting: The Art and Science of Prediction" [39].

Superforecasting involves a group of individuals known as "superforecasters". These individuals use techniques such as breaking down complex problems into smaller, more manageable pieces, and actively look for multiple perspectives by analyzing multiple different relevant data sources. They update their forecast models regularly by using probabilistic thinking techniques that focus on assigning specific probabilities to different outcomes, rather than making binary (true or false) decisions, that may be out of touch with the underlying levels of uncertainty involved in such complex questions.

Superforecasting can be applied to different industries, including geopolitical forecasting, technological advancements, financial predictions, etc. Its reason for existence is being able to utilize a group's collective intelligence towards more dependable and accurate forecasts, by building them on more refined and nuanced understandings and predictions, in the vast world of uncertain domains.

2.5.2 Use Case

Collaborative Bayesian reasoning is a process in which two or more people try to answer a research question by sharing their beliefs about the reasons that bear on the given question. In order to facilitate collaborative Bayesian reasoning, it would be helpful to use a technology for representing beliefs and the reasons that bear on those beliefs graphically.

2.6 Graph Theory

Pairwise relationships with objects can be modeled using mathematical structures called graphs according to the field of mathematics known as graph theory. A set of interconnected nodes, also referred to as vertices, compose a graph along with its connecting edges, also referred to as links. Edges are either undirected, where connections between two vertices occur symmetrically, or directed when connections between node pairs are asymmetrical. The graphs used in this document contain directed edges only.

2.6.1 Graph Data Structure

Various terms are used within graph theory to describe the graph structures and the relationships between its components. These are some of the most important terms in the context of this document:

A **source node** is the node from which an edge is emanating.

Similarly, a **target node** is the node to which an edge is directed.

A **parent node** is a node that has one or more child nodes connected to it. A parent of a child node will have an edge directed from the parent to the child.

A **child node** is a node that is a descendant of another node in a graph. It is essentially the opposite of a parent node, in that it will have an edge directed to itself from the parent.

The **root node** is the most central node. It does not have any parents, and serves as the ancestor of all other nodes in the graph.

The **out-degree** of a node is the number of edges leaving that node. It is a measure of how many other nodes can be reached directly from the given node.

Similarly, the **in-degree** of a node is the in-degree of a node is the number of edges coming into that node. It's a measure of how many other nodes can reach the given node directly.

Chapter 3

Relevant Technologies

In this chapter, some relevant technologies that are used during the development are described.

3.1 PageRank

PageRank, founded by Stanford University graduate students Larry Page and Sergey Brin (the eventual founders of Google), is an innovative algorithm that determines which website pages should show up first in response to user search queries on Google Search [45]. To do this, PageRank applies complex calculations based on how many inbound links each web page on the internet has (and how high quality these links are).

Essentially, if lots of other reputable websites refer traffic to a given page via shared hyperlinks, the page will appear more important than sites without such connections. After weighing all pages against each other using mathematical graph theory models, using pages represented as nodes, connected by links represented by edges, this system delivers a meaningful probability distribution. This indicates how valuable or relevant different web pages are, relative to each other. Because of these features, the algorithm turned out to be useful when determining the "connectedness" of nodes in the force-directed graph.

3.2 React

React is a free and open source front-end JavaScript library for building user interfaces based on "components" [46]. These components consist of parts of a user interface, meaning that by simply combining multiple components together in a certain configuration, an entire user interface can be built. It is maintained by Meta (previously known as Facebook) as well as a community of individual developers and companies. It can be used as a base on the front-end side of the development of single-page, mobile, or

even server-rendered applications. The only job of React is to render the combination of components to the Document Object Model (DOM), meaning that other operations such as state management, routing, and API interaction need to be managed elsewhere. "Redux" is an example of a state management tool that is often used in conjunction with React. The library abstracts away the DOM by virtualizing it, which offers both a simpler programming model and better performance [40]. React is already in use by Ponder, which needs to be taken into account when integrating the new graph system.

3.3 D3-Force

D3-force is a library, and a module within D3.js, that uses physics-based rules to position nodes and edges in a force-directed graph (see Section 2.4) visualization [15, 34]. The forces are calculated using a physics-based simulator that uses the velocity Verlet numerical integrator [31]. The force layout requires a larger amount of computation than other D3 layouts since FDG layouts are calculated in an iterative manner [8].

Developers can design and set forces that control the behavior of data components to start the visualization process with D3-force. Forces can then be applied to those components in the visualization while they are being drawn. The API for D3-force offers many ways of applying forces alone or in combination, enabling complicated force systems. Developers have the option to selectively vary, enable, or disable forces, allowing them to experiment and see how each force affects the visualization. This level of control makes sure that the visualization appropriately depicts the relationships and patterns in the underlying data, according to the desired "look and feel". Many different forces can be constructed and adjusted using the API [16], or they can even be custom made. To fine-tune the behavior of each force, characteristics can be selected, like strength, distance thresholds, and directionality. The interaction and arrangement of the data components inside the visualization can be controlled however it is desired because of this flexibility. Some of the most central built-in force functions that can be used to position the elements include, but are not limited to the following:

"d3.forceCenter()" is the centering force, which attracts nodes towards a specified center point. It helps achieve a balanced layout by pulling nodes towards the center, and preventing them from drifting too far away. This essentially functions by setting a center of gravity for the graph.

"d3.forceCollide()" is the collision force, which prevents nodes from overlapping with each other. It applies a repulsive force to nodes that are colliding, causing them to move away from each other.

"d3.forceLink()" is the link force and represents connections between nodes. It applies a force along all edges, pulling and pushing each pair of nodes towards each other. This force is essentially like a spring that gives node pairs a natural target distance between them. Essentially, it will push nodes further apart if their distance is smaller than the link's target distance, or pull them together if they are further apart.

“d3.forceManyBody()” is the charge force, simulating the effects of magnetic-like charge forces between nodes, as if there was a magnetic field across the graph. It applies both attractive and repulsive forces between all pairs of nodes, based on their distances. Nodes that are closer experience a repulsive force, while nodes that are farther apart from each other experience an attractive force.

“d3.forceX()” and **“d3.forceY()”** are the positioning forces, allowing the developer to explicitly set a desired “target” position for nodes, meaning that the specified nodes will tend to move towards the specific location. Coordinates can be assigned to nodes, or custom positioning rules can be specified. This force is often used to achieve specific node layouts or enforce a certain structure in the visualization.

In addition to forces, the API also provides functions for interacting with the elements. This allows the user to interact with the nodes and edges, or the entire graph itself, in various ways. Some of the options that can be provided for this are, among others, zooming and panning, dragging and dropping nodes, manipulating the graph, etc. The API is overall very feature-rich and well documented, making these types of implementations easy.

The physics simulation, which manages the forces over time, is typically ran for many iterations until the positions of the elements stabilize as the system reaches equilibrium. During each iteration, the positions of the elements are updated based on the forces applied to them [8], and the updated graph is drawn using a “tick” function. The tick function can either be set to iterate in an automatic and “natural” way, or it can be manually controlled in order to manipulate time, for example by temporarily pausing the simulation.

D3-force simulates physical forces on particles using a “velocity Verlet numerical integrator” [15]. “The simulation assumes a constant unit time step $\Delta t = 1$ for each step and a constant unit mass $m = 1$ for all particles”. The forces are calculated based on the properties of each element, such as size, mass, initial position, and speed [26]. The forces can be set up between elements, such as by making elements attract or repel one another based on their charge. D3-force can calculate the position and velocity of elements similar to the way real world physics determines the position and velocities of real elements. Some examples are the spring force, gravitational force, and magnetic force, as mentioned previously. Technically, these forces do not act on the graphical elements themselves, they act on the graph data. However, the results from the calculation of D3-force can be visualized using graphical elements such as SVG (Scalable Vector Graphics) or HTML Canvas. Using SVG seems to be the most popular option for rendering with D3-force.

3.4 Springy

Springy, also known as Springy.js, is a force-directed graph layout algorithm (see Section 2.4) made in JavaScript. The algorithm simulates physical forces between nodes and

edges in the graph to determine their positions. According to the source code [25], the physics simulations are calculated by using the following techniques:

“Coulomb’s Law” models the repulsion force between nodes. According to Coulomb’s Law, nodes that are closer to each other, repel each other with a force that decreases with distance. This force is inversely proportional to the square of the distance between the nodes.

“Hooke’s Law” models the attraction force between connected nodes (nodes connected by an edge). According to Hooke’s Law, connected nodes that are farther apart from their natural/resting length experience an attractive force that tries to bring them closer, and vice versa. This force is proportional to the displacement from the natural length of the spring connecting the nodes.

A center attraction force is applied to all nodes to bring them closer to the center of the graph. This force helps prevent nodes from drifting too far away from the center, and keeps the graph balanced.

The forces applied to the nodes are used to update their velocities and positions. The velocity of each node is updated by considering the applied forces, as well as a damping factor, which reduces the velocity over time. The updated velocity is then used to update the position of the node.

The above steps are repeated for multiple iterations, or time steps. In each iteration, the forces are recalculated based on the current positions, and applied to the nodes. This iterative process continues until the system reaches a stable state and the stopping criteria is met. This occurs when the total kinetic energy of the system falls below a fixed minimum energy threshold. The rendering of the simulation is then temporarily stopped as a result, to save computation power.

After iteratively updating the node positions according to the forces through the simulation, the node positions will eventually lead to the system’s total energy being minimized. This usually results in a visually pleasing arrangement of the nodes and edges in the graph, with connected nodes typically being closer together and disconnected nodes being farther apart.

Springy.js provides a basic graph API for creating and manipulating nodes and edges. It also includes some built-in rendering functionality, but this can be customized or replaced with a custom renderer. This means that the core part of the library only consists of the layout algorithm [23].

3.5 TypeScript

TypeScript is an open source programming language maintained and developed by Microsoft. It can be described as a strict syntactical superset of JavaScript. This means that anything written in JavaScript can also be written in TypeScript, while also offering other features that aren’t present in JavaScript. The language has gained more and more

popularity among developers because of this, and other features.

The main feature is static and strong typing abilities. Static typing simply means that variable types are checked during compilation, while strong typing means that the declaration of variable types is strictly enforced. With static typing comes the added ability to explicitly specify the type of variables, function parameters, and return values. This can make development easier by catching any errors related to typing during compilation rather than at runtime.

Type inference allows TypeScript to infer types when they're not explicitly specified based on the way functions are used or values are assigned to variables. Additionally. With type annotations and aliases, it's easy to create custom types for use throughout the code, resulting in improved readability and maintainability.

Since TypeScript was already heavily used within Ponder, it became a priority to develop the new system functionality using this programming language.

3.5.1 TypeDoc

TypeDoc is a documentation tool used to provide documentation for TypeScript projects. This tool can convert TypeScript comments into documentation in the form of HTML (Hypertext Mark-up Language) or JSON (JavaScript Object Notation). Since TypeScript was used in all of the source code in the project, TypeDoc was able to infer some of the documentation, such as parameters, return values, and even the D3-force library since it contains existing type definitions.

3.6 Node.js

Node.js, or simply Node, is an open-source, cross-platform, JavaScript runtime environment that allows developers to run JavaScript code on the back-end (server side) [44]. It was initially released in 2009 by Ryan Dahl, and is built on top of the JavaScript engine, which enables execution of JavaScript code outside of a web browser. It also comes with a package manager called NPM (Node Package Manager), which contains many packages, including the technologies mentioned previously: React, D3-force, Springy, TypeScript, and TypeDoc. This allows for a simplified installation process where each of the technologies can be used in conjunction with each other while being kept up-to-date by NPM.

Chapter 4

Research and Planning

This chapter goes over the research and planning required for the implementation of the Force-Directed Graph (FDG) drawing algorithm. It first introduces the research method used. Then, the requirements for the product are defined. Finally, a decision is made on the appropriate technology to use as a starting point in the development.

4.1 Design Science as a Research Method

The design science research (DSR) method was chosen for several reasons. This method suits the research question well since DSR focuses on solving a practical problem that can have a real-world impact, by systematically exploring, designing, and developing a concrete "artifact" that meets the specific requirements for the product.

The seven guidelines for conducting design science research, as outlined in "Design Science in Information Systems Research" [21], are central within design science. The points can be described as follows:

1. Design as an Artifact: The research must produce an artifact as an essential part of the research process. This artifact can be a software system, a device, a method, a model, or any other tangible or intangible object that provides a solution to the identified problem.

   In this case, the artifact is the implementation and customization of the force-directed graph drawing algorithm, and the integration of this into the Ponder platform made by Disputas.

2. Problem Relevance: The artifact created should address a significant and relevant problem in the domain of study. It should be designed to solve a real-world problem or improve an existing situation, making a practical impact in the field.

   The developed artifact here is a prototype force-directed graph drawing layout algorithm that aims to cover key aspects of graphical visualization of collaborative probabilistic reasoning.

3. Design Evaluation: The resulting artifact should be rigorously evaluated to demonstrate its utility, quality, and efficacy. The evaluation process should utilize well-executed evaluation methods and metrics that effectively measure the artifact's performance and effectiveness.

   An evaluation of the artifact was made in Chapter 6.

4. Research Contributions: The research must provide clear contributions to the field. This includes contributing to the theoretical knowledge base through the design and evaluation of the artifact. The design artifact itself can be considered a research contribution, as it adds to the understanding of the problem domain.

   In this case, the artifact itself is considered the research contribution.

5. Research Rigor: Both the construction and evaluation of the artifact should be conducted rigorously. The research process should adhere to established scientific principles and methods while maintaining relevance to the research context. Rigor ensures that the research outcomes are trustworthy and can withstand criticism.

   To ensure rigor, experiments and analyses such as performance and compatibility tests were performed, according to the artifact's requirements. The development itself was also tied up against the relevant requirements. All of the results produced are reproducible through the provided source code.

6. Design as a Search Process: Design is considered a search process aimed at finding effective solutions to problems. The research should follow an iterative approach, where the researcher generates proposed solutions, tests them, analyzes the results, and iterates on the design to improve the artifact's performance and effectiveness.

   Iterative design on a prototype, according to the artifact's requirements, was a large part of the development process.

7. Communication of Research: The research should be presented effectively to both technology-aware and management-oriented audiences. The research findings and the artifact should be presented in a manner that demonstrates how they can be applied in practical settings, providing guidance for implementation and utilization.

   The documentation contains both written explanations, as well as detailed code snippets for the feature implementations.

These guidelines provide a framework for conducting design science research, ensuring that the research process is systematic, relevant, rigorous, and contributes to both theory and practice.

4.1.1 Artifact and Evaluation

As per the design science guidelines, the goal of this thesis is to develop a product based on existing technologies by extending its functionality through adding new features. The next step is to incorporate this product into the already-existing Ponder platform, which was created by the company Disputas. The proposed solution will have its requirements

listed and specified in Section 4.2. Throughout the development, an iterative process of feedback and implementation made sure that the organization was constantly informed about the development direction, and allowed them to give their own opinions and feedback. Afterwards, the fulfillment degree of the artifact will be evaluated in Chapter 6 through a requirement fulfillment analysis. In this analysis, each requirement will be compared to the finished prototype. The artifact is to be built as a prototype solution, and will need further development in order to potentially become a commercially available product. It will therefore be evaluated as such.

4.2 Requirements

The requirements were initially loosely defined since it was unclear what would be more or less possible to implement within the given time frame. In addition, some of the requirements were modified and iterated on throughout the project. They were therefore reviewed and concretized at a later time. The requirements were split into two groups - primary and secondary, in addition to the functional and non-functional requirement groups. The primary requirements were stated as required for the system, while the secondary group was described as “nice-to-haves”. The final requirements are defined in Table 4.1, and are ordered from primary to secondary, and functional to non-functional. According to Disputas, these are the key demands that are required in order for such a technology to be more useful than already existing technologies. They are also specified in Sections 4.2.2.1 and 4.2.2.2, where more details are described. It should be noted that these are requirements that are meant for a first-iteration prototype of the system only. The requirements therefore reflect a foundation that can be considered a minimum viable product (MVP). A review going over the degree of accomplishment for each requirement is later defined in Section 6.5.

4.2.1 Requirement List

ID	Priority	Description
Functional Requirements		
FR1	Primary	The system should produce a force-directed graph drawing for representing non-layered graphs.
FR2	Primary	The system should be integrated into Ponder.
FR3	Primary	The system should represent propositions using nodes, and arguments using edges.
FR4	Primary	The nodes should display a text description.
FR5	Primary	The system should set each node's size depending on its "connectedness" in the graph.
FR6	Primary	The system should be easy to develop further.
FR7	Primary	The system should be implemented using TypeScript.
FR8	Primary	The user should be able to modify the graph.
FR9	Primary	Edges should have directionality.
FR10	Primary	The graph drawing should work with at least 80 nodes.
FR11	Secondary	The user should be able to drag and drop nodes around.
FR12	Secondary	The system should be able to handle multiple graphs within the same view at once.
FR13	Secondary	The system should support multi-person collaboration within the same graph.
FR14	Secondary	The user should be able to expand nodes, or click to open a modal for them.
FR15	Secondary	The nodes should display a credence value.
FR16	Secondary	The system should support zooming and panning inside the graph view.
FR17	Secondary	The system should infer new credence values based on existing ones in the graph.
Non-Functional Requirements		
NFR1	Primary	The system should perform with adequate speed in regards to human perception of slowness.
NFR2	Secondary	The system should be cross-browser compatible.

