CodeGraph: an Interactive Dependency Analyzer for Javascript Projects

Robert van Barlingen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 24.05.2021

Thesis supervisor:

Prof. Petri Vuorimaa

Thesis advisor:

M.Sc. Alex Prokhorov

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Robert van Barlingen

Title: CodeGraph: an Interactive Dependency Analyzer for Javascript Projects

Date: 24.05.2021 Language: English Number of pages: 6+106

Department of Computer Science

Professorship: Web Services, Web Applications

Supervisor: Prof. Petri Vuorimaa

Advisor: M.Sc. Alex Prokhorov

Gaining an understanding of a codebase is a vital task for both software developers and project managers. Traditional text-based tools are sub-optimal for achieving this. We have developed CodeGraph, a web application that visualizes the dependency graph of JavaScript projects. This allows users to visually explore the relation between the different files in the project. Additionally, it provides the user with statistics on the project and information about the individual files. Through a small number of usability tests, we have determined that CodeGraph facilitates better understanding of a JavaScript project and allows for a more pleasant user experience than traditional text-based tools.

Keywords: network graph visualization, dependency graphs

Preface

I submit this thesis for the completion of my Master's degree ICT Innovation at Aalto University. This degree is part of the EIT Digital Master School Program. I have completed the first year of this two-year program at KTH in Stockholm and the second year at Aalto University in the track Visual Computing and Communication (VCC).

This thesis was completed under the supervision of professor Petri Vuorimaa at Aalto University. Alex Prokhorov, UX designer at Nebulae in Ghent, was my advisor on the project. Joshua Hulpiau, project manager at Nebulae, served as my mentor and I worked in close collaboration with him. The project was accompanied by a internship at Nebulae.

I want to thank professor Petri Vuorimaa for his constant guidance. He has helped me tremendously by providing frequent feedback, answering my questions and guiding me though the entire process of writing this thesis. I want to thank Joshua Hulpiau for being such a great mentor. He has continuously reviewed my work, spent hours deliberating with me on how to make the project better and has helped me tremendously navigating the challenges of developing a product in an industrial setting. Also Alex Prokhorov, Robby Prima Suherlan and Ega Kamalludin for their valuable contributions to CodeGraph.

I would also like to express my gratitude to my long-term girlfriend, Aya van Dam. She has been a tremendous emotional support throughout this, often stressful, process. She is the joy of my life. I would also like to thank my family. My sister, Annemieke van Barlingen, her husband, Gunter Timmermans, and their daughter, Alexandra, who always welcome me with open arms. It is always a great pleasure to spend time with them. I could not have asked for a more loving family. My mother and father, who have always supported me as much as they possibly could. They have raised me with so much love and care and they have shaped who am I today. Gerry Martin and Shelly MacBain, even though the distance has prevented my from seeing them all these years, they will always remain in my mind and in my heart. Finally, my girlfriend's family, Nicoline van Dam, Alain Le Sage, their daughter, Cariad and Grady Koeslag. From day one, they have treated me as one of their own and throughout the years, they have helped me tremendously.

Contents

Abbreviations and Terms

D3	Data-Drive Documents
CY	Cytoscape
React	A JavaScript UI framework
Material	A UI library designed and maintained by Google
CI/CD	Continuous Development / Continuous Integration
git	version management framework
GUI	Graphical User Interface
MCL	Markov Clustering
RDF	Resource Description Framework
IDE	Integrated Development Environment
W3C	World Wide Web Consortium
URI	Universal Resource Identifier

1 Introduction

Code bases, more specifically JavaScript projects, can become very big and complex with time. Consider the example of FreeCodeCamp[1], the most popular public JavaScript project on GitHub, consisting of more than 5.000 files, which together contain more than 668.000 lines of code. This is by no means an exception and there exist many projects with far more files and lines of code. Managing such a massive code base is far from an easy task. Files are often tightly interconnected and often depend on external libraries to function correctly. This complexity can quickly become a problem for both developers who have to maintain and improve the code base and project managers who have to maintain an overview of the project in order to effectively manage the development.

In this project, I aim to create a tool for developers and project managers to quickly get a visual overview of such JavaScript projects. This tool will henceforth be referred to as CodeGraph. CodeGraph will visualize all the files in the project and all the interdependencies among the files as a network graph. Such dependency graphs, as they are called, are already highly used in compilers, software installers, software building scripts and module bundlers. However, in these instances, they are solely used as a means for the compiler, installer or script to get a better understanding of a JavaScript code base. These dependency graphs have rarely been used as a tool for humans to get a better understanding of the project. CodeGraph will nicely complement existing techniques for understanding JavaScript projects, such as reading the code directly and reading the documentation.

More precisely, through this project I will aim to answer the following research question:

How can dependency graphs be used to provide a better understanding of JavaScript projects? (Q1)

In order to answer this question, I will also explore the following subquestions:

- *What requirements should an application providing such a dependency graph fulfill in order to provide the user with as many useful insights as possible? (Q2)*
- *How can dependency graphs be visualized in order to optimize the usability for the user? (Q3)*
- *How can such a tool be integrated in the current project development workflow as seamlessly as possible? (Q4)*

In Chapter 2, I will first review the state-of-the-art in the field of dependency visualization. In Section 2.1, I will review different applications and research projects that have similar or related objectives as CodeGraph. I will follow this, in Section 2.2, with a review of studies on various techniques that will be useful in the development

[1] https://github.com/FreeCodeCamp/freecodecamp

of CodeGraph. After the theoretical Section, I will move on to the main Chapter of this paper, Chapter 3, where I explain the process of developing CodeGraph. I will start by explaining the methodology of the development process in Section 3.1. In Section 3.2, I will review the motivation for the development of CodeGraph. The objectives that CodeGraph should accomplish are introduced in Section 3.3. Section 3.4 will give an overview of all the relevant stakeholders and all the requirements that CodeGraph should fulfill in order to create an optimal tool for the stakeholders. In Section 3.5, I will describe the architecture of CodeGraph, followed by Section 3.6, where I will explain the development process. Finally, in Sections 3.7 and 3.8, I will describe the design of the usability test and present the results followed by a discussion in Chapter 4. Finally in Chapter 5, I will provide conclusions of the project.

This project is executed under supervision of Prof. Dr. Petri Vuorimaa at Aalto University and in close collaboration with Nebulae, a web development company in Belgium. I was directly guided by Joshua Hulpiau, the founder and project manager of Nebulae. I was the main contributor to the project, but other employees of Nebulae, including Robby Prima Suherlan, Alex Prokhorov, Ega Kamalludin and Joshua Hulpiau, also regularly contributed to the project. In this thesis, I will mainly focus on my contributions towards CodeGraph, but I will also mention other people's contributions when appropriate.

A presentation of the entire project is available at `http://bit.ly/codegraph`.

2 Theory

2.1 Similar Project and Applications

We are hardly the first ones to create visual tools to aid software developers. What follows is list of visual tool that are similar or related to CodeGraph.

- Researchers at Zhejiang University have created a application that shares many of the same goals as CodeGraph. They outlined their research in *"Visual exploration of dependency graph in source code via embedding-based similarity"*[25]. They have create a *"visual analytics system to explore code dependencies between files for visually understanding software architecture and interactively analyzing bad dependencies."* This application is shown in Figure 1. Their application's main feature is a visual representation of the dependency graph between the different files. They recognize that the dependency graph of a large projects can become chaotic. In order to reduce the number of visualized nodes, they have implemented an abstraction based on the file-structure of the project. More specifically, in their visualization, all the files in a single folder can be collapsed into a single node in the dependency graph. This folder-node can then be expanded to a single node for each file in the folder.

  Their application also offers other attractive features such as the detection of bad dependencies. The user's code is analyzed and if bad dependencies, such as circular or conflicting dependencies, are detected, these are visually shown to the user. The user is also provided information on the individual files, including the file contents, number of functions, number of classes, etc. Apart from the visualization of the dependency graph, the application also provides other graphs, such as a sunburst chart to present file information including the file hierarchy and dependency details, and a t-SNE chart[36] to project bad dependencies.

  This project shares many similarities with CodeGraph. To start, they are both developed as JavaScript applications using the D3 library for rendering the visualizations. They are both specifically designed to analyze JavaScript projects and they both use many of the same techniques for creating an optimal visualization, such as clustering. Their application does, however, offer more features than CodeGraph. But in contrast to CodeGraph, their application cannot be easily integrated into the existing development workflow. For example, CodeGraph offers the ability to integrate it in the CI/CD progress of the users, while their application only functions as a completely stand-alone application.

  Of all the existing projects, this is the project that is most similar to CodeGraph.

- Another interesting research project is described in *"Exploring Relations within Software Systems Using Treemap Enhanced Hierarchical Graphs"*[4]. This project is once again focused on the visualization of dependency graphs. It has some distinct benefits over other solutions including CodeGraph. To begin with, their application isn't restricted to dependencies through import and

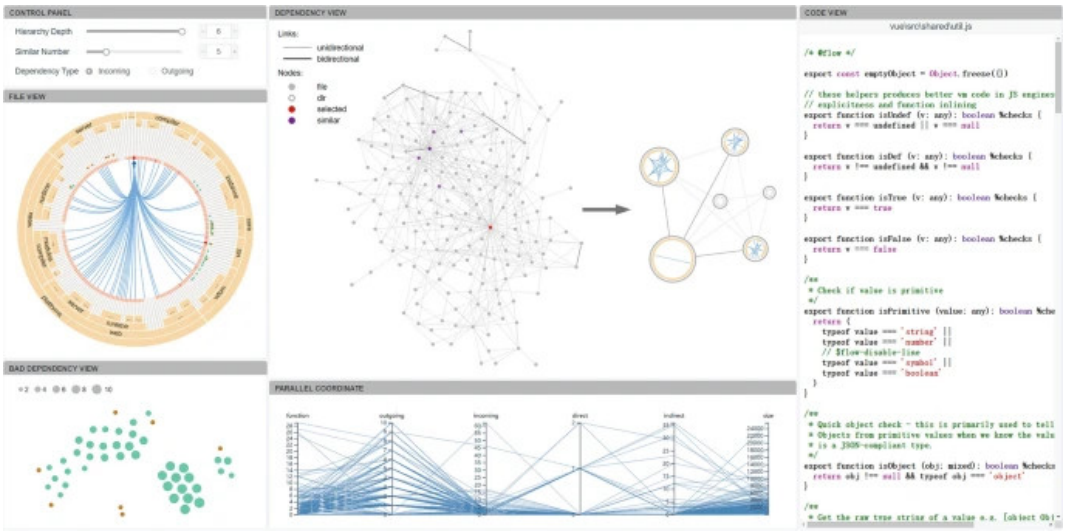


Figure 1: The GUI of the application described in "Visual exploration of dependency graph in source code via embedding-based similarity" [25].

export statements, but also includes method calls, read or write accesses to attributes, and inheritance of classes. The visualizations in this application puts heavy emphasis on the abstraction of hierarchical levels. Depending on the level of abstraction the application will show dependencies between packages, the relation between specific methods or attributed or a mix of both. When collapsed, the abstracted group of nodes are represented by a treemap [22]. This is a visualization that represents a hierarchical structure as a compact rectangular graph. Figure 2 shows the result of this abstraction. These abstractions can interactively be collapsed and expanded to allow the user to explore the dependency graph in a stepwise manner. Additionally, the application also indicates groups using colors. The result is very well-organized as shown in Figure 3.

