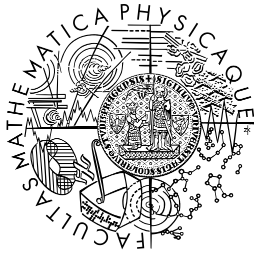

**FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University**

BACHELOR THESIS

Aleš Kakos

A Tool for Graph Visualisation of Social Networks

Department of Software Engineering

Supervisor of the bachelor thesis: Doc. RNDr. Irena Holubová, Ph.D.

Study programme: Informatics

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, Doc. RNDr. Irena Holubová, Ph.D. for her guidance and support as well as the people making up the Software Engineering department at Charles University for giving me the opportunity to study the ever-so-interesting field of computer magic.

I would also like to thank my family for supporting me on this path of life. Specical thanks belongs to my mother who didn't allow me to give up when things got tough and to my granfather to whom I promised the title of Bachelor bearing his surname.

Last but not least I want to thank my girlfriend and future wife for her patience and understanding during the hard times of my long overdue studies.

Title: A Tool for Graph Visualisation of Social Networks

Author: Aleš Kakos

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: This work is about the design and implementation of a website called Aphantasia - a social network for graph enthusiasts which allows users to create posts, link them to others, and view them as a part of a shared, ever-expanding interactive graph. This strategy presents an alternative to user-post interaction on social networks compared to the mainstream infinite scroll approach. We developed Aphantasia into a fully functional prototype and compared it to other graph-rendering software. Particular attention was paid to handling large numbers of posts without sacrificing the graph view's performance, readability, and user experience.

Keywords: graph data, visualization, Big Data

Název práce: Nástroj pro grafickou vizualizaci sociálních sítí

Autor: Aleš Kakos

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. RNDr. Irena Holubová, Ph.D., Katedra softwarového inženýrství

Abstrakt: Tato práce se zabývá návrhem a implementací webové stránky nazvané Afantázie – sociální sítě pro grafové nadšence, která umožňuje uživatelům vytvářet příspěvky, propojovat je s ostatními a zobrazit je jako rostoucí sdílený interaktivní graf. Tento přístup představuje alternativu k interakci uživatelů s příspěvky na sociálních sítích v porovnání s přístupem nekonečného posouvání. Afantázii jsme vyvinuli do stavu plně funkčního prototypu a porovnali ji s jiným softwarem pro vykreslování grafů. Zvláštní pozornost jsme věnovali zobrazení velkého množství příspěvků, aniž bychom omezili výkon, čitelnost a uživatelskou přívětivost grafového zobrazení.

Klíčová slova: grafová data, vizualizace, velká data

Contents

Introduction **7**

1 Graph Layout Algorithms **9**
1.1 Circular Layout . . . 9
1.2 Hierarchical Layout . . . 9
1.3 Radial Layout . . . 10
1.4 Force-Directed Layout . . . 11

2 Related Software **13**
2.1 Obsidian . . . 13
2.2 Gephi . . . 14
2.3 Cytoscape.js . . . 15
2.4 Final comparison . . . 16

3 Design Analysis **22**
3.1 Intended Use . . . 22
3.2 Functional Requirements . . . 22
3.2.1 Graph View . . . 22
3.2.2 User Management . . . 23
3.2.3 Thought Creation . . . 23
3.2.4 Routing and Pages . . . 24
3.3 Non-functional Requirements . . . 24
3.3.1 Graph View UX . . . 24
3.3.2 Extendability . . . 24

4 Implementation **25**
4.1 Preparation . . . 25
4.1.1 Used Technologies and Libraries . . . 25
4.1.2 Plan of Execution . . . 26
4.2 Basic Web Application . . . 27
4.2.1 Database Schema . . . 27
4.2.2 Backend Architecture . . . 28
4.2.3 Initial Frontend Implementation . . . 30
4.2.4 Hosting and Server Management . . . 31
4.3 Small Graph Vizualization . . . 32
4.3.1 FDL Implementation . . . 33
4.3.2 Parametrization . . . 33
4.4 Big Graph Visualization . . . 34
4.4.1 Frontend Architecture . . . 35
4.5 Final List of Features . . . 36
4.5.1 AuthX/Y . . . 37
4.5.2 Pages . . . 37
4.5.3 Localization . . . 38
4.5.4 Custom Graph Rendering Engine . . . 38

5 Testing 50
5.1 Graph View Performance . . . 50
5.1.1 Limits of the Graph View . . . 50
5.2 User Feedback . . . 50
5.3 Aphantasia Versus Related Software . . . 52

6 Documentation 53
6.1 User Documentation . . . 53
6.1.1 Registering and Logging in . . . 53
6.1.2 Opening a Thought . . . 53
6.1.3 Graph View . . . 54
6.1.4 Creating a New Thought . . . 55
6.1.5 Settings . . . 55
6.2 Installation Guide . . . 56
6.2.1 Running Aphantasia Locally . . . 56
6.2.2 Deploying Aphantasia . . . 59
6.3 Administrator Documentation . . . 61
6.3.1 Backend . . . 61
6.3.2 Frontend . . . 61

7 Conclusion 63

Glossary 65

References 66

List of Figures 67

List of Tables 68

Introduction

Motivation

In recent years, it has been hard to escape the infinite feed. It is a design pattern where the user is presented with a list of posts that can be scrolled through seemingly indefinitely. Infinite feed is the primary way of presenting posts used by, for example TikTok, Instagram, Facebook, Twitter, Reddit, and, to a lesser extent, YouTube. This approach is easy to implement, intuitive, and requires minimal user interaction - A swipe or scroll is all the input is needed.

While the infinite feed is not necessarily bad or insufficient, it does have some drawbacks. Namely:

- **Echo chambers** - When the recommendation system behind the infinite feed also includes or even centers around user preferences as a factor, the ideological or political similarity of the content consumed can narrow the user's worldview. This effect is often called an ”echo chamber” [1].
- **Addictive design** - The infinite feed might encourage mindless, addictive consumption of content with each swipe akin to the pulling of a slot machine lever. The term ”doomscrolling” [2] has been coined to describe this behavior.
- **Lack of autonomy** - The user does not decide on the next post in the feed. Instead, it is decided by the recommendation algorithm (colloquially referred to as ”The algorithm” [3]), which can be opaque and hard for content creators and consumers to understand.

In this work, we will explore a graph-based approach as an alternative to the infinite feed pattern.

Aphantasia

Aphantasia (sometimes also referred to as Afantázie) is the implementation part and the final product of this thesis, currently available at:

- `https://aphantasia.io` internationally
- `https://afantazie.cz` for Czech user base

It is a social network concept based on graph visualization of its contents. It lets users create posts called thoughts, interlink them, and explore them as animated, interactive, and colorful graph.

In the following text, we will go through the design and implementation of Aphantasia and explore whether it can mitigate or at least alleviate the drawbacks of the infinite feed pattern.

- We will start by introducing the concept of Graph Layout Algorithms in Chapter 1.

- Next, we will compare software products that provide graph visualization in Chapter 2.
- In Chapter 3, we will look at the analysis, requirements, and use cases of Aphantasia.
- Then we will implement the software accordingly in Chapter 4.
- Chapter 5 is where we test Aphantasia. We will compare it to the software from Chapter 2, try its quantitative limits, and evaluate the user experience when compared to the infinite feed.
- And finally, in Chapter **??**, we will provide the User, Developer, and Administrator documentation.

1 Graph Layout Algorithms

Graph layout algorithms (or GLAs for short) are a class of algorithms used for computing positions of nodes so that they make a nice-looking or helpful diagram. These algorithms accept graph-representing data as input and produce positions of individual nodes as output.

In this work, we are going to assume that GLAs produce 2-dimensional layouts, but it is worth mentioning that the number of dimensions can be arbitrarily high.

There are many types of GLA, each with its strengths and weaknesses. Let us look at some of the common ones.

1.1 Circular Layout

Circular layout (Figure 1.1) algorithms arrange nodes in a circle, often emphasizing the structure of the graph by placing nodes with similar properties close together. In a circular layout, nodes can be distributed evenly along the circumference of the circle, or their placement can be weighted by specific properties (e.g., node degree or importance).

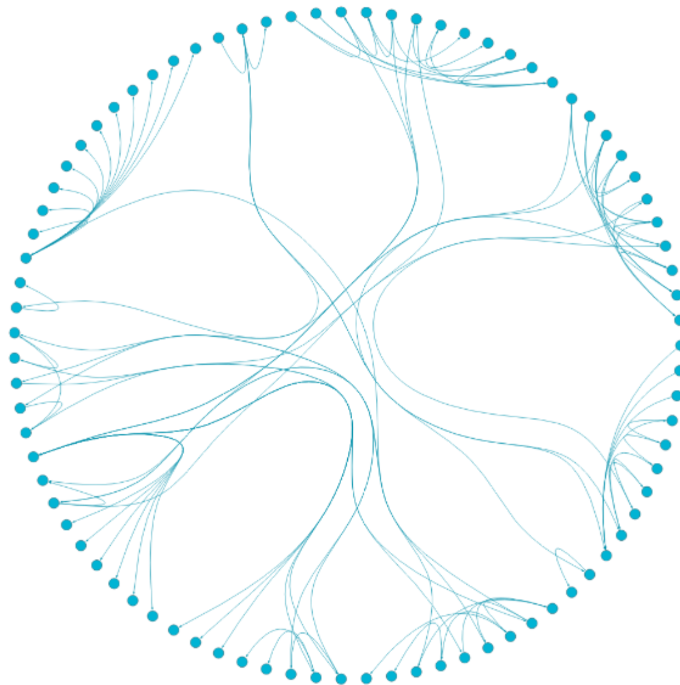

Figure 1.1 Circular layout[4]

1.2 Hierarchical Layout

Hierarchical layout (Figure 1.2)algorithms are designed to emphasize directional relationships, such as those found in flowcharts, dependency graphs, or

organizational charts. These algorithms arrange nodes in layers or levels, with edges generally flowing in a single direction (e.g., top to bottom or left to right).

This layout can be applied to general graphs but is most effective for directed acyclic graphs (DAGs) or undirected trees. Note that the structure in Figure 1.3 is not a tree but a general graph.

Figure 1.2 Hierarchical layout[4]

1.3 Radial Layout

Radial layout (Figure 1.3) is a type of hierarchical layout where nodes are arranged in concentric circles around a central node (root). Its immediate neighbors are arranged in the first circle. Subsequent layers represent nodes further away. Radial layout can help highlight relationships between the central node and its surroundings.

Figure 1.3 shows a tree. In the context of social networks, many of the contemporary social networks are composed of trees:

- **Reddit** - The root represents a post, and the rest are its comments
- **Twitter** - The root is a standalone tweet, and the rest form a series of replies
- **Youtube** - The root is a video, and the rest are its comments

As such, if one visualized the content of these social networks, it might look like a set of radial layouts. We are highlighting this fact because Aphantasia's content

structure is not a tree but a DAG which might have interesting consequences for navigation, user experience, and usage.

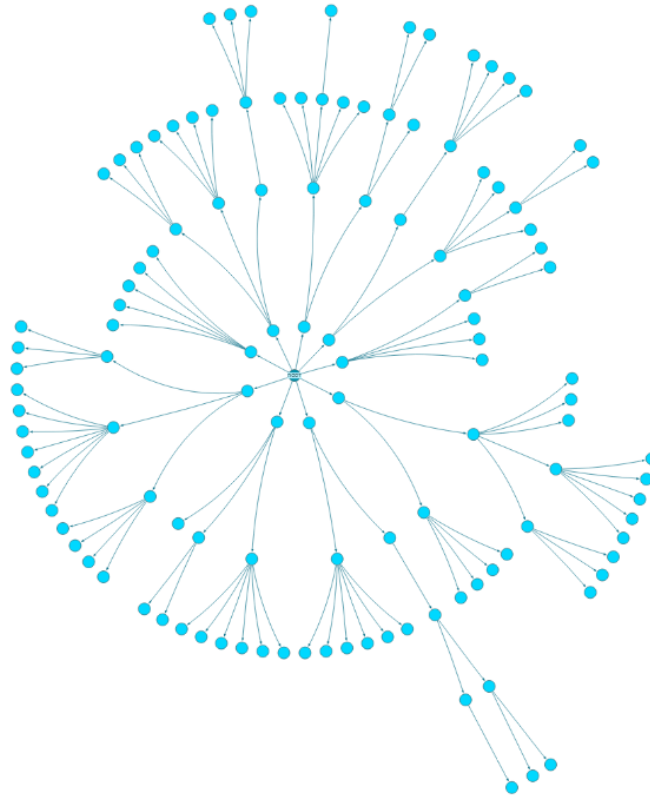

Figure 1.3 Radial layout[4]

1.4 Force-Directed Layout

This type of layout is going to be the focus of this work. All of the layouts we will present in later chapters were produced by force-directed layout (FDL) algorithms.

The main idea behind FDL algorithms is very intuitive: Nodes are attracted to each other when connected by an edge and repelled otherwise. Elegant in concept and easy to implement, this approach is endlessly customizable.

One particularly useful feature is that FDL algorithms are incremental and update the layout continuously. This can be utilized to produce visually pleasing animations of the data and allows for user interaction (e.g., dragging nodes around or changing parameters). The dragging feature can also be used to aid the algorithm in achieving a more desired layout during its run.

Thanks to this aspect, a developer implementing an FDL algorithm has the ability to adjust the strength of the forces, add new ones, or implement entirely new behaviors and parameters to fit specific use cases - all by watching the animation run, interacting with it, and inferring what behavior needs to be changed or added.

These positives are, however, balanced by two drawbacks:

- **high time complexity**

  The basic version of FDL has a time complexity of $O(n^2)$ per step, where n is the number of nodes in the graph. For stabilization, usually n steps are considered sufficient, and thus, the time complexity for producing a stabilized graph is often computed as $O(n^3)$.

- **sensitive parameterization**

  As for parameterization, the algorithm is highly dependent on the parameters set by the user (or developer), some of which can be very sensitive and radically change the resulting layout with only a small change in value. This can be a challenge for users unfamiliar with the algorithm or the data they are working with. Achieving a specific look or quality for the final graph render often requires a significant amount of time spent tweaking the parameters.

  This fact also means that it is difficult to create a ”one size fits all” FDL algorithm - the parameters that work well for one dataset might not work well for another.

2 Related Software

There are plenty of programs and libraries for graph visualization with different purposes. There are D3.js, Neo4j, Graphviz, Tulip, Wolfram Grapher, Pajek, and many more, just to name a few.

In this chapter, however, we will focus on just three software products, each with a very different use case and target user base.

- The first one is **Obsidian** which is primarily a note-taking application but provides a graph view of the notes as an interesting and easy-to-use feature.
- The second is **Gephi**, a software focused on in-depth analysis and visualization of large graphs.
- And finally there is **Cytoscape.js**. A JavaScript library for graph visualization in the browser.

We chose these in particular because Aphantasia could be viewed as a hybrid of the trio.

We will look into three aspects of these products:

- Primary use-case and target user base
- User experience
- Ability to visualize large graphs

As the source of data for testing the large-graph visualization capabilities, we chose the dataset of citations between papers in the field of high-energy physics [5] (Later refered to as CitHep). It contains 34 546 nodes, 4 215 78 edges, and temporal data (i.e., dates of publication).

This dataset is suitable for our purposes because:

- It is large enough to test the capabilities of both the mentioned software and Aphantasia
- It is a real-world dataset with temporal data that is going to play a role in visualizing the contents of Aphantasia

2.1 Obsidian

Obsidian [6] was a direct inspiration for Aphantasia. It allows users to create, edit, and, most importantly, interlink markdown file notes in the file system. One Obsidian project is a system directory called a Vault - a set of markdown notes, user settings, plugins, and other files. The interlinked notes in a Vault form a directed graph, which can be visualized with just a click of a button. The graph is animated and provides the ability to replay the history of the Vault from the very first note to the current state.

It is apparent from this description that Obsidian is aimed at a general audience of note-takers with maybe a slight bias towards graph/data visualization enthusiasts. It is available for all the major operating systems and has a large community of users and plenty of extensions available through the community plugins.

Obsidian graph view

The graph visualization in obsidian (called graph view) is based on a force-directed layout algorithm. [1] It is easy to use and provides an appealing visual representation of the notes. The graph is animated and interactive, meaning the user can drag nodes around, zoom in and out, and click on nodes to open the associated note.

It is also customizable to some extent. The nodes' colors can be set based on different filters, such as path, tags, or text search. Users can also adjust four sliders - central force, repel force, link force, and link distance. The graph view is swift for small graphs, but the program needs to spend some time indexing the notes for larger graphs. (Though this is a one-time operation, the indexes are then saved in the Vault).

In Figure 2.1, one can see a typical Obsidian graph view depicting a small Vault of one of our projects.

In Figure 2.2, we converted a part of the CitHep dataset with only the first 3000 nodes [2] to markdown notes and visualized them in Obsidian. It took our machine over 10 minutes to index this graph. Once indexed, the application ran smoothly.

We were unable to visualize the whole dataset without the program crashing or taking too long to index. However, we found a few examples of larger datasets rendered in Obsidian online, with one containing 22 000 nodes [7].