Table 4.1: Requirements

4.2.2 Requirement Details

4.2.2.1 Functional Requirements

FR1: The system should produce a force-directed graph drawing for representing non-layered graphs.
The graph drawing should be structured and function as a force-directed graph, as described in Section 2.4.

FR2: The system should be integrated into Ponder.
The system should coexist with the already implemented solution for DAG, as described in Section 2.3.1. Therefore, a method for switching between the two views should also be available. The rendering of the graph should be performed by Ponder.

FR3: The system should represent propositions using nodes, and arguments using edges.
Nodes should have a proposition attached, while the edges connecting them should contain arguments that lead to new propositions.

FR4: The nodes should display text content.
Each node should have a description as its text content, which represents its proposition.

FR5: The system should set each node's size depending on its "connectedness" in the graph.
The size of the nodes should vary based on their connectedness to other nodes. Specifically, the number of possible paths pointing to each node should be the focus of this calculation.

FR6: The system should be easy to develop further.
Being a prototype, the system should be easy to develop further with new features. It should therefore be flexible, customizable, and maintainable.

FR7: The system should be implemented using TypeScript.
The system should preferably be implemented using the programming language TypeScript, since this is already used in Ponder and is familiar to Disputas.

FR8: The user should be able to modify the graph.
Users should be able to add, edit, and delete nodes and edges in the graph.

FR9: Edges should have directionality.
Edges should have an assigned direction. This means that there should be a path explicitly from one node to another. This direction should be visualized using an arrow.

FR10: The graph drawing should work with at least 80 nodes.
The graph should be able to handle at least 80 nodes and still maintain usability.

FR11: The user should be able to drag and drop nodes around.
The system should provide easy-to-use drag-and-drop functionality for freely moving nodes around.

FR12: The system should be able to handle multiple graphs within the same view at once.
Multiple graphs can be created and interacted with at once. These should exist within the same view, and be able to be connected or disconnected from each other.

FR13: The system should support multi-person collaboration within the same graph.
Multiple users should be able to collaborate on the same graph simultaneously.

FR14: The user should be able to expand nodes, or click to open a modal for them.
Users should be able to expand nodes or click on them to open a modal window for more details.

FR15: The nodes should display a credence value.
The system should display the credence value of each node using visual cues such as labels. This value represents the truth value or strength of each proposition. The user should be able to position the labels.

FR16: The system should support zooming and panning inside the graph view.
The system should provide zooming and panning functionality for navigating graphs, using an infinite canvas style. While zooming, the node text should dynamically change its visibility depending on the zoom level.

FR17: The system should infer new credence values based on existing ones in the graph.
The system should include calculation capabilities for probability inference. This means that given a set of arguments and propositions with credence/truth values, new values should be inferred based on these. Specifically, the truth values from leaf nodes should be used to calculate new values for all non-leaf nodes by using Bayesian probability calculations. The calculation may include multiple different models for different use cases and/or to improve flexibility.

4.2.2.2 Non-Functional Requirements

NFR1: The system should perform with adequate speed in regards to human perception of slowness.
The system should provide adequate speed in terms of human perception, ensuring a responsive user interface. This implies an absolute runtime in terms of loading the graph, and rendering it smoothly. The system should continue to perform fast enough, even with at least 80 nodes, as described in FR10.

NFR2: The system should be cross-browser compatible.
The system should be compatible with different web browsers, ensuring compatibility across major browsers. The four most popular web browsers should be supported.

4.3 Evaluation of Existing Technologies

Before beginning the development process, a review of existing work first had to be performed. The intention of this was to determine which relevant FDG (force-directed graph) drawing technologies already exist, and which ones are the best candidates for further development according to the requirements.

4.3.1 Relevant Technologies

In order to evaluate the most relevant algorithms and libraries, a research process involving a review of existing technologies was required. Since requirement FR6 mentions the importance of flexibility and customizability, Disputas suggested beginning this process by sorting relevant open source repositories by popularity on the platform "GitHub". A search process was performed among a set of candidate libraries by browsing GitHub repositories using the search term *force directed graph*, and sorting by the number of "stars". Star count is a strong indication of popularity, and was therefore used to quickly gain an overview over good candidates, as well as filtering out lower-quality technologies. The list of these technologies is shown in Table 4.2.

Table 4.2: Comparison of Relevant Technologies Based on GitHub Star Rating

Technology	**Stars**
Springy.js [23]	1783
d3-force [15]	1257

graph-force [20]	159
ngraph.forcelayout [2]	135
d3-force-reuse [41]	111
ForceDirectedLayout [7]	75
nodesoup [32]	40
cytoscape.js-euler [11]	28
ForceDirectedPlacement [3]	26
Force-Directed-Layout [22]	20
Fastest-Force-Directed-Graph [27]	19

4.3.2 Reviewing Relevant Technologies

The two options that stood out were the libraries “D3-force” (see Section 3.3) and “Springy” (see Section 3.4). These were determined to be the most relevant because of the fact that they were about an order of magnitude more popular than the other related technologies, while they both seemed to be compatible with most of the primary requirements, and some of the secondary ones “out of the box”. These two libraries would therefore need to be explored further in order to determine if they could be a viable option as a starting point in the development process.

4.3.3 Performance Test and Analysis

An experiment for comparing the performance between Springy and D3-force was conducted by creating test graphs of varying sizes, using each of the libraries. This test would serve as a comparison between the two, for assessing whether they could pass requirements FR10 and NFR1, which are related to performance.

It should be noted that the web browser used during the test was Google Chrome (version 109.0.5414.119).

For both algorithms, the graphs were displayed in varying sizes, and two types of performance metrics were measured - rendering and load times. The test graphs each had twice the number of edges as nodes. These were varied between steps of 100, 1 000, 10

000, and 100 000 nodes in order to test how each library performed under different loads. For each test, a sample was collected 10 times for every graph size. The samples were then aggregated into average and standard deviation (std. dev.) numbers. The full data collection can be found in Appendix B, and the code used is listed in Appendix A.

4.3.3.1 Initial Graph Layout Load Time

For the first part of the test, time was measured for the generation of the initial graph layout. Note that this is the time passed before the rendering of any frames at all. The resulting measurement therefore represents the time the user spends looking at a blank screen (lower is better). This is important because of the attention span available before the user may become impatient and decide to close the page. This was found by measuring the initial delay from the time the script starts until the rendering begins. The first frame is rendered when the first call to the "tick" function is made by the physics simulation. This metric is referred to as the "initial graph layout load time". Table 4.3 shows the average initial graph layout load time, while Table 4.4 shows the standard deviation for the initial graph layout load time.

Table 4.3: Initial Graph Layout Load Times - Average

Nodes	Springy average (ms)	D3-force average (ms)
100	9.47e-1	1.78
1 000	11.9	9.97
10 000	231	75.2
100 000	1.55e+04	779

Table 4.4: Initial Graph Layout Load Times - Std.Dev.

Nodes	Springy std. dev. (ms)	D3-force std. dev. (ms)
100	3.88e-1	6.13e-1
1 000	4.82	3.70
10 000	80.6	27.2
100 000	5.23e+3	280

4.3.3.2 Graph Rendering Frame Rate

For the second part of the test, the performance while rendering the graphs was evaluated. This measurement looked at the rate of the frames being rendered (frame rate), meaning

the number of new frames being shown to the screen over time. To measure this, a counter kept track of the frame count by incrementing every time a call to the "tick" function was made. This accumulated over one-second intervals before it was checked and reset back to zero at the end of each interval.

The metric is known as "frames per second" (FPS), and is commonly used in video graphics to determine the overall smoothness of animations (higher is better). The consequence of a low frame rate is "stuttering", where the animations no longer look smooth, which can be jarring to interact with for the user. Table 4.5 shows the average frame rate while rendering the graph, and Table 4.6 shows the standard deviation for the graph rendering frame rate.

Table 4.5: Graph Rendering Frame Rate - Average

Nodes	Springy average (FPS)	D3-force average (FPS)
100	128	129
1 000	14.8	19.1
10 000	2.20e-1	1.74
100 000	1.57e-3	1.67

Table 4.6: Graph Rendering Frame Rate - Std. Dev.

Nodes	Springy std. dev. (FPS)	D3-force std. dev. (FPS)
100	2.87	3.38
1 000	8.22e-1	1.54
10 000	6.38e-3	1.94e-1
100 000	4.16e-5	4.33

4.3.3.3 Results Analysis

Looking at the requirements in Section 4.2, the interface should perform "with adequate speed in regards to human perception of slowness" according to requirement NFR1. It should also "be able to handle at least 80 nodes and still maintain usability", as specified by requirement FR10. It is therefore most interesting to look at the data for graphs containing 100 nodes.

According to "Section", the following is stated about how page load time affects how often the user may decide to exit a website before it loads: "some estimates say up to 1% loss for every 100 ms delay in page load time" [35]. The results from the initial graph

layout load times test show that even with 10 000 nodes, the load times are usually less than 100 ms for both algorithms. For 100 nodes, the load times are only about one or two milliseconds, meaning that even at 10 times the load time, there would in theory still be less than a 1% loss of users.

When looking at the rendering frame rates, we can see that the frame rate is kept at more than 60 FPS for the graphs with the size of 100 nodes. According to Healthline Media, "In the past, experts maintained that most people's maximum ability to detect flicker ranged between 50 and 90 Hz, or that the maximum number of frames per second that a person could see topped out around 60." [29]. This means that the rendering performance is about twice that of 60 FPS, which is what is desired at a minimum.

In both of these scenarios, the requirements are met with a solid margin to spare. The results of the performance test therefore show that both Springy and D3-force perform sufficiently for the intended purpose, in terms of performance.

4.3.3.4 Implementation of Performance Testing

The graph generation was done in a random fashion, where each node was created with a random name, and edges were added between the new node and two randomly chosen other nodes. The variable "numberOfNodes" was adjusted throughout the performance tests to vary the graph size between 100, 1 000, 10 000, and 100 000 nodes. The graph building for Springy is shown in Listing 4.1, and for D3-force in Listing 4.2.

Listing 4.1: Graph Building Springy

```
const graph = new Springy.Graph();
const nodes = [graph.newNode({ label: "initialNode" })];
const edgeColor = "#EB6841";
const numberOfNodes = <number_of_nodes>;

// Create a graph with a certain number of nodes and twice the
   number of edges
for (let i = 0; i < numberOfNodes; i++) {
  // nodes
  const name = (Math.random() + 1).toString(36).substring(7);
  const newNode = graph.newNode({ label: name });
  nodes.push(newNode);

  // edges
  const otherNode1 = nodes[Math.floor(Math.random() *
     nodes.length)];
  graph.newEdge(otherNode1, newNode, { color: edgeColor });

  const otherNode2 = nodes[Math.floor(Math.random() *
     nodes.length)];
```

```
  graph.newEdge(otherNode2, newNode, { color: edgeColor });
}
```

Listing 4.2: Graph Building D3-Force

```
// d3-test.js

let edges = [];

const numberOfNodes = 10;

// Create a graph with a certain number of new nodes and 2
    times the number of edges
for (let i = 0; i < numberOfNodes; i++) {
  let newNode = (Math.random() + 1).toString(36).substring(7);
  let otherNode1 = edges[Math.floor(Math.random() *
      edges.length)].source;
  let otherNode2 = edges[Math.floor(Math.random() *
      edges.length)].source;
  edges.push({ source: otherNode1, target: newNode });
  edges.push({ source: otherNode2, target: newNode });
}

let nodes = [];

for (let i = 0; i < 100; i++) {
  let newNode = (Math.random() + 1).toString(36).substring(7);
  nodes.push({ name: newNode });
}
```

The FPS measurement was set up using a simple method that could be used for both layout algorithms. An initial timestamp, "prevTime", was recorded using "performance.now()", and the counter "frames" was initialized to 0. A function "measureFPS()" was called with each animation frame update during the test. Every time this function was called, the frame counter was incremented. When the time elapsed ("time" minus "prevTime") exceeded 1000 milliseconds (1 second), the FPS was calculated as ("frames" * 1000) / ("time" minus "prevTime"), while resetting "prevTime" to the current time and the frame counter back to 0. The calculated FPS value was then logged to the console. This way, FPS was continually monitored and logged every second throughout the performance test. The code implementation for this is shown in Listing 4.3.

Listing 4.3: measureFPS Function

```
// springy-test.js
// d3-test.js

// Variables to measure frames per second (FPS)
```

```
let prevTime = performance.now();
let frames = 0;

// Function to measure FPS
const measureFPS = () => {
  const time = performance.now();
  frames++;
  if (time > prevTime + 1000) {
    const fps = (frames * 1000) / (time - prevTime);
    prevTime = time;
    frames = 0;
    console.info("FPS:", fps);
  }
};
```

The measurement of the initial layout loading time was more straightforward to find, and could also be used for both algorithms. Using JavaScript's built-in console timing function, a timer was started at the very beginning of the process with "console.time()". After the layout was fully loaded and was about to render its first frame, "console.timeEnd()" was called, automatically calculating the elapsed time and logging it to the console. This is shown in Listing 4.4.

Listing 4.4: Initial Layout Load Time Measurement

```
// springy-test.js
// d3-test.js

console.time("initial layout loading time"); // Start timing
    the initial layout loading process

// (...)

console.timeEnd("initial layout loading time"); // Stop timing
    the initial layout loading process and print the elapsed
    time
```

4.3.4 Deciding on a Technology

Since both algorithms performed well, either one would suffice. After reviewing the advantages and disadvantages of each, D3-force was chosen on the basis of the following advantages, which seemed to make D3-force appear more established, and set itself apart from Springy.

Firstly, the API is very comprehensive and well documented [16, 14]. Secondly, there exists a vast number of dedicated open source user repositories that explore use cases of D3-force, especially available on Observable, but also on GitHub [13, 12]. Thirdly,

the general “umbrella” D3 ecosystem is also very popular, and provides even more code examples and a large developer community, which can be helpful for further development. These points are especially significant when considering requirement FR6, which states that “the system should be easy to develop further” by being “flexible, customizable, and maintainable”. See Section 3.4 about Springy and Section 3.3 about D3-force for more details.

Chapter 5

Implementation

This chapter goes over the process of designing and developing the product by implementing new features into existing technologies. Various decisions about feature implementations are made along the way. Eventually, the developed product is integrated into Disputas' platform called Ponder. The illustrations shown throughout the chapter are made from recreating the graph used in the initial prototype design from Section 1.2. All the code used in this chapter can be found in Appendix A.

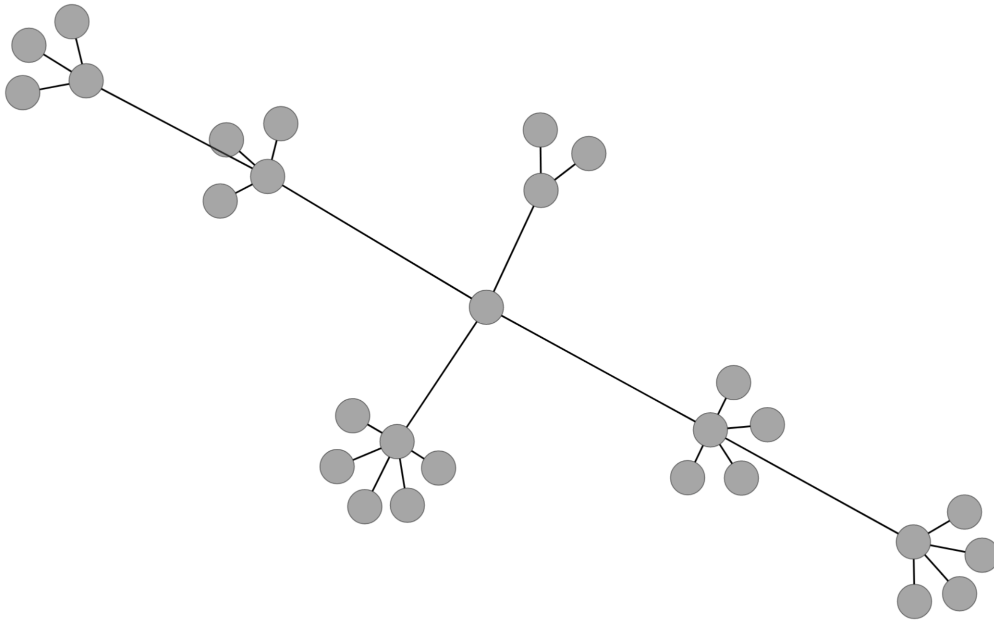
5.1 Feature Implementations

After d3-force was chosen as a starting point, the next step involved experimenting with feature implementations that were compatible with the D3-force module. Many iterations were made by communicating back and forth with Disputas during weekly meetings. The main feature iterations are described in this Section.

5.1.1 Starting Point

The first repository explored was an implementation that presented a graph in a hierarchical tree structure [6]. This was tested using a hierarchical data set, but was later discarded. It was decided to instead use a normal graph implementation, which was more freely structured [5]. This decision was made because of the fact that collaborative Bayesian networks should not have a direction, and should preferably be able to grow in all directions, as mentioned in Section 2.5. An initial design can be seen in Figure 5.1.