- Hunter[13] is another application with some novel features. Its main feature is, once again, the visualization of the dependency graph. The novelties of Hunter include the fact that, in the file browser, the folders are marked with a color. Each node in the dependency graph is marked with the color of the parent folder of the file represented by the node. This allows the user to visually relate the view of the file browser to the visualization of the dependency graph. Additionally, the size of each node in the dependency graph is directly proportional to the number of lines of code contained in the file. This gives more important node, i.e. nodes with more lines of code, more visual weight. A final exciting feature of Hunter's dependency graph is the fact that it distinguishes between library files imported from external libraries and files belonging to the project. Regular files are represented by a circle, while library files are represented by a square. In addition to the dependency graph, Hunter also has a window that shows the source code of the selected node, a search panel

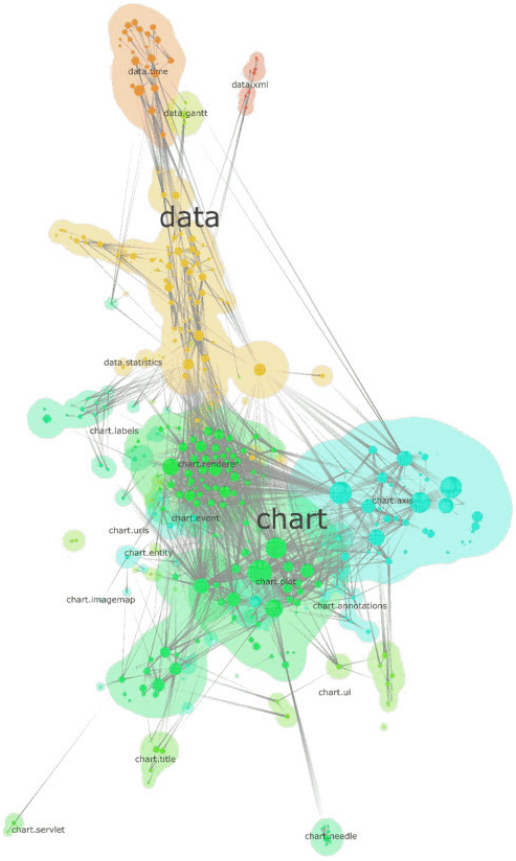


Figure 2: Treemaps allow hierarchical groups of nodes to be represented by a single rectangular graph.

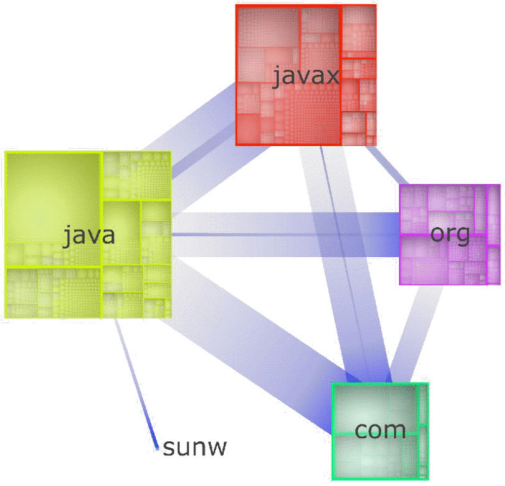


Figure 3: Colors allow the user to quickly identify the different groups of nodes.

that allows the user to easily search for a specific file and a window that shows a treemap that shows the inner structure of the file. Figure 4 shows GUI of Hunter.

- *"Visualization of Program Dependence Graphs"*[39] describes another similar application. In contrast to the previous projects, this project is focused on the visualization of the dependency graph of Java projects. It has two distinct, unique features. The first feature is its rich filter system. It offers the user a rich filter menu that can be used to remove, add, merge, split and color nodes and edges. Custom filters can be defined using rich regular expressions referencing the properties of the nodes and edges and sets of filters can be saved as profiles. This is by far the most rich filtering functionality of all the mentioned applications. The other distinct feature is its ability to visually show the difference between two snapshots or case bases. This is useful for comparing what has changed between two versions of a code base. These changes are marked as such in the dependency graph visualization. The application is also capable of working with large dependency graphs containing a few thousand nodes, it contains clusters of nodes and long edges are cut such that only

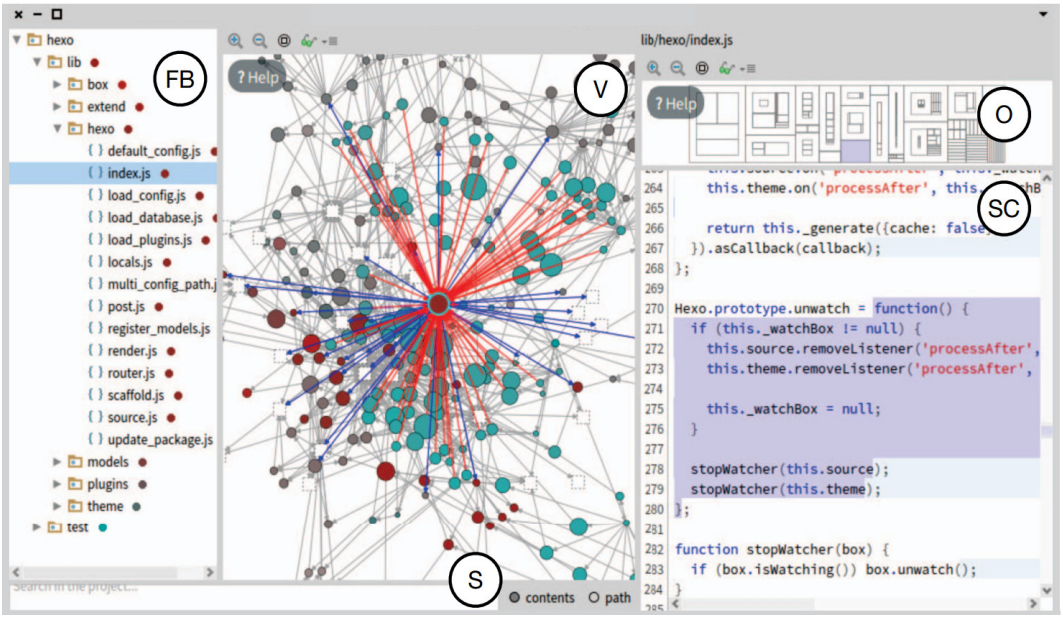


Figure 4: The GUI of the Hunter.

the beginnings and ends are drawn in order to improve the overview and performance. Figure 5 shows the GUI.

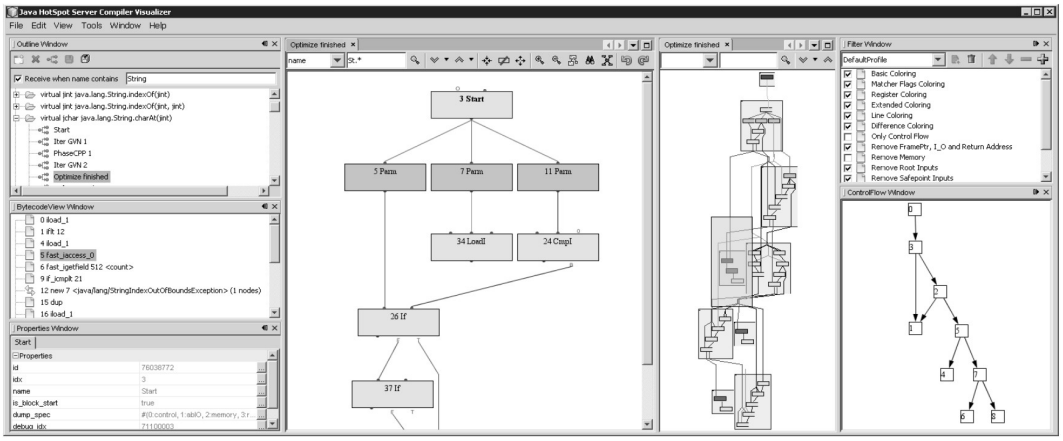


Figure 5: The GUI of the application described in *"Visualization of Program Dependence Graphs"*[39].

- Polyptychon[10], shown in Figure 6, is another application for visualizing the dependency graphs of Java projects. Polyptychon is focused on the incremental exploration of a project. To this end, when a node is selected this so-called view root becomes the center of the visualization. All the child nodes of the view root are then grouped and displayed in their respective groups. The groups are constructed based on the characteristics of the nodes. For example, the tangled groups contains groups of nodes that are strongly connected, i.e. a group in which every node is connected to every other node. The visualization

also contains nodes which are not directly connected to the root node, but are somehow related to the root node or it siblings. These groupings allow the user to quickly get an impression of the kinds of dependencies that are related to the root node.

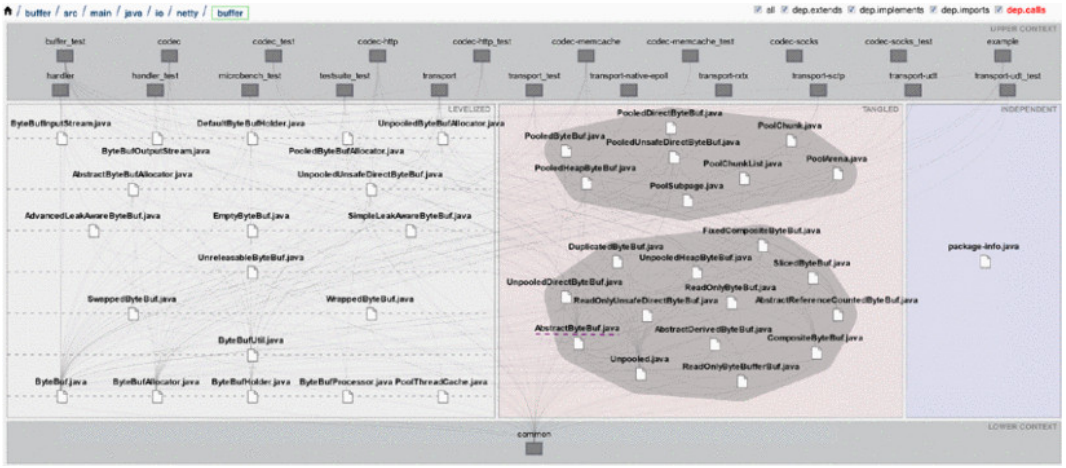


Figure 6: The GUI of Polyptychon.

- The application described in *"Visualizing Modules and Dependencies of OSGi-based Applications."* [33] has many of the same features as the aforementioned projects. It allows users to reduce the complexity of the dependency graph using filtering, selection and abstraction. It also provides information on individual nodes and the project as a whole and the size of each node depends on the number of lines of code in the corresponding file. One truly unique feature of this application is the inclusion of a Virtual Reality visualization which allows the user to view the dependency graph using a VR headset. This VR visualization is shown in Figure 8.

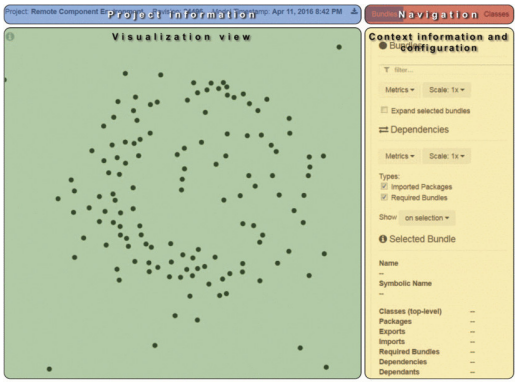


Figure 7: The GUI of *"Visualizing Modules and Dependencies of OSGi-based Applications."*[33]

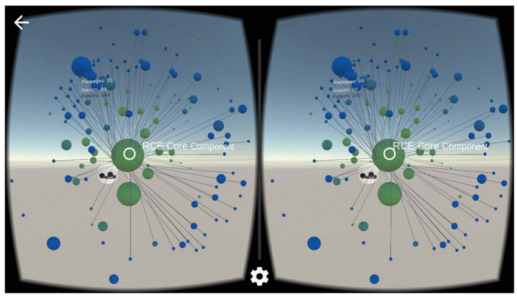


Figure 8: A Virtual Reality visualization of the dependency graph.