Obsidian is not very good at clustering nodes with a high degree of connectivity. To our knowledge, it also does not support automatic node coloring based on clusters, modularity, or other data indicators. Color must be user-defined based on the content and location of the notes. CitHep nodes do not contain content apart from id and date, so the nodes in Figure 2.2 are left default gray.

2.2 Gephi

Much more specialized than Obsidian, Gephi [8] is an open-source software focused on visualization and quantitative analysis of large graphs. It provides several algorithms for graph layout and quantitative analysis of various data and is extendable through community plugins.

Gephi can compute quantitative characteristics of graph-based data such as modularity, clustering coefficient, degree distribution, and many more. It can visualize the working data in the viewport and export visually appealing images of the visualized graph.

Gephi graph export

In Figure 2.3, one can see, again, the first 3000 nodes of the CitHep dataset visualized in Gephi. Compared with Obsidian, the Gephi render export provides more visually identifiable characteristics of the data:

- The layout is more structured, and communities are more visible

[1] We weren't able to find any specifics about the algorithm used in Obsidian. However, from the way its graph view behaves, it is safe to assume a variation of an FDL algorithm was used.

[2] The first as in the order they are stored in the dataset file.

- Colors of the nodes represent their associated modularity class
- Size of the nodes represents the number of citations the associated paper has

Figure 2.4 is an exported image of the entire CitHep dataset visualized in Gephi. The process of exporting this image took a few hours. We had to learn to use the program itself, import the data, tweak the layout using various provided algorithms (though we primarily relied on ForceAtlas 2), and finally export the resulting image.

The software crashed a few times during the process, and with the citHep dataset loaded in memory, it wasn't always buttery smooth, but it was otherwise very usable. Considering the amount of data, that is a commendable feat.

Again, one can find even larger datasets visualized using Gephi. One post on Gephi forum [9] shows an exported image of 212 600 nodes and 4 045 203 edges.

The user experience of Gephi is one of a technical tool - something one has to learn to use and spend time with to get the most out of it. But the reward is the ability to visualize and analyze large graphs better than with any other software we could find.

2.3 Cytoscape.js

Cytoscape.js is a JavaScript library for graph visualization in the browser. Its homepage starts with a Demos section followed by the headline 'Introduction' and almost 8200 lines of text [10]. Among the demos, there are various examples of usage ranging from simple FDLs to more complex use cases such as an interactive graph of wine and cheese pairings. [11]

To assess the software, we used two projects - a simple project of our own and one official template showcasing FDL features of the library.

Firstly, we attempted to create a react + cytroscpe.js project. We initialized a fresh react ts project using Vite, added the dependency to Cytoscape.js, and, using the official instructions on the Cytoscape.js homepage, we created a simple application capable of rendering a graph. See the result in Figure 2.5. There is nothing particularly noteworthy about this attempt apart from the fact that the web application uses Cytoscape.js alongside react TypeScript, which seems to be a valid combination of libraries.

The second project was an attempt to render the CitHep dataset in Cytoscape.js. In order to do so, we cloned the GitHub repository of one of the official examples from the cytoscape.js homepage [12]. The example is called 'Euler' and showcases several small demos using a Cytoscape.js proprietary FDL.

The demo called 'Large graph' showcases a path of 5000 nodes and the result can be seen in Figure 2.6.

This example is animated, and when the page refreshes, the graph stabilizes in real-time. We had to increase the simulation time as the default value resulted in a tangled, unhelpful layout. In this case the animation was realtively smooth and with the simulation time set to 10 seconds the graph layout worked well.

We modified this project to use, again, the first 3000 nodes from the CitHep dataset. We tweaked several parameters such as force strength, edge length, and

repulsion strength, and in the end, we achieved the layout shown in Figure 2.7. The performance during the runtime of the FDL data dropped under 10 FPS, but the application remained usable and stable. The layout we produced was very similar to the one we made with Obsidian.

We also attempted to render the entire CitHep dataset, but unfortunately, the application could not handle it and crashed.

While we did not use the full capabilities of the library, the experience with Cytoscape.js was very positive. The animations are smooth, there are plenty of parametrization options, and the resulting layout is satisfactory.

2.4 Final comparison

As can be seen in Table 2.1, the three demonstrated software products vary greatly. The intended use case and target user base are very different, and so is their large graph rendering capability.

Obsidian is by far the most accessible but offers the least amount of control over the graph layout.

Gephi is the most powerful but also the most complex and time-consuming to use.

Cytoscape.js is a good middle ground between the two but only accessible to developers.

	Obsidian	**Gephi**	**Cytoscape.js**
Use-case	note-taking	data analysis and visualization	graph visualization in browser
Target Userbase	general audience	researchers, technical users	web developers
User Experience	easy to use	technical, steep learning curve	programmatic, mostly parametrization
3 000 nodes handling	slow indexing but smooth afterwards	stable, smooth	stable but visibly lower FPS
34 546 nodes handling	skipped as indexing took too long	mostly stable, lower FPS while running FDL	crashed immediately

Table 2.1 Comparison of Obsidian, Gephi, and Cytoscape.js

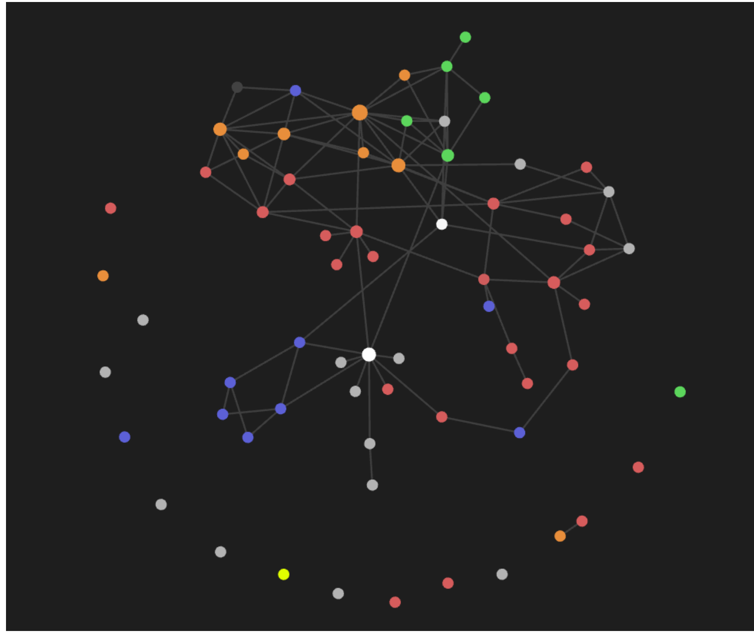

Figure 2.1 A common graph view of a small Vault in Obsidian

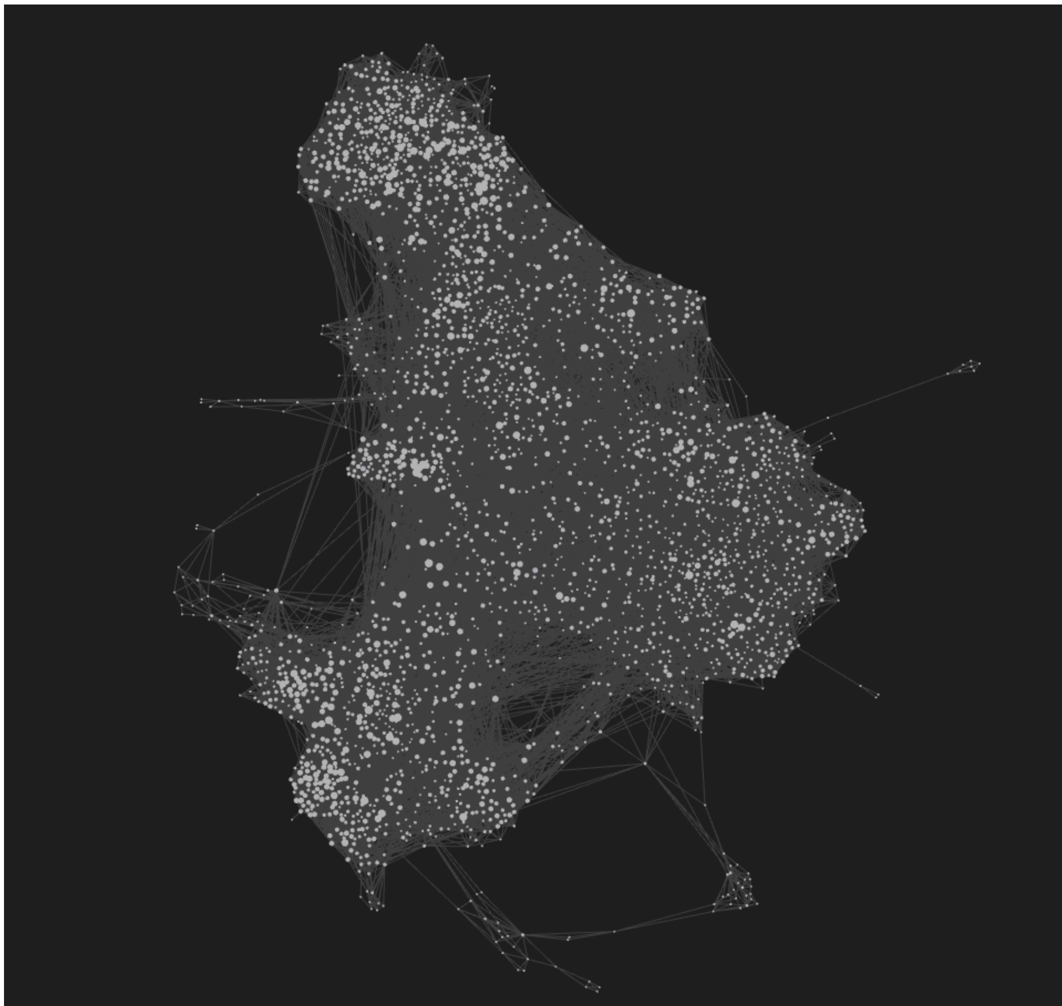

Figure 2.2 The first 3000 nodes of the CitHep dataset visualized in Obsidian

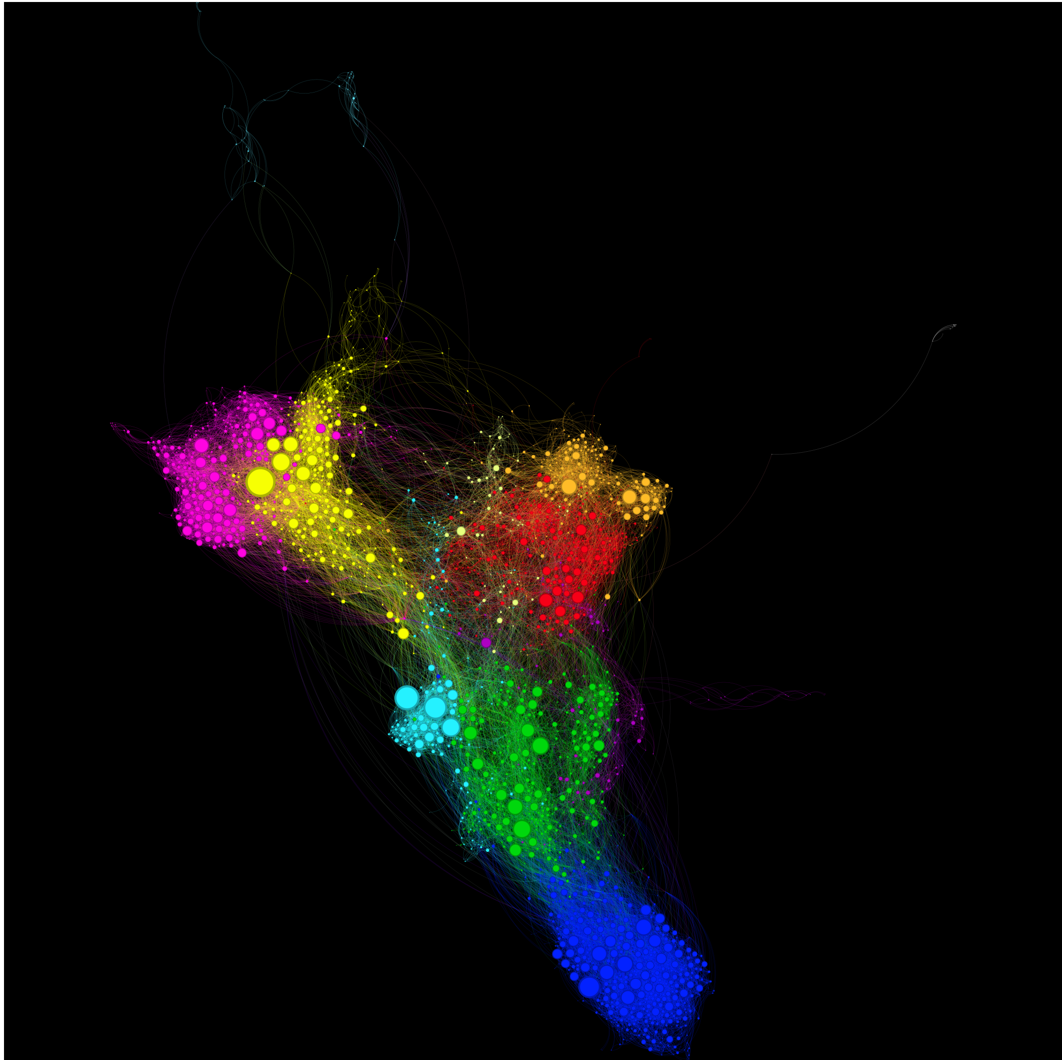

Figure 2.3 The first 3000 nodes of the CitHep dataset visualized in Gephi

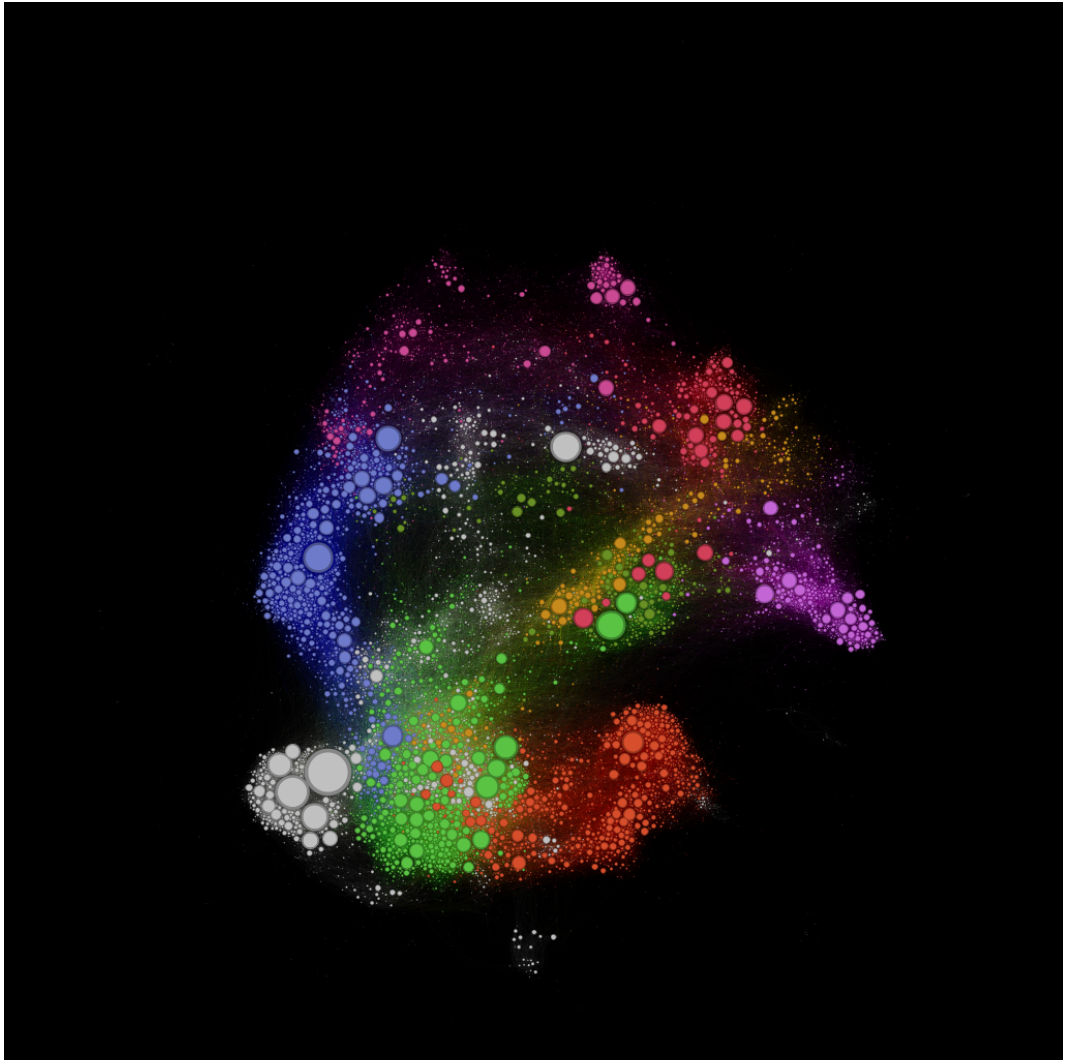

Figure 2.4 CitHep dataset visualized in Gephi (34546 nodes)