Figure 5.1: Initial Graph Design



5.1.2 Edge Directionality

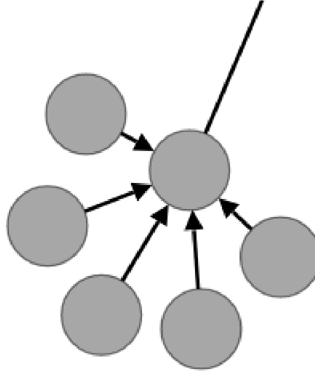
From the requirements in Section 4.2, FR11 states that edges in the graph should have a directionality, including a visible arrow. This was implemented by drawing a custom SVG element. The "d" attribute can be used to draw an element of any desired shape and size, such as the arrow head drawn in this case. After defining the element, it gets attached to every edge at the end of their line. See Listing 5.1 for the code. In this way, the edge will point from its source node to its target node.

Listing 5.1: Arrow Head SVG Definition

```
// drawFDG.ts

  // arrow head
  g.append("svg:defs")
    .append("svg:marker")
    .attr("id", "arrowhead")
    .attr("viewBox", "0 -5 10 10")
    .attr("markerWidth", 5)
    .attr("markerHeight", 5)
    .attr("orient", "auto")
    .append("svg:path")
    .attr("d", "M0,-5L10,0L0,5");

// attaches arrow head to the edges
edge
    .attr("marker-end", () => "url(#arrowhead)")
```

After adding this feature, the final look for the arrow head can be seen in Figure 5.2.

Figure 5.2: Edge Directionality



5.1.3 Node Dragging

According to requirement FR9, "the user should be able to drag and drop nodes around". This was implemented using the "drag" function in Listing 5.2. The function creates a drag behavior for a simulation in a D3.js-based visualization. Inside the function, three event handler functions for the drag behavior are defined: "dragStarted", "dragged", and "dragEnded".

The "dragStarted" function is called when the drag behavior starts on an element. It checks if the behaviour is active, i.e., if the user is currently dragging the element, using "event.active". If the behaviour is not active, it sets the "alpha target" of the simulation to 0.3 and restarts the simulation. This increases the simulation's "alpha value", which temporarily energizes the simulation by increasing the node velocities, making it continue to run smoothly during dragging [16].

The "dragged" function is called when the element is being dragged. It updates the "fx" (fixed x-position) and "fy" (fixed y-position) properties of the "subject" of the event, which represents the dragged element (the node). This sets the position of the element to the current mouse coordinates ("event.x" and "event.y"), effectively dragging it on the screen.

The "dragEnded" function is called when the drag behavior ends on the element. If the drag behavior is not active, it sets the alpha target of the simulation back to 0, effectively stopping the simulation's velocity decay, and slowing the simulation back down. Additionally, it resets the "fx" and "fy" properties of the dragged event to "null", allowing the element to move freely again.

Finally, the "drag" function returns a "d3.drag()" object configured with the event handlers. The "d3.drag()" function creates a drag behavior that can be applied to D3 elements. The "on" method is used to specify the event handlers for various drag events: "start" (calls "dragStarted"), "drag" (calls "dragged"), and "end" (calls "dragEnded").

This "drag" function can be used to enable dragging functionality for D3 elements within a simulation, allowing users to interactively move and reposition elements in the visualization. In this case, dragging is enabled for all nodes (including the text within them).

Listing 5.2: Drag Function

```
// drawFDG.ts

function drag(
simulation: d3.Simulation<Node, undefined>
): d3.DragBehavior<any, any, any> {
function dragStarted(event: d3.D3DragEvent<SVGRectElement, any,
    any>) {
  if (!event.active) simulation.alphaTarget(0.3).restart();
}

function dragged(event: d3.D3DragEvent<SVGRectElement, any,
    any>) {
  event.subject.fx = event.x;
  event.subject.fy = event.y;
}

function dragEnded(event: d3.D3DragEvent<SVGRectElement, any,
    any>) {
  if (!event.active) simulation.alphaTarget(0);
  event.subject.fx = null;
  event.subject.fy = null;
}

return d3
  .drag()
  .on("start", dragStarted)
  .on("drag", dragged)
  .on("end", dragEnded);
}

// calling the function from the relevant elements
node
  .call(drag(simulation));

text
.call(drag(simulation));
```

5.1.4 Zoom and Pan

Requirement FR16 in Section 4.2 refers to the functionality for zooming and panning while interacting with the graph. Both of these functionalities are provided by D3 in the module "d3.zoom" [9]. Despite the name not including the "pan" keyword, the module allows for both zooming and panning of SVG elements. Once the zoom behaviour is set up using "d3.zoom()", it can be applied to any selected elements using the "call" method. The "scaleExtent" function from d3.zoom was then used to define the minimum and maximum zoom levels to be 0.25 and 10 respectively, which were the same values as Ponder previously had used for their DAG implementation (see Section 2.3.1). This limitation was implemented to control the zooming capacity of the visualization, preventing excessive accidental zooming in or out, that could potentially cause the user to lose track of the positioning of the graph drawing itself. This necessity had become apparent since the graph was no longer centered in the view, but instead gave the user the freedom to move it around manually using the zoom/pan function.

Event listeners were also established for the zoom behavior. Using D3's "on()" method, three specific zoom events were monitored: "zoom", "start", and "end".

During the "zoom" event, a function called "handleZoom" was triggered. This function was made to adjust the SVG elements' position and scale them on the page according to the user's "zoom action", meaning any form of zooming or panning. The transform attribute in the "g" SVG element was modified based on the current transformation, triggered by the zoom event. This would be used to store the current state of the view.

In response to the "start" event, which fired when the zoom action began, the cursor was changed to "grabbing". This change indicated to users that they had initiated a zoom action. Once the zoom action was completed, the "end" event triggered, reverting the cursor back to its original state. This cursor change again acted as a visual cue to inform users that the zoom action had concluded.

Finally, the zoom behavior was attached to the SVG container with the ID "main" using the call method. This made it possible for the container to respond to user actions that trigger zoom events. The code for implementing zooming and panning is available in Listing 5.3.

Listing 5.3: D3 Zoom

```
// drawFDG.ts

const zoom = d3
.zoom()
.scaleExtent([0.25, 10])
.on("zoom", handleZoom)
.on("start", () => d3.select("#main").attr("cursor",
   "grabbing"))
.on("end", () => d3.select("#main").attr("cursor", "initial"));
```

```
d3.select("#main").call(zoom as any);

function handleZoom(event) {
  d3.select("g").attr("transform", event.transform.toString());
}
```

5.1.5 Variable Node Size

According to Requirement FR5 from Section 4.2, “the system should set each node’s size depending on its ‘connectedness’ in the graph”. The “PageRank” algorithm (see Section 3.1) was thought to be a fitting solution to this problem. The reason for this was first of all that the algorithm is well established, having famously been used originally by Google. Secondly, it fits the requirement specification, which states that “the number of possible paths pointing to each node should be the focus of this calculation”. According to Google, “PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.” [45]. Since the PageRank method seemed to align well with the requirement, it was decided to be implemented.

The PageRank algorithm was used for calculating and setting the “radius” property for all the nodes in the force-directed graph. This was important to the visualization aspect, as the algorithm could determine what the visual prominence of each node in the graph should be. Specifically, a high PageRank value would mean that the node should also have a high radius value, since a high PageRank value indicates high connectedness in the graph. In the same way, a larger node will visually signify a higher degree of importance for a given node.

Initially, each node was assigned an arbitrary radius value of 10 as its initial PageRank value. To ensure the accuracy and stability of the computation, the algorithm was set to iterate 100 times through the nodes, which was more than enough time for the values to converge.

The way this was calculated in every iteration started from each and every node. For each node, the algorithm identified its direct parents - the nodes that have edges pointing to it (direct parents) (see Listing 5.5). The function “findDirectParents” works by first filtering out edges that are pointing to the given child node. Then, it finds all the source nodes that are connected to these edges.

For each parent node, a value is calculated based on its PageRank value and the number of edges emanating from it (outDegree) (parentNode.radius / parentNode.outDegree). If the parent node had no outgoing edges (outDegree = 0), a value of 1 was used instead to avoid zero division. After accumulating these calculated values from each parent node, the radius of the child node was updated. This process was repeated until it eventually converged.

The inDegree and outDegree of each node have been calculated beforehand using the "setNodeDegree" function (see Listing 5.6). This function simply sets the "inDegree" of each node depending on how many edges have the given node as its "target" node, and vice versa for the "outDegree".

Once the iterations were completed, a normalization process was performed to customize the visual appearance by reducing the amount of differentiation among the nodes when displayed on the graph. The radius of each node was scaled using a logarithmic function to decrease the size differences, and then multiplied by a factor of 15, as the graph drawing more closely resembled the prototype after tweaking it this way.

The resulting graph, with updated radius values, was then returned by the function. The new radii of the nodes represented their PageRank values, and therefore their relative importance or influence within the network/graph. The function for calculating node radius using PageRank is shown in Listing 5.4.

Listing 5.4: Set Node Radius Using PageRank Algorithm

```
// utils.ts

/**
 * Calculates and sets the node radius property for all the
    nodes in the graph
 * using the PageRank algorithm.
 *
 * @export
 * @param {FDGGraph} { nodes, edges } The graph to be modified.
 * @returns {FDGGraph} A modified graph with radius values
    updated.
 */
export function setNodeRadius({ nodes, edges }: FDGGraph):
   FDGGraph {
  const initialValue = 10; // the initial PageRank value for
     each node, used to iterate into a final radius value
  const pageRankIterations = 100; // the number of iterations
     of the PageRank algorithm - set to a high value to make
     sure it converges

  // set initial radius value before iterating
  let modifiedNodes: Node[] = nodes.map((node) => ({
    ...node,
    radius: initialValue,
  }));

  // main radius calculation using the PageRank algorithm
  for (let i = 0; i < pageRankIterations; i++) {
    for (const node of modifiedNodes) {
```

```
      const parentNodes = findDirectParents(node, {
        nodes: modifiedNodes,
        edges,
      });
      const parentRadiusValues = parentNodes.map(
        (parentNode) => parentNode.radius /
          Math.max(parentNode.outDegree, 1)
      );
      if (parentRadiusValues.length !== 0) {
        node.radius = parentRadiusValues.reduce(
          (accumulator, value) => accumulator + value,
          0
        );
      }
    }
  }

  // normalize the radius values and adjust them for
     visualization purposes
  modifiedNodes = modifiedNodes.map((node) => ({
    ...node,
    radius: Math.log(node.radius) * 15,
  }));
  return { nodes: modifiedNodes, edges };
}
```

Listing 5.5: findDirectParents Function

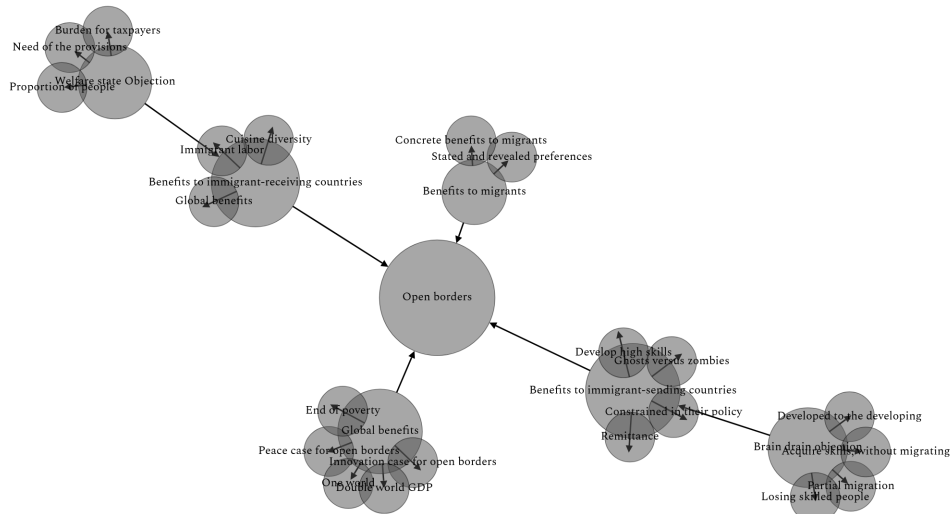
```
// utils.ts

/**
 * Finds all directly connected (first degree) parent nodes.
 *
 * @param {Node} node The child node to be assessed.
 * @param {FDGGraph} { nodes, edges } The graph containing the
     node.
 * @returns {Node[]} The directly connected parents.
 */
function findDirectParents(node: Node, { nodes, edges }:
   FDGGraph): Node[] {
  const incomingEdges = edges.filter((edge) => edge.target ===
     node.id);
  return nodes.filter((_node) =>
    incomingEdges.find((edge) => _node.id === edge.source)
  );
}
```

Listing 5.6: Set Node Degree

```
/**
 * Sets the properties 'inDegree' and 'outDegree' for every
    node in the graph.
 *
 * @export
 * @param {FDGGraph} { nodes, edges } The graph to be modified.
 * @returns A modified graph with updated properties.
 */
export function setNodeDegree({ nodes, edges }: FDGGraph):
   FDGGraph {
  const modifiedNodes = nodes.map((node) => ({
    ...node,
    inDegree: edges.filter((edge) => edge.target ===
        node.id).length,
    outDegree: edges.filter((edge) => edge.source ===
        node.id).length,
  }));
  return { nodes: modifiedNodes, edges };
}
```

After the variable node size feature was implemented, the node sizes were similar the originally proposed design (1.1). However, the edges were no longer proportionate to the new node sizes, and had to be corrected. This can be seen in Figure 5.3.

Figure 5.3: Variable Node Size



5.1.6 Edge Length Correction

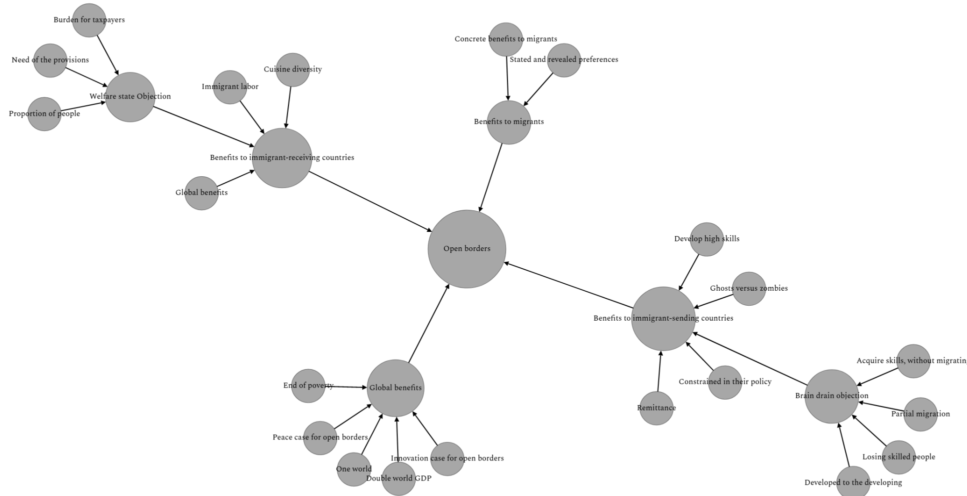
As a result of the varying node radius, the visible part of the edges between nodes were often too short, and even non-existent in some cases. The reason for this was that the appropriate distance for each edge was measured between the center points of each node. This meant that larger nodes would often cover up parts of its connected edges. The solution to this problem was to change the link distance parameter through the D3-force API, which effectively sets a new target length for each edge. The actual length of each edge is determined by a "spring force", and can be compared to the natural "resting length" of a physical spring, meaning that the value set in the link distance parameter is the target resting length (see 3.3). The goal of this change was to achieve a constant value for the visible part of the edge. Since the distance of this force is measured between the center points of each node, the new link distance value had to be set to the sum of the desired visible edge length and the radius of both the target and source nodes combined. This value was tweaked until an appropriate length was found. The small update to the code is shown in Listing 5.7.

Listing 5.7: Set Link Force Distance

```
link.distance(
    link => link.target.radius + link.source.radius + 70
);
```

The result, shown in Figure 5.4, was a more balanced ratio between node and edge size. The edges were therefore not looking too short anymore.

Figure 5.4: Edge Length Correction



5.1.7 Credence Value Visualization

Disputas also wanted to express the credence value of each node in some visual way, as per requirement FR15 (Section 4.2). What they suggested, was to use a gradient of

gray color tones that varied based on this value. This was achieved by using a constant shade of gray for the fill value, while varying the "fill-opacity" instead. The fill-opacity attribute controlled the transparency of the fill color.

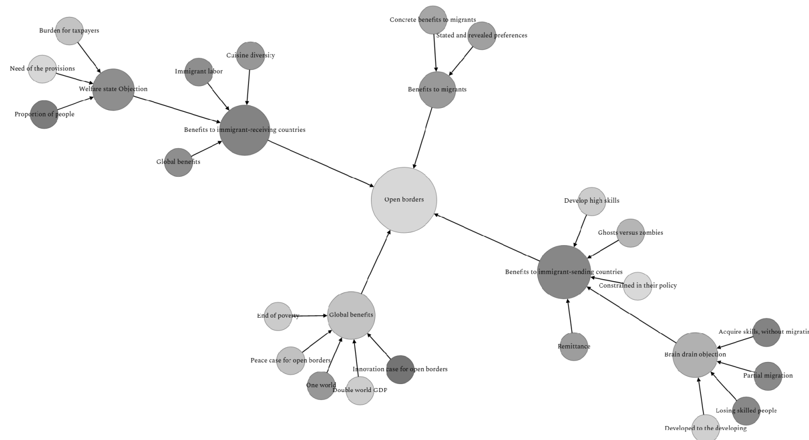
The "truthValue" property represents the credence value of the nodes, and ranges from 0 to 100. To convert this value into an appropriate fill opacity, it had to be transformed into a range between 0.2 and 0.8. By excluding the upper and lower opacity ranges, the contrast was still kept away from the white background and the black node text, since these have an equivalent of 0.0 and 1.0 opacity, respectively. This meant that a node with a truthValue of 0 would get a fill opacity of 0.2 (20% opaque), while a node with a truthValue of 100 would have a fill opacity of 0.8 (80% opaque). Nodes with truthValues between 0 and 100 would therefore have fill opacities between 20% and 80%. The code implementation can be found in Listing 5.8.