- SolidSX[30] is one of the more full-featured applications. In fact, it is constantly being improved and it is being used by hundreds of students. In contrast to the previous applications, which offer a single, closed-off, fixed solution, SolidSX offers a far more modular solution. For one thing, users can create scripts to define selections. But more importantly, SolidSX can very easily be integrated in third party tools through a thin wrapper. For example, they have created a plug-in for Visual Studio which allows the user to fully view and interact with SolidSX inside Visual Studio. SolidSX is so optimized for the integration by third party applications that the plug-in wrapper only consists of around 200 lines of code. They have achieved this modularity by making SolidSX listen to asynchronous Windows command messages, e.g. load a dataset, user interaction events, etc. Moreover, SolidSX stores all its data in a simple SQL database that the user can interact with. Figure 9 and 10 shows SolidSX.

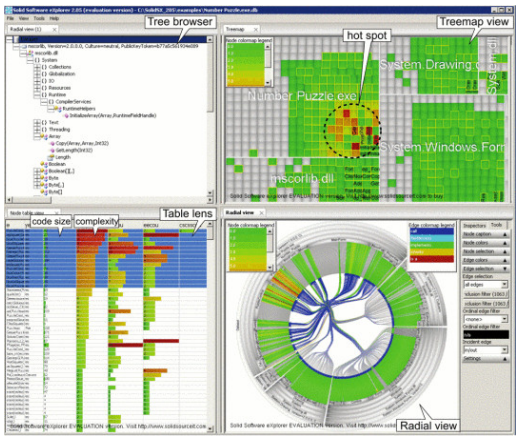


Figure 9: The GUI of SolidSX.

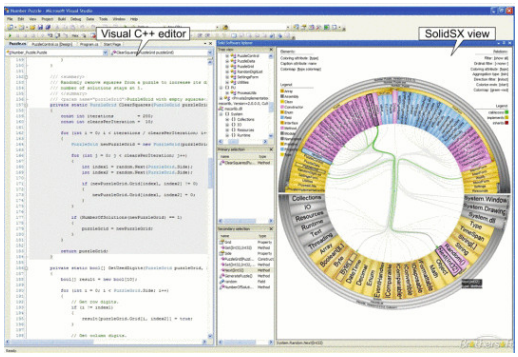


Figure 10: The GUI of SolidSX.

- One of the most interesting and unique applications comes from *"Animated Visualization of Software History using Evolution Storyboards"*[6]. The authors argue that large software systems have a rich history of development accompanied by major refactors, architecture changes, the removal of old features and the introduction of new features. Throughout this process, the developers learn from their mistakes. This wisdom is gathered in the minds of the senior developers on the project and passed along to new developers through anecdotes. It is rarely documented. When senior developers leave the project, part of their wisdom goes with them, which might lead to mistakes being repeated. The authors set out to create a system that documents the history of a software project. It does this as follows: the application is linked to the version control system that houses the software. The application then extracts and analyzes how the code has changed over time. It then presents the user with a sequence of panels, each containing the dependency graph of the codebase at a specific moment in time. This sequence of panels is very similar to a movie in which movement is conveyed through the display of slightly changed pictures in rapid succession. In contrast to a regular movie, the software presents each panel

individually allowing the user to manually decide how much time he spends viewing each panel. The user can also define how much development time is conveyed between two consecutive panels. For example, a new panel can be defined every three months, or whenever 1.000 new dependencies have been added or after every major release. The application also offers some other innovative features such as the use of a heat map to further emphasis the change over time and the fact that each edge also includes how the connected nodes depend on each other. Figure 11 shows the first and last panel of an animation.

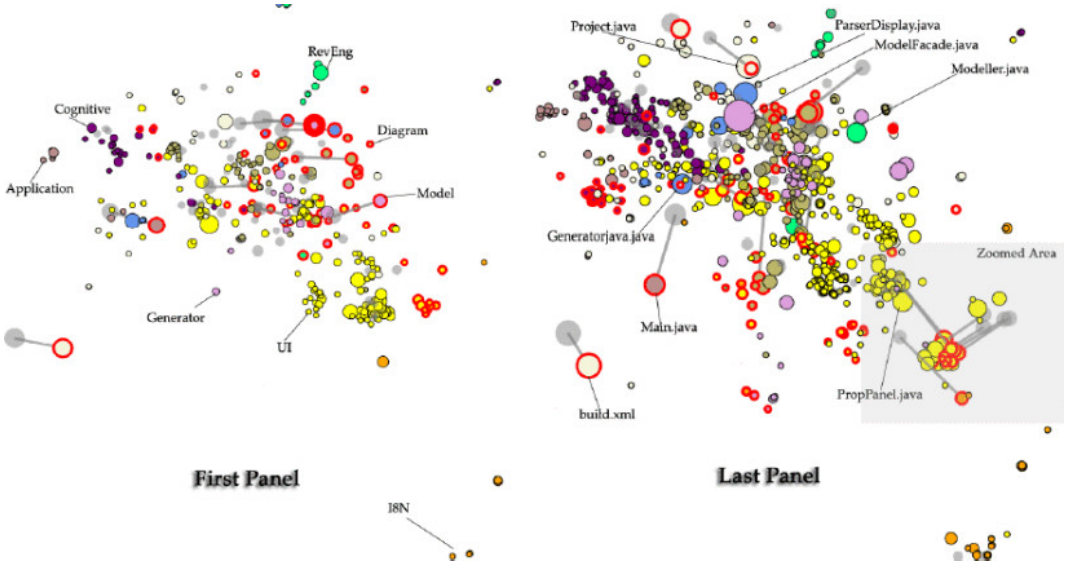


Figure 11: The first and last panel of an animation described in *"Animated Visualization of Software History using Evolution Storyboards"*[6]

- Another approach to visualizing the change of a project over time is described in *"Visual exploration of software evolution via topic modeling"*[26]. With their application, the researches aim to visualize three aspects: (1) when major changes in the codebase have occurred, for example between v1 and v2 of the application, the code base might have changed drastically, while between v1.0 and v1.1 the code might have changed very little. (2) How the software features have evolved over time, for example, what major features have been added or removed over time and (3) why those evolutions have occurred, for example, code refactoring, bug fixing, addition of a new feature, etc. They envision that their application can help new developers understand the history of the project and allows project managers to assess the progress of the development process.

  In contrast to the previously mentioned application, this application aims to visualize the project evolution over time using graphs and statistics rather than an animation. Examples of the application's graphics include a word cloud showing common vocabulary, a trend line which is a 2D graph showing how features have evolved over time, a file comparison showing what files

have changed between two points in time. Figure 12 shows the GUI of the application.

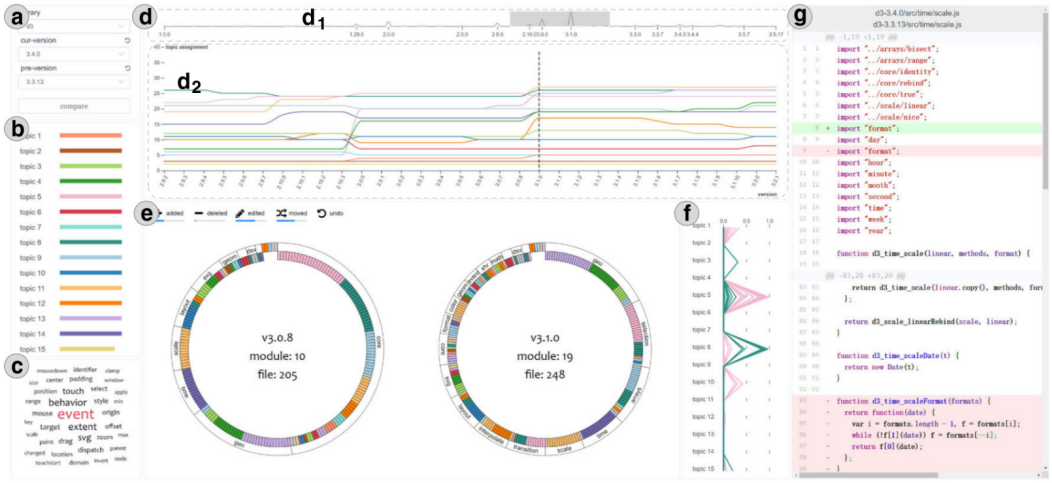


Figure 12: The GUI of the application described in *"Visual exploration of software evolution via topic modeling"*[26].

- So far, all the discussed projects have focused on analyzing a single project. However, the developers of the Small Project Observatory (SPO)[27] argue that software is rarely developed in isolation. In fact, most software is developed as part of a larger ecosystem. Whether it be at a company, a research institution or an open-source community, most software is developed in parallel alongside other software that has to work together. This is especially prevalent in recent years with the gradual transition towards architectures centered around micro-services. To this end, SPO is a tool for analyzing and visualizing entire software ecosystems. It gathers information from all the version control repositories of the all the projects, analyzes how they are related and presents the result to the user in the form of an interactive web application. One of the most interesting features of SPO, with regards to CodeGraph, is the interproject dependency graph. This is a graph that visualizes how the different projects are linked. This is essentially what CodeGraph aims to do but on a higher abstraction level. Another interesting aspect of their graph is the fact that they have implemented color-coding based on the age of the project. The older the project, the darker the node is colored as shown in Figure 13. Other interesting features of SPO include a vocabulary map that visualizes the most common vocabulary used across the ecosystem, a collaboration chart which shows which users have worked together in the past, a timeline view which shows how the projects have changed over time, etc.

- In *"Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study"*[35] the researchers perform a comparison of different dependency graph visualization tools. However, in order to perform this comparison, the researchers have developed their own pipeline for extracting

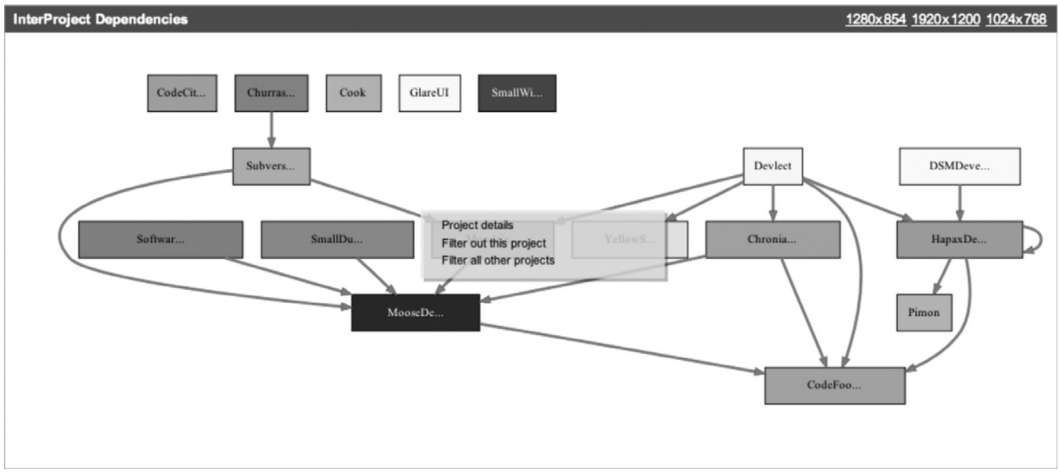


Figure 13: The interproject dependency graph of Small Project Observatory.

the dependency information from C/C++ projects. A strong aspect of this pipeline is that it extract many different types of nodes and relations. Consequently, the dependency graph contains nodes representing directories, nodes representing files, nodes representing functions, edges representing containment and nodes representing function calls. This offers a richer visualization than the other projects which only contain nodes representing files and edges representing imports. Figure 14 shows different dependency graphs generated by the researchers.