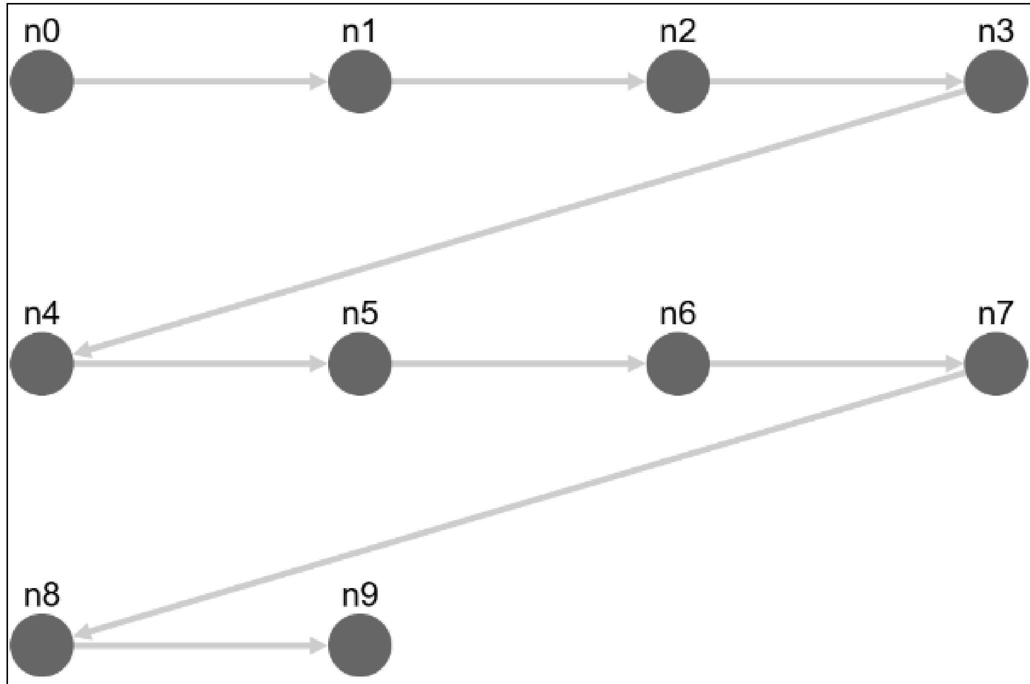

Figure 2.5 A simple application in react + cytoscape rendering a simple graph

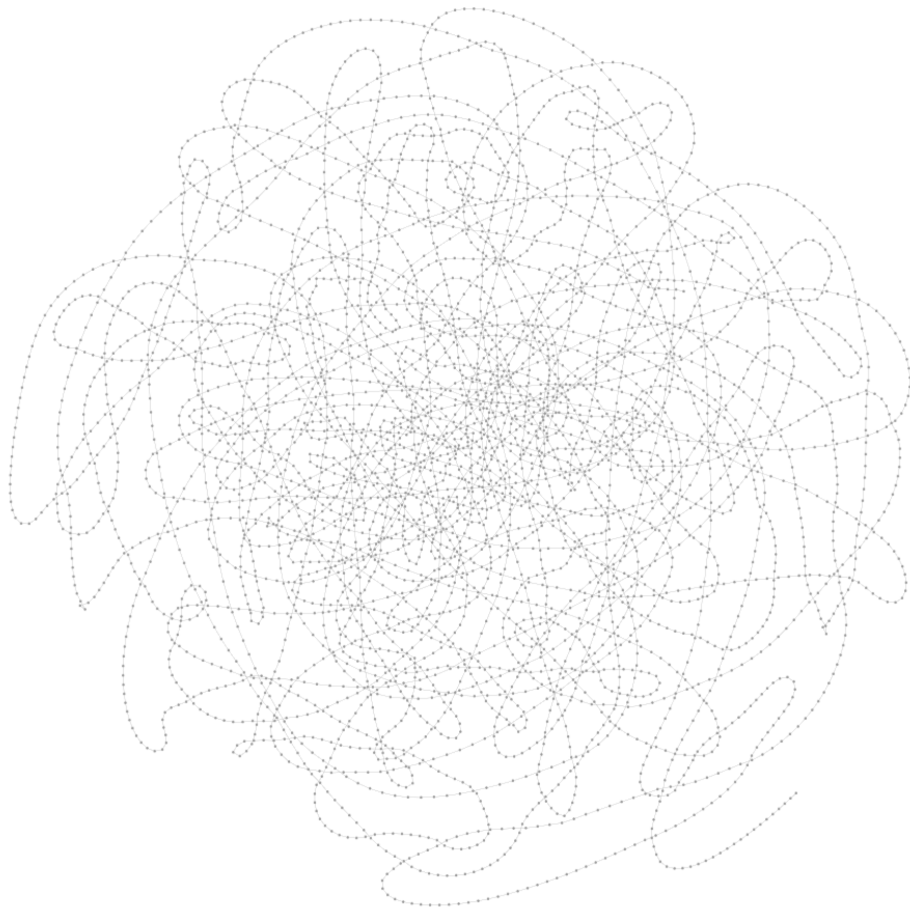

Figure 2.6 An official example of large graph rendering in Cytoscape.js (path of 5000 nodes)

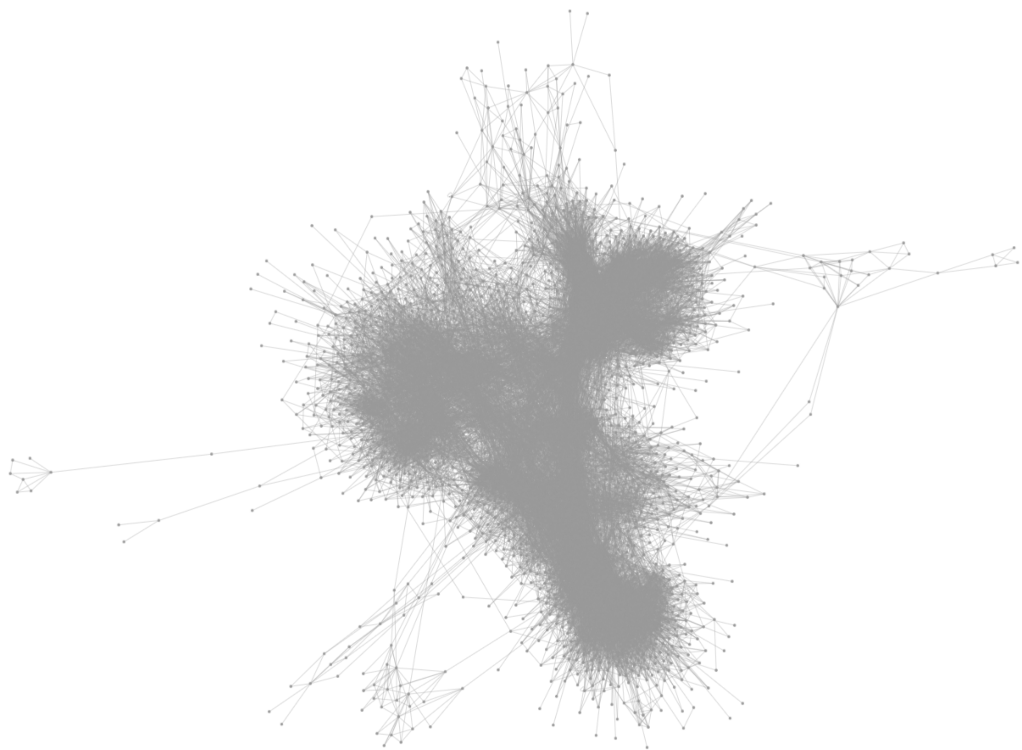

Figure 2.7 The first 3000 nodes of the CitHep dataset visualized in Cytoscape.js

3 Design Analysis

Before implementing Aphantasia, we will introduce its requirements and set expectations. In short, Aphantasia will be a social network based on an Obsidian-like graph view.

3.1 Intended Use

The use case of Aphantasia is very similar to other social networks. Users will be incentivized to create text-based posts called thoughts, link them to other thoughts, and try to collect replies from other users.

Apart from the graph aspect, Aphantasia will be novel in the structure of its content. Whereas most other similar social networks allow only one link per post, Aphantasia will allow up to 5. In graph theory, the structural difference between the two approaches is that while other networks contain forests, Aphantasia will be made of DAG components.

We expect this element to make the posts more interconnected. On that basis, we want to create an online experience that encourages two aspects:

- **Exploration** of the thoughts of others
- and creation of new **associations** between them.

The ability to reply to more than one thought at once combined with graph view means users will be able to interlink two separate connected components together and thus create bridges of associations.

3.2 Functional Requirements

The functional requirements of Aphantasia are as follows:

3.2.1 Graph View

The focus of Aphantasia is the graph view. We want to create a comparable user experience to Obsidian's graph view, and thus, we will need to implement the following features:

- **Node Rendering** - Each thought should be rendered as a node
- **Edge Rendering** - Links between thoughts should be rendered as edges
- **Animated Graph Layout** - The graph should stabilize using an FDL algorithm and animate the stabilization process
- **Interactivity** - Nodes should be draggable to influence the layout algorithm
- **Movement and Zoom** - A viewport with the ability to move around and zoom in and out through the graph

- **Content Preview** - Users should be able to click on a thought to see its content
- **Clickable links and backlinks** - The content preview needs to provide clickable links and backlinks (i.e., replies)
- **Floating Titles** - When zoomed in past a certain threshold, titles of thoughts should be displayed under their nodes in the graph view

To then elevate the experience beyond Obsidian and towards its own online shared experience, we will need to implement the following features:

- **User-specific Coloring** - Thoughts in graph view should be colored according to the user who created them
- **Replies-dependent Node Size** - Thoughts that have more replies should appear larger to indicate their importance [1]
- **Graph Exploration** - The graph should be explorable by traversing the thoughts through their edges
- **Dynamic loading** - Only a subset of thoughts should be loaded at once based on the user's position in the graph
- **Big Graph Support** - The graph view should be able to handle arbitrarily big graphs

3.2.2 User Management

Aphantasia needs to provide a simple user management system. Concretely:

- **Registration**
- **Login**
- **Logout**
- **Account Personalization**

The personalization aspect should include the ability to choose color. Users' thoughts will be displayed in this color.

3.2.3 Thought Creation

Registered users should be able to create new thoughts and link them to other existing thoughts. Each thought should have a title, a body, and up to 5 links to other thoughts.

[1] Obsidian does have this feature as well, but it is not as pronounced as we would like

3.2.4 Routing and Pages

Aphantasia should have at least the following pages:

- **Graph View**
- **User Settings**
- **Post Creation Form**
- **Login and registration pages**

3.3 Non-functional Requirements

Our non-functional requirements will be focused mainly on the graph view and the user experience it provides.

3.3.1 Graph View UX

The graph view of Aphantasia should offer the following qualities:

- **Fluidity** - The graph should be animated without stuttering or FPS drops
- **Responsive Design** - The graph view should be usable on both desktop and mobile devices
- **Performance** - The graph should be able to handle at least a hundred thoughts on screen at once
- **Intuitive Exploration** - The graph exploration should be intuitive and easy to understand

3.3.2 Extendability

We would like to design Aphantasia in a way that allows for easy extension and modification. The app should be able to easily accommodate additional features such as:

- **Notifications** - In the future, a page with replies to user's thoughts is planned
- **Search and Filters** - In time, users should be able to search for thoughts and filter them based on various criteria
- **Graph Parameters Modification** - The graph view parametrization (such as the strength of the forces in the FDL algorithm) does not need to be available to the users right away, but the application should allow for it in the future

4 Implementation

In this chapter, we will go through the implementation process of Aphantasia and the technical challenges we faced. The relevant source code is available at `https://github.com/0rbit3r/afantazie_bachelors_thesis`.

4.1 Preparation

Aphantasia will be a publicly accessible web application, and as such, we will need to set up the following:

- **Frontend** - a webpage serving an interactive graph and UI
- **Backend** - for business logic and data processing
- **Database** - for persistence of data
- **Hosting** - to make the website accessible

4.1.1 Used Technologies and Libraries

Before implementing Aphantasia, we need to decide on the technologies we will utilize.

Hosting, Database and Backend

Let us start from the bottom with hosting. We will use a Virtual Private Server (VPS) from a hosting company, Váš Hosting. They will provide us with a Linux server fully prepared for hosting, including SSL certificates, domains, databases, and tools to manage the server.

For the backend, we will use **.NET 8.0** with **C#**. While the main reason for this decision is familiarity with the technology, the .NET platform is regardles as a good choice as:

- It is Linux-compatible
- The provided the **Entity Framework Core** library provides ORM, complete code-first solutions and makes it easy to replace database technology later on if needed

For the database, we will use **PostgreSQL** (version 15.10) as it is one of the most popular database engines and is known for its reliability and performance. We will use code-first approach to database design and will use Entity Framework Core (version 8.0.4) to scaffold and migrate the database (programmatically create and update the database schema based on defined Entities in the project).

Graph View

This is where the most crucial decisions will be made as its graph view is the our primary focus and we are striving for a good user experience.

We already saw a library that might be usable in our case in Section 2.3. We decided against using it as we would use only a fraction of its features, and it might be overkill for our needs. Instead, we will implement our own graph rendering engine using a custom GLA implementation and render the nodes using a rendering library.

The reasoning behind this is primarily based on customizability - we will be able to implement only the features we need and optimize the rendering for our use case.

To render the graph, we will use an existing JavaScript library capable of 2D graphics. We considered four possibilities to render the graph:

- **HTML and CSS** - the most barebones solution would be to use a set of divs to represent nodes and SVG lines to represent edges
- **HTML5 Canvas** - a more sophisticated solution to draw both nodes and edges in HTML5 Canvas API
- **PixiJS** - library for 2D rendering in the browser
- **Three.js** - library for 3D rendering in the browser

We chose to use **PixiJS** (version 7.4.2) as it seemed to provide a good compromise between ease of use and performance for 2D rendering in the browser.

Pages and UI

To implement graph view UI and pages for user management, login, and post creation, it makes sense to take advantage of a JavaScript framework. We have experience with two - Angular and React.

We will use **React** (version 18.2.0) with TypeScript. React has an unopinionated approach to architecture, which should make it easier to integrate with our rendering solution. We have already seen that React can be integrated with Cytoscape.js in Section 2.3. We will attempt to integrate React with PixiJS to create a graph view page with UI controls friendly on both desktop and mobile devices.

4.1.2 Plan of Execution

With technological decisions made, the roadmap for the implementation of Aphantasia is as follows:

1. In the first stage (Section 4.2), we will implement a basic web application, including user management. We will host this application on the VPS, set up a PostgreSQL database, and configure the reverse proxy server.
2. Next, in Section 4.3 we will implement small graph rendering engine. The result of this stage should contain an Obsidian-like graph view capable of rendering at least a few hundred nodes.

3. And lastly, in Section 4.4, we will extend the application to handle large graphs and thus finish the implementation.

4.2 Basic Web Application

In this section we will describe our process of creating a simple web application, scaffold the backend architecture, set up the database and implement user management. We also decided to implement a simple chat service during this phase. The reasoning behind this decision was the following:

- It will serve as preparation for the real-time server-client communication, which will be useful for the graph view (see Section 4.2.3)
- The user management system, just on its own, is not useful or testable
- The chat is locked behind login, and thus, we can develop and test authorization early on
- The messages inherit the color of their author, and thus, we can test the color selection feature

4.2.1 Database Schema

We set up the database of Aphantasia, including the graph-specific tables:

- **Users** - holds the user data
- **Thoughts** - holds the thought data
- **ThoughtReferences** - holds the links between the thoughts

A diagram of the database schema can be seen in Figure 4.1.

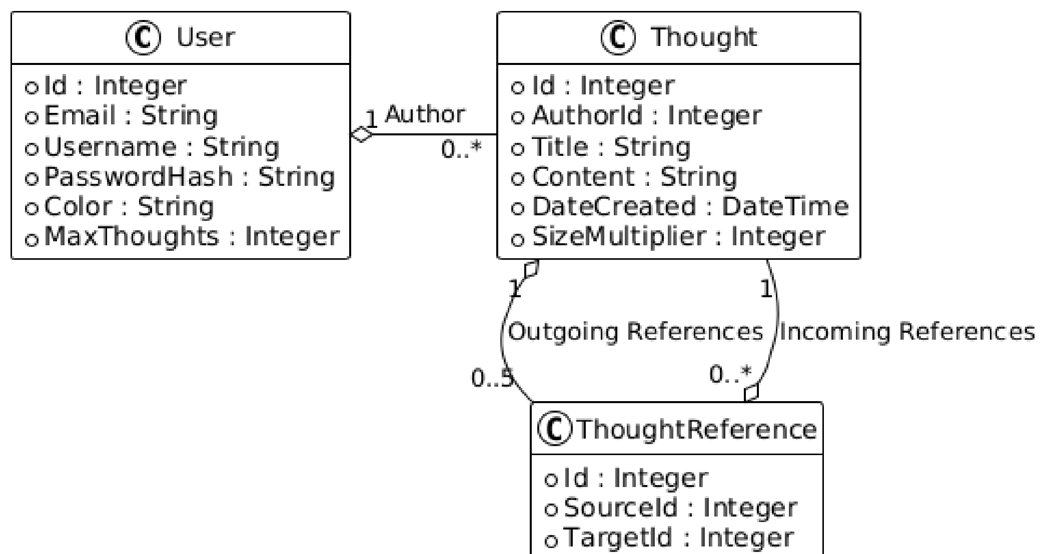

Figure 4.1 Aphantasia Database schema

We created the database using code-first approach and scaffolded the local development database and production database using Entity Framework Core. With that, the database is ready to be used by the backend.