Listing 5.8: Set Node Opacity

```
node
  .attr("fill", "#505050") // gray
  .attr("fill-opacity", (d) => 0.2 + (0.6 * (d.truthValue as
      number)) / 100); // scales truthValue from 0-100 to
      0.8-0.2
```

Figure 5.5 shows the result of this feature. It should be noted that the credence values in this and the following examples are randomly generated, and are only used for demonstration purposes.

Figure 5.5: Credence Value Visualization



5.1.8 Dynamic Text Visibility

Another feature wanted by Disputas according to requirement FR16 in Section 4.2, was for dynamic text visibility to be implemented depending on the zoom level. This requirement was added in order to "clean up" the amount of text shown by every node on the screen at once. The solution to this problem was implemented in three parts.

Firstly, a cutoff limit was set for the maximal number of characters shown by each node description. Any overflowing text would be replaced with "...".

Secondly, functionality was implemented in order to hide more node descriptions when zooming further out of the graph, and showing more when zooming further in. This was calculated by looking at the zoom scale (magnification level) value "event.transform.k", representing the current zoom level, provided by the zoom event "D3ZoomEvent". If the product of this zoom scale and a node's radius was greater than 20, then the node's description text turned visible, otherwise, the text was removed (set to an empty string). This was added to the "handleZoom" function, shown in Listing 5.9.

Thirdly, a "tooltip" function was implemented as a way of displaying the entire node description, since most of the text is now hidden by default. A tooltip is simply a function that displays a text label whenever the user hovers the mouse pointer over something (here, a node). This was implemented by simply adding the full node description (as well as the credence value) to the "title" attribute for each node. These features together help avoid clutter and keep the visualization readable at different zoom levels. As shown in Listing 5.10, both the node description and its credence value were set as a tooltip for each node.

Listing 5.9: Dynamic Text Visibility

```
// drawFDG.ts

function handleZoom(event: d3.D3ZoomEvent<any, any>) {
    g.selectAll("text").text((d) => {
      const node = d as Node;

      const isRootNode = node.outDegree === 0;
      if (isRootNode)
        return node.description.length > 40
          ? node.description.substring(0, 38) + "..."
          : node.description;

      if (node.radius * event.transform.k > 20)
        return node.description.length > 30
          ? node.description.substring(0, 28) + "..."
          : node.description;

      return "";
    });
    d3.select("g").attr("transform",
        event.transform.toString());
}
```

Listing 5.10: Set Node Tooltip

```
  node
```

```
.append("title")
.text((node) => `truth value:
    ${node.truthValue}\ndescription: ${node.description}`);
```

The illustrations in Figures 5.6 and 5.7 show the improved text visibility, at two different zoom levels.

Figure 5.6: Text Visibility Improvements - Zoomed In

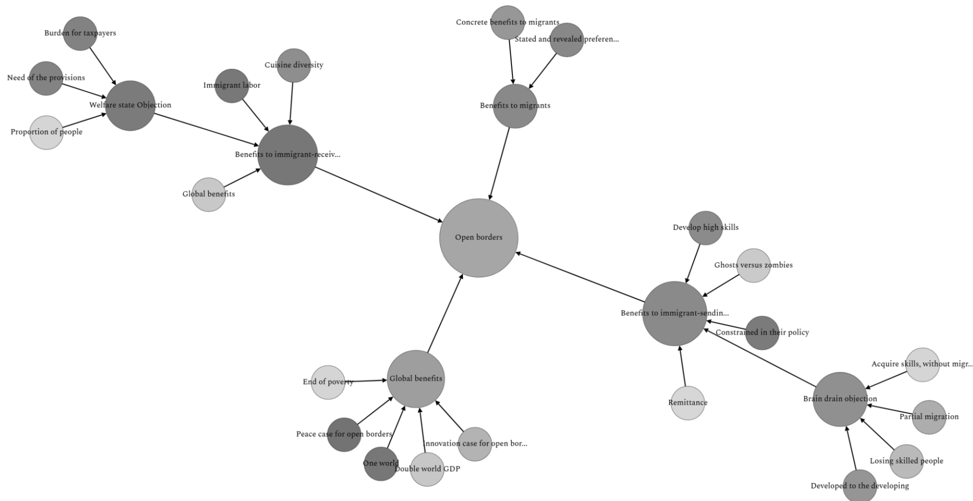
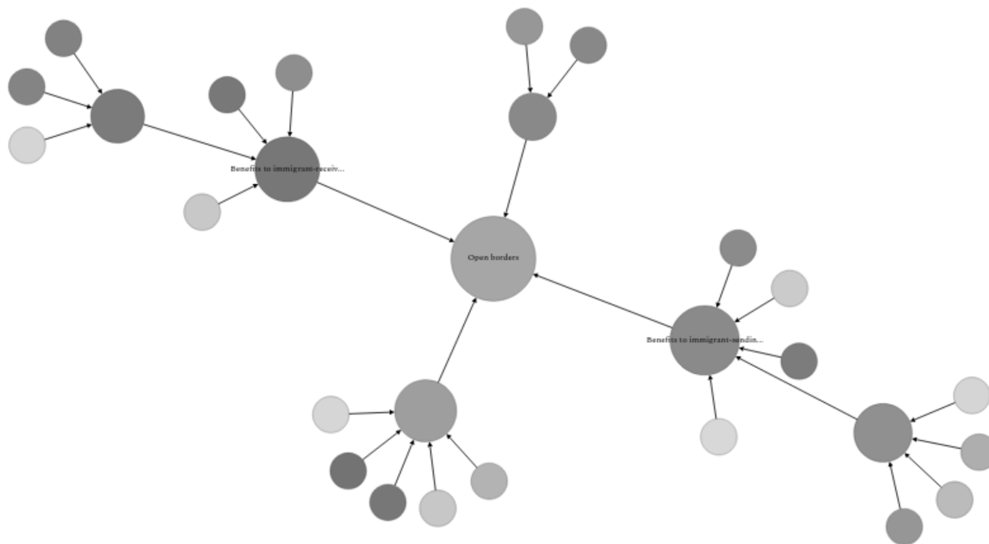


Figure 5.7: Text visibility Improvements - Zoomed Out



5.1.9 Bayesian Inference

Requirement FR17 (see 4.2) states that “truth values from leaf nodes should be used to calculate new values for all non-leaf nodes by using Bayesian probability calculations”. This implementation applies Bayesian reasoning to the graph by inferring Bayesian val-

ues for each node based on given truth values. Bayesian reasoning is a mathematical framework for updating probabilities based on new evidence or information. In this case, the leaf nodes are used as starting points for inferring new values to the rest of the nodes. The requirement states that the bayesian reasoning functionality may include multiple different models. One example of inference computation was therefore implemented to demonstrate how it could be done.

5.1.9.1 Dice Throwing Probability Model

When two events are mutually exclusive (i.e., a proposition has distinctive outcomes), their union can be found by simply summarizing their individual probabilities, since they are each distinct from each other [37]. However, this is not necessarily the case. If the events are *not* mutually exclusive, the probabilities cannot simply be added together, since the overlap of the events also needs to be accounted for in order to avoid double counting. In addition, whether the events are independent or not, meaning that their outcomes do not affect each other, does matter as well. This leads to a lot of different possibilities for modeling a Bayesian network, that should be explored further in future versions of the system. For now, a modeling of dice throws is implemented as an example.

As mentioned previously, the leaf nodes should be the source of credence values, and should be used to infer new values. They are therefore considered the source of information, and are effectively dice throwing events. This means that the credence value of a leaf node equates to the probability of showing any of a given set of sides. For example, a leaf node with a credence value of 50%, would be considered a dice throw where one of the sides 1, 2, or 3 would show. The probability of their children would then be calculated as the probability that at least one of their parents' events turned out to be true. This goes on until all the nodes have inferred truth values. The root event would therefore be equal to the probability that at least one of the leaf nodes is true.

This specific calculation implemented for dice throws has a set of assumptions for the calculations to be valid. It assumes that for a given child proposition, every direct parent propositions are independent from each other, meaning that their outcomes do not affect each other. It also assumes that each event is not mutually exclusive, meaning that the outcomes of each event cannot happen at the same time. This would be applicable to a series of dice throws. In this case, each event (throw) does not affect each other, and only one side can show at a time. The calculation used is the probability of the union of parent propositions, which is the probability that at least one of the propositions occurs. This is typically calculated using the "principle of inclusion-exclusion".

If we have sets representing parent nodes $A1, A2, ..., An$, then the probability of their union (the chance that at least one is true), denoted as $P(A1 \cup A2 \cup ... \cup An)$, is given by:

$$P(A1 \cup A2 \cup ... \cup An)$$

The union of three sets in the case of events that are not mutually exclusive [38]:

$$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$$

Similarly, the union of four sets in this case can be stated as follows:

$P(A \cup B \cup C \cup D) = P(A) + P(B) + P(C) + P(D) - P(A \cap B) - P(A \cap C) - P(A \cap D) - P(B \cap C) - P(B \cap D) - P(C \cap D) + P(A \cap B \cap C) + P(A \cap B \cap D) + P(A \cap C \cap D) + P(B \cap C \cap D) - P(A \cap B \cap C \cap D)$

The principle of inclusion-exclusion, which is at work here, says that for any number of sets, the probability of their union is equal to the sum of the probabilities of the individual sets, minus the sum of the probabilities of their pairwise intersections, plus the sum of the probabilities of their triple intersections, minus the sum of the probabilities of their quadruple intersections, and so forth. From these two equations, we can generalize the function for n number of sets:

$P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \le i_1 < i_2 < \cdots < i_k \le n} P(A_{i_1} \cap \cdots \cap A_{i_k}) \right)$

$\sum_{1 \le i_1 < i_2 < \cdots < i_k \le n}$ represents the summation over all distinct k-tuples of indices from 1 to n. $(-1)^{k-1}$ is the alternating sign that determines whether the intersection term should be added or subtracted in the calculation. $P(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k})$ represents the probability of the intersection of the k sets $Ai1, Ai2, ..., Aik$. To simplify, the probability of the union of n sets is obtained by summing up the probabilities of all possible intersections of the sets, with alternating signs for subtraction, depending on whether the intersection is of an even or odd number of sets.

This formula accounts for the overlapping probabilities between the sets using alternating addition and subtraction, ensuring that we do not double-count the intersections. This approach implies that the total outcome, being the root proposition, will be limited to a maximum of 100% likelihood of occurring.

5.1.9.2 Code Implementation

The graph traversal for this needs to happen in a recursive way, since it begins at the root node, and calls itself until hitting a leaf node before returning back. While returning, each node takes Bayesian values from its parents and calculates a new value for itself based on those. Eventually, the function calls return back to the root node, and every node is assigned a value.

The main function "applyBayesianReasoning", takes a graph (FDGGraph) as input and applies Bayesian reasoning to it. It starts by finding the root nodes in the graph, which are nodes with no outgoing edges (outDegree = 0). Then, for each root node, it calls the "setBayesianValue" function to calculate and set the Bayesian value for that node. Finally, it returns the modified graph with Bayesian values set to the "truthValue"

property.

Listing 5.11: applyBayesianReasoning Function

```
// utils.ts

/**
 * Applies Bayesian reasoning to the given graph.
 * @param graph - The graph to apply Bayesian reasoning to.
 * @returns The modified graph with Bayesian values set.
 */
export function applyBayesianReasoning(graph: FDGGraph):
    FDGGraph {
  const rootNodes = graph.nodes.filter((node) => node.outDegree
      === 0);
  for (const rootNode of rootNodes) {
    setBayesianValue(graph, rootNode);
  }
  return graph;
}
```

The setBayesianValue function (Listing 5.12) is a recursive function that sets the Bayesian value for a given node, from a calculation. If the node has no incoming edges (inDegree = 0), it returns the truth value of the node as its Bayesian value. Otherwise, it finds its direct parents using the "findDirectParents" function (see Listing 5.5) and recursively calls setBayesianValue for each parent to obtain their Bayesian values. Then, it calculates the Bayesian value for the current node based on the parent values using the "calculateBayesianValue" function. The Bayesian value is then both assigned to the truthValue property of the node, and returned by the function.

Listing 5.12: setBayesianValue Function

```
// utils.ts

/**
 * Sets the Bayesian value for the given node in the graph.
 * @param graph - The graph containing the node.
 * @param node - The node to set the Bayesian value for.
 * @returns The calculated Bayesian value for the node.
 */
function setBayesianValue(graph: FDGGraph, node: Node): number {
  if (node.inDegree === 0) return node.truthValue as number;
  const parents = findDirectParents(node, graph);

  const parentValues = parents.map((parent) =>
      setBayesianValue(graph, parent));
  const bayesianValue = calculateBayesianValue(parentValues) as
```

```
    number;
  node.truthValue = bayesianValue;
  return bayesianValue;
}
```

The calculateBayesianValue function (Listing 5.13) takes an array of parent values and calculates the Bayesian value based on those values. It converts the parent values from percentages to decimals for calculation purposes and then iterates over the parent values. For each value, it checks if the index is even or odd and performs addition or subtraction accordingly, using the “intersectAddCombinations” function. After the iteration, the Bayesian value is converted back from decimal to percentage and returned.

Listing 5.13: calculateBayesianValue Function

```
// utils.ts

/**
 * Calculates the Bayesian value based on the given parent
    values.
 * @param parentValues - An array of parent values.
 * @returns The calculated Bayesian value.
 */
function calculateBayesianValue(parentValues: number[]): number
   {
  // convert from percent to decimal
  const decimalParentValues = parentValues.map(
    (parentValue) => parentValue / 100
  );

  let bayesianValueDecimal = 0;

  for (let n = 1; n < parentValues.length + 1; n++) {
    if (n % 2 === 0) {
      bayesianValueDecimal -=
          intersectAddCombinations(decimalParentValues, n);
    }
    if (n % 2 === 1) {
      bayesianValueDecimal +=
          intersectAddCombinations(decimalParentValues, n);
    }
  }

  // convert back from decimal to percent
  return bayesianValueDecimal * 100;
}
```

The intersectAddCombinations function (Listing 5.14) calculates the sum of products for all combinations of size n from the given array. It calls the "nCombinations" function to obtain the combinations and then iterates over each combination. For each combination, it calculates the product of all the values and returns the sum of all the products.

Listing 5.14: intersectAddCombinations Function

```
/**
 * Calculates the sum of products for all combinations of size
    n from the given array.
 * @param array - The array to generate combinations from.
 * @param n - The size of the combinations.
 * @returns The sum of products for all combinations.
 */
function intersectAddCombinations(array: number[], n: number):
    number {
  const combinations = nCombinations(array, n);
  const multipliedCombinations: number[] = [];

  for (const combination of combinations) {
    const product = combination.reduce((acc, val) => acc * val,
        1);
    multipliedCombinations.push(product);
  }

  return multipliedCombinations.reduce((acc, val) => acc + val,
      0);
}
```

The nCombinations function (Listing 5.15) generates all combinations of size n from the given array using recursion and iteration. It starts by checking if n is 1, in which case it simply returns every element by itself as the combinations. Otherwise, it iterates over the array and recursively calls itself with a smaller n value on the remaining elements. The combinations are generated by combining the current element with each subcombination obtained from the recursive call. The total combinations are then returned in the end.

Listing 5.15: nCombinations Function

```
// utils.ts

/**
 * Generates all combinations of size n from the given array.
 * @param array - The array to generate combinations from.
 * @param n - The size of the combinations.
 * @returns An array of combinations.
 */
function nCombinations(array: number[], n: number): number[][] {
```

```
  if (n === 1) {
    return array.map((a) => [a]);
  }

  const combinations: number[][] = [];
  for (let i = 0; i <= array.length - n; i++) {
    const subCombinations = nCombinations(array.slice(i + 1), n
        - 1);
    for (const c of subCombinations) {
      combinations.push([array[i], ...c]);
    }
  }
  return combinations;
}
```

5.2 Ponder Integration

The next step in the development process, after iterating on the most essential features for a minimal viable product (MVP), was to integrate the system into the Ponder development branch in their repository, as of requirement FR2. This required some preparation before being able to merge the code bases together. Firstly, the graph type used by Disputas had to be merged with the types used by the graph drawing algorithm. Secondly, the graph drawing had to be converted to a "React component" in order to fit into Disputas' website. Thirdly, a series of changes related to making rendering itself work within Ponder were made. After several revisions, the code base was merged into the development branch in the Ponder repository.