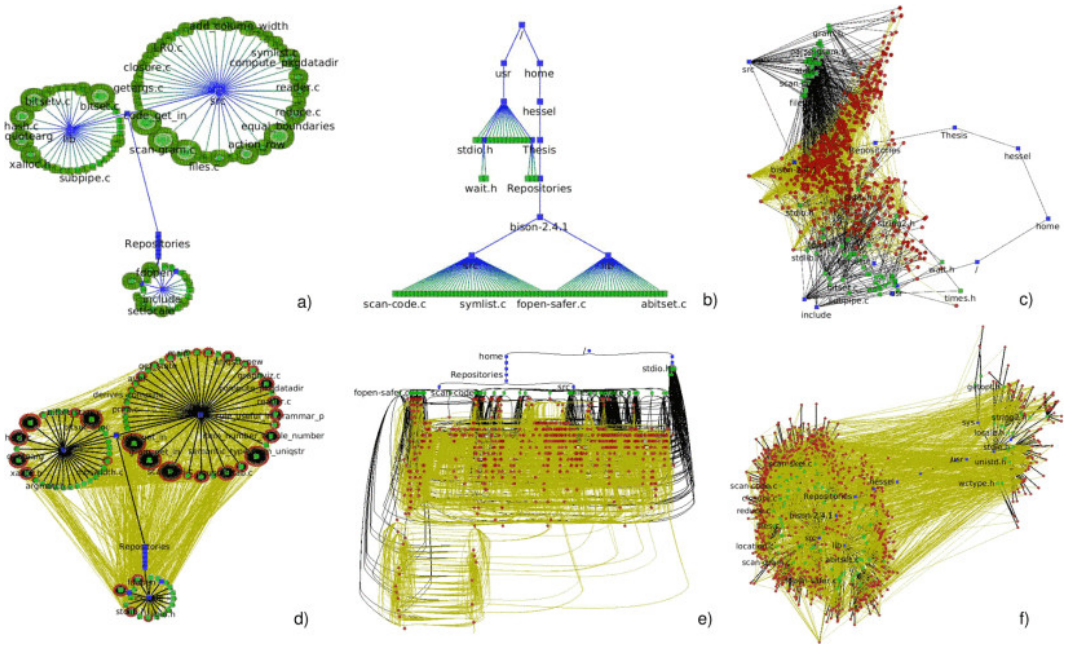


Figure 14: The dependency graphs contains different nodes and edges indicated by the different colors and shaped of the nodes and edges.

- Yet another form of dependency graph is presented by EXAPUS[11]. This application is focused on the exploration of API usage in Java projects. As such, it provides a tree-like dependency graph of all the API uses withing the project. The nodes corresponding to API that are used heavily receive a heavier border than those that are used less frequently. It also makes a distinction between different types of API, which are represented by different shapes. Finally, EXAPUS also provides many additional statistics about the API usage in the project. Figure 15 shows the API dependency graph of EXAPUS.

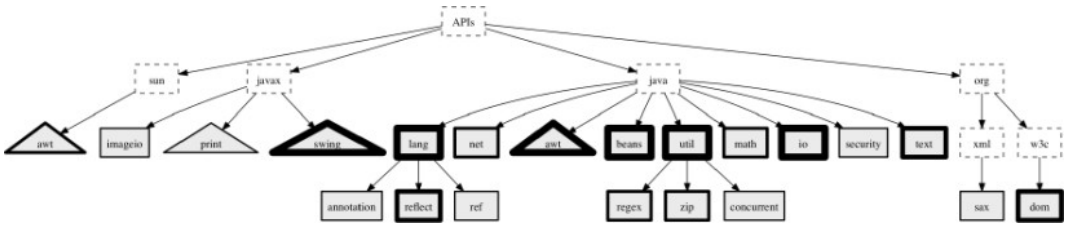


Figure 15: The API dependency graph of EXAPUS.

All of these projects are established mostly for research purposes. In contrast, we have the explicit intention of making CodeGraph a commercial product in the long term. Because of this, it is very important that CodeGraph fits seamlessly in the existing development workflow of software companies. This was not really the case in the above applications, but this was also not their intention.

2.2 Methods

Apart from similar or related projects, we also draw inspiration from studies on various useful techniques.

- One of the most useful papers is *"A survey on visualization approaches for exploring association relationships in graph data"*[7]. In this paper, the researchers give an overview of many of the most prominent graph visualization and simplification techniques. They present seven different types of visualization. The node-link diagram is a very common visualization. It represents the entities as nodes and the relations between the entities as edges. The placement of the nodes and edges is then defined by the layout algorithm. These layouts can either be constraint-based or force-directed. The constraint-based layouts work by trying to optimize some parameter while being bounded by a set of constraint. Force-directed layouts place imaginary springs between connected nodes. The layout then runs a physical simulation, in which connected nodes are attracted to each other, while disconnected nodes repulse each other. This produces very nice, well-organized graph visualizations. Figure 16 shows many different examples of node-link graphs.

  The adjacency matrix represents a graph with N nodes as an $N \times N$-matrix. Assume all the nodes are numbered. If there exists an edge from node X to node Y, then the adjacency matrix M will have a value of K at row X, column

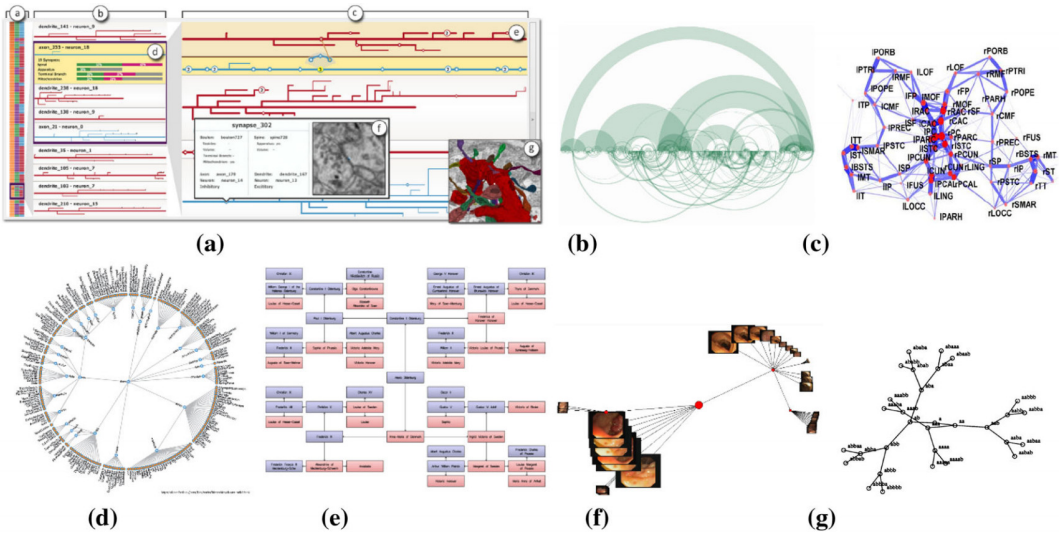


Figure 16: Examples of node-link graphs.

Y, with K being the weight of the edge or 1 if the edges have no weight. If there exists no edge from node X to node Y, then the value of row X, column Y is 0. This representation allows the user to quickly discover patterns or certain structures in the graph data. However, it is not very intuitive. Figure 17 shows some examples of adjacency matrices.

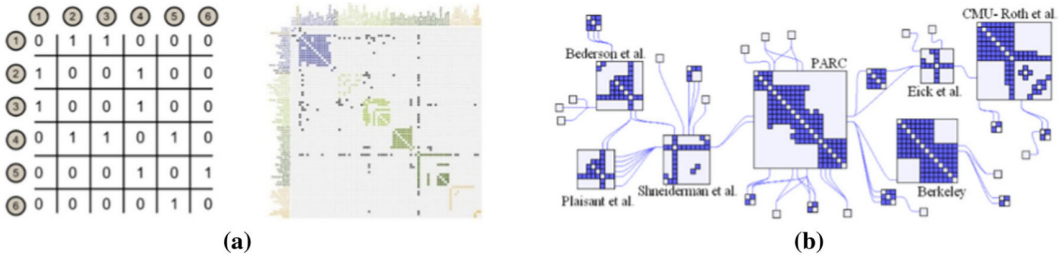


Figure 17: Examples of adjacency matrices.

In a node-link graph, each edge is connected to exactly two nodes. In a hypergraph, on the other hand, an edge connects an indefinite number of nodes. The edge represents a common relationship between all the connected nodes. This visualization reduces clutter and is very intuitive, however, it does lose some information. Graphs (a) and (b) in Figure 18 show hypergraphs.

Flow diagrams represent data flow between entities by making the width of the line between the entities proportional to the amount of data flow. It is a very intuitive way to display flow, however, for large datasets, this visualization can become quite cluttered. Graphs (c) and (d) in Figure 18 are flow diagrams.

Graphs with geospatial data relate data to geographical locations, for example, an abstract representation of the power grid over the map of a city. Graph (a) in Figure 19 shows such a graph.

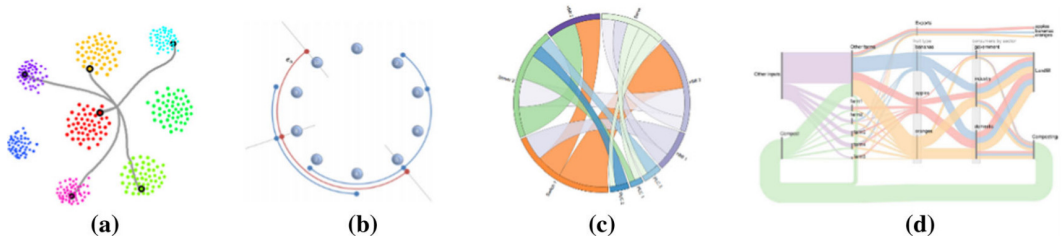


Figure 18: Graphs (a) and (b) are examples of hypergraph and graphs (c) and (d) are examples of flow diagrams.

Multi-attribute graphs are graphs that focus on the visualization of data with multiple attributes or multiple dimensions. Graphs (b) and (c) in Figure 19 are examples of such multi-attribute graphs.

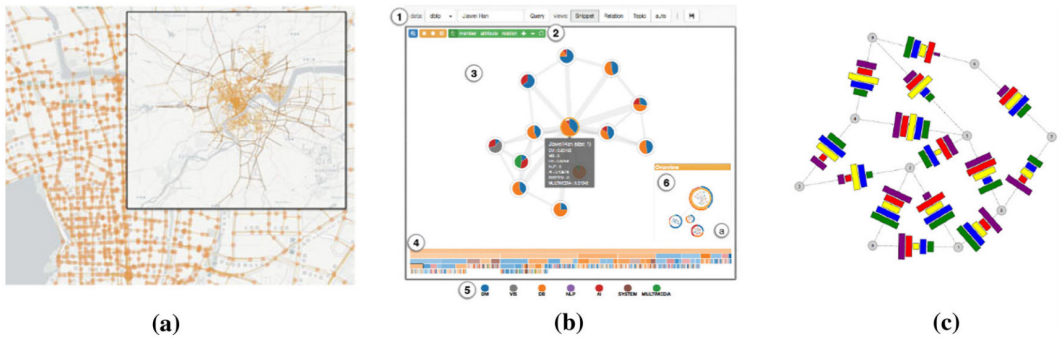


Figure 19: Graph (a) is an example of a graph with geospatial data and graphs (b) and (c) are examples of multi-attribute graphs.