4.2.2 Backend Architecture

The backend of the basic application is an ASP.NET application written in C#. Similarly to the database schema, we decided to develop architectural design early on and it paid off as development of later features was much easier. The initial architectural blueprint remained mostly unchanged until the end of the project.

The solution of the backend consists of 19 projects implementing the backend to Aphantasia and one project for secondary tools (see Section 6.2.1 for more details).

The backend projects are divided into four directories (forming conceptual layers):

- **Presentation**
- **Service**
- **Core**
- **Data**

There is also one project outside of these layers that serves to Bootstrap the application and set up dependency injection. Figure 4.2 describes the architecture of the backend with its dependencies.

This division is loosely based on Onion architecture [13]. Notice that the Presentation layer doesn't directly depend on the Service layer. Instead, it depends on the Service Interface layer, and the actual implementation is injected at runtime. This allows for clean separation of concerns, makes the code more testable, and allows for easy swapping of implementations.

The following sections describe the individual layers and their responsibilities in more detail.

Presentation Layer

The Presentation layer sits in the outermost layer of the application and is responsible for handling HTTP requests and responses. It contains two projects - **Api** and **Model**.

The Api project holds API controllers implemented using ASP.NET Core MVC. The controllers and signalR hubs only handle incoming traffic and for handling business logic, they call the appropriate service in the Service layer.

The Presentation Model contains definitions for the data transfer objects (DTOs). These objects are used to transfer data between the client and the server.

Service Layer

The Service contains the business logic of Aphantaisa. It is a set of 5 projects (and their respective interfaces):

- **Auth** - handles user registration, login, and JWT token generation
- **Chat** - handles the chat functionality
- **Site Activity** - handles the real-time stats of the site (such as the number of users online)
- **Thoughts** - handles the creation and retrieval of thoughts
- **User Settings** - handles the user settings (such as selected color or on-screen thoughts limit)

Core Layer

The Core contains three projects:

- **Core Model** - contains the core model of the application used in Repository and Service interfaces as arguments
- **Localization** - contains the localization resources for the application
- **Constants** - contains constants (currently only the default on-screen thoughts limit for newly registered users)

This layer sits in the midle of the application dependency wise and is used by both the Service and Data layers.

Data Layer

The Data layer is called by services to access the database using Entity Framework Core. It contains the Repository project and its interface. The Repository project is responsible for handling the database queries and updates.

AfantazieServer Project

Since the application is using dependency injection we cannot put its entry point inside any of the projects mentioned above without breaking the principles of the Onion architecture. If we, for example, wanted to run the API by setting setting the Presentation.Api as the startup project, we would have to add references to the Service layer and Data layer to add their classes to dependency injection container and thus break the separation of concerns.

Instead, we created a separate project that serves as the entry point for the application. It is responsible for all the setup and configuration of the application, which includes but is not limited to:

- Configuration files
- Dependency injection setup
- CORS setup (Cross-Origin Resource Sharing)
- Logging setup

Note that not all of these setups are done in the AfantazieServer project itself. All individual projects contain the bootstrapping logic, and expose an ”AddModule” method that is called from the AfantazieServer project to add the respective service. For example, the AfantazieServer project calls the AddDataModule() method from the Data project, which then registers the Data module with all its responsibilities, allowing it to swap modules for different implementations if needed (as long as they implement the corresponding interface).

Models

We use three different models to represent data in the application - each serving a different purpose in its respective layer:

- **Presentation Model** - DTOs used for transmission between client and server (The client has to implement the exact same model to be able to communicate with the server)
- **Data Model** - Entities correspond to Database tables (which is created code-first from the entities by Entity Framework)
- **Core Model** - Serves as an intermediary between the Presentation and Data models. Business logic should be implemented in the Service layer using this model.

Both Repository and Service interfaces use the Core Model as arguments and return values. This means that the Service layer is not dependent on either the Entities or the DTOs, but it also requires mapping between the Core Model and the Data Model. For mapping, we used the Mapster library inside the Data and Presentation Model projects.

4.2.3 Initial Frontend Implementation

During the first phase, we initialized the web application and implemented authX/Y, real-time server-client communication and routing with a few pages:

- Home page
- Login page
- Registration page
- Chat page
- About page

We also styled the application using plain css. The styling is minimalistic and not exactly modern but is functional and responsive.

Cache Busting

Browsers routinely cache files to speed up the loading of websites. This is a good thing as it reduces the load on the server and speeds up the user experience. However when the website is in active development and the files are changing frequently the cache can become a problem. Users might not see the changes made to the website because the browser is loading the old cached version of the file. The techniques for solving this problem are called Cache busting. We used a React library called React cache buster to resolve this issue.

At first, we had no success with the library as it was not working as expected. The reason, as we found out, was that the library uses a meta.json file to store the version of the application. The client then checks the version of the application, and if it is different from the version in the meta.json file, it forcefully reloads the page to get the new version of the application.

The problem was that Nginx was caching the meta.json file, and the client was not getting the new version of the file. Instead, it was getting a response with a 304 status code (not modified). We solved this by adding a directive to the Nginx configuration file that forces the server to provide a new version of the file at all times. See Section 6.2.2 for the exact configuration used.

Server-Client Communication

For the chat application, we tried to use WebSockets for real-time communication. While we did manage to get this approach working, we did not like the developer experience.

SignalR is a library that simplifies the process of setting up WebSockets and provides a nice API for server-client communication. It is available for both .NET and JavaScript and is easy to set up, with much less boilerplate code than using WebSockets directly.

SignalR uses so-called hubs to communicate between server and client, which automatically handle the connection and disconnection of clients.

We implemented two hubs - ChatHub and StatsHub, with the former being used for chat messages while on the chat page and the latter used across the entire application to keep track of and display the number of users online on the homepage.

This feature can be exploited further, for example, to inform the client that new thoughts were created.

4.2.4 Hosting and Server Management

The VPS we rented comes ready with a lot of tools to manage the server, is fully prepared for hosting, is accessible via SSH and has a web-based control panel for managing the server.

After installing the .NET 8 runtime, the process of running a .NET application was as simple as transferring the program via SFTP and inputting 'dotnet Afantazie.dll' into the terminal. Deploying the frontend required only copying the files to the necessary directory in the server filesystem. DNS and SSL certificates were set with a mouse click in the provided server management.

Nginx

The greatest challenge when hosting the application came with setting up a reverse proxy server (although that was mostly due to our inexperience with the technology). Two provided reverse proxy server technologies were the default Apache and Nginx. After a short research through forums we decided to make the switch to Nginx as it is more modern alternative to Apache and can have performance benefits. If needed, we could always switch back to Apache without much hassle.

The provided template for the Nginx configuration was a good starting point, but in order to make the application functional, we had to make a few additions to the afantazie.cz.conf file:

- Reroute the API requests to the backend
- Set up redirection from HTTP to HTTPS
- Force the server to always provide a new version of the meta.json file used for cache busting (see Section 4.2.3)

For a complete configuration file, see Section 6.2.2.

4.3 Small Graph Vizualization

Small graph implementation, the second phase of development, required two things - rendering and forces simulation. As mentioned previously, we decided to render the graph using PixiJS and implement our proprietary FDL.

To integrate PixiJS into the React application, we first followed the official documentation [14]. We were not happy with the result as the Pixi/react components were hard to work with programmatically.

So, instead, we used a guide by Adam Emery [15] to integrate PixiJS into the React application. His approach is different in that it only uses the Stage component from the Pixi/react library , and the rest of the PixiJS code is written in plain JavaScript (in our case, TypeScript). This allows us to better control the PixiJS code and use the full potential of the library instead of using the pixi/react wrapper.

After implementing a simple node and edges rendering, we quickly hit a roadblock with massive memory leaks. As we found out, we were instantiating new PixiJS objects every time the graph was updated. To mitigate this issue, we created an interface - RenderedThought - that holds the PixiJS objects for each thought (the Title text and the Circle graphics).

The RenderedThought interface remained to the end of the development and we incrementaly added more properties to it as we needed them. It contains all the properties that are needed to render the thought on the screen and to interact with it. For example, each rendered thought has a boolean property 'held' which is used to determine if the thought is currently being dragged by the user.

Before big graph solution we stored the loaded thoughts in an array of rendered thoughts kept in the React state.

Once we were able to render nodes and edges, we created a simple force directed layout algorithm with two forces - pull connected and push unconnected.

At this point, we had an application with a comparable graph view to obsidian (Figure 4.3).

Figure 4.4 showcases the first 3000 nodes of the citHep dataset now visualized using Aphantasia in this stage of development. With this many nodes on the screen, the application lagged a lot, although it remained somewhat usable.

4.3.1 FDL Implementation

The core of our FDL implementation is a set of two forces - pull and push. These forces are defined as functions accepting distance between the nodes and returning a force that should be applied to them.

These functions are defined in the graphParameters.ts file and currently look like this:

```
export const pullForce = (borderDist: number) => {
    if (borderDist <= 0) {
        return borderDist;
    }
    const computed = 0.01 *
        (borderDist - IDEAL_LINKED_DISTANCE);
    const limited = computed > MAX_PULL_FORCE
        ? MAX_PULL_FORCE
        : computed < -MAX_PULL_FORCE
            ? -MAX_PULL_FORCE
            : computed;

    const final = Math.sign(limited) === -1
        ? limited / EDGE_COMPRESSIBILITY_FACTOR
        : limited;

    return final;
};
export const pushForce = (borderDist: number) => {
    if (borderDist === 0) {
        return 0;
    }
    if (borderDist < 0) {
        return -borderDist;
    }
    const computed = 5 / Math.sqrt(borderDist);
    return Math.min(MAX_PUSH_FORCE, computed);
};
```

Note that the pull force can be negative. This is intentional as the connected nodes should be attracted not towards each other but towards a certain ideal edge distance. This distance is parameterized as IDEAL_LINKED_DISTANCE.

The numbers inside the functions are the respective force strength parameters.

4.3.2 Parametrization

As stated in Section 1.4, the FDL algorithm is highly dependent on the parameters set by the user. Thus, as expected, we had to implement and configure number of parameters in order to make the graph view a smooth and enjoyable experience. The full list of parameters we implemented can be found in Administrator documentation in Chapter **??**.

Here, we will discuss what led us to implement many of these parameters - jitter. In some situations, particularly when the graph was not yet fully stabilized, and there were a larger amount of nodes in a small area, the nodes had the tendency to oscillate quickly.

Parametrizing the forces themselves helped a little but not enought to mitigate the issue. Instead we implemented a momentum system, where the nodes would not react to the forces immediately but would instead gradually accelerate based on forces applied. The momentum system is influenced by dampening system - a set of parameters controlling how much the forces applied to the momentum and the momentum itself are reduced each frame.

The jitter was not completely eliminated but was reduced to a level where it was not noticeable in normal use. We also believe that if the problem arises again, we will be able to alter the parameters accordingly and reduce the problem further.

4.4 Big Graph Visualization

In the final stage of the development, we implemented the big graph solution.

State management

So far, we used React states and contexts to hold and manage the the graph context. This worked for a simple small graph approach, but we were not happy with this approach once the context started to become more complicated as we had little control over the state from the PixiJS code.

A solution we used is a react library called Zustand [16]. This library allows minimal state management in React that is compatible with external (meaning non-React) code.

Dynamic loading

At the beginning of the development, we used one endpoint for all the data at once. That worked until around 500 thoughts, after which the loading time started to become noticeable.

The obvious solution was to load only a subset of the data at a time. We implemented this idea in two ways:

- **Temporal API endpoints** - Client requests a new subset of nodes in the form of ”beforeId / afterId / aroundId”. Thanks to the ascending ID increment approach, it translates to the chronological order of the nodes. When the time window exceeds the currently loaded data, it gets updated with missing nodes from the relative past or future.

- **Neighborhood API endpoints** - Breadth-first search starting in a given node up to a given depth is used to implement graph exploration.

This approach worked not just as an optimization technique but we ended up building our big graph handling on it. In Figure 4.5, we can see the logic flow of

the big graph rendering solution. Aphantasia uses two arrays of rendered thoughts - temporal array and neighborhood array.

To keep track of which thoughts should be visible on screen as well as fetching and updating temporal and neighborhood thoughts, we created the file thoughtsProvider.ts.

4.4.1 Frontend Architecture

The frontend architecture is much less clear-cut than the backend but we visualized the relationship between the main source code files in Figure 4.6. All of the graph-related code sits inside the src/pages/graph directory of the AfantazieWeb project and the containers in the diagram correspond to the folder structure.

Let us look at the individual parts of the frontend architecture:

React

Graph Page contains the UI elements of the graph view:

- Content preview
- Controls (zoom and time slider)
- Time window date label
- New thought button

It also handles much of the initialization logic, such as getting the thought ID from the URL and fetching the appropriate data (either the latest or around the requested thought).

Graph Container This component is the bridge between React and PixiJS code. It calls the run() method of the **Graph Runer**.

Simulation

In the Simulation directory, we find three files:

- **Graph Runner** - Contains the main logic loop of the graph simulation. Its run() method is called by the Graph Container, and it initializes the graphics, runs the render function and the simulation functions.
- **Forces Simulation** - Custom FDA implementation.
- **ThoughtsProvider** - Responsible for providing the temporal and neighborhood thoughts as well as fetching data from the backend based on time window position and graph exploration state.

The source code of the main logic loop of the application (located in the file graphRunner.ts) is shown below. Note that the code is simplified for the sake of brevity.

```
export default function runGraph(app: Application) {
    const renderGraph = initGraphics(app);
    useGraphStore.getState().setFrame(0);

    // main application loop
    app.ticker.add((_) => {
        const graphState = useGraphStore.getState();

        // cache thoughts
        if (graphState.frame === THOUGHTS_CACHE_FRAME) {
            ...
            localStorage.setItem('thoughts-cache',
                JSON.stringify(thoughtsCache));
        }

        // Update temporal thoughts if needed
        updateTemporalThoughts();

        // FDL simulation
        const frame = graphState.frame;
        if (frame < SIMULATION_FRAMES) {
            simulate_one_frame();
        }
        graphState.setFrame(frame + 1);

        // render the graph
        renderGraph();
    });
}
```

Graphics

This directory contains two files:

- **Graphics** - Contains the initializeGraphics() method which returns a callback function render() called by the Graph Runner every loop.
- **ViewportInitializer** - Contains the addDraggableViewport() method which sets up the viewport for the graph.

State and parameters

Here we find two files:

- **Graph Store** - Zustand store for the graph state.
- **Graph Parameters** - Constants used in the graph view, FDL and other behavior.

4.5 Final List of Features

This section provides a deeper look into the user-facing features and technical aspects of the application. More user-focused documentation will be available in Chapter 6.1.

4.5.1 AuthX/Y

The application includes user account functionality supporting authentication and authorization. Users can register, log in, and log out. Most pages and features are accessible only to logged-in users.

Passwords are hashed using SHA256 encryption and must meet minimal requirements:

- At least eight characters
- At least one uppercase letter
- At least one lowercase letter
- At least one number

Login is managed via JWT stored in local storage. Tokens expire after one day, and a refresh token mechanism is not implemented, requiring users to re-login every 24 hours.

Tokens are currently sent in URL parameters for SignalR WebSocket connections. This practice is not ideal and should be replaced in the future. A possible solution involves a ticketing system where a logged-in client would request a ticket from the API. This ticket would then authorize the WebSocket connection, avoiding token exposure in the URL.

4.5.2 Pages

Thanks to React, the graph view is one of many pages in the application. There are currently ten pages:

- **Homepage**: Displays a feed of recent thoughts and navigation buttons (Figure 4.7)
- **Welcome Page**: Similar to the homepage but tailored for unregistered users (Figure 4.7)
- **About Page**: Provides information about the project
- **Chat Room**: A basic real-time chat
- **Settings Page**: Contains user settings and a logout button
- **Login and Registration Pages**
- **Notifications Page**: Displays replies from other users
- **Graph View**: Enables users to view thoughts
- **Thought Creation Page**: Allows users to create new thoughts

4.5.3 Localization

Initially, we developed a Czech version of the application and later added an English version. For frontend localization, we utilized two JSON files and a Localization object which returns either Czech or English text based on Vite configuration. This makes the application easily extensible to more languages.

Backend localization was also required, as the API returns localized authentication and validation messages. To achieve this, we created a Localization project with classes implementing localization interfaces. During bootstrapping, a specific localization is registered based on configuration. While this approach is somewhat cumbersome, it suffices for the limited localization needs of the backend.