5.2.1 Merging Graph Types

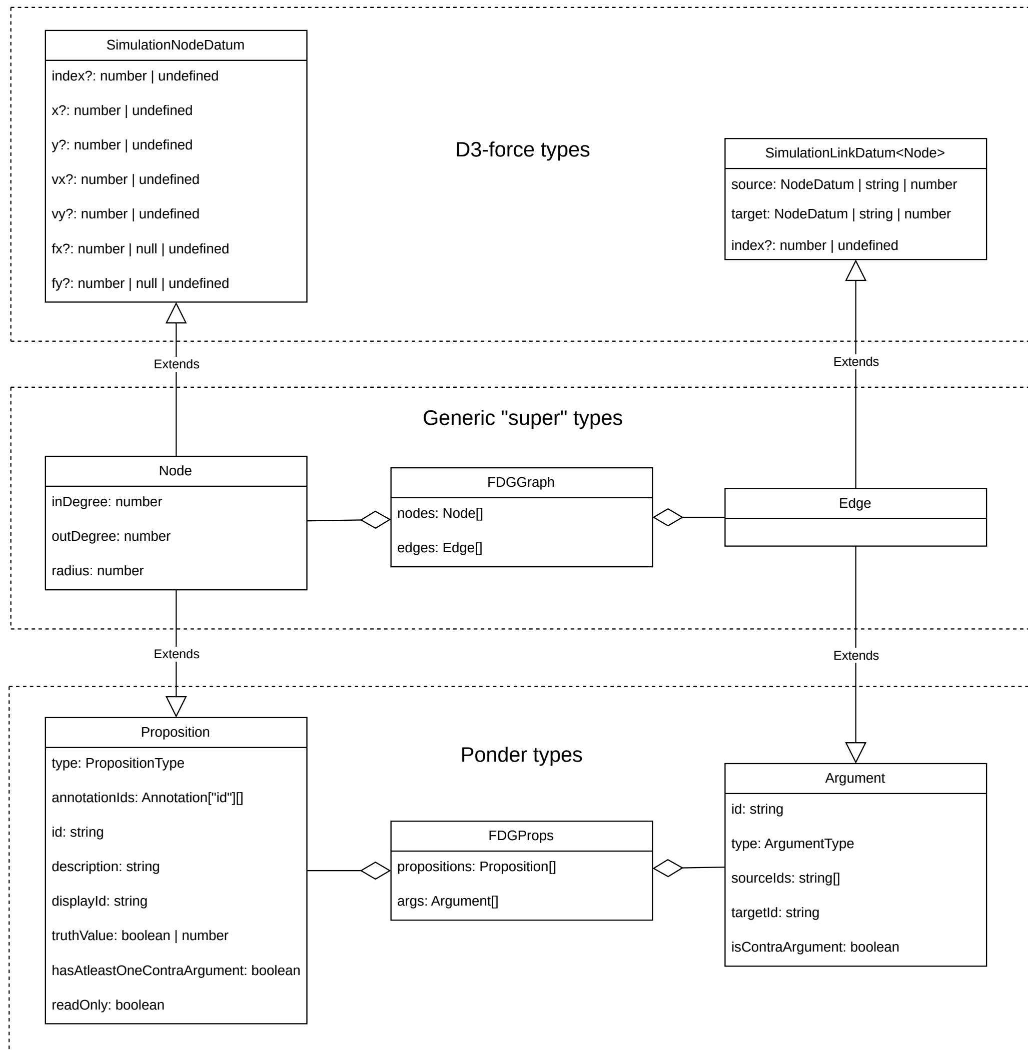
Since there was a gap between the types used for nodes and edges in the FDG drawing algorithm and the corresponding types used in Ponder, these types had to be merged together into generic "super" types. Ponder uses types called "Propositions" and "Arguments", which are collectively identified as "FDGProps". In the same way, the D3-force uses "SimulationNodeDatum" and "SimulationLinkDatum". A Proposition object is meant to be represented by a node, while an Argument object is meant to function as an edge between propositions. A Node object is a type of the "SimulationNodeDatum" from D3-force, while the Edge object is a type of the "SimulationLinkDatum", also from D3-force.

In order to solve this, the Node type was extended to inherit the properties from the Proposition and SimulationNodeDatum types, while similarly, the Edge type was extended to inherit the Argument and SimulationLinkDatum types. This allowed a graph to be built from Disputas' existing graph data within d3-force by converting propositions and

arguments (FDGProps) into nodes and edges (FDGGraph). The result of this can be seen in the interface diagram in Figure 5.8. This diagram defines three different sections. The “D3-force types” section describes the data types strictly required from D3-force, in order for the library to keep track of its nodes and edges during the simulation. The “Ponder types” section contains types that are used for the specific graph data set used by Ponder. These two sections are essentially merged into the generic types found in the “Generic ’super’ types” section.

Listing 5.16 shows the process of converting the FDGProps type (Propositions and Arguments) into the FDGGraph type (Nodes and Edges).

Figure 5.8: Interface Diagram



Listing 5.16: Merging Graph Types

```
// index.tsx

export default function FDG() {
  const { arguments: args, propositions } =
```

```
    useSelector(analysisSelector);

  // maps out proposition IDs and formats them into nodes with
     initialized values
  const propositionList = propositions.allIds.map(
    (id) => propositions.byId[id]
  );
  const nodes: Node[] = propositionList.map((proposition) => ({
    ...proposition,
    inDegree: 0,
    outDegree: 0,
    radius: 0,
  }));

  // maps out argument IDs and formats them into edges with
     initialized values
  const edges: Edge[] = args.allIds
    .map((id) => args.byId[id])
    .flatMap((arg) =>
      arg.sourceIds.map((source) => ({
        ...arg,
        source: source,
        target: arg.targetId,
      }))
    );
}
```

5.2.2 React Component

A new React component had to be made for the graph drawing (see Section 3.2). This was necessary to be able to render the graph properly inside of Ponder. The new component receives the graph data (FDGProps), and adds its own data properties to this ("inDegree", "outDegree", and "radius"), which is necessary for the pre-processing of the graph. The pre-processing consists of the functions "applyBayesianReasoning", "setNodeRadius", and "setNodeDegree", which are called in Listing 5.17. The FDG component is a function that produces the graph as normally, but instead of drawing the graph itself, it returns it as an "SVG" (Scalable Vector Graphics) element back to Ponder. This in turn allows Ponder to decide when and where to draw it instead. In addition to this, a React "useEffect" function was necessary (see Section 3.2). This function is used to synchronize the component with the external system (here, Ponder) [30]. This means that each time the component is rendered, the useEffect function is triggered, which then performs any necessary actions. In this case, the actions consist of updating the graph with the utility functions used for pre-processing the graph. Essentially, every time a change is made to the graph, it will apply those functions to it, before rendering it again

automatically. Listing 5.17 shows this updated use of the graph drawing.

Listing 5.17: React Component

```
// index.tsx

/**
 * Uses a set of arguments and propositions as nodes and edges
    to generate and draw an interactive force directed graph
    as an SVG element, using the D3-force library.
 *
 * @returns {React.SVGProps<SVGSVGElement>} An interactive
    force directed graph as an SVG.
 */
export default function FDG() {
  // (...)

  useEffect(() => {
    const updatedGraph = applyBayesianReasoning(
      setNodeRadius(setNodeDegree({ nodes, edges }))
    ); // calculates and sets new properties each time the
       graph is modified
    drawFDG(updatedGraph);
    const cleanup = () => {
      d3.select("#main␣>␣*").remove();
    };
    return () => cleanup();
  }, []);

  return (
    <svg
      viewBox="-500␣-500␣1000␣1000"
      width="100%"
      height="100%"
      id="main"
    ></svg>
  );
```

5.2.3 Rendering

Finally, a series of infrastructural changes were made to Ponder to make room for the FDG. A “graphType” property was added to a shared state within Ponder, which is managed by Redux (see Section 3.2). This was given the value of either “DAG” or “FDG”, which determines which view is to be shown. By using React’s conditional rendering technique, it is possible to switch seamlessly between rendering either of the

two graphs. The graphs are each represented by their own react component, which is then inserted into the Document Object Model (DOM) in order to be displayed. Essentially, the graphType value controls which graph is shown to the user at any moment in time.

The graph data used by Disputas was already being stored within Redux and used for the DAG, meaning that when rendering the FDG drawing, the necessary graph data could simply be fetched in the same way, from the same storage. To make the switching possible from a user's perspective, a button was added for changing the graph mode. This button switches the value of the graphType property between the two values, which in turn changes the graph view.

The final FDG view inside Ponder can be seen in Figure 6.2, and the DAG equivalent graph is shown in Figure 5.10.

Figure 5.9: Ponder GUI - FDG View

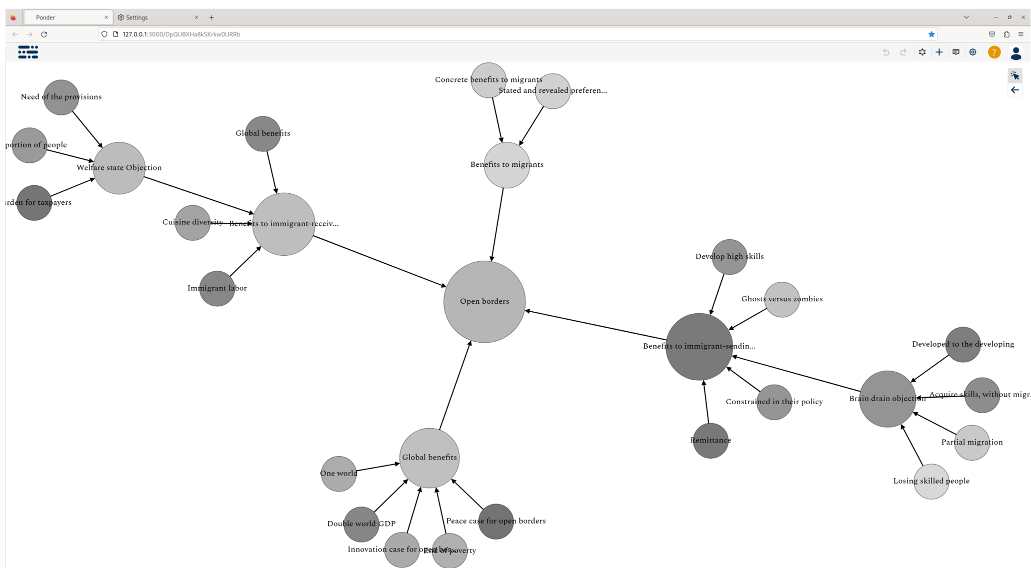
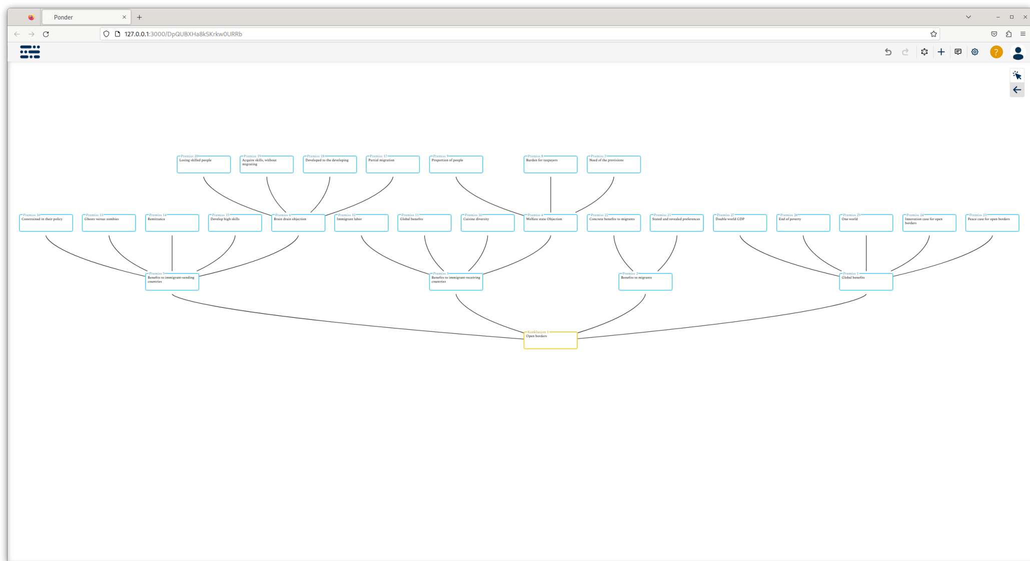


Figure 5.10: Ponder GUI - DAG View



Chapter 6

Evaluation

This chapter is an evaluation of the implementations made to the system. The system design is compared to the initially proposed design. Then, a description of the system interaction is given, as are multiple system tests related to the requirements. Finally, a review of the requirements is performed, where an overview of the fulfillment of each of them is given.

6.1 Initial Prototype Comparison

Once the final version of the prototype had been developed, it was interesting to compare it to the initially proposed design idea. The initial design from Chapter 1 can be seen in Figure 6.1. The final developed FDG drawing is comparatively shown in Figure 6.2.

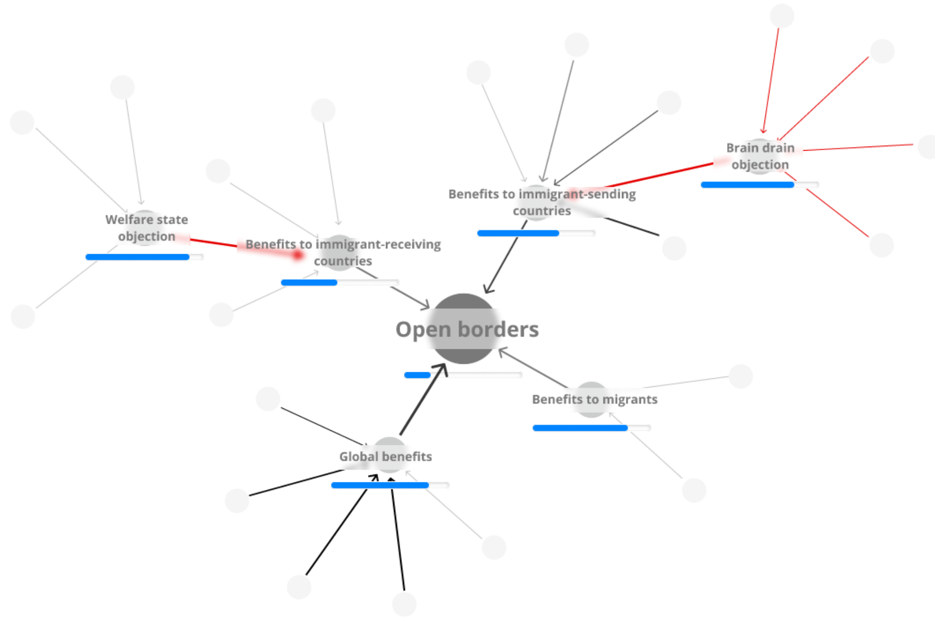
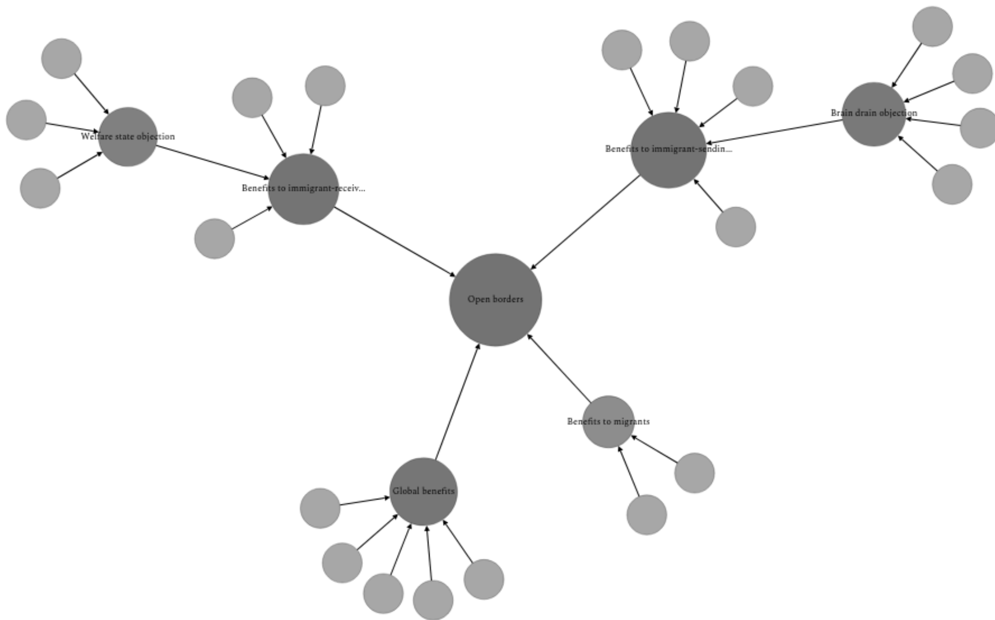