Finally, space-filling diagrams are graphs that take up as much space of some predefined area as possible. It has a high utilization rate and it works well for visualizing the inclusion of entities within other entities. Figure 20 shows examples of space-filling diagrams.

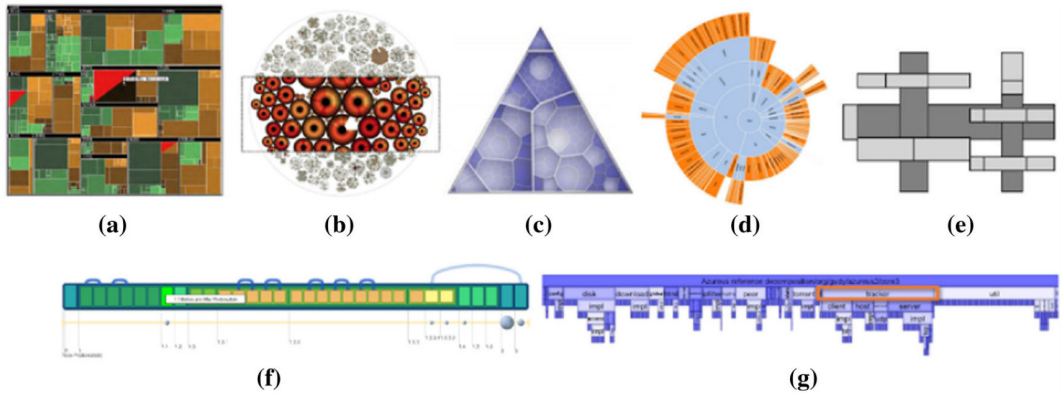


Figure 20: Examples of space-filling diagrams.

However, visualization techniques can only get you so far towards the orga-

nization of data. Large datasets will almost always remain cluttered, even with the best visualization techniques. Therefore, it is very important to also simplify the data. The paper[7] introduces five different techniques of data simplification. The first technique is graph filtering. This technique involves the careful selection of which nodes to include in the graph and which nodes to remove. There are many different metrics for performing this filtering. Often times, the user simply manually decides, which nodes to filter out.

Node clustering is another simplification technique. It groups together similar nodes into clusters. Such a cluster can then be abstracted into a single node or all the nodes in the cluster can be placed close together and be made visually distinct, for example, by giving them all the same color. In either case, the visualization becomes more organized. Different algorithms exist for deciding which nodes to group together into a cluster.

Edge bundling is a technique, which groups together edges. These edges then get reduced into a single edge. Often times, the color and thickness of these new edges represent how many edges are included in the bundle. This technique reduces the clutter caused by many edges overlapping each other.

Dimension reduction is often applied to visualization of data with many dimensions. It reduces the visualization to only display certain dimensions of the data. This process does hide some of the data.

Finally, topology-based graph transformations provide ways of putting more emphasis on important information. One technique is the, so-called, fish-eye view. This technique enlarges the information that the user in interested in, while making the other data smaller. Another technique is to provide two different views: an overview and a detailed view. The overview shows all the data and can be used to navigate and select what data the user is interested in. The detailed view provides detailed insights into the data that the user is interested in.

- In *"Cluster Analysis of Java Dependency Graphs"*[14], the researchers share their experience using the GirvanNewman clustering algorithm for clustering Java dependency graphs. This is interesting since this is one of the few studies on clustering algorithms specifically for the use case of dependency graphs. Their results were very promising. They also created a visualization tool for displaying the clusters. Figure 21 shows the result of the clustering. We can see that the clusters are nicely formed around groups of tightly connected nodes.

- In *"On the Shape of Circular Dependencies in Java Programs"*[1], the researchers define different types of circular dependencies and they analyze their typologies. They provide a detailed account of what they look like. This is useful information for implementing a feature that detects circular dependencies.

- In *"Slimming javascript applications: An approach for removing unused functions from javascript libraries"*[37], the authors describe a novel way to remove unused code from JavaScript bundles. This is an important topic as it leads to

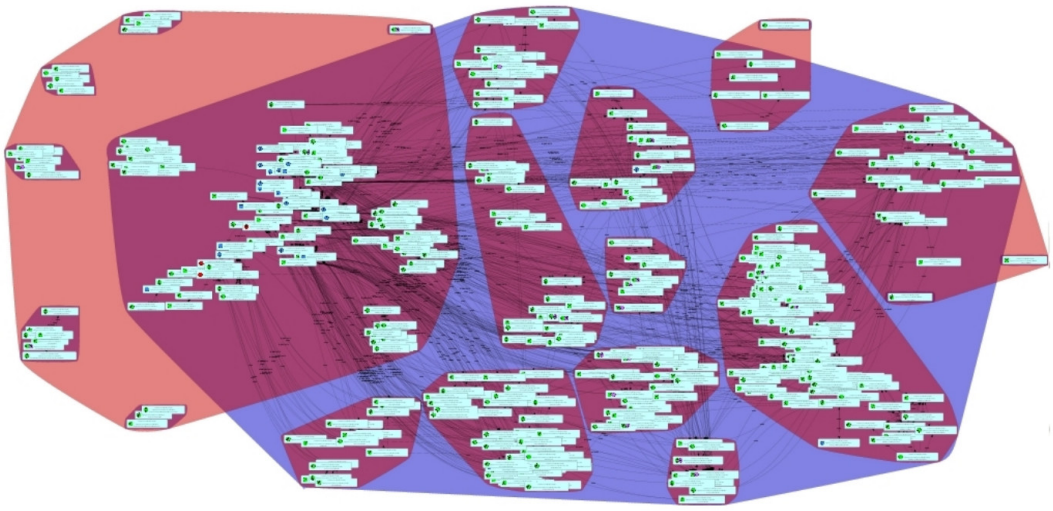


Figure 21: The result of the dependency graph cluster.

the user having to download less data when loading a website, which reduces loading times, and memory and power usage on mobile devices. The reason why this research is relevant to CodeGraph is because their technique focuses on unused code in the source code of library dependencies. These are a lot harder to remove than unused code in the developer's own codebase. Their new approach has been shown to reduce the size of JavaScript bundles by 26% on average. Information on the bundle size of dependencies and amount of unused code in these dependencies would be a useful addition to CodeGraph.

3 Practice

3.1 Methodology

For the development of CodeGraph, we will adopt the Design Science Research Methodology (DSRM)[29] consisting of the following steps: the problem identification and motivation, the definition of the objectives for a solution, the design and development of artifacts, the demonstration, an evaluation and the communication of the results.

Section 3.2 will detail the problem identification and motivation for the creation of CodeGraph. The objectives for CodeGraph are discussed in Sections 3.3 and 3.4. The former outlines the general aims and goals of CodeGraph, while the latter defines the exact formal requirements that CodeGraph should fulfill in order to be considered complete. The design of CodeGraph is outlined in Section 3.5, while the development process is discussed in Section 3.6. In Section 3.7, CodeGraph will be demonstrated to real users and tested based on usability tests. Finally, in Section 3.8 the results of the usability tests will be evaluated and presented.

3.2 Motivation

Applications are often developed over a long period of time and improved and maintained for many years after the initial release. In fact, a survey shows that on average, approximately 30% of a developers time is spent on code maintenance[2]. Furthermore, the development team often changes over time with original developers leaving the project and new developers being added to the project. All this makes tools for learning about the project and maintaining high-level overview of the project an absolute necessity. Current tools that are used by new developers to get acquainted with a project include mailing lists, discussion boards (e.g., slack[3] and Microsoft teams[4]), wikis, documentations, source code repositories, and issue tracking systems (e.g., trello[5] and jira[6]) [31]. Studies have shown these are sufficient for development by long-standing developers [28]. However, for new developers just getting started with a project, these tools are suboptimal. This is not because of lack of information. On the contrary, there is usually too much information and because traditional tools offer limited organization of the information, it is very hard for new developers to find specific information. This results in developers having to go through a long and frustrating learning process before being able to contribute to the project. Visualization tools can help in this regard as they leverage the human's strong ability to recognize visual patterns. They allow the user to grasp, discover and understand large data sets far more efficiently than text-based tools. This is especially useful for open-source communities as they rely almost exclusively on written, long-distance communication. In contrast to a company, new developers are not guided by a senior

[2] https://blog.tidelift.com/developers-spend-30-of-their-time-on-code-maintenance-our-latest-
[3] https://slack.com
[4] https://www.microsoft.com/nl-be/microsoft-teams/group-chat-software
[5] https://trello.com/
[6] https://www.atlassian.com/software/jira

developer, but rather have to discover everything themselves. Sometimes there exists a document aimed at introducing new members to the project, but most often they simply have to learn by reading discussions, engaging with members and trial and error. In this case a visual tool showing the structure of the project would be very useful. In fact, a study has shown that new developers on open-source project are indeed more effective in finding information using visual tools compared to purely text-based tools. The frustration of the difficult learning process using text-based tools may also diminish the motivation of new developers, which makes it harder to recruit new developers. For commercial companies, it can save money as program comprehension reportedly costs as much as 40% of the software lifecycle. [30].

Existing developers can also benefit from visual tools as they allow the developer to reason about the project on a more abstract level. This is useful for different common tasks. For example, when maintaining the software, the developer will often reverse engineer some code of which the developer no longer remembers how it works exactly. Visual tools speeds up this process by giving clear insights into the architecture. This also makes it easier to reason about the project when refactoring code, introducing new features or removing old features. It can also be used as a tool to improve communication between developers as they can support their verbal arguments with visuals.

Visual tools, such as CodeGraph, can also benefit project managers as they need to maintain an abstract overview of the whole project in order to know how to allocate resources, make architectural decisions and communicate with the developers and stakeholders. These tools can also be used for quality assessment. For example, the visual tool shows an abstract overview of the project's architecture, which can then be analyzed to see if it fulfills the project requirements and can be compared to the initial architectural plan.

How would CodeGraph fit in the current workflow for understanding a project? Currently, the two main methods for getting an understanding of a code base include reading the code and reading the documentation. Analyzing the code graph using our method would fall somewhere between these two. It doesn't provide the same level of detail as reading the code directly and it doesn't provide the save high-level overview as documentation. However, in contrast to documentation, it requires very little time to set up, and it provides a nice quick understanding of the project as a whole and how all the different components work together. Table 1 shows how CodeGraph compares to other code comprehension techniques.

CodeGraph cannot replace either the process of reading code or the process of reading code documentation. However, it provides a some unique insights that should make it a valuable tool, especially since it takes little to no time to set up.

3.3 Objectives

CodeGraph is meant to be a development tool aiding developers and project managers in the organization and understanding of their JavaScript projects. This is accomplished by providing a visual, high level overview of the project in the form of a dependency graph. This dependency graph shows, which files in the project import

	reading code	reading documentation	CodeGraph
time to set up	++	- -	++
detailed understanding	++	-	0
high level understanding	- -	++	+
easy of understanding the project	- -	++	0
gained understanding of the inter-relation between files	0	+	++

Table 1: Comparison of techniques to get an understanding of a code base

which other files. In this visualization, the files are represented by nodes and the relation *"file A is imported by file B"* is depicted by an an arrow going from node A to node B as shown in Figure 22.