4.5.4 Custom Graph Rendering Engine

The graph view is the primary feature of Aphantasia and the most complex part of the application. Its basic features include:

- **Zooming and Panning**
- **Dragging Thoughts**
- **Floating Text Titles** (Figure 4.9)
- **Thought Highlighting** (On Figure 4.8 and 4.9)

On-screen Thoughts Limit

The On-screen thoughts limit is a critical part of our big graph rendering solution. The default value is 100, but users can adjust it in the settings.

The idea behind it is to always render, at most, this number of thoughts on screen. And to view more user input is required - either by moving the time slider or by using the graph exploration feature, both of which we will talk about briefly.

The limit is demonstrated in Figure 4.11 and 4.12 with the values set to 300 and 700, respectively.

Time Slider

Combining the On-screen thoughts limit, dynamic loading, and two UI buttons resulted in a feature we call the Time slider. It allows users to move a conceptual time window smoothly into the past or future by holding the corresponding button. The resulting layout using the time slider is demonstrated in Figure 4.11 with 641 thoughts on afantazie.cz viewed in three different time windows of length 300.

Above the time slider controls, there is a label showing the current time window's position - the creation date of the newest thought on the screen. The label can be seen in the bottom left in Figure 4.8.

New thoughts appear either in their cached positions (see Layout caching section below) or in a circular pattern around the simulation container's center. When not yet cached, the appearance of newer/older nodes creates a visually appealing effect as the thoughts gradually appear in what resembles a loading spinner.

Live Preview

When the time slider moves beyond the last thought, the application enters live preview mode indicated by 'Now...' apearing in place of the time window's date in bottom left. In this mode, the client listens to new thoughts and adds them to the graph in real-time.

While this feature enhances interactivity, it has only been tested with two users creating thoughts simultaneously. Higher activity levels could potentially overwhelm the system, but until there is an active user base, this remains a theoretical concern.

We implemented this feature using long polling, meaning that the client periodically sends requests to the server to check for new thoughts. This approach is a bit wasteful, especially considering that we already have an active signalR connection to use across the entire application. In the future we will replace long polling with the active signalR connection.

Graph Exploration

The neighborhood API endpoint powers graph exploration demonstrated in Figure 4.8. After clicking on a node, link, or reply, the client loads the neighborhood of the newly highlighted thought, enabling interactive exploration.

In the example and many other screenshots provided in this work, some thoughts appear hollowed out. This effect triggers when the thought's direct neighbors (links or replies) are not currently rendered on screen, signaling that there is more to explore behind it. When the neighbors of a node are all visible, the node is filled with its author's color.

Currently, the graph exploration neighborhood array does not respect the on-screen thoughts limit and loads all neighbors of the highlighted thought up to a given depth. This didn't pose a problem with the datasets we used but could be a significant issue with highly connected datasets.

Thoughts Layout Caching

The browser's local storage caches the thoughts layout after a period of inactivity. The length of this period is parametrizable by a number of frames, with the current value set to 1 000 frames, which corresponds to around 30 seconds of inactivity on most devices. When a thought leaves the screen and later reappears, it retains its previous position. This feature facilitates graph stability during time sliding and between sessions and removes the need for the graph to stabilize again and again.

Paired with the time slider, this approach produced an unexpected emergent behavior. As the production dataset grew beyond 500 thoughts (five times the default on-screen limit), it remained possible to create a stabilized graph across the entire dataset. Moving the time slider across such a stabilized layout is a uniquely satisfying experience, which we believe sets Aphantasia apart. To some extent, this feature is visible in Figure 4.11 but it is best experienced in the live application.

The cache currently has no size limit and is not cleared automatically, which could be a potential issue with big datasets and longer graph view sessions. Logged-

in users can, however, delete cached positions in settings to force the graph to re-stabilize.

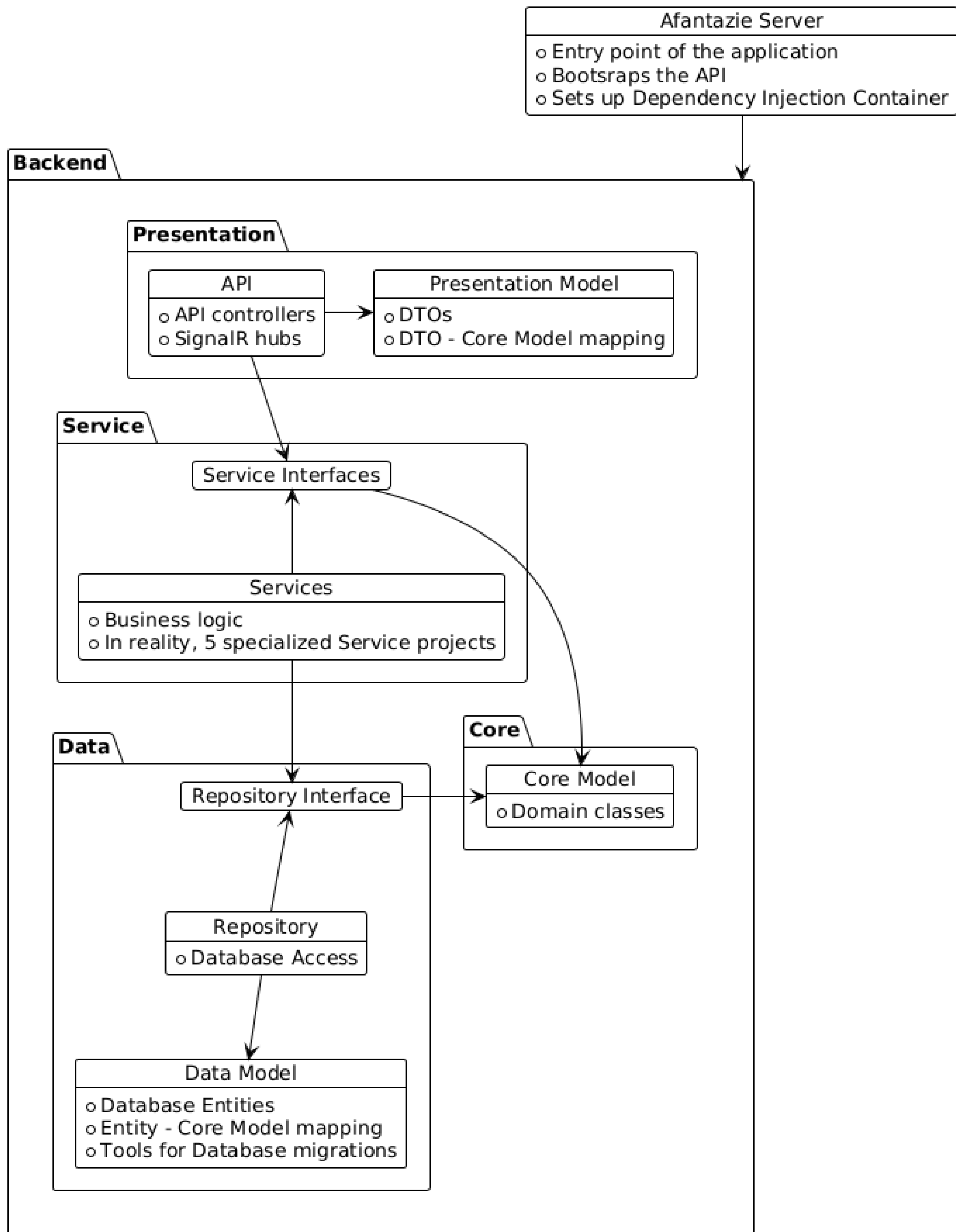

Figure 4.2 Aphantasia Backend Architecture

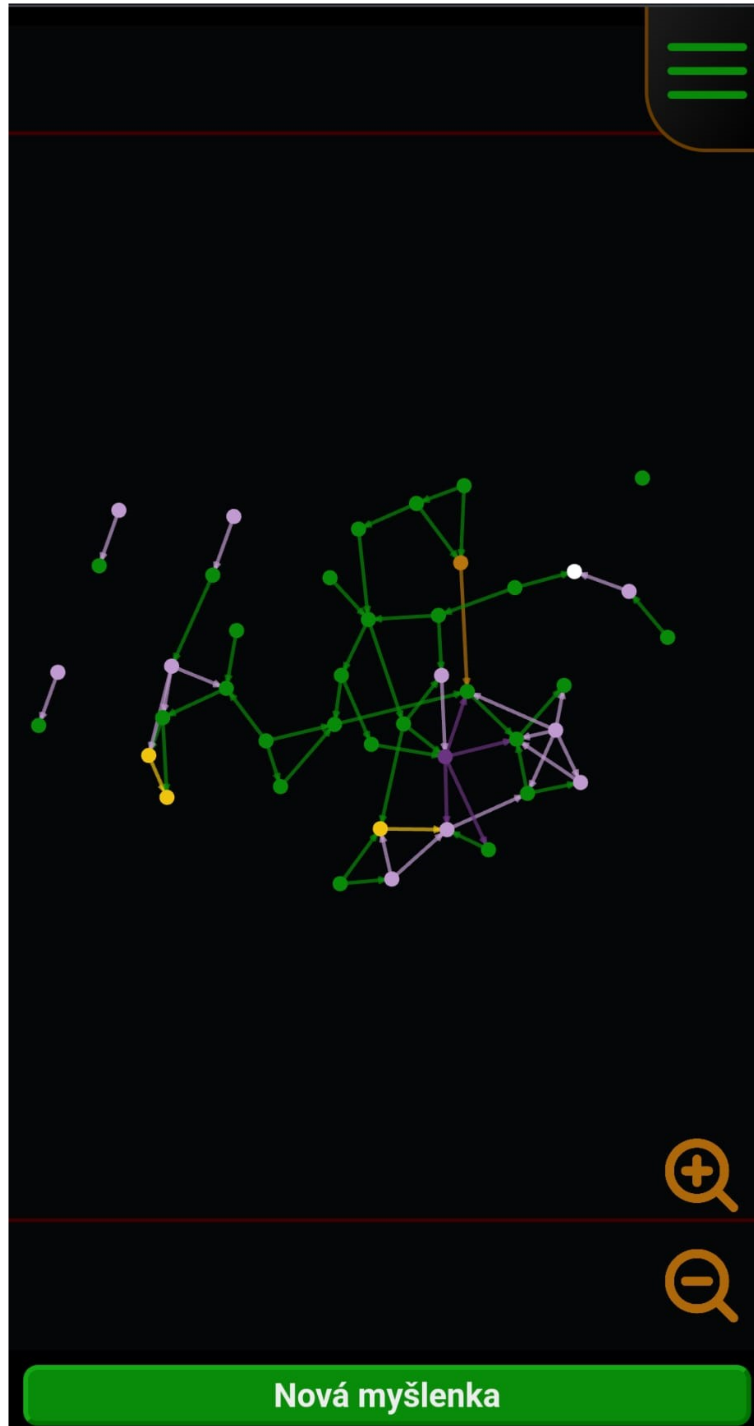

Figure 4.3 Aphantasia graph view at the end of the small graph development stage

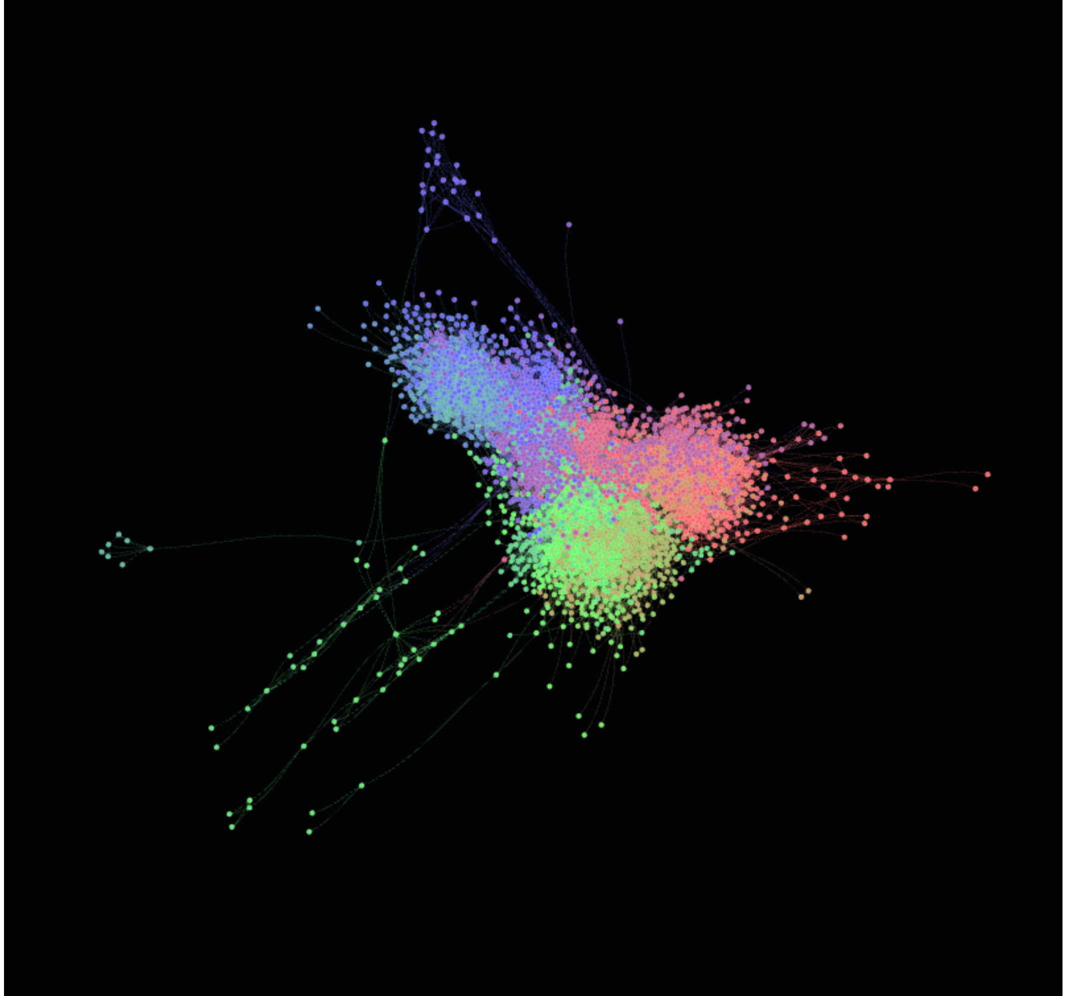

Figure 4.4 The first 3000 nodes of the CitHep dataset visualized in Aphantasia (before big graph solution)