Figure 6.1: Initially Proposed Design Idea

Figure 6.2: FDG Drawing

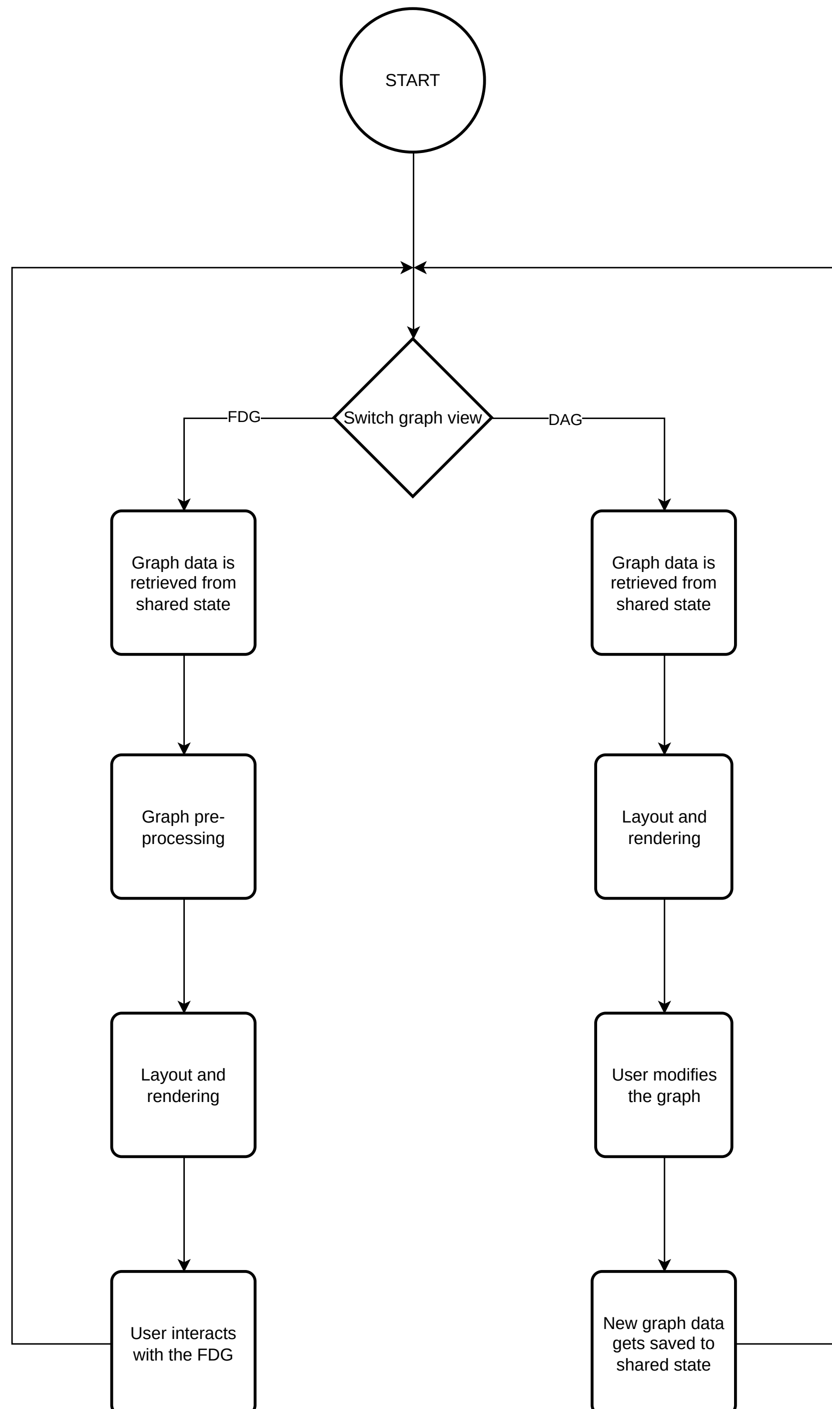


6.2 System Interaction

The flow chart in Figure 6.3 describes the system state depending on the user interaction. Since the system is currently a prototype only, a set of functionality therefore depends on the DAG layout for graph manipulation. To begin with, the user selects either "FDG" or "DAG". Graph data is then retrieved from a shared state. For the

FDG, the pre-processing methods are performed on the graph before layout and rendering. These include “applyBayesianReasoning”, “setNodeRadius”, and “setNodeDegree”. Graph modification is then only available from the FDG view, meaning that the user will have to generate the desired graph there before switching the view.

Figure 6.3: System Interaction Diagram



6.3 Cross-Browser Compatibility Test

A browser test was necessary in order to evaluate requirement NFR2 from Section 4.2, which states the following: "The system should be compatible with different web browsers, ensuring compatibility across major browsers. The four most popular web browsers should be supported.". To set up the test, a custom graph was created beforehand based on the graph structure used in the initial prototype. A desktop monitor with a resolution of 1920 by 1080 pixels was used. A browser list was determined based on the most popular browsers. According to StatCounter, the following were the four most popular browsers: Google Chrome, Apple Safari, Microsoft Edge, and Mozilla Firefox [36].

Since the Safari browser is exclusively available on Apple operating systems, a replacement for this had to be made. "Epiphany" is built on the "WebkitGTK" browser engine, which is a port of the WebKit browser engine. These browsers are fundamentally very similar to Safari: they all share the same WebKit browser engine [10]. However, the Epiphany browser was unfortunately not able to function properly on the test system. The Safari browser therefore had to be omitted completely from this list. The resulting browser list for the test was left as follows:

- Google Chrome (version 113.0.5672.92)
- Microsoft Edge (version 113.0.1774.50)
- Mozilla Firefox (version 113.0.1)

All the inspection points in the table were tested and verified for each of the browsers listed. All the functionality checks were passed for all the browsers, as shown in Table 6.1.

Table 6.1: Browser Test Results

Inspection Point	**Google Chrome**	**Microsoft Edge**	**Mozilla Firefox**
The graph appears as it should and is not missing any components.	Passed	Passed	Passed
The nodes can be dragged around.	Passed	Passed	Passed
The zooming functionality works.	Passed	Passed	Passed
The text changes visibility when zooming.	Passed	Passed	Passed
The panning functionality works.	Passed	Passed	Passed
Switching the graph view works.	Passed	Passed	Passed
The node tooltip displays correctly.	Passed	Passed	Passed
The runtime seems to not be within human perception of slowness.	Passed	Passed	Passed

6.4 System Performance Test

A final system performance test was made in order to evaluate whether the performance of the system still holds since initially starting out with the plain graph data, after implementing new features, and integrating the system into Ponder. As there currently is no easy way of creating test data within Ponder, a graph was manually created as an approximation to the previously synthetically generated graph with 100 nodes, and about twice the number of edges. The test is comparable to the previous one found in Section 4.3.3, and this test similarly looks at both "initial graph layout load times", and "graph rendering frame rate". It should be noted that the web browser used for the test was Google Chrome (version 113.0.5672.92). The results from the initial graph layout load times are listed in Table 6.2, while the results from the graph rendering frame rate are shown in Table 6.3.

Table 6.2: Initial Graph Layout Load Times

Nodes	**Average (ms)**	**St.Dev. (ms)**
100	3.60	8.28e-1

When looking at the load time as compared to the previous result, we can see that it has increased from 1.78 to 3.60 milliseconds on average. As expected, the pre-processing of the graph might have contributed to this additional time delay. This is about a doubling of the previous time, but is still well within the requirement, as previously discussed

Table 6.3: Graph Rendering Frame Rate

Nodes	Average (FPS)	St.Dev. (FPS)
100	131	2.55

in the analysis for the initial test in Section 4.3.3.3. The standard deviation increased slightly from 0.6 to 0.8 ms, which does not really make a difference.

On the other hand, the rendering frame rate has also increased slightly, from 129 to 131. This seems to be within the margin of error, and might be related to the sample size of only 10. As a conclusion, we can say that the system still performs well within the acceptance criteria for the requirements, as discussed in the previous performance test analysis. The standard deviation was slightly reduced from 3.4 to 2.6 FPS, which does not make a significant difference either.

6.5 Requirement Review

This review looks at how the system compares to the requirements defined initially. See Section 4.2 for the full requirement description.

6.5.1 Functional Requirements

FR1: The system should produce a force-directed graph drawing for representing non-layered graphs.
Since the system is implemented using D3-force, force-directed graph drawing is already provided "out of the box".

FR2: The system should be integrated into Ponder.
As explained in Section 5.2, the system was eventually integrated with Ponder, and the code base was therefore merged into its development repository branch. The system coexists with the already implemented DAG solution. This was made possible by allowing the user to switch between the two, as described in Section 5.2.3. It also describes how the rendering is performed by Ponder, as per the requirement specification.

FR3: The system should represent propositions using nodes, and arguments using edges.
The system combines nodes with propositions, and edges with arguments. The implementation of this is described in Section 5.2.1.

FR4: The nodes should display text content.
As described in Section 5.1.8, each node has some visible text, according to the proposition's text description. The full description can also be shown to the user by hovering over the node with the mouse.

FR5: The system should set each node's size depending on its "connectedness" in the graph.
Section 5.1.5 describes how the PageRank algorithm was implemented in order to determine each node's "connectedness". This was used as a basis for setting each node's size.

FR6: The system should be easy to develop further.
Since the system is built on top of D3-force, it inherits all its benefits. D3-force is easy to develop further with new feature implementations, as it provides a modular and extensible architecture, allowing for the ability to easily add new features or customize existing ones. Custom forces can be created, or the behavior of existing forces can be modified to suit future requirements. This flexibility makes it convenient to extend the system and develop new features. It is also flexible, customizable, and maintainable by offering a wide range of forces, such as charge forces, centering forces, collision forces, and link forces, among others. These forces can be combined, tweaked, or removed as needed to achieve the desired layout behavior. Additionally, D3-force allows the developer to control various parameters of the forces, such as strength, distance, and velocity, making it highly customizable. In addition, its popularity means that there is a lot of documentation and support surrounding it, making maintenance easier as well.

FR7: The system should be implemented using TypeScript.
The system is fully implemented using TypeScript, and the full code listing can be found in Appendix A.

FR8: The user should be able to modify the graph.
Users are currently not able to modify the graph through the FDG drawing interface section of Ponder. However, as described in Section 6.2, it is possible to work around this to modify the graph data by switching to the existing DAG view. Since these two components use the same graph data, the user can seamlessly switch back to the FDG view after modifying the graph this way, to see and interact with the modified graph.

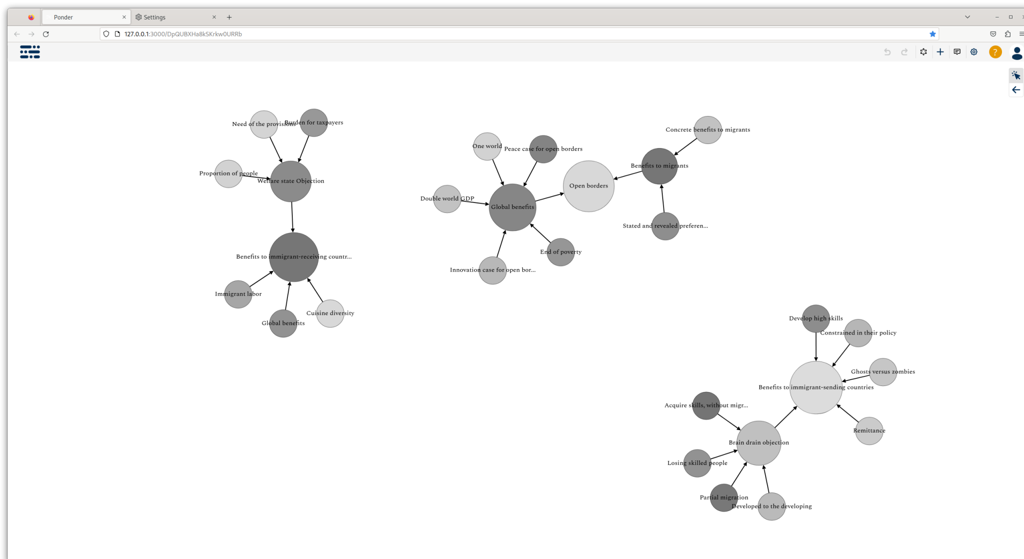
FR9: Edges should have directionality.
Edge directionality was implemented as described in Section 5.1.2. The implementation adds an arrow head pointing from the source node to the target node.

FR10: The graph drawing should work with at least 80 nodes.
The performance of the system was evaluated in Section 6.4. This describes the system performing well within adequacy in a scenario with 100 nodes drawn at once.

FR11: The user should be able to drag and drop nodes around.
Section 5.1.3 explains how node dragging is implemented in the system. This allows the user to easily "drag-and-drop" nodes around freely.

FR12: The system should be able to handle multiple graphs within the same view at once.
Multiple graphs can be shown in the same view at once. A demonstration of this can be seen in Figure 6.4, where a recreation of the graph used in the initial design (see Figure 1.1) was split into three disconnected graphs in the same view.

Figure 6.4: Multiple Graphs Simultaneously



FR13: The system should support multi-person collaboration within the same graph.
The system currently does not support any form of multi-person collaboration.

FR14: The user should be able to expand nodes, or click to open a modal for them.
The system currently does not provide functionality for the user to either expand or open a modal for the nodes. However, it is possible to see details such as the text description and credence value by hovering over each node, as described in Section 5.1.8.

FR15: The nodes should display a credence value.
The user can observe the credence value of a node in two ways, as described in Section 5.1.7. One way is through the color tone of the node, and another is through reading the specific value through a "tooltip" while hovering the mouse over the node. However, it is currently not possible for the user to reposition this tooltip.

FR16: The system should support zooming and panning inside the graph view.
The implementation and functionality of both zooming and panning is described in Section 5.1.4. In addition, varying text visibility according to the zoom level has been implemented, and is described in Section 5.1.8.

FR17: The system should infer new credence values based on existing ones in the graph.
The system includes a partial implementation for inferring new truth values. The truth values from leaf nodes are used to calculate new values for all non-leaf nodes, as per the requirement specification. However, the specific implementation is limited to a single specific model, and the feature therefore needs to be extended further with more and/or better models. The implementation is described in Section 5.1.9. In addition, the argu-

ments, represented by edges, should also weigh in on the calculation, instead of only the propositions.

6.5.2 Non-Functional Requirements

NFR1: The system should perform with adequate speed in regards to human perception of slowness.
The speed of the system has been tested and analyzed in Section 6.4, in terms of the initial load time, as well as the rendering frame rate. This was measured and evaluated to be of adequate speed in terms of human perception, with more than 80 nodes. The evaluation took the absolute runtime into consideration, as per the requirement specification.

NFR2: The system should be cross-browser compatible.
The system was tested for three out of four of the most popular browsers in Section 6.3. Out of these, all of them passed all the functionality tests. However, the Apple Safari browser still needs to be tested for compatibility.

6.5.3 Requirement Evaluation Overview

Table 6.4 gives an overview over the degree of fulfillment. Each requirement is color coded in either red for “not fulfilled”, orange for “partially fulfilled”, and green for “fulfilled”. All the primary requirements were fulfilled, except for FR8, which was partially fulfilled. Out of the secondary requirements, four were fulfilled, three were partially fulfilled, and one was not fulfilled.

ID	Priority	Description
		Functional Requirements
FR1	Primary	The system should produce a force-directed graph drawing for representing non-layered graphs.
FR2	Primary	The system should be integrated into Ponder.
FR3	Primary	The system should represent propositions using nodes, and arguments using edges.
FR4	Primary	The nodes should display a text description.
FR5	Primary	The system should set each node's size depending on its "connectedness" in the graph.
FR6	Primary	The system should be easy to develop further.
FR7	Primary	The system should be implemented using TypeScript.
FR8	Primary	The user should be able to modify the graph.
FR9	Primary	Edges should have directionality.
FR10	Primary	The graph drawing should work with at least 80 nodes.
FR11	Secondary	The user should be able to drag and drop nodes around.
FR12	Secondary	The system should be able to handle multiple graphs within the same view at once.
FR13	Secondary	The system should support multi-person collaboration within the same graph.
FR14	Secondary	The user should be able to expand nodes, or click to open a modal for them.
FR15	Secondary	The nodes should display a credence value.
FR16	Secondary	The system should support zooming and panning inside the graph view.
FR17	Secondary	The system should infer new credence values based on existing ones in the graph.
		Non-Functional Requirements
NFR1	Primary	The system should perform with adequate speed in regards to human perception of slowness.
NFR2	Secondary	The system should be cross-browser compatible.

Table 6.4: Requirement Evaluation

Chapter 7

Discussion and Conclusion

The chapter discusses the advantages and disadvantages of the current state of the artifact produced. Further work is then derived from this, followed by a conclusion to the thesis by looking back at the original research question.

7.1 Discussion

7.1.1 Advantages

After exploring the possibilities within the D3-force API, it seems clear to the author that this has been a good option as it provides a wide variety of customization options (see Section 3.3). The features implemented on top of this, seem to work well and fill the use case. This potential gives it room to grow into a more feature-complete product. All the primary requirements were fulfilled, except for FR8, which was partially fulfilled. Overall, the system works well considering it is only a prototype.

7.1.2 Limitations

The probability calculation may need more work, in order to be relevant and fully functional. In its current state, probabilities are calculated based on dice-throwing events, which may or may not be relevant. Other probability models need to be explored, and potentially be implemented in a combination, allowing for a variety of settings for different probabilistic situations. In addition, the arguments, represented as edges, should weigh in on the calculation as well, in order to give further nuance to the probability of an outcome.

When it comes to browser support, the Apple Safari web browser still needs to be tested to make sure of compatibility. According to Statcounter, this accounts for 20% of the desktop browser market share [36].

The FDG view does not contain its own graph manipulation functionality, since it relies on the DAG view as a current workaround.

7.1.3 Further Work

When it comes to further work, the main goal is to iterate on the prototype until the system is feature-complete enough to potentially turn into a commercially available product. To begin with, the specific points mentioned in Section 7.1.2 should be applied.