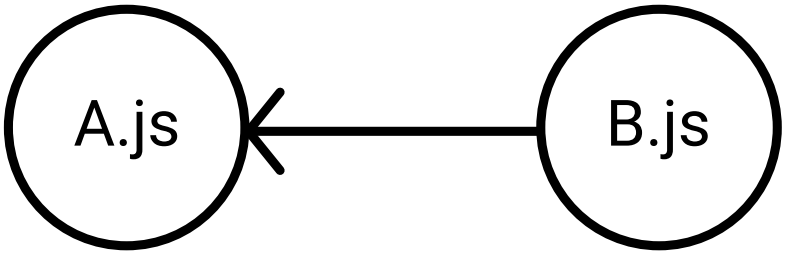


Figure 22: File A.js imports a module from B.js.

As JavaScript projects become more complex, then number of files increases. This makes it harder for project managers to keep the overview of the project and for new developers to get started working on a project. This is where CodeGraph will shine as a tool. However, as the number of files increases, so do the number of nodes in the dependency graph. Therefore, it is of the utmost importance that CodeGraph maintains order, especially since network graph are prone to becoming messy and chaotic as the number of nodes in the graph increases. Figure 23 shows an example of a network graph with many nodes and edges. With this many nodes, it is hard to maintain an overview over the project.

CodeGraph should aid the user in getting a grasp of such chaos. In order to do so, CodeGraph should first provide the ability to limit the data, i.e. to filter out any nodes which the user is not interested in. For example, the user could be interested only in some specific file A and all the files that file A imports, or in the files that a specific users has worked on. Apart from filtering the data, CodeGraph should

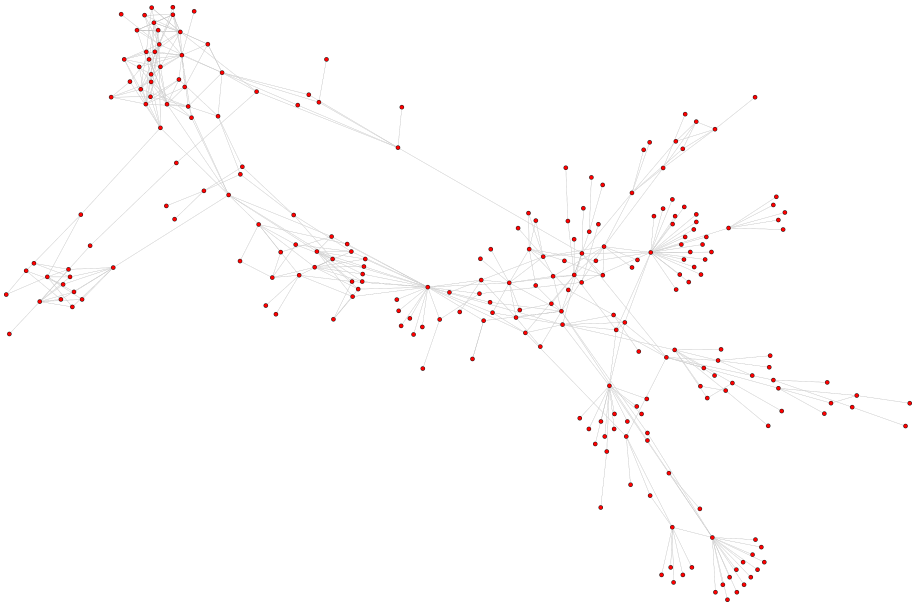


Figure 23: Network graphs with many nodes and edges become chaotic quickly.

also display the dependency graph in an organized fashion. This is mainly achieved through different layouts defining where the nodes and edges a placed. Figure 24 and 25 show how a well-defined layout can help organize and make sense of a large graph network.

Lastly, CodeGraph should aid the user by analyzing the data and proving the user with useful insight. For example, this can be achieved by clustering the dependency graph. This means grouping together nodes that together form a tightly connected group. This usually corresponds a group of files, which together make up a component in a JavaScript project. Figure 26 and 27 show the strength of clustering in a network graph. The clusters are indicated by the different colors.

Apart from the clustering, CodeGraph should also provide insights into the data through statistics, visualized using tables, charts, histograms, etc. This data should be provided on a per-node basis and on the project as a whole.

As all projects and teams are different, CodeGraph should provide the ability for developers and project managers to personalize the CodeGraph project to their wishes. Examples include layout parameters, labeling, project settings, etc. On the other hand, CodeGraph should also be as easy and quick to set up as possible. To this end, CodeGraph should have a set of sensible default values that work for most projects, while allowing users to change these values.

CodeGraph should work both as a tool for an individual as for teams as a whole. For example, CodeGraph should allow for easy exchange of the data and insights. This could be achieved through, for example, the ability to download the dependency graph as an image or as a dataset in a format like json or csv. The same goes for statistics. This process could be streamlined even more through integration with 3rd party team-collaboration tools such as slack, jira, etc. CodeGraph could also include features for collaboration within the application itself, such as the ability to place

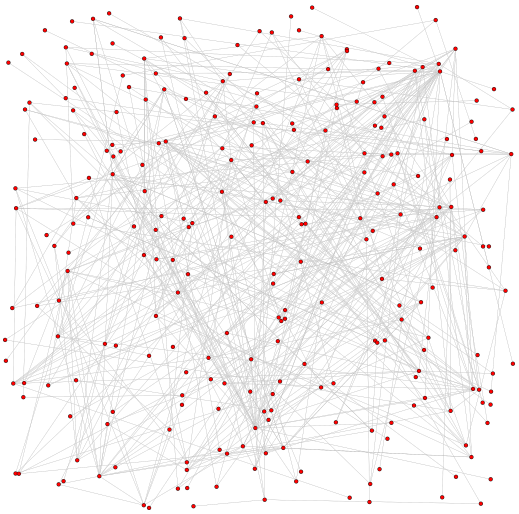


Figure 24: A network graph in which all nodes are placed randomly.

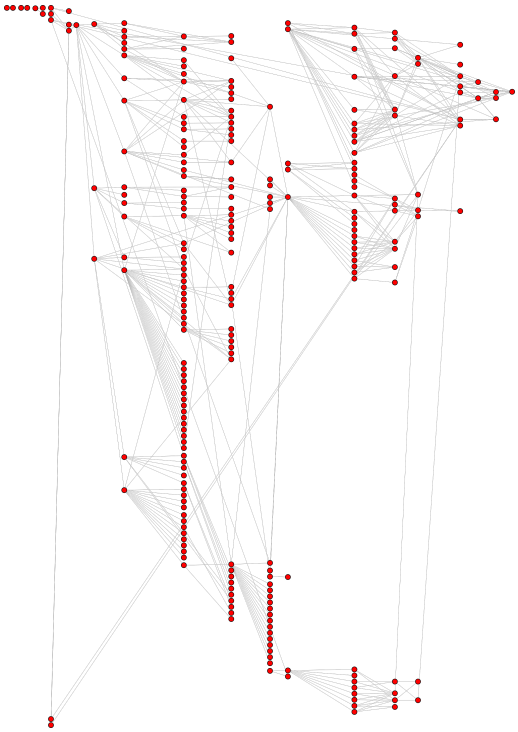


Figure 25: The same network graph, with all nodes placed according to a layout algorithm.

comments associated with nodes and/or edges.

Setting up and maintaining the CodeGraph instance for a JavaScript project should require as little work as possible. To this end, a CodeGraph project should be automatically synchronized with the codebase through the CI/CD process. This ensures that the dependency graph in CodeGraph always reflects the most up-to-date version of the project.

3.4 Requirements

The requirements is a set of features that CodeGraph has to contain. They are defined using the MoSCoW method [21]. This method ranks all requirements as one of the following four categories from most critical to least critical:

- **Must have:** This indicates that the requirement is critical to the correct working of the application and the successful achievement of all the objectives. The project will not be considered completed until all these requirements are fulfilled.
- **Should have:** These requirements can be equally critical as *must have* requirements, but they are less time critical. They are very important for the completion of the objectives, but they can potentially be temporarily postponed

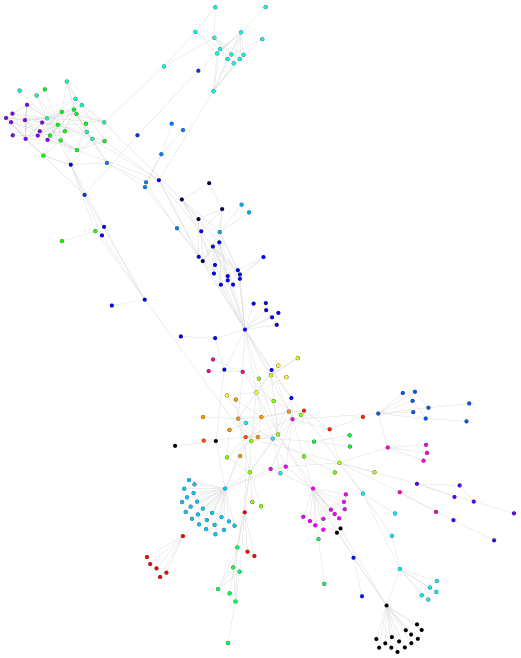


Figure 26: A network graph with a simple force layout and coloring based on clustering.

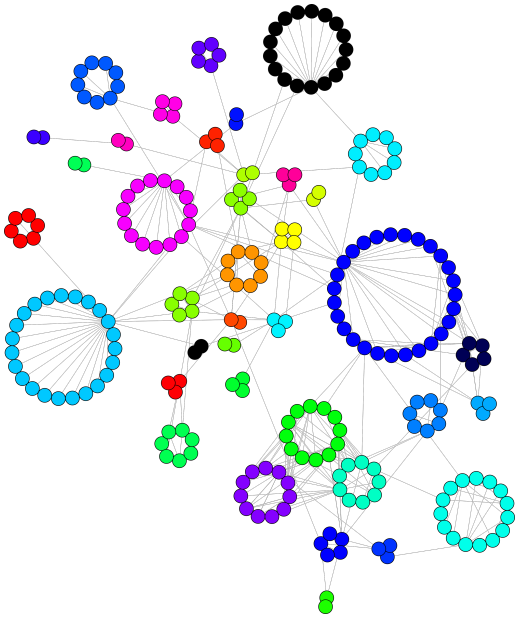


Figure 27: The same network graph with a layout based on clustering.

in favor of a less-qualitative, but easier and faster-to-implement solution. An MVP does not require all these requirements to be completed.

- **Could have:** These requirements have a high likelihood of improving the user-experience, but they are not critical to the achievement of the objectives.
- **Won't have:** This indicates that the requirement will probably increase the quality of the application, but less than *could have* requirements. These requirements will not be developed in the first iteration of the application.

Traditionally, these requirements form a contract between the developer, me, Robert van Barlingen, and the client, Nebulae represented by Joshua Hulpiau. The requirements define exactly what is expected of the developer. However, in this case I was not responsible for the completion of the entire CodeGraph application, but rather the completion of certain specific aspects of CodeGraph. Therefore these requirements do not reflect what is expected of me, but rather how the objectives of CodeGraph will be achieved.

Section C in the appendix contain the entire list of requirements. They are split up in six categories:

- **Project:** These requirements pertain to CodeGraph project configurations.
- **GUI:** These requirements specify what Graphical User Interface elements should be present in CodeGraph.