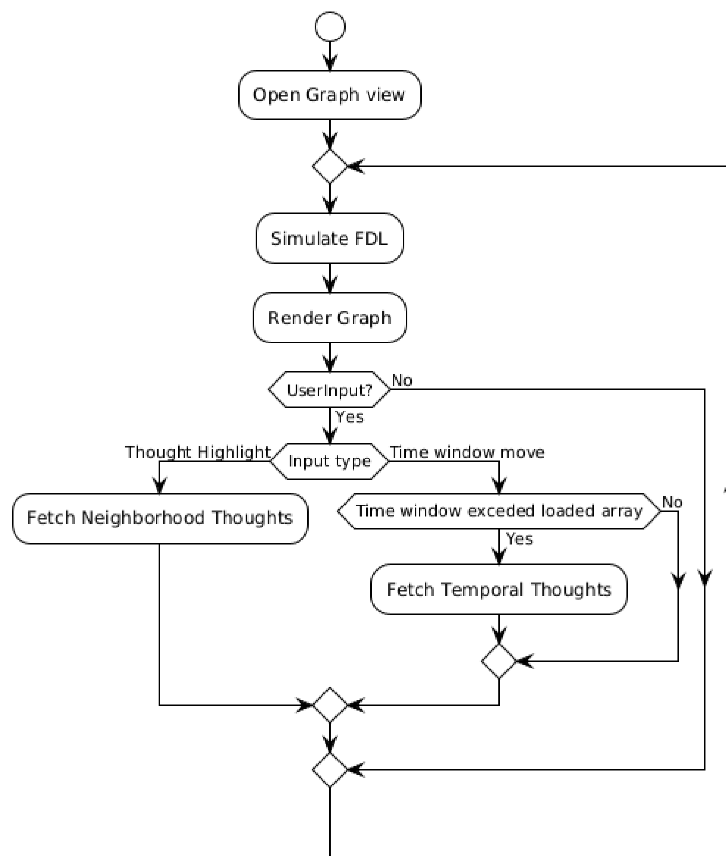

Figure 4.5 The logic flow of the big graph rendering solution

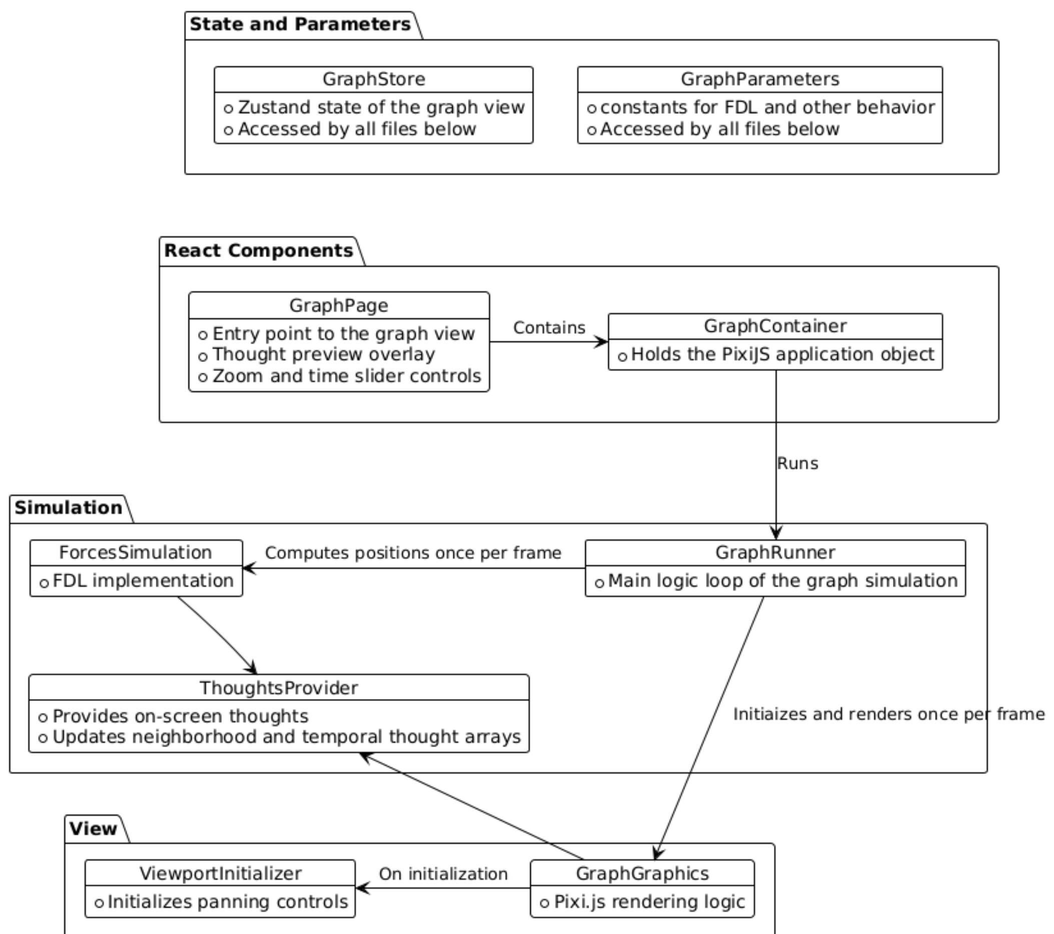

Figure 4.6 The frontend architecture of Aphantasia

Figure 4.7 The Welcome page and homepage of Aphantasia

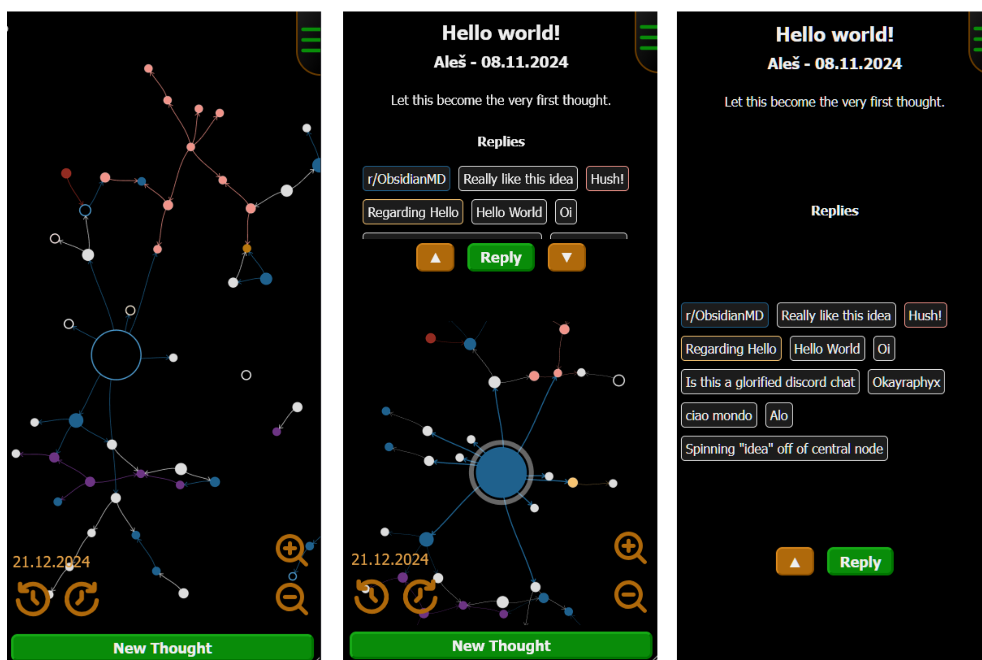

Figure 4.8 The graph view on a mobile device - non-highlighted mode, half-screen preview, and fullscreen preview, respectively

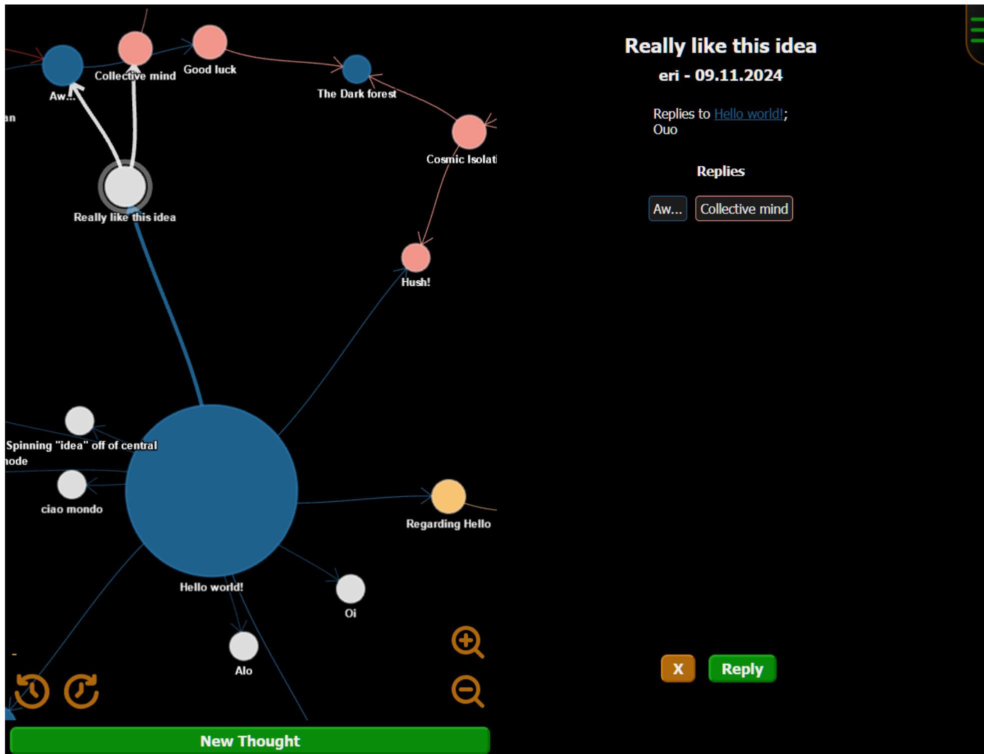

Figure 4.9 Floating titles in the graph view on desktop

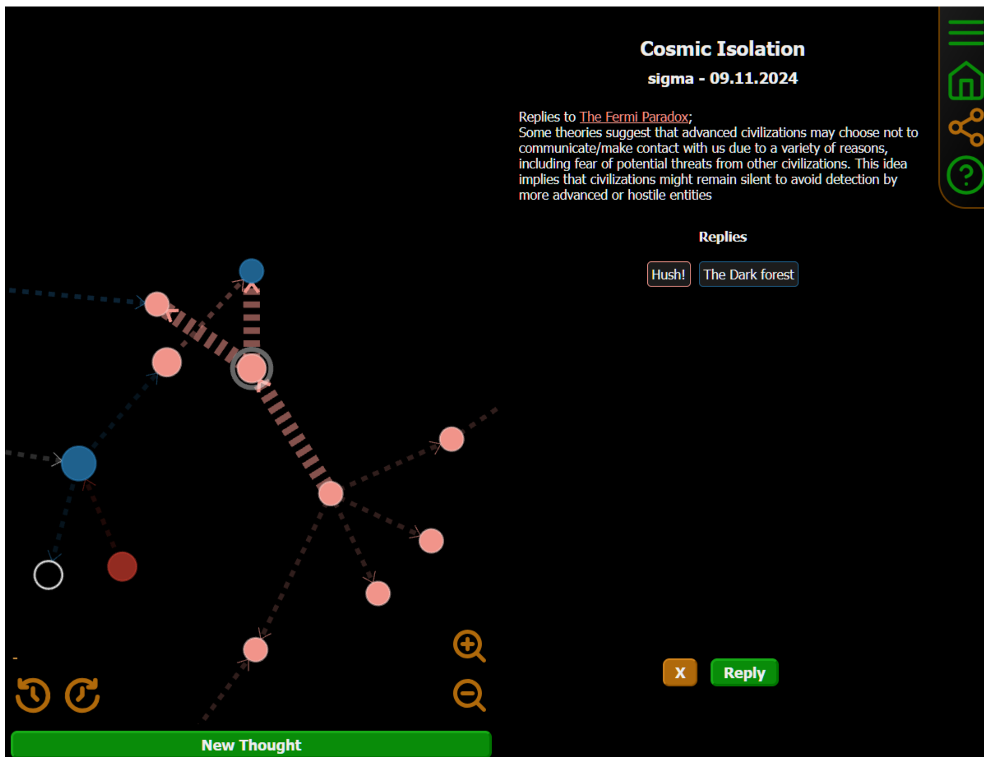

Figure 4.10 Animated edges in the graph view

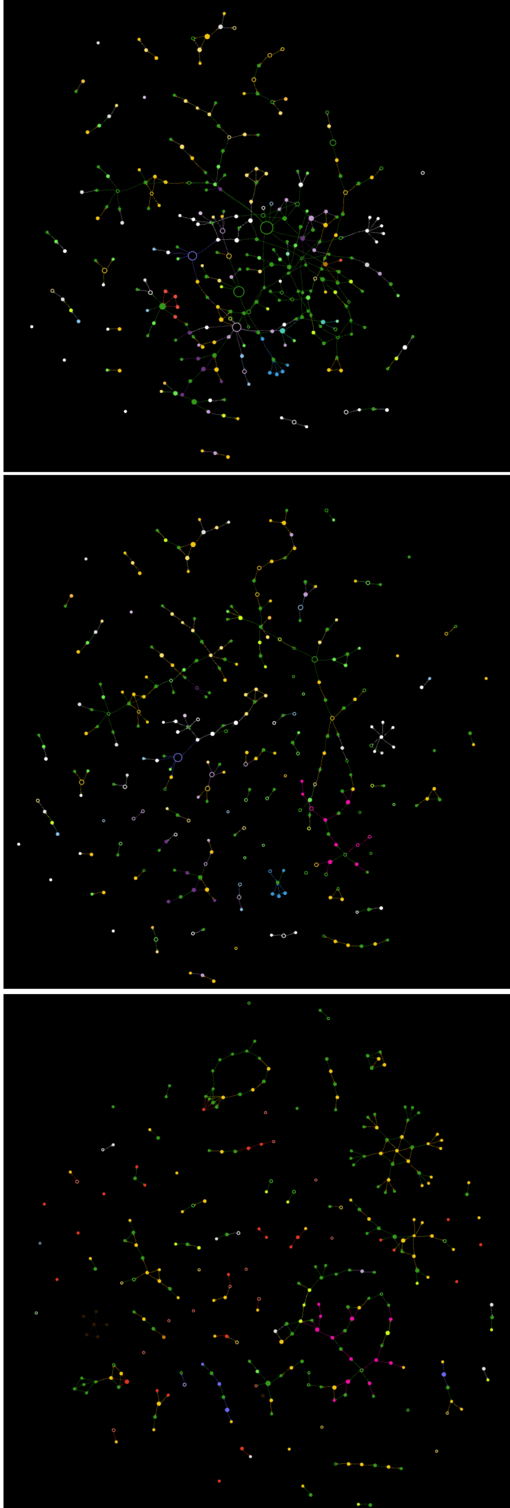

Figure 4.11 Aphantasia with the Czech production dataset in stabilized temporal layout (641 nodes in three time windows of length 300)

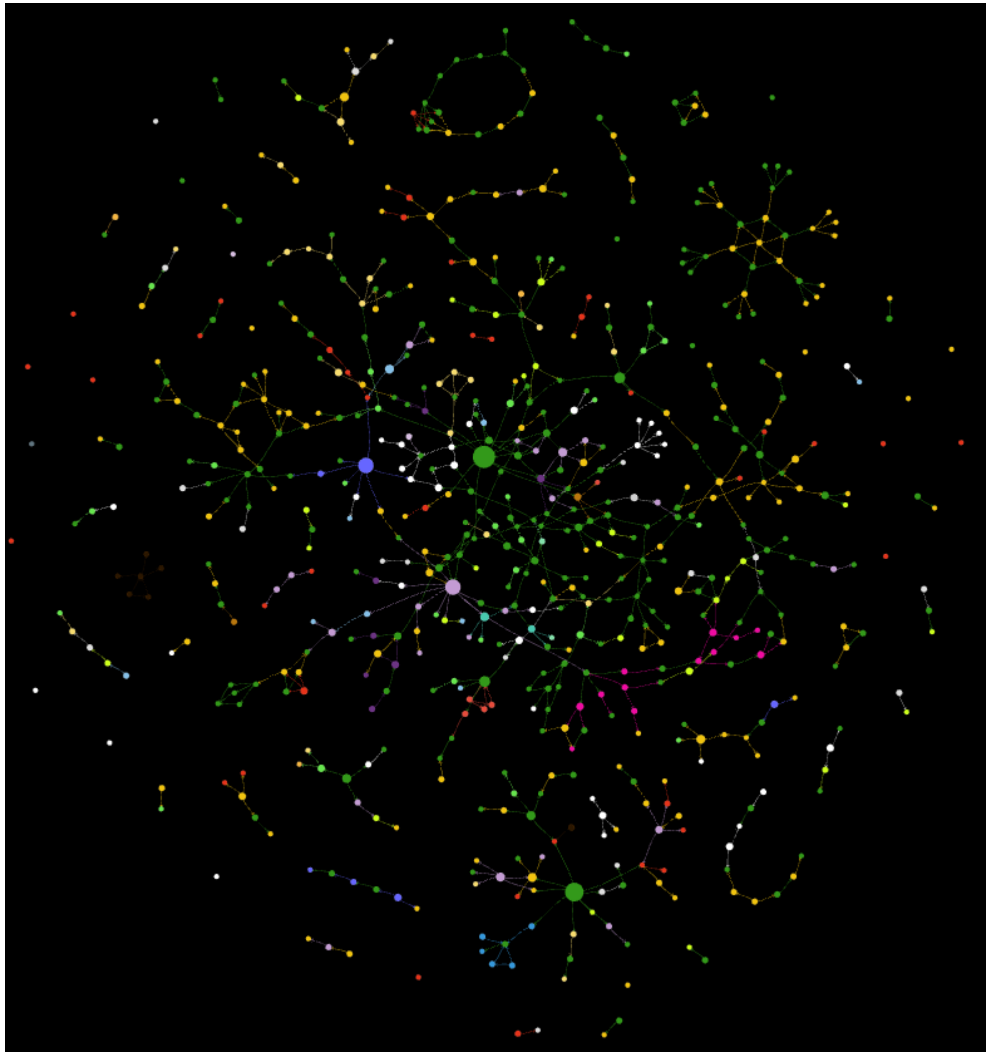

Figure 4.12 The entire dataset of afantazie.cz (641 nodes) in a single time window

5 Testing

5.1 Graph View Performance

The graph can handle the entire CitHep dataset and, thanks to dynamic loading, could handle even larger graphs. In Figure 5.1, one can see a highlighted thought from the CitHep dataset.

To achieve this, we wrote a C# script to import the CitHep data files into the Aphantasia database. In the graph view, we used the same parameter settings we used for the much smaller graphs in production environment, which at the time consisted of around 600 and 150 thoughts, respectively. To our surprise the resulting graph behaved well and except for increased tendency to jitter the CitHep graph view was smooth and stable. The only parameter we had to change was neighborhood BFS depth - from three down to just one. CitHep graph is bigger and much more interconnected than the small production graphs, so even at depth two, the graph exploration feature often resulted in an unreasonable amount of on-screen nodes.

5.1.1 Limits of the Graph View

In Figure 4.4, we have seen how Aphantasia handles 3 000 nodes even before the dynamic loading was implemented.

The application is not meant to display large graphs at once, but out of curiosity we tried to render as many thoughts as possible.

First, we tried to set the on-screen thought limit to 40 000 (and thus load and render the entire CitHep dataset) and we were not able to fetch the data from the backend. We are not sure why this happened but one possible explanation is that the API response is too large and either the server and/or the client were not able to handle it.