Since starting the project, newer repositories have appeared, which could be worth looking at. "vasturiano/force-graph" is also based on D3-force, and has become very popular on GitHub [42].

Regarding the limitation on graph manipulation, this could be implemented using something like the "Build Your Own Graph" D3 repository available on Observable [4]. It provides a simple implementation for adding nodes and edges. Another example that could contribute to this by providing a complex graph editor with collapsible graph building, would be the "adamfeuer/d3js-tree-editor" repository available on GitHub [1].

For an even better graph visualization, tuning of the multitude of forces and force combinations could be beneficial. This can be experimented with by simply tweaking different values. Details about the most central forces are described in Section 3.3, but this could also be explored further by using the D3-force API.

7.2 Conclusion

When looking back at what the project started out to achieve, we can try to answer the original research question found at the end of Section 1.1. This states: "How can a force-directed graph drawing algorithm be implemented for a Bayesian network while maintaining visual clarity and user interaction?". The artifact produced is one of many possible answers to the research question about how this can be implemented. Through all the implementations described in Chapter 5, the artifact has piece by piece gained the necessary features to satisfy most of the requirements for such a product. The need for both visualization and calculation for collaborative forecasting of different events therefore seems to be satisfiable with such a product. However, this is still only a prototype, which needs to be iterated upon until the system is feature-complete enough to potentially turn into a commercially available product.

Bibliography

[1] *adamfeuer/d3js-tree-editor: d3js-tree-editor.* (Accessed on 06/01/2023). URL: https://github.com/adamfeuer/d3js-tree-editor.

[2] *anvaka/ngraph.forcelayout: Force directed graph layout.* (Accessed on 05/15/2023). URL: https://github.com/anvaka/ngraph.forcelayout.

[3] *Benjoyo/ForceDirectedPlacement: Java implementation of Fruchterman and Reingold's graph layout algorithm using force-directed placement.* (Accessed on 05/15/2023). URL: https://github.com/Benjoyo/ForceDirectedPlacement.

[4] Mike Bostock. *Build Your Own Graph! / D3 — Observable.* (Accessed on 05/14/2023). Jan. 2022. URL: https://observablehq.com/@d3/build-your-own-graph?collection=@d3/d3-force.

[5] Mike Bostock. *Force-Directed Graph / D3 — Observable.* (Accessed on 05/14/2023). May 2022. URL: https://observablehq.com/@d3/force-directed-graph.

[6] Mike Bostock. *Force-Directed Tree / D3 — Observable.* (Accessed on 06/03/2023). Aug. 2020. URL: https://observablehq.com/@d3/force-directed-tree?collection=@d3/d3-force.

[7] *chaangliu/ForceDirectedLayout: A simple implementation of force directed layout method in Java and JavaScript.* (Accessed on 05/15/2023). URL: https://github.com/chaangliu/ForceDirectedLayout.

[8] Peter Cook. *D3 Force layout.* (Accessed on 05/20/2023). 2023. URL: https://www.d3indepth.com/force-layout/.

[9] Peter Cook. *D3 Zoom and Pan.* (Accessed on 05/14/2023). 2023. URL: https://www.d3indepth.com/zoom-and-pan/.

[10] *Cross Browser Testing with Safari Alternative: WebKitWebDriver-Epiphany Docker Container Image - codeluge.* (Accessed on 05/22/2023). URL: https://www.codeluge.com/post/cross-browser-testing-with-webkitwebdriver-epiphany-container-image-and-webdriverio/.

[11] *cytoscape/cytoscape.js-euler: Euler is a fast, high-quality force-directed (physics simulation) layout for Cytoscape.js.* (Accessed on 05/15/2023). URL: https://github.com/cytoscape/cytoscape.js-euler.

[12] *D3 — Observable.* (Accessed on 05/15/2023). URL: https://observablehq.com/@d3?tab=public.

[13] *d3-force / D3 — Observable.* (Accessed on 05/15/2023). URL: https://observablehq.com/collection/@d3/d3-force.

[14] *d3/API.md at main · d3/d3.* (Accessed on 05/18/2023). URL: https://github.com/d3/d3/blob/main/API.md.

[15] *d3/d3-force: Force-directed graph layout using velocity Verlet integration.* (Accessed on 05/13/2023). URL: https://github.com/d3/d3-force.

[16] *d3/d3-force: Force-directed graph layout using velocity Verlet integration.* (Accessed on 05/14/2023). URL: https://github.com/d3/d3-force#api-reference.

[17] *dagrejs/dagre: [DEPRECATED] - Directed graph layout for JavaScript.* (Accessed on 05/13/2023). URL: https://github.com/dagrejs/dagre.

[18] Andrew Disney. *Force-directed graph layouts explained - Cambridge Intelligence.* (Accessed on 05/20/2023). Feb. 2021. URL: https://cambridge-intelligence.com/keylines-faq-force-directed-layouts/.

[19] DisputasAS. *Disputas — disputas.no.* [Accessed 13-May-2023]. URL: https://disputas.no/#about.

[20] *H4kor/graph-force: Python library for embedding large graphs in 2D space, using force-directed layouts.* (Accessed on 05/15/2023). URL: https://github.com/H4kor/graph-force.

[21] Alan Hevner et al. 'Design Science in Information Systems Research'. In: *Management Information Systems Quarterly* 28 (Mar. 2004).

[22] *hijiangtao/Force-Directed-Layout: A Force Directed Layout work wrote in ES2015.* (Accessed on 05/15/2023). URL: https://github.com/hijiangtao/Force-Directed-Layout.

[23] Dennis Hotson. *dhotson/springy: A force directed graph layout algorithm in JavaScript.* (Accessed on 05/13/2023). URL: https://github.com/dhotson/springy/.

[24] Dennis Hotson. *getspringy.com/demo.html.* (Accessed on 05/31/2023). URL: http://getspringy.com/demo.html.

[25] Dennis Hotson. *springy/springy.js at master · dhotson/springy · GitHub.* (Accessed on 05/20/2023). URL: https://github.com/dhotson/springy/blob/master/springy.js.

[26] Lucas Jellema. *Introduction to D3 Force for Simulation and Animation - AMIS, Data Driven Blog - Oracle & Microsoft Azure.* (Accessed on 05/21/2023). May 2021. URL: https://technology.amis.nl/frontend/introduction-to-d3-force-for-simulation-and-animation/.

[27] *justinormont/Fastest-Force-Directed-Graph: Implements a Force Directed Graph Layout Engine backed by an Oct Tree (8-way spatial partitioning).* (Accessed on 05/15/2023). URL: https://github.com/justinormont/Fastest-Force-Directed-Graph.

[28] Paal Fredrik Skjørten Kvarberg. *Two directions for research on forecasting and decision making - EA Forum.* (Accessed on 05/31/2023). Mar. 2023. URL: https://forum.effectivealtruism.org/posts/dsG5SYjhPqnxhystM/two-directions-for-research-on-forecasting-and-decision.

[29] Jennifer Larson. *Human Eye FPS: How Much Can We See and Process Visually?* (Accessed on 05/25/2023). Oct. 2020. URL: https://www.healthline.com/health/human-eye-fps.

[30] Meta. *useEffect – React*. (Accessed on 05/21/2023). 2023. URL: https://react.dev/reference/react/useEffect.

[31] Nico. *D3-Force Directed Graph Layout Optimization in NebulaGraph Studio*. (Accessed on 05/21/2023). Apr. 2020. URL: https://www.nebula-graph.io/posts/d3-force-layout-optimization.

[32] *olvb/nodesoup: Force-directed graph layout with Fruchterman-Reingold*. (Accessed on 05/15/2023). URL: https://github.com/olvb/nodesoup.

[33] Olivier Pourret. *Bayesian Networks: A Practical Guide to Applications*. 2008.

[34] princiya. *D3.js force layout – P's Blog*. (Accessed on 05/21/2023). Nov. 2017. URL: https://princiya777.wordpress.com/2017/11/18/d3-js-force-layout/.

[35] Section. *How Page Load Time Affects Bounce Rate and Page Views — Section*. (Accessed on 05/25/2023). 2022. URL: https://www.section.io/blog/page-load-time-bounce-rate/.

[36] StatCounter. *Statcounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share*. (Accessed on 05/22/2023). May 2023. URL: https://gs.statcounter.com/.

[37] StudyWell. *Mutually Exclusive & Independent Events - StudyWell*. (Accessed on 06/05/2023). 2023. URL: https://studywell.com/probability/mutually-exclusive-independent/.

[38] Courtney Taylor. *Probability of Union of 3 or More Sets*. (Accessed on 06/07/2023). Aug. 2019. URL: https://www.thoughtco.com/probability-union-of-three-sets-more-3126263.

[39] Philip E. Tetlock. *Superforecasting: The Art and Science of Prediction*. McClelland & Stewart, 2016.

[40] Tutorialspoint. *ReactJS - Overview*. (Accessed on 05/22/2023). URL: https://www.tutorialspoint.com/reactjs/reactjs_overview.htm.

[41] *twosixlabs/d3-force-reuse: Faster force-directed graph layouts by reusing force approximations*. (Accessed on 05/15/2023). URL: https://github.com/twosixlabs/d3-force-reuse.

[42] *vasturiano/force-graph: Force-directed graph rendered on HTML5 canvas*. (Accessed on 05/30/2023). URL: https://github.com/vasturiano/force-graph.

[43] Wikipedia. *Force-directed graph drawing - Wikipedia*. (Accessed on 05/20/2023). Nov. 2022. URL: https://en.wikipedia.org/wiki/Force-directed_graph_drawing.

[44] Wikipedia. *Node.js - Wikipedia*. (Accessed on 06/12/2023). May 2023. URL: https://en.wikipedia.org/wiki/Node.js.

[45] Wikipedia. *PageRank - Wikipedia*. (Accessed on 05/31/2023). June 2023. URL: https://en.wikipedia.org/wiki/PageRank.

[46] Wikipedia. *React (software) - Wikipedia*. (Accessed on 05/22/2023). June 2023. URL: https://en.wikipedia.org/wiki/React_(software).

[47] yWorks. *Force-Directed Graph Layout*. (Accessed on 05/20/2023). 2023. URL: https://www.yworks.com/pages/force-directed-graph-layout.

Appendix A

Code

A.1 D3-Force Performance Test

Listing A.1: d3-force-performance-test.js

```
// Variables to measure frames per second (FPS)
let prevTime = performance.now();
let frames = 0;

// Function to measure FPS
const measureFPS = () => {
  const time = performance.now();
  frames++;
  if (time > prevTime + 1000) {
    const fps = (frames * 1000) / (time - prevTime);
    prevTime = time;
    frames = 0;

    console.info(`FPS: ${fps}`);
  }
};

console.time("initial␣layout␣loading␣time"); // Start timing
    the initial layout loading process

// Set a width and height for the SVG
const width = 1000,
  height = 1000;

let edges = [];
```

```
const numberOfNodes = 10;

// Create a graph with a certain number of new nodes and 2
   times the number of edges
for (let i = 0; i < numberOfNodes; i++) {
  let newNode = (Math.random() + 1).toString(36).substring(7);
  let otherNode1 = edges[Math.floor(Math.random() *
     edges.length)].source;
  let otherNode2 = edges[Math.floor(Math.random() *
     edges.length)].source;
  edges.push({ source: otherNode1, target: newNode });
  edges.push({ source: otherNode2, target: newNode });
}

let nodes = [];

for (let i = 0; i < 100; i++) {
  let newNode = (Math.random() + 1).toString(36).substring(7);
  nodes.push({ name: newNode });
}

let svg = d3
  .select("body")
  .append("svg")
  .attr("width", width)
  .attr("height", height);

let force = d3.layout
  .force()
  .size([width, height])
  .nodes(d3.values(nodes))
  .links(edges)
  .on("tick", tick)
  .linkDistance(1000)
  .start();

let edge = svg
  .selectAll(".link")
  .data(edges)
  .enter()
  .append("line")
  .attr("class", "link");

let node = svg
```

```
  .selectAll(".node")
  .data(force.nodes())
  .enter()
  .append("circle")
  .attr("class", "node")
  .attr("r", width * 0.03)
  .on("click", (d) => console.log(d));

console.timeEnd("initial␣layout␣loading␣time"); // Stop timing
    the initial layout loading process and print the elapsed
    time

// Function to update the positions of nodes and edges on each
    tick
const tick = () => {
  measureFPS();

  node
    .attr("cx", (d) => d.x)
    .attr("cy", (d) => d.y)
    .call(force.drag);

  edge
    .attr("x1", (d) => d.source.x)
    .attr("y1", (d) => d.source.y)
    .attr("x2", (d) => d.target.x)
    .attr("y2", (d) => d.target.y);
};
```

A.2 Springy Performance Test

Listing A.2: springy-performance-test.js

```
// Variables to measure frames per second (FPS)
let prevTime = performance.now();
let frames = 0;

// Function to measure FPS
const measureFPS = () => {
  const time = performance.now();
  frames++;
  if (time > prevTime + 1000) {
    const fps = (frames * 1000) / (time - prevTime);
    prevTime = time;
    frames = 0;
```

```
    console.info("FPS:", fps);
  }
};

console.time("initial␣layout␣loading␣time"); // Start timing
    the initial layout loading process

const graph = new Springy.Graph();
const nodes = [graph.newNode({ label: "initialNode" })];
const edgeColor = "#EB6841";
const numberOfNodes = 10;

// Generate nodes and edges for the graph. Each node is
    connected to 2 edges.
for (let i = 0; i < numberOfNodes; i++) {
  // nodes
  const name = (Math.random() + 1).toString(36).substring(7);
  const newNode = graph.newNode({ label: name });
  nodes.push(newNode);

  // edges
  const otherNode1 = nodes[Math.floor(Math.random() *
      nodes.length)];
  graph.newEdge(otherNode1, newNode, { color: edgeColor });

  const otherNode2 = nodes[Math.floor(Math.random() *
      nodes.length)];
  graph.newEdge(otherNode2, newNode, { color: edgeColor });
}

jQuery(function () {
  measureFPS();

  // Create Springy instance and attach it to the HTML element
      with ID "springydemo"
  const springy = jQuery("#springydemo").springy({
    graph: graph,
    nodeSelected: (node) =>
      console.log("Node␣selected:␣" +
          JSON.stringify(node.data)),
  });
  window.springy = springy;
});
```

```
console.timeEnd("initial␣layout␣loading␣time"); // Stop timing
    the initial layout loading process and print the elapsed
    time
```

A.3 Types

Listing A.3: types.ts

```
import { SimulationLinkDatum, SimulationNodeDatum } from "d3";
import { Proposition, Argument } from "@disputas/types";

/**
 * Represents a node in the FDG.
 */
export interface Node extends SimulationNodeDatum, Proposition {
  inDegree: number;
  outDegree: number;
  radius: number;
}

/**
 * Represents an edge in the FDG.
 */
export interface Edge extends SimulationLinkDatum<Node>,
    Argument {}

/**
 * Represents a FDG.
 */
export interface FDGGraph {
  nodes: Node[];
  edges: Edge[];
}

/**
 * Represents the properties for configuring an FDG.
 */
export interface FDGProps {
  propositions: Proposition[];
  args: Argument[];
}
```

A.4 Index

Listing A.4: index.ts

```
import React, { useEffect } from "react";
import { setNodeDegree, setNodeRadius, applyBayesianReasoning }
    from "./utils";
import { useSelector } from "react-redux";
import { analysisSelector } from "selectors";
import { Edge, Node } from "./types";
import drawFDG from "./drawFDG";
import * as d3 from "d3";

/**
 * Uses a set of arguments and propositions as nodes and edges
     to generate and draw an interactive force directed graph
     as an SVG element, using the D3-force library.
 *
 * @export
 * @returns {React.SVGProps<SVGSVGElement>} An interactive
     force directed graph as an SVG.
 */
export default function FDG() {
  const { arguments: args, propositions } =
      useSelector(analysisSelector);

  // maps out proposition IDs and formats them into nodes with
      initialized values
  const propositionList = propositions.allIds.map(
    (id) => propositions.byId[id]
  );
  const nodes: Node[] = propositionList.map((p) => ({
    ...p,
    inDegree: 0,
    outDegree: 0,
    radius: 0,
  }));

  // maps out argument IDs and formats them into edges with
      initialized values
  const edges: Edge[] = args.allIds
    .map((id) => args.byId[id])
    .flatMap((arg) =>
      arg.sourceIds.map((source) => ({
        ...arg,
        source: source,
        target: arg.targetId,
```

```
      }))
    );

  useEffect(() => {
    const updatedGraph = applyBayesianReasoning(
      setNodeRadius(setNodeDegree({ nodes, edges }))
    ); // calculates and sets new properties each time the
        graph is modified
    drawFDG(updatedGraph);
    const cleanup = () => {
      d3.select("#main > *").remove();
    };
    return () => cleanup();
  }, []);

  return (
    <svg
      viewBox="-500 -500 1000 1000"
      width="100%"
      height="100%"
      id="main"
    ></svg>
  );
}
```