- **Dependency Graph:** These requirements describe all the features that the visualization of the dependency graph should contain.
- **Statistics:** These requirements list all the statistics about the project and individual nodes that should be provided by CodeGraph.
- **Filtering:** These requirements pertain to the user's ability to filter out certain data such that the dependency graph contains fewer nodes.
- **Data:** These requirements specify what data CodeGraph should contain and how it is handled.
- **Setting:** These requirements specify what settings should be available to the user to adapt the application to their personal preferences.

3.5 Architecture

CodeGraph is designed as a web application that runs almost entirely client side. It gets its data from an remote git repository and then it performs all the processing and visualization in the user's browser.

It works as follows: The user maintains his JavaScript project in a remote git repository such as github [7], gitlab [8] and bitbucket [9]. He then goes to the CodeGraph site where he creates a project. This mainly involves providing all the necessary information and permissions for CodeGraph to connect with the remote git repository. From then on, whenever the user accesses the CodeGraph project, CodeGraph fetches the JavaScript project from the git repository. This happens through the Gloow API. Gloow [10] is a knowledge management tool that is currently in development at Nebulae. It allows the user to create network graphs of knowledge. A possible use-case could be a journalist researching a story and mapping all the different facets of the story in a network graph. For this application, Nebulae had already developed a data-structure for network graphs and an accompanying API-endpoint that allows data to be fetched and pushed. For CodeGraph, Joshua enhanced the API-endpoint to take a JavaScript project as input, compute the dependency graph data and push all this data to the client in a concise data structure. The computation of the dependency graph based on the JavaScript project is completed using dependency cruiser [11]. This is an open-source library that computes the dependency graph of a JavaScript or TypeScript project and exports it to any number of different formats, including JSON. The Gloow API takes this JSON output from dependency cruiser, turns it into its own JSON format and passes it to CodeGraph. Figure 28 shows how CodeGraph fits within its environment. This part of CodeGraph has not yet been fully developed at the time of writing. Therefore, this architecture is still prone to change.

[7] https://github.com/
[8] https://gitlab.com/
[9] https://bitbucket.org/
[10] https://nebulae.dev/projects/gloow
[11] https://github.com/sverweij/dependency-cruiser

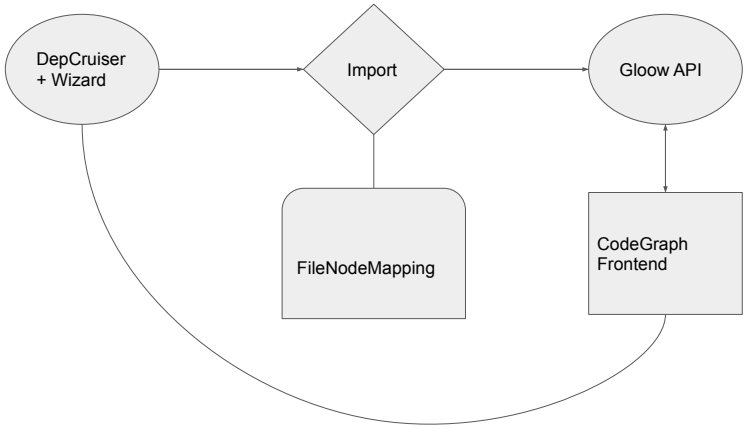


Figure 28: Layout: The structure of CodeGraph.

CodeGraph is developed using Facebook's React library [12] using TypeScript [13]. The GUI consists of four main sections shown in Figure 29: the navigation bar, the left menu bar, the right menu bar and the graph window.

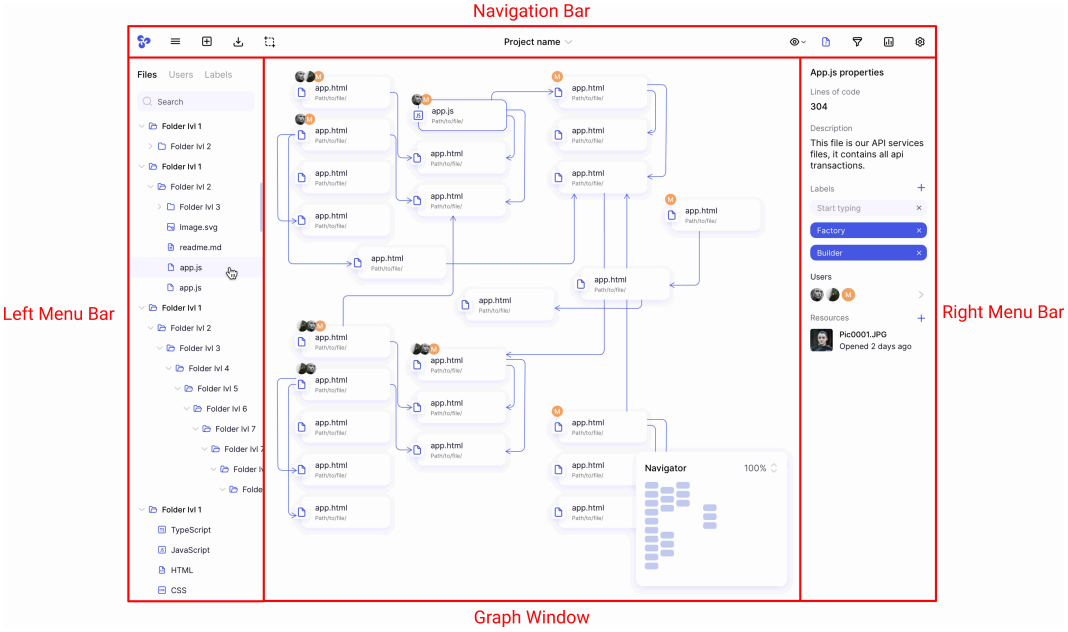


Figure 29: The GUI consists of four parts: the navigation bar, the left menu bar, the right menu bar and the graph window.

The navigation bar allows the users to access all the features of the application. As you can see on Figure 30, the navbar contains a group of icons on the left side and

[12] https://reactjs.org/
[13] https://www.typescriptlang.org/

icons on the right, with a project selector in the middle. From the left to the right, the icons include the CodeGraph logo, which redirects the user to the homepage. The selector menu toggle, which hides/shows the selector menu in the left menu bar. The Add button, which allows the user to add elements such as comments and resources to the project. The Download button, which allows the user to download the dependency graph as a JSON or CSV file. Finally, the screenshot button, which allows the user to download the dependency graph as an image in either SVG or PNG format. In the middle of the navbar, the project selector is located. This is a dropdown that allows the user to select the CodeGraph project that should be loaded. The icons on the right side include, from left to right: the mode selector, which allows the user to switch between the general and the difference mode. In the general mode, the dependency graph shows the state of the project after a git commit, by default the latest commit in the main branch. In the difference mode, the focus is on the difference between two commits, i.e., which files were added, removed or changed between two commits. The layout selector, which allows the user to change the layout that is applied to the dependency graph. The node information toggle, which hides/shows the node information in the right menu bar. The statistics toggle, which hides/shows the project statistics in the right menu bar. And finally, the settings button, which opens a settings windows which allows the user to change the project settings.

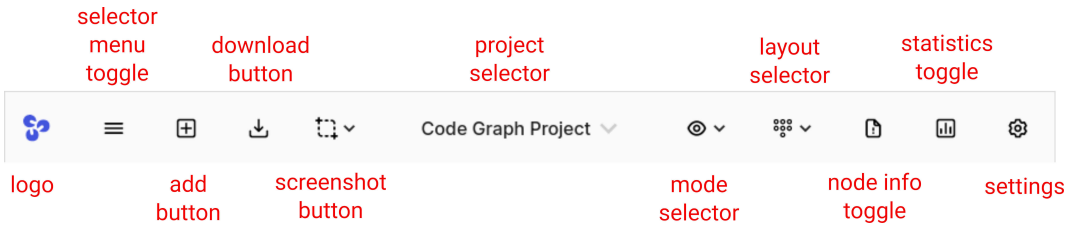


Figure 30: The navigation bar gives access to all CodeGraph's features.

When the left menu bar is opened, it shows a menu with three tabs: files, users and labels, as shown in Figure 31. Underneath the tab navigation, a search bar is placed. This search bar allows you to search for specific files, labels or users. The files tab contains the hierarchical file and folder structure of the JavaScript project. On the left side of each file or folder is an icon indicating the type of file or folder. On the right side of each file or folder is an eye icon and a focus icon. These allow the user to filter the nodes to be visualized in the dependency graph. The eye is toggled on by default. Toggling it off for a file hides the corresponding node in the dependency graph. Toggling it off for a folder hides all the nodes corresponding to all the files in the folder. This works recursively, so hiding a folder also hides all the files in a subfolder of the original folder. The focus button next to each file and folder does the exact opposite. It is off by default and pressing it makes only that file or folder visible. This allows the user to quickly focus on a few files or folders instead of having to hide all the ones the user is not interested in. Hiding/Showing works based on blacklisting, while focusing works based on whitelisting. Finally, hovering over a file or folder in the files tab also highlights the corresponding node(s) in the

dependency graph.

The Users tab contains a list of all the users that have worked on the JavaScript project. This is information is directly available from the git repository. For each user listed, the same filtering functionality is available as in the files tab. For each user, you can hide or focus, which shows or hides all the files that the corresponding user has worked on.

The labels tab contains a nested listing of all the labels. Each node can have one or more labels associated with it. These labels can either freely and manually be assigned by the user, or they can be generated automatically by CodeGraph. CodeGraph automatically analyzes the files and assigns some basic labels, for example, each file gets a label based on their file type: HTML (.html), JavaScript (.js), TypeScript (.ts), CSS (.css), etc. Ideally, more sophisticated file recognition could be executed, for example, CodeGraph could be able to recognize and label react components, redux reducers[14] and actions and angular [15] components, services and guards. The user is also able to create custom labels and assign them to different nodes. For each label, the user can then once again filter the dependency graph to hide or focus all the nodes with the specified label.

The right menu bar can either contain the information of the selected node or the statistics of the project as shown in Figure 32. The contents of the right menu bar is changed using the icon buttons in the navigation bar. The node information is based on the selected node. The user can select a node by clicking on the node in the dependency graph, which will also highlight the node in the dependency graph. The node information includes the following information: the number of lines of code contained in the file, a description of the file, which the user can add himself, a list of labels associated with the file, a list of the users that have edited the file in the past and a list of resources. Resources are files and links to external resources that the user can manually add to a node. Examples include images, wikipedia articles, documentation, etc.

The project statistics include the number of nodes, the number of links or edges, the number of clusters, the number of connected components, the ratio of links to nodes, the diameter of the dependency graph, a list of the nodes that have been imported the most frequently by other files, a histogram of the number of times a file has been imported, a list of the nodes that import the most other nodes and a histogram of the number of imports in the files.