In the second test we set the on-screen thought limit to 10 000 and we were able to fetch the data and render it. The result was not unexpected - a big hairball of nodes and edges running at less than one frame per second. See Figure 5.2.

From our tests, we learned that Aphantasia's performance begins to degrade noticeably at around 400-700 on-screen thought limit and gradually drops to just a few frames per second with the limit set to around 1 500. These values are highly dependent on the connectivity of the data, with more connections leading to more computation time and thus lower performance.

5.2 User Feedback

We advertised the application on Reddit, sharing both the Czech and English versions across several subreddits.

Most comments were positive, praising the application's concept and the experience of exploring thoughts with one user describing the experience as feeling like an ”archeologist” uncovering ideas.

Negative feedback focused on the lack of practicality, outdated UI, and occasional bugs. Users also suggested missing features such as pinch zoom, image

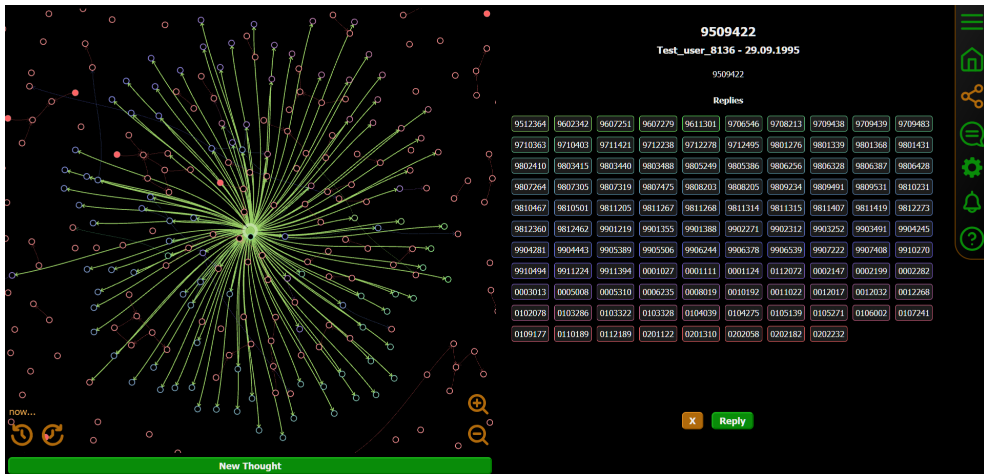

Figure 5.1 A highlighted thought of the CitHep Dataset in Aphantasia

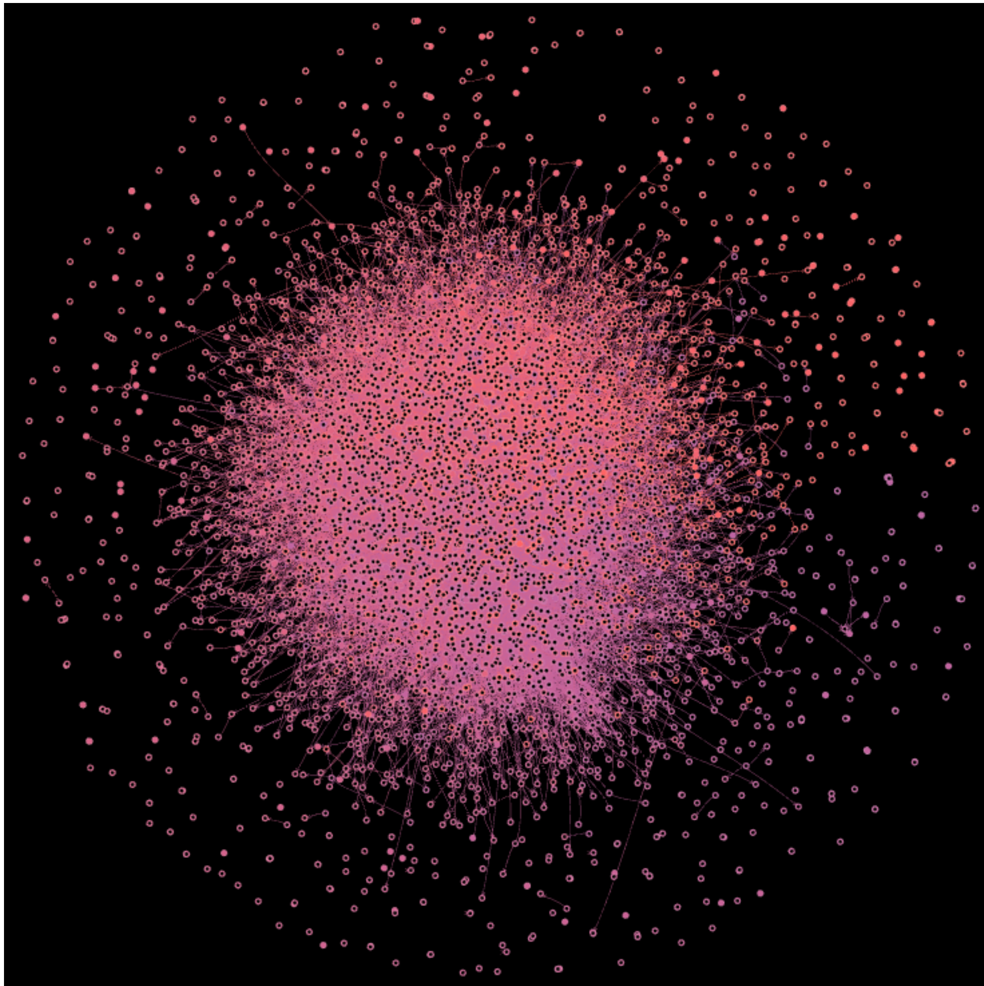

Figure 5.2 Cithep dataset rendered in Aphantasia with 10000 on-screen thought limit

posting, a better landing page, and filtering. Browser compatibility issues were mentioned as well.

The feedback was mostly constructive, and we are grateful for it. It helped us hone on the experience users already found fun and stimulating, specifically interaction and exploration. However, some users admitted they did not understand the application, highlighting the need for a proper tutorial.

5.3 Aphantasia Versus Related Software

Ideologically, Aphantasia is closest to Obsidian. There is of course an obvious difference between the two - Obsidian is local note-taking system with graph view while Aphantasia is an online social experience based on graph view. Both are, however, meant for the general audience, and their graph views bear a similarity.

Compared to Gephi and Cytoscape.js, Aphantasia has lower performance on large graphs rendered at once. However, thanks to the time slider and graph exploration features, it can handle much larger datasets in an intuitive way. The dynamic loading could, in theory, handle millions of nodes. In such case the main limiting factor would be the performance of the backend and the database.

Finally, in table 5.1, we added Aphantasia to the comparison table from the first chapter. We can see that Aphantasia is a good compromise between the ease of use of Obsidian and the performance of Gephi and Cytoscape.js.

	Obsidian	**Gephi**	**Cytoscape**	**Aphantasia**
Use-case	note-taking	data analysis and visualization	graph visualization in browser	social network
Target Userbase	general audience	researchers, technical users	web developers	general audience, graph enthusiasts
User Experience	easy to use	technical, steep learning curve	programmatic, mostly parametrization	intuitive, slight learning curve
3 000 nodes handling	slow indexing but smooth afterwards	stable, smooth	stable but visibly lower FPS	stable, smooth, explorable
34 546 nodes handling	skipped as indexing took too long	mostly stable, lower FPS while running FDL	crashed immediately	stable, explorable, slightly increased jitter

Table 5.1 Comparison of Obsidian, Gephi, Cytoscape.js and Aphantasia

6 Documentation

The remaining chapter is dedicated to the documentation of Aphantasia.

6.1 User Documentation

All of the pages are accessible from the homepage and/or the collapsable navbar in the top right.

6.1.1 Registering and Logging in

While not logged in, only the graph view, about page, and welcome page are accessible.

To register, click on the register button on the home screen. The registration requires selecting a unique username, email, and password. The password has security requirements which are displayed immediately after opening the form and in case of not meeting the criteria on submit.

To log in click on the login button on the homescreen. The login requires a username or email and password.

At this point, the email is not used anywhere and is only included in preparation for email verification functionality.

6.1.2 Opening a Thought

There are multiple ways to open a thought:

- **New thoughts log on homepage** - On the homepage, there is a feed of the last three thoughts created on the website. Clicking on one of them will open the thought in graph view.

  Clicking on the "All Thoughts" button under the feed leads to the list of all thoughts.

- **Notifications** - The Notifications button and the bell icon in the navbar lead to the list of replies. Replies are thoughts of other users linked to any of the logged-in user's thoughts. Clicking on a reply opens the respective thought.

- **Graph view** - The main way to access thoughts is through the graph view. We will take a closer look at it in the next section.

- **Direct link** - Every thought has a unique ID which can be shared and accessed directly using the URL in format '/graph/{thoughtId}'.

  Example of the full URI leading to the thought with ID 1: https://aphantasia.io/graph/1

6.1.3 Graph View

To access the graph view, click the ”Graph” button on the main page or click on the graph icon in the navbar (three connected nodes).

In the graph view, the following controls are available:

- **Mouse wheel or bottom right buttons** - Zooms in and out

  Once zoomed past a threshold, titles of the thoughts appear. (Figure 4.9)

- **Draging the background** - Pans the viewport

- **Dragging a node** - Moves the grabbed thought around

  This is useful for customizing the layout, ”untying” thoughts that are too close to each other or to speed up the process of the layout algorithm.

- **Clicking a node** - Highlights a thought and switches to highlighted mode.

Highlighted Mode

In default mode, the whole display is used to view the graph, and the user can interact with it as described above. When a thought is accessed, the graph view switches to highlighted mode. (Figure 4.8)

In highlighted mode half of the screen gets dedicated to the highlighted thought preview and the other half to the graph view.

The graph view in highlighted mode shares almost all behavior with the default mode, with a few exceptions:

- **Visual node highlight** - The currently opened thought is also visually highlighted by a white circle around it.

- **Visual edges highlight** - All edges connected to the opened thought are exaggerated in thickness and color while all other edges are dimmed.

- **Neighborhood thoughts** - On highlighting a thought, the application loads its neighborhood.

Every time a thought is selected, the viewport also smoothly centers on it.

The thought preview shows the title, author, time of creation, content with clickable links, and replies section. Both the links in content and titles in the replies section are color-coded based on the author’s selected color, and one-click will highlight the respective thought. At the bottom of the preview there are Reply button and a close button (up icon on mobile and X on desktop).

Neighborhood Thoughts and Graph Exploration

When a thought is highlighted, the application loads its neighborhood. The neighborhood is defined as thoughts accessible through BFS up to a given depth. The currently used depth of the search is fixed at 3.

Some of the thoughts rendered on screen can be filled with black color. 4.8 This indicates that some neighbors of the node are not visible on screen, and thus, the node is explorable.

6.1.4 Creating a New Thought

There are two ways to access the thought creation page:

- **From the graph view** - Click on the ”New thought” button at the bottom of the graph view.
- **From the thought preview** - Click on the ”Reply” button at the bottom of the thought preview.

Both of these ways lead to the thought creation page (Figure 6.1). The difference between them is that when accessed through the Reply button, the respective thought link is automatically added to the content input.

The thought creation page consists of two text fields: Title and Content.

Both the title and content are required. Title has a minimum length of 1 character. The content’s minimum length is 5 characters. Both of these requirements are enforced by validation rules, and the user is notified by notification messages if the submit fails.

Referencing Other Thoughts

When a link (or a reference) is added to a thought, it will be connected and attract the corresponding node in graph view. Links can be added to the content field as a link with the format ”[id](text)”, where id is the id of the thought and text is the text that will be displayed. The text does not have to necessarily be the original title of the linked thought.

Adding links can be done in three ways:

- **Manually** - By typing the link in the content field
- **By ’Add reference’ button** - This button opens an overlay with a list of thought titles and a search bar. Click on a thought title to add it to the content at the cursor position.
- **Reply** - As mentioned, the Reply button in the graph view adds the link to the respective thought automatically.

Each thought can have up to five references.

6.1.5 Settings

The settings page (Figure 6.2) is accessible from the navbar by clicking on the gear icon or by clicking the ”User settings” button on the homepage. Currently, there are two settings that can be changed:

- **On-screen thought limit** - This setting changes the maximum number of thoughts that are displayed on screen at once. The default value is 100. but can be changed to virtually any positive value.
- **Color** - This setting changes the color of users’ thoughts in the graph view. There are predefined colors accessible by clicking the username but users are free to choose any color they like using a hexcode.

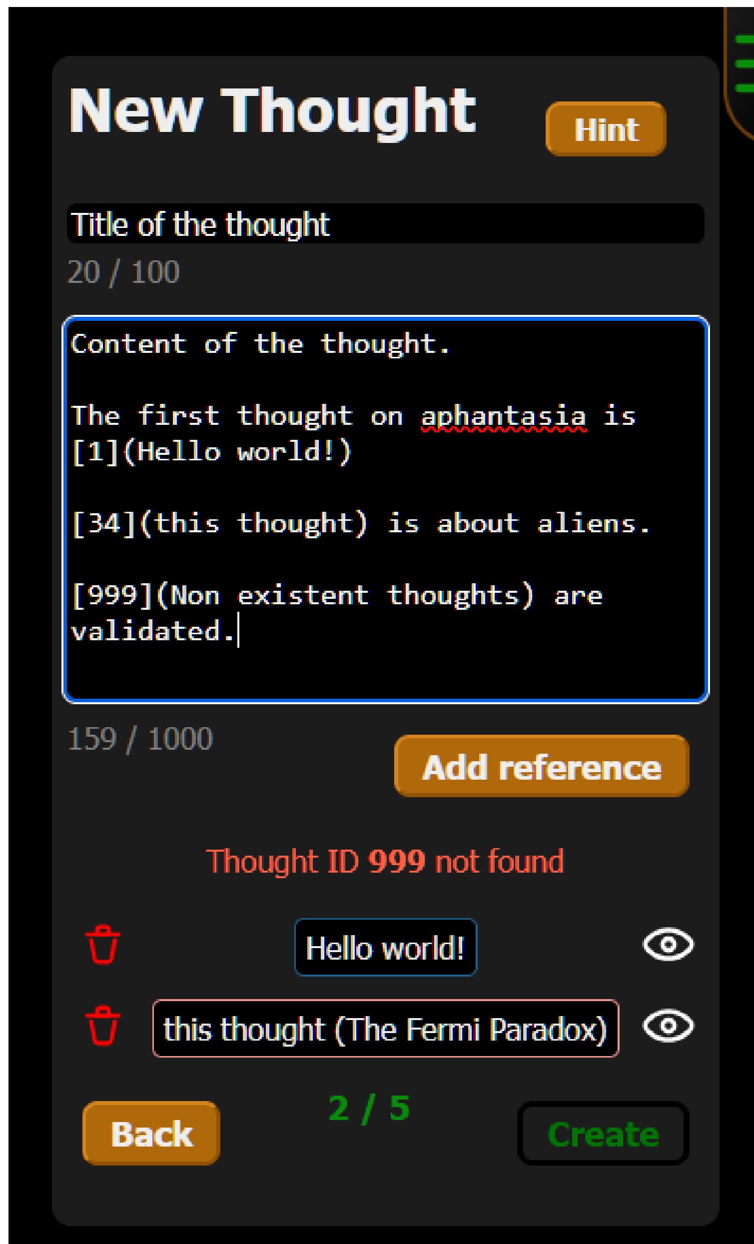

Figure 6.1 Aphantasia - Thought creation page

Apart from these settings, there is also a **Log out** button and **Delete thought positions** button, which erases positions of thoughts saved in the browser - forcing the graph to restabilize on the next load.

6.2 Installation Guide

Here we will look at how to set up and run both locally and how to deploy a public instance.