A.5 Utility Functions

Listing A.5: utils.ts

```
import { Node, FDGGraph } from "./types";

/**
 * Sets the properties 'inDegree' and 'outDegree' for every
     node in the graph.
 *
 * @export
 * @param {FDGGraph} { nodes, edges } The graph to be modified.
 * @returns A modified graph with updated properties.
 */
export function setNodeDegree({ nodes, edges }: FDGGraph):
    FDGGraph {
  const modifiedNodes = nodes.map((node) => ({
    ...node,
    inDegree: edges.filter((edge) => edge.target ===
        node.id).length,
```

```
    outDegree: edges.filter((edge) => edge.source ===
        node.id).length,
  }));
  return { nodes: modifiedNodes, edges };
}

/**
 * Finds all directly connected (first degree) parent nodes.
 *
 * @param {Node} node The child node to be assessed.
 * @param {FDGGraph} { nodes, edges } The graph containing the
    node.
 * @returns {Node[]} The directly connected parents.
 */
function findDirectParents(node: Node, { nodes, edges }:
    FDGGraph): Node[] {
  const incomingEdges = edges.filter((edge) => edge.target ===
      node.id);
  return nodes.filter((_node) =>
    incomingEdges.find((edge) => _node.id === edge.source)
  );
}

/**
 * Calculates and sets the node radius property for all the
    nodes in the graph, using the PageRank algorithm.
 *
 * @export
 * @param {FDGGraph} { nodes, edges } The graph to be modified.
 * @returns {FDGGraph} A modified graph with radius values
    updated.
 */
export function setNodeRadius({ nodes, edges }: FDGGraph):
    FDGGraph {
  const initialValue = 10; // the initial PageRank value for
      each node, used to iterate into a final radius value
  const pageRankIterations = 100; // the number of iterations
      of the PageRank algorithm - set to a high value to make
      sure it converges

  // set initial radius value before iterating
  let modifiedNodes: Node[] = nodes.map((node) => ({
    ...node,
    radius: initialValue,
```

```
  }));

  // main radius calculation using the PageRank algorithm
  for (let i = 0; i < pageRankIterations; i++) {
    for (const node of modifiedNodes) {
      const parentNodes = findDirectParents(node, {
        nodes: modifiedNodes,
        edges,
      });
      const parentRadiusValues = parentNodes.map(
        (parentNode) => parentNode.radius /
            Math.max(parentNode.outDegree, 1)
      );
      if (parentRadiusValues.length !== 0) {
        node.radius = parentRadiusValues.reduce(
          (accumulator, value) => accumulator + value,
          0
        );
      }
    }
  }

  // normalize the radius values and adjust them for
      visualization purposes
  modifiedNodes = modifiedNodes.map((node) => ({
    ...node,
    radius: Math.log(node.radius) * 15,
  }));
  return { nodes: modifiedNodes, edges };
}

/**
 * Applies Bayesian reasoning to the given graph.
 * @param graph - The graph to apply Bayesian reasoning to.
 * @returns The modified graph with Bayesian values set.
 */
export function applyBayesianReasoning(graph: FDGGraph):
    FDGGraph {
  const rootNodes = graph.nodes.filter((node) => node.outDegree
      === 0);
  for (const rootNode of rootNodes) {
    setBayesianValue(graph, rootNode);
  }
  return graph;
```

```
}

/**
 * Sets the Bayesian value for the given node in the graph.
 * @param graph - The graph containing the node.
 * @param node - The node to set the Bayesian value for.
 * @returns The calculated Bayesian value for the node.
 */
function setBayesianValue(graph: FDGGraph, node: Node): number {
  if (node.inDegree === 0) return node.truthValue as number;
  const parents = findDirectParents(node, graph);

  const parentValues = parents.map((parent) =>
     setBayesianValue(graph, parent));
  const bayesianValue = calculateBayesianValue(parentValues) as
     number;
  node.truthValue = bayesianValue;
  return bayesianValue;
}

/**
 * Calculates the Bayesian value based on the given parent
    values.
 * @param parentValues - An array of parent values.
 * @returns The calculated Bayesian value.
 */
function calculateBayesianValue(parentValues: number[]): number
   {
  // convert from percent to decimal
  const decimalParentValues = parentValues.map(
    (parentValue) => parentValue / 100
  );

  let bayesianValueDecimal = 0;

  for (let n = 1; n < parentValues.length + 1; n++) {
    if (n % 2 === 0) {
      bayesianValueDecimal -=
          intersectAddCombinations(decimalParentValues, n);
    }
    if (n % 2 === 1) {
      bayesianValueDecimal +=
          intersectAddCombinations(decimalParentValues, n);
    }
```

```
  }

  // convert back from decimal to percent
  return bayesianValueDecimal * 100;
}

/**
 * Calculates the sum of products for all combinations of size
    n from the given array.
 * @param array - The array to generate combinations from.
 * @param n - The size of the combinations.
 * @returns The sum of products for all combinations.
 */
function intersectAddCombinations(array: number[], n: number):
   number {
  const combinations = nCombinations(array, n);
  const multipliedCombinations: number[] = [];

  for (const combination of combinations) {
    const product = combination.reduce((acc, val) => acc * val,
       1);
    multipliedCombinations.push(product);
  }

  return multipliedCombinations.reduce((acc, val) => acc + val,
     0);
}

/**
 * Generates all combinations of size n from the given array.
 * @param array - The array to generate combinations from.
 * @param n - The size of the combinations.
 * @returns An array of combinations.
 */
function nCombinations(array: number[], n: number): number[][] {
  if (n === 1) {
    return array.map((a) => [a]);
  }

  const combinations: number[][] = [];
  for (let i = 0; i <= array.length - n; i++) {
    const subCombinations = nCombinations(array.slice(i + 1), n
       - 1);
    for (const c of subCombinations) {
```

```
      combinations.push([array[i], ...c]);
    }
  }

  return combinations;
}
```

A.6 Graph Drawing

Listing A.6: drawFDG.ts

```
import * as d3 from "d3";
import { FDGGraph, Node } from "./types";

/**
 * Draws a force directed graph to an SVG placed inside the
     #main HTML element.
 * @export
 * @param {FDGGraph} { nodes, edges } A set of nodes and edges.
 */
export default function drawFDG({ nodes, edges }: FDGGraph) {
  let nodeText = (d: Node) => {
    const isRootNode = d.outDegree === 0;
    if (isRootNode)
      return d.description.length > 40
        ? d.description.substring(0, 38) + "..."
        : d.description;
    if (d.radius > 20)
      return d.description.length > 30
        ? d.description.substring(0, 28) + "..."
        : d.description;
    return "";
  };

  const zoom = d3
    .zoom()
    .scaleExtent([0.25, 10])
    .on("zoom", handleZoom)
    .on("start", () => d3.select("#main").attr("cursor",
      "grabbing"))
    .on("end", () => d3.select("#main").attr("cursor",
      "initial"));

  d3.select("#main").call(zoom as any);
```

```
// initialize the simulation with forces
const simulation = d3
  .forceSimulation(nodes)
  .force(
    "link",
    d3
      .forceLink(edges)
      .id(({ index: i }) => nodes.map((node) => node.id)[i ||
         0])
      .distance(
        (d) => (d.target as Node).radius + (d.source as
           Node).radius + 70
      )
  )
  .alphaTarget(0)
  .alphaDecay(0.0228)
  .velocityDecay(0.4)
  .force("charge", d3.forceManyBody().strength(-600))
  .force("center", d3.forceCenter().strength(1))
  .on("tick", ticked);

// main container
const svg = d3.select("#main");

// graph container
const g = svg.append("g");

// arrow head
g.append("svg:defs")
  .append("svg:marker")
  .attr("id", "arrowhead")
  .attr("viewBox", "0 -5 10 10")
  .attr("markerWidth", 5)
  .attr("markerHeight", 5)
  .attr("orient", "auto")
  .append("svg:path")
  .attr("d", "M0,-5L10,0L0,5");

const edge = g
  .selectAll("line.link")
  .data(edges)
  .enter()
  .append("path")
  .style("stroke", "black")
```

```
  .attr("marker-end", () => "url(#arrowhead)") // sets arrow
     head position
  .style("stroke-width", 2);

const node = g
  .append("g")
  .attr("stroke", "black")
  .attr("stroke-opacity", 0.5)
  .selectAll("circle")
  .data(nodes)
  .join("circle")
  .attr("id", (d) => "g" + d.id)
  .on("click", (e: any) => console.log(e.target.__data__))
  .style("stroke-width", 1)
  .attr("fill", "#505050") // gray
  .attr("fill-opacity", (d) => 0.2 + (0.6 * (d.truthValue as
     number)) / 100) // scales truthValue from 0-100 to
     0.8-0.2
  .attr("r", (d) => d.radius)
  .call(drag(simulation));

// displays tooltip while hovering over a node
node
  .append("title")
  .text((d) => `truth value: ${d.truthValue}\ndescription:
     ${d.description}`);

const text = g
  .selectAll("text")
  .data(nodes)
  .enter()
  .append("text")
  .attr("fill", "black")
  .attr("text-anchor", "middle")
  .attr("dominant-baseline", "middle")
  .text((d) => nodeText(d))
  .call(drag(simulation));

function ticked() {
  edge.attr("d", (d) => {
    const target = d.target as Node;
    const source = d.source as Node;
    const deltaX = target.x! - source.x!;
    const deltaY = target.y! - source.y!;
```

```
    const distance = Math.sqrt(deltaX * deltaX + deltaY *
       deltaY);
    const normX = deltaX / distance;
    const normY = deltaY / distance;
    const sourcePadding = source.radius;
    const targetPadding = target.radius + 10; // offset to
       avoid the arrow head overlapping with the target node
    const sourceX = source.x! + sourcePadding * normX;
    const sourceY = source.y! + sourcePadding * normY;
    const targetX = target.x! - targetPadding * normX;
    const targetY = target.y! - targetPadding * normY;

    return "M" + sourceX + "," + sourceY + "L" + targetX +
       "," + targetY;
  });

  node.attr("cx", (d) => d.x!).attr("cy", (d) => d.y!);

  text.attr("x", (d) => d.x!).attr("y", (d) => d.y!);
}

function drag(
  simulation: d3.Simulation<Node, undefined>
): d3.DragBehavior<any, any, any> {
  function dragStarted(event: d3.D3DragEvent<SVGRectElement,
      any, any>) {
    if (!event.active) simulation.alphaTarget(0.3).restart();
  }

  function dragged(event: d3.D3DragEvent<SVGRectElement, any,
      any>) {
    event.subject.fx = event.x;
    event.subject.fy = event.y;
  }

  function dragEnded(event: d3.D3DragEvent<SVGRectElement,
      any, any>) {
    if (!event.active) simulation.alphaTarget(0);
    event.subject.fx = null;
    event.subject.fy = null;
  }

  return d3
    .drag()
```

```
      .on("start", dragStarted)
      .on("drag", dragged)
      .on("end", dragEnded);
  }

  function handleZoom(e: d3.D3ZoomEvent<any, any>) {
    g.selectAll("text").text((d) => {
      const node = d as Node;

      const isRootNode = node.outDegree === 0;
      if (isRootNode)
        return node.description.length > 40
          ? node.description.substring(0, 38) + "..."
          : node.description;

      if (node.radius * e.transform.k > 20)
        return node.description.length > 30
          ? node.description.substring(0, 28) + "..."
          : node.description;

      return "";
    });
    d3.select("g").attr("transform", e.transform.toString());
  }
}
```

Appendix B

Data from Performance Testing

B.1 Data Collection for Initial Graph Layout Load Time

Table B.1: Initial Graph Layout Load Time

Nodes	**Springy (ms)**	**D3-force (ms)**
100	1.29	2.42
1 000	19.4	10.6
10 000	287	110
100 000	1.63E+04	694
100	0.908	1.86
1 000	9.84	14.7
10 000	295	72.8
100 000	1.53E+04	760

100	0.947	2.01
1 000	13.1	13.1
10 000	269	74.8
100 000	1.76E+04	890
100	0.933	1.89
1 000	12.6	10.7
10 000	229	75.4
100 000	1.77E+04	1.11E+03
100	0.957	1.93
1 000	13.7	11
10 000	244	78.8
100 000	1.67E+04	808
100	1.64	2.01
1 000	10.8	10.2
10 000	240	72.9
100 000	1.67E+04	930
100	0.876	1.83
1 000	15.7	11.3
10 000	241	83.9
100 000	1.87E+04	785
100	0.959	1.88

1 000	12.7	9.42
10 000	211	89.1
100 000	1.78E+04	825
100	0.989	1.92
1 000	9.62	8.73
10 000	250	79.7
100 000	1.64E+04	907
100	0.917	1.86
1 000	13.7	9.93
10 000	274	90.1
100 000	1.75E+04	856

B.2 Data Collection for Graph Rendering Frame Rate

Table B.2: Graph Rendering Frame Rate

Nodes	Springy FPS	D3-force FPS
100	133	132
1 000	13.2	17.6
10 000	0.219	2.14
100 000	1.53e-3	0.832
100	122	124
1 000	14.9	21.7
10 000	0.221	1.97
100 000	1.61e-3	0.204
100	126	130
1 000	15.5	19.9
10 000	0.234	1.69
100 000	1.52e-3	0.182
100	127	129
1 000	15.3	21.5
10 000	0.217	1.50
100 000	1.56e-3	0.198
100	127	127

(continued)

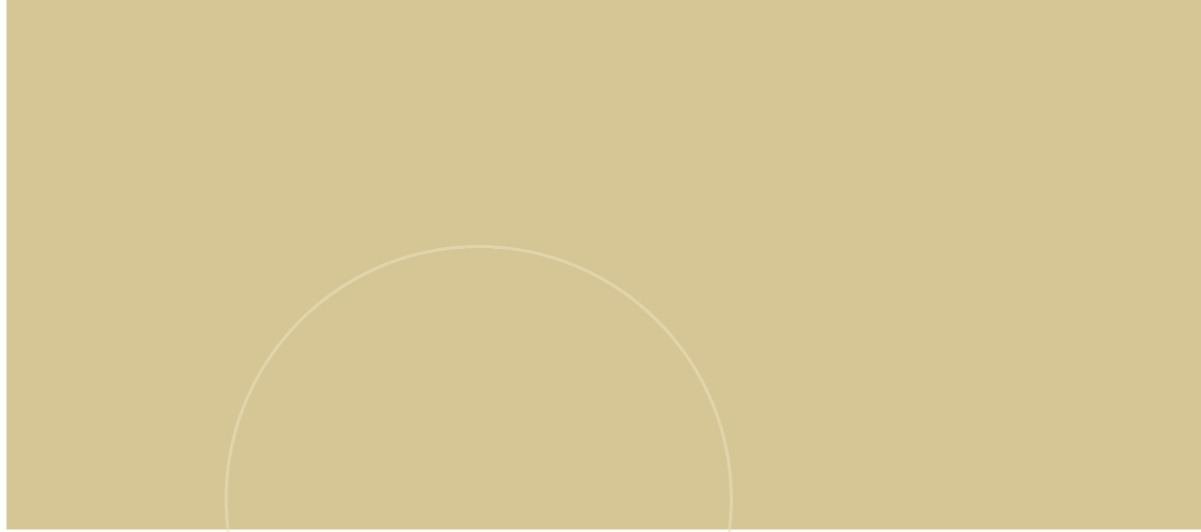
Nodes	Springy FPS	D3-force FPS
1 000	15.1	19.3
10 000	0.210	1.77
100 000	1.50e-3	0.197
100	128	132
1 000	15.9	19.2
10 000	0.216	1.67
100 000	1.56e-3	0.202
100	128	130
1 000	14.6	17.6
10 000	0.216	1.52
100 000	1.58e-3	0.357
100	126	130
1 000	15.3	18.8
10 000	0.220	1.68
100 000	1.61e-3	13.97
100	131	134
1 000	14.1	17.9
10 000	0.224	1.79
100 000	1.62e-3	0.397

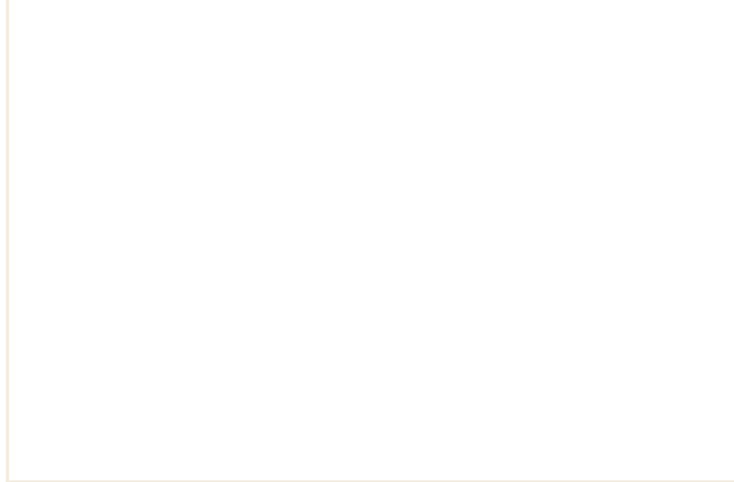
(continued)

Nodes	Springy FPS	D3-force FPS
100	128	124
1 000	14.1	17.5
10 000	0.219	1.71
100 000	1.61e-3	0.188

B.3 Data Collection for the Finalized System Performance Test

Nodes	Initial Graph layout load time (ms)	Graph Rendering (FPS)
100	3.48	131
100	3.26	129
100	3.91	132
100	5.66	132
100	3.26	133
100	3.40	132
100	4.13	133
100	2.92	127
100	2.82	132
100	3.20	126





Norwegian University of
Science and Technology