6.2.1 Running Aphantasia Locally

To run Aphantasia locally, follow these steps:

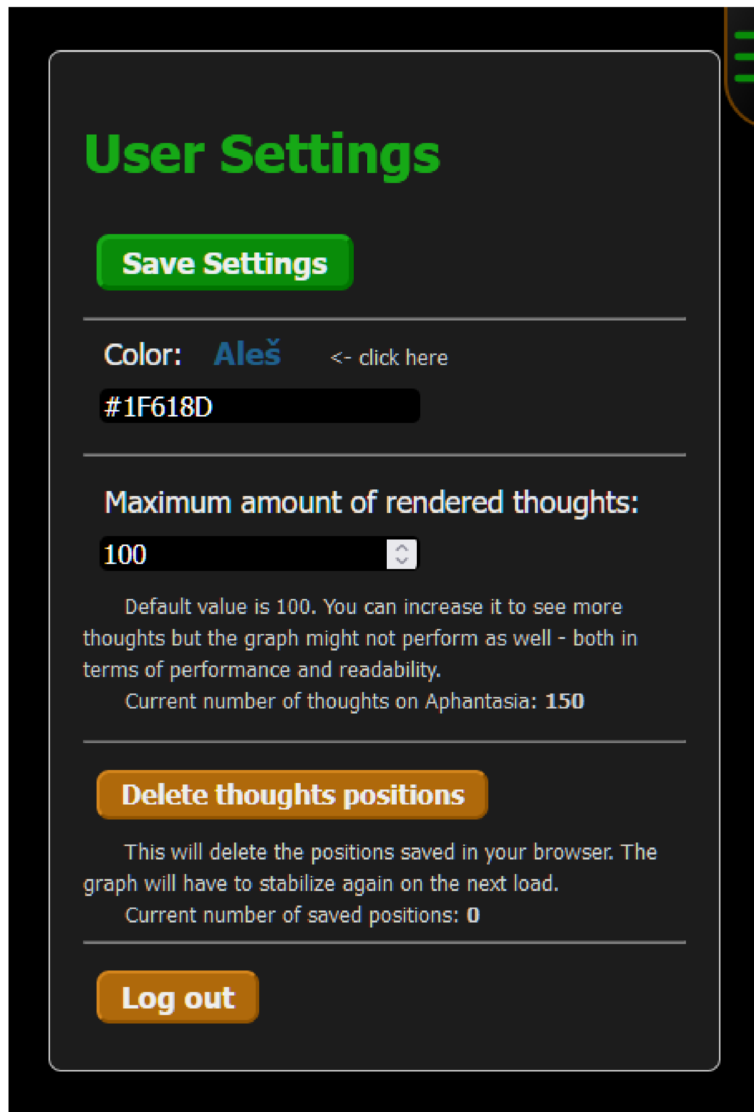

Figure 6.2 Aphantasia - User settings

Database

The database is necessary for the application to run. It is set up using Entity Framework Core and PostgreSQL but one can choose any other database engine supported by EF Core. To use a different database engine, install the necessary packages and replace the **UseNpsql** methods in these two files appropriately:

- **Afantazie.Data.Model/DatabaseContextProvider.cs**
- **Afantazie.Data.Model/DesignTimeDataContextFactory.cs**

With the database running the next step is to scaffold the database (ie. create the schema compatible with Aphantasia). To do so, follow these steps:

1. Add a file named **migrationsettings.json** in the root of project **Afantazie.Data.Model**.
2. Add the following content to the file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "DATABASE CONNECTION STRING"
  }
}
```

3. Run the following command in the Data.Model project root:

```
dotnet ef database update
```

The command should print the connection string provided in the migrationsettings.json and ask for confirmation. If everything is correct, confirm the prompt by inputting ”y”, and the database should be created.

Frontend

Before running the frontend make sure to have Node.js with npm installed on the machine. Open the root of the AfantazieWeb directory in a command line and run:

```
npm install
npm run dev
```

The application will then be accessible at the address seen in the console output. The language of the application can be changed by replacing **VITE_LANGUAGE** to either ’en’ or ’cz’ inside the .env.development file.

Backend

Before running the backend, make sure to have .NET 8 SDK installed on the machine.

To run the Aphantasia backend, we recommend using Visual Studio and following these steps:

1. Open the AfantazieServer.sln solution file in Visual Studio.
2. Right-click the AfantazieServer project and click on Manage User Secrets.
3. Copy the JSON with the connection string we provided in the database setup section and paste it into the secrets.json file.
4. Set the AfantazieServer project as a startup project and run it.

The project can be configured in the appsettings.Development.json file, but it is not necessary for the application to run.

Testing Data

Apart from adding testing data manually throught the appliaction or directly inserting into the database one can also use the Afantazie.Tools project. It is a .NET 8 console application inside the AfantazieServer solution we created to help with various tasks regarding Afantazie development. Among other things, it can generate random thoughts and users and add them to the database.

Before running the tool to generate thoughts, follow these steps:

1. Host and scaffold a database as described above.
2. Manually remove foreign key constraints from the ThoughtReferences table. [1]
3. Create an appsettings.json in the root of the project. Its content should be the same as the migrationsettings.json shown above.

Then, set the project as the startup project and run it. Again the tool wil print the connection string and ask for confirmation after which it will present several options to pick from. There are a few random data generators, but we recommend using option 3 - 'Generate random rainbow thoughts'.

This option will generate random clusters of thoughts where each generated thought has a set probability of being linked to a thought in a different cluster. User will then be asked for various parameters - fill them in and let the tool run. Once it finishes, the database will be filled with random clustered thoughts and users with unique colors across the color spectrum.

6.2.2 Deploying Aphantasia

To host Aphantasia publicly, we recommend using a Linux server (We used Debian 12.8).

Set up the database using the same steps as in the local setup. The database can be hosted on the same server or on an external service (server or cloud).

Frontend

To prepare the frontend for deployment, first modify the .env.production file in the root of the AfantazieWeb directory. Language and the backend URL need to be configured. Here is an example of our configuration:

```
VITE_LANGUAGE=cz
VITE_URL=https://afantazie.cz
```

Then run the following commands in the root of the AfantazieWeb directory:

```
npm install
npm run build
```

This will create a dist directory with the compiled frontend code. Copy this code to a public directory on the server.

Backend

To deploy the backend, first, modify the app settings.Production.json file at the root of the AfantazieServer project. Here is an example of our configuration:

```
{
  "ApplicationLanguage": "cz",
  "JwtSecurityKey": "SECRET_KEY_HERE",

  "ConnectionStrings": {
```

[1] This is necessary as the tool doesn't generate the data optimally and violates these constraints. They can be added back after the import but it is not necessary for the application to run.

```
      "DefaultConnection": "DATABASE CONNECTION STRING"
   }
}
```

In the JwtSecurityKey field, insert a secure string that will be used to sign JWT tokens. In the ConnectionStrings field, insert the connection string to the database. One can also change the language of the application by changing the ApplicationLanguage field.

We recommend using Visual Studio IDE to build the backend. Create a new publish profile and set the target runtime to "Portable". Then, publish the project in a directory and copy the files to the server. Make sure to have the .NET 8 runtime installed on the server.

Finally, run the AfantazieServer.dll file with the following command:

```
dotnet AfantazieServer.dll
```

This way, the server will only run as long as the terminal is open. To run the process in the background, we highly recommend using a program called **tmux** (another alternative is program **screen**). Utilizing tmux makes it possible to run the server in the background.

Nginx Configuration

As stated in Section 4.2.4 we use Nginx as a reverse proxy to reroute requests to the backend and to force the meta.json file to be always returned. Here is how we set up the Nginx configuration file:

```
server {
  server_name www.afantazie.cz afantazie.cz;
  root /www/hosting/afantazie.cz/www;

  index index.php index.html;

  include /etc/Nginx/sites-available/domains_conf/afantazie.cz.conf;
  if ($scheme != https)
  {
    return 308 https://$host$request_uri$is_args$args;
  }

  location /api {
    proxy_pass          http://127.0.0.1:5000;
    proxy_http_version 1.1;
    proxy_set_header    Upgrade $http_upgrade;
    proxy_set_header    Connection keep-alive;
    proxy_set_header    Host $host;
    proxy_cache_bypass $http_upgrade;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Proto $scheme;
  }

  location /hub {
    proxy_pass          http://127.0.0.1:5000;
    proxy_http_version 1.1;
    proxy_set_header    Upgrade $http_upgrade;
    proxy_set_header    Connection "Upgrade";
    proxy_set_header    Host $host;
    proxy_cache_bypass $http_upgrade;
```

```
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;
  }

  location /meta.json {
    add_header Cache-Control "no-cache, no-store, must-revalidate";
    expires -1;
  }
}
```

6.3 Administrator Documentation

This section is dedicated to the documentation of the Aphantasia administration.

6.3.1 Backend

To understand the architecture of the backend, see the detailed architecture description in Chapter 4.2.2. There is not much to be done on the backend side apart from the initial setup and deployment.

The application has an active console logging, and one can see the activity on the site, including the number of currently active users, thoughts currently being explored, as well as the creation of new thoughts.

6.3.2 Frontend

To understand the architecture of the frontend, see the detailed architecture description in Chapter 4.4.1.

Graph Layout Algorithm Parameters

While running the Aphantasia's frontend publically it is important to keep in mind that the graph parametrization might become insufficient as the graph grows. This is the main administration task on the frontend side - adjusting the graph layout algorithm parameters.

- **Forces**: Pull force, push force, and gravity force, including maximum allowed values and distance thresholds
- **Gravity On**: Boolean value that turns the gravity force on or off
- **Link Distance**: Controls the spacing between connected thoughts
- **Momentum Dampening Divisor**: Forces acting on thoughts are divided by this number, resulting in a smoother with less jitter, but the graph will be slower to stabilize.
- **Stage Size**: The size of the simulation container
- **Radius Controls**: Base radius, size multiplier (how much larger more-referenced thoughts are), and maximum radius

- **Simulation Falloff Time**: Gradually slows down the GLA after each user interaction (dragging or selecting thoughts and time sliding)
- **Frames with Overlap**: Allows newly appeared thoughts to overlap for a given amount of frames
- **Layout Caching Frequency**: Specifies how often the layout is saved to local storage
- **Node Mass**: Allows differently sized thoughts to have proportional influence on each other, with maximum and minimum values also parametrized
- **Backlinks Count Force Divisor**: Reduces forces acting on highly connected nodes
- **Edges Appearance**: Adjusts thickness and color of highlighted and unhighlighted edges (edges leading in or out of the currently highlighted node)
- **Animated Edges**: Turns on animated edges feature (seen in Figure 4.10)
- **Frames With Less Influence**: New thoughts start with no influence over the simulation and gradually gain it over time

7 Conclusion

In this thesis, we have presented a novel approach to social network content presentation using a graph view instead of the traditional infinite feed. We discussed the motivation and the potential benefits it might bring.

We have implemented a proof-of-concept web application called Aphantasia, which utilizes such an approach, including a custom GLA implementation and a rendering engine based on PixiJS. During the development, we have solved several technical challenges regarding user management, server-client communication, hosting, and, of course, graph rendering.

The finished application provides:

- **Graph view** able to render several hundred nodes at once
- **Dynamic loading** ensuring the capability to explore large graphs in order of tens of thousands of nodes (and potentially much more)
- **User management** including registration and login
- **User interface**, including not just graph view but also pages with user settings, post creation form, chat and rudimentary notifications
- **Live preview** of the graph view

We believe that we have proven graph view as a viable alternative to interact with social media content, provided that the content structure is allowed to have a form of DAG. To look again at the three drawbacks of infinite scroll design we mentioned in the introduction:

- **Echo chambers** - Aphantasia's graph view encourages exploration and serendipity, which can help users break out of their echo chambers.
- **Addictive design** - Aphantasia's graph view requires more user interaction and engagement, which can help users avoid mindless consumption.
- **Lack of autonomy** - Aphantasia's graph view allows users to decide which thoughts to explore and how to navigate the graph.

We are satisfied with the result and believe the application is a good starting point for further development.

Glossary

authX/Y - authentication and authorization, set of systems verifying user identity and granting access to resources 32

BFS - breath first search - algorithm for traversing a **graph** while prioritizing earlier visited **nodes** 52, 56

business logic - the code that implements the business rules of a program (ie. what the application is designed for) 30

CitHep - dataset of citations between papers in the field of high-energy physics. Contains 34546 **nodes**, 421578 **edges** and temporal data... 15, 17, 18, 52

code-first - a development approach where the database schema is generated from the code, as opposed to the database-first approach 27, 30, 32

DAG - directed acyclic graph, a **graph** with **directed edges** and no **cycles** 13, 24, 65

dependency injection - a programming technique in which class instances resolve their dependencies using a DI container set up during the startup. 30, 31

FDL - force-directed layout - a type of **GLA** that simulates physical forces between **nodes** to determine their positions 17, 34, 38

FPS - frames per second - a measure of fluidity of animation and speed of a simulation 18, 26

GLA - graph Layout Algorithm - used for computing positions of **nodes** so that they make a nice or useful diagram 11, 28, 64, 65

graph - a set of **nodes** connected by **edges** 15

JWT - JSON Web Tokens, a compact and self-contained way to securely transmit information between parties as a JSON object 39

local storage - web storage feature that allows JavaScript to store and retrieve data in the browser betwen sessions.............................. 39, 64

localization - adaption of a product to a specific locale or market (language, currency, time format etc.).. 40

memory leak - a situation where a program fails to release memory it no longer needs, leading to wasting memory and potentialy lower performance... 34

ORM - Object-Relational Mapping - a programming technique for converting data between backend and database 27

production (environment) - the instance of an application that is accessible to its userbase, as opposed to development or testing environments . . . 41, 52

SHA256 - cryptographic hash function commonly used to securely hash passwords and tokens . . . 39

thought - a post on Aphantasia represented by a colored node in the graph view 24

tree - a **graph** with no **cycles** and exactly one path between any two **nodes** 13

Vite - frontend build tool that provides configuration and optimized production builds . . . 17, 40

WebSockets - a two-way communication protocol allowing server sending messages to the client without client requesting them . . . 33

References

1. WIKITIONARY. *Echo chamber wikipedia page*. 2024. Available also from: https://en.wikipedia.org/wiki/Echo_chamber_(media).
2. WIKITIONARY. *Doomscrolling wikipedia page*. 2024. Available also from: https://en.wikipedia.org/wiki/Doomscrolling.
3. PAGE, Wikitionary wikitionary. *Algorithm*. 2024. Available also from: https://en.wiktionary.org/wiki/algorithm#Noun.
4. YFILES. *yFiles Demos - Layout styles*. 2024. Available also from: https://www.yworks.com/demos/showcase/layoutstyles/.
5. PROJECT, Stanford Network Analysis. *High-energy physics citation network*. 2024. Available also from: https://snap.stanford.edu/data/cit-HepPh.html. Accessed: 2024-09-10.
6. OBSIDIAN. *Obsidian homepage*. 2024. Available also from: https://obsidian.md/.
7. IMAGE4N6. *Graphview: At the beginning it starts with cool patterns, then turns into total chaos, and then surprisingly structures emerge from the chaos! (22k nodes)*. 2024. Available also from: https://www.reddit.com/r/ObsidianMD/comments/1fyw2te/graphview_at_the_beginning_it_starts_with_cool/.
8. GEPHI. *Gephi homepage*. 2022. Available also from: https://gephi.org/.
9. PEGERP. *Large Steam network visualization with Google Maps + Gephi*. 2012. Available also from: https://forum-gephi.org/viewtopic.php?f=28&t=2314.
10. CYTOSCAPE.JS. *Cytoscape.js homepage*. 2024. Available also from: https://js.cytoscape.org/.
11. FRANZ, Max. *Wine and cheese graph - Interactive demonstration*. 2024. Available also from: http://www.wineandcheesemap.com/.
12. FRANZ, Max. *Cytoscape.js Euler demonstration - GitHub repository*. 2024. Available also from: https://github.com/cytoscape/cytoscape.js-euler.
13. KAPOOR, Ritesh. *Onion architecture*. 2022. Available also from: https://medium.com/expedia-group-tech/onion-architecture-deed8a554423.
14. PIXIJS. *PixiJS official React guide*. 2024. Available also from: https://pixijs.io/pixi-react/#1-create-a-new-react-project-with-vite.
15. EMERY, Adam. *PixiJS guide by Adam Emery*. 2023. Available also from: https://adamemery.dev/articles/pixi-react.
16. ZUSTAND. *Zustand homepage*. 2024. Available also from: https://zustand.docs.pmnd.rs/getting-started/introduction.

List of Figures

1.1 Circular layout[4] 9
1.2 Hierarchical layout[4] 10
1.3 Radial layout[4] 11

2.1 A common graph view of a small Vault in Obsidian 17
2.2 The first 3000 nodes of the CitHep dataset visualized in Obsidian 17
2.3 The first 3000 nodes of the CitHep dataset visualized in Gephi . . 18
2.4 CitHep dataset visualized in Gephi (34546 nodes) 19
2.5 A simple application in react + cytoscape rendering a simple graph 20
2.6 An official example of large graph rendering in Cytoscape.js (path of 5000 nodes) 20
2.7 The first 3000 nodes of the CitHep dataset visualized in Cytoscape.js 21

4.1 Aphantasia Database schema 27
4.2 Aphantasia Backend Architecture 41
4.3 Aphantasia graph view at the end of the small graph development stage 42
4.4 The first 3000 nodes of the CitHep dataset visualized in Aphantasia (before big graph solution) 43
4.5 The logic flow of the big graph rendering solution 44
4.6 The frontend architecture of Aphantasia 45
4.7 The Welcome page and homepage of Aphantasia 46
4.8 The graph view on a mobile device - non-highlighted mode, half-screen preview, and fullscreen preview, respectively 46
4.9 Floating titles in the graph view on desktop 47
4.10 Animated edges in the graph view 47
4.11 Aphantasia with the Czech production dataset in stabilized temporal layout (641 nodes in three time windows of length 300) 48
4.12 The entire dataset of afantazie.cz (641 nodes) in a single time window 49

5.1 A highlighted thought of the CitHep Dataset in Aphantasia . . . 51
5.2 Cithep dataset rendered in Aphantasia with 10000 on-screen thought limit 51

6.1 Aphantasia - Thought creation page 56
6.2 Aphantasia - User settings 57

List of Tables

2.1 Comparison of Obsidian, Gephi, and Cytoscape.js 16

5.1 Comparison of Obsidian, Gephi, Cytoscape.js and Aphantasia . . 52