A cluster of nodes is a group of tightly connected nodes. Clusters are computed using a clustering algorithm. There are many different clustering algorithms. We use the Markov Clustering Algorithm (MCL) [3] provided by the cytoscape library [16]. Cytoscape is a an extensive open-source library focused on the visualization and manipulation of network graphs. It is heavily used for bioinformatics studies, the study of social networks and the semantic web. MCL is based on the idea that if you start walking from a node to other nodes along the connecting edges (random walk), that you are then more likely to end up within the cluster. Discovering clusters

[14] https://redux.js.org/
[15] https://angular.io/
[16] https://js.cytoscape.org/

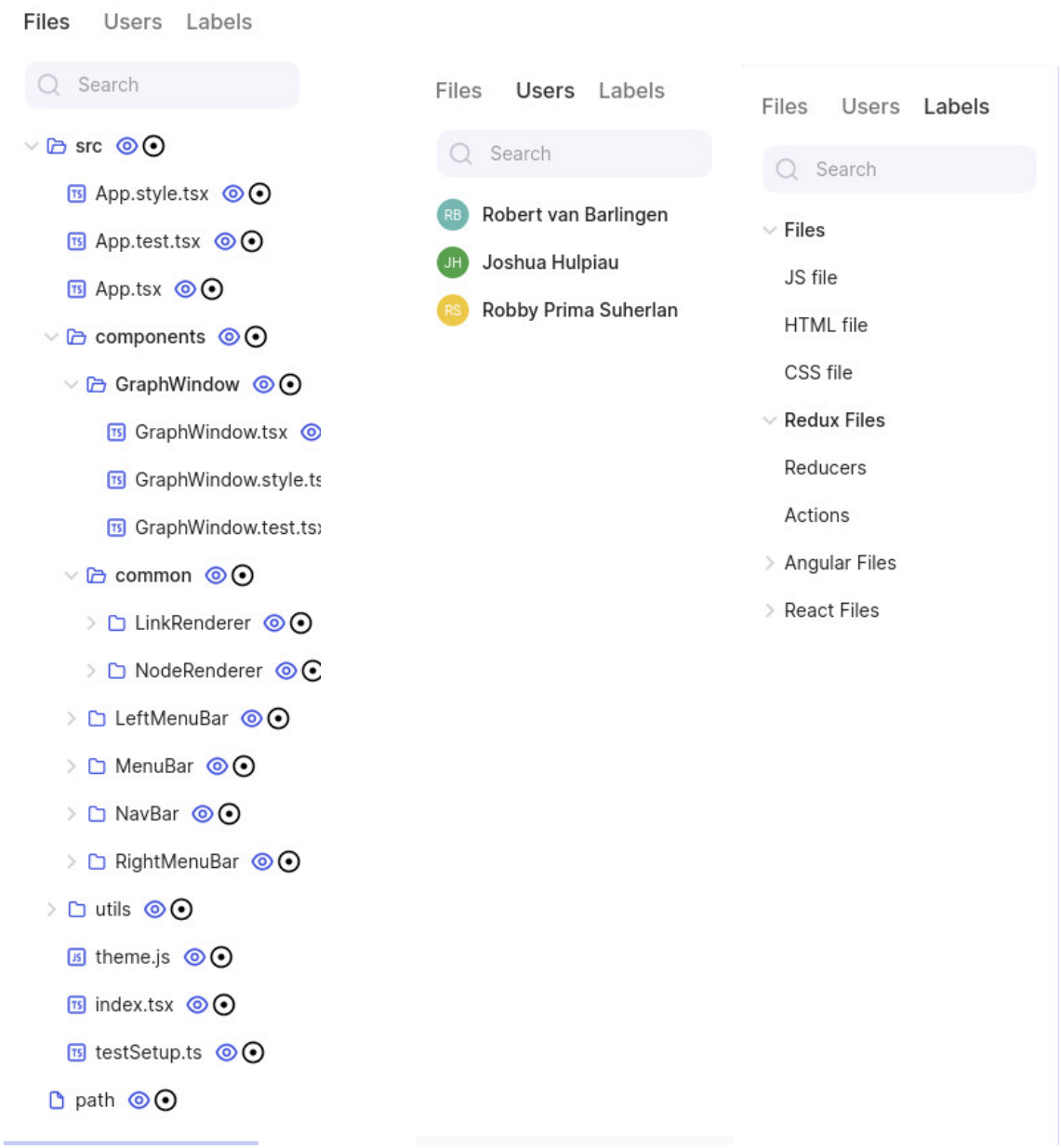


Figure 31: The left menu bar contains three tabs: files (left), users (middle) and labels (right).

is then a matter of finding the probability of ending up in a node given that you started in a certain node after X steps, with X large. This is done by taking the Markov matrix to a large power. Many other clustering algorithms exist, but we chose MCL, because it is sufficiently fast for the size of network graphs that we aim to work with. But more importantly, many other clustering algorithms require the user to first define the number of clusters that need to be computed, while this isn't the case for MCL. MCL simply creates as many clusters as it finds based on the defined parameters. This is important for our use case because we want to make the

clustering process as passive as possible for the user. We do not want the user to have to define how many clusters are needed. This also wouldn't make sense as there is no clear ideal number of clusters. It should simply compute the best clusters, i.e., the clusters, which are the most tightly connected. In the future, however, I might experiment with different clustering algorithms. If a clustering algorithm were to require a predefined number of clusters, then I could define a function that scales with the number of nodes in the dependency graph. This would ensure that a decent number of clusters is chosen.

Through my tests, MCL has proven to give good results for dependency graphs, however, more research would be required to investigate what the exact nature of the clusters is. For example, one could image that a cluster might correspond with a single reusable react component. However, this would have to be closely investigated.

The diameter of a network graph is the longest of all the shortest paths between any two nodes. This gives an indication of the size of the graph. However, one caveat is that if the dependency graph consists of two or more disconnected graphs, then the diameter is infinity. This happens more often than not as most projects include at least a few files that neither import any other files nor are imported themselves. However, if the user filters out these small disconnected graphs, then the statistics show a relevant diameter once again. The diameter is computed by first computing the shortest path between any two nodes using Dijkstra's algorithm [9] and then iterating over all these shortest paths to find the longest one. We once again use Cytoscape's implementation of Dijkstra's algorithm.

The graph window shown in Figure 33 is the biggest and most important part of the GUI. This contains the visualization of the dependency graph. The visualization is rendered using D3[17], an open-source, JavaScript data visualization library. This allows us to completely define what the nodes and edges look like. D3 then takes our dependency graph data, including the position of the nodes and edges, together with our definition of the what the nodes and links look like, and renders the dependency graph as an interactive and dynamic SVG image. D3 also includes the ability to zoom and translate the viewport. Zooming is performed using the scroll-wheel and translating is accomplished by dragging the canvas. This allows the user to zoom in or out and drag the viewport to whatever part he is interested in.

In order to maintain a sense of orientation while performing zooming and translation, the graph window also contains a minimap in the bottom right corner. This minimap shows a simplified view of the full dependency graph. When zoomed in, the dependency graph also shows a rectangle representing the viewport. The user can also perform viewport translations by dragging the viewport rectangle in the minimap. Figure 34 shows an example of a zoomed-in viewport with the minimap showing the viewport rectangle.

The placement of the nodes and edges is defined by the different layouts that the user can choose between. A layout is an algorithms that decides the position of each node and graph in the canvas [34][12]. Good layouts provide order to the chaos that is a network graph and the best layouts even provide insights into the data, for

[17] https://d3js.org/

Figure 32: The right menu bar can either contain the information on the selected node (left) or statistics on the project (middle and right).

example, by grouping together related nodes. Coming up with good layout algorithms has been the subject of many different studies. Generally these layouts fall in one of two categories: discrete and continuous layouts. Continuous layouts place the nodes through an iterative process, often based on some physical simulation. These layouts start by placing all the nodes at some initial location, often randomly defined. Then they perform a number of iterative steps, in which the translation of each node in each step is defined by the position of the other nodes. For example, nodes that are linked together often exert a pulling force towards each other, while disconnected nodes

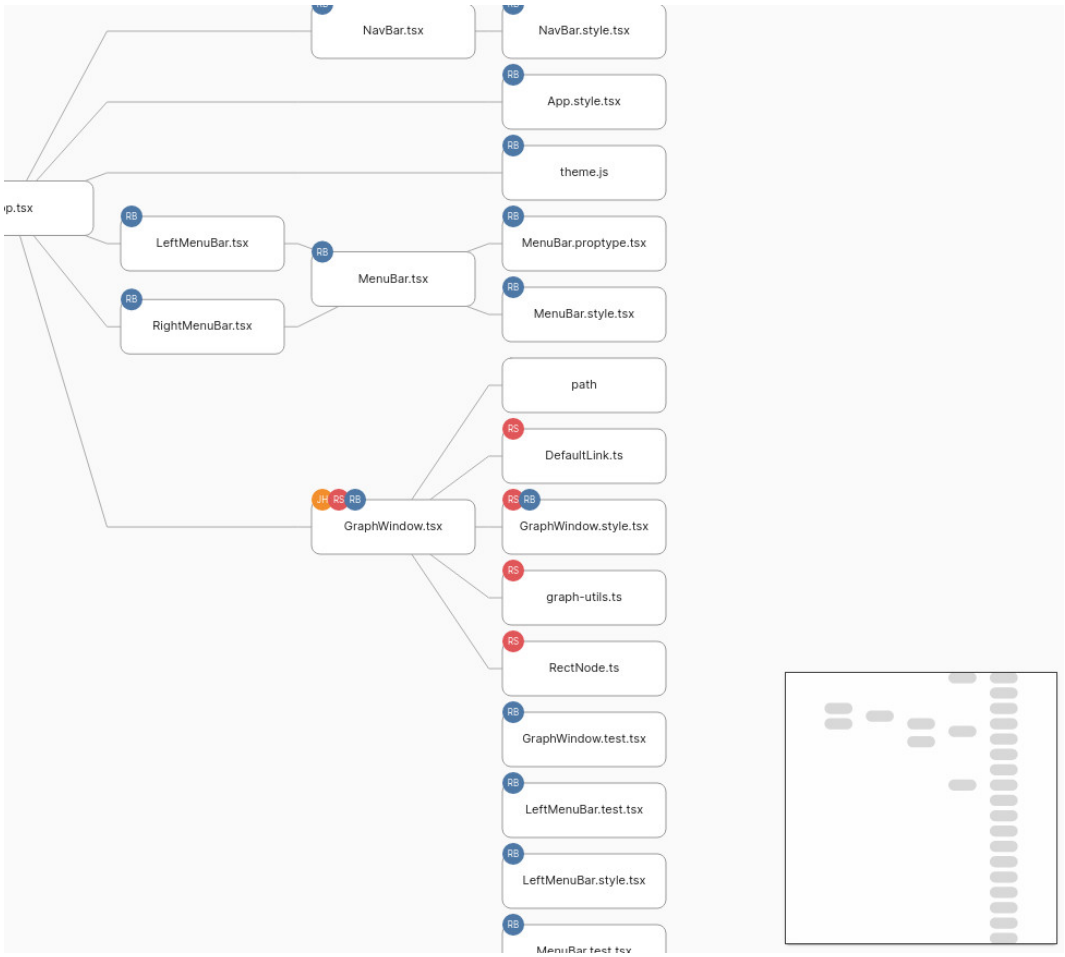


Figure 33: The graph window.

exert a pushing force. Discrete layouts, on the other hand, compute the position of the nodes and edges in a single step. They often accomplish this by maximizing one or more quality measures, for example, the number of crossing edges. Layouts usually also provide different parameters that the user can use to modify the layout. For example, a layout based on a physical simulation could provide a parameter that defines the strength of the pulling force between two connected nodes.

I have implemented the following different layouts: dagre, force, radial, tree, multi-focal, circle, concentric, breadth first, cose, cola, avsdf, cise, cose-bilkent, fcose, klay and random. Dagre [18] was implemented using a stand-alone external library. Force, radial, tree and multi-focal were implemented using the D3-force library [19]. Finally, circle, concentric, bread first, cose, cola, avsdf, cise, cose-bilkent, fcose, klay and random were implemented using cytoscape and corresponding 3rd party cytoscape plugins. Cytoscape offers some layouts out of the box, but they also provide the possibility to use plugins. Appendix A contains figures of all the different layouts. What follows is a short description of how each layout works:

[18] https://github.com/dagrejs/dagre
[19] https://github.com/d3/d3-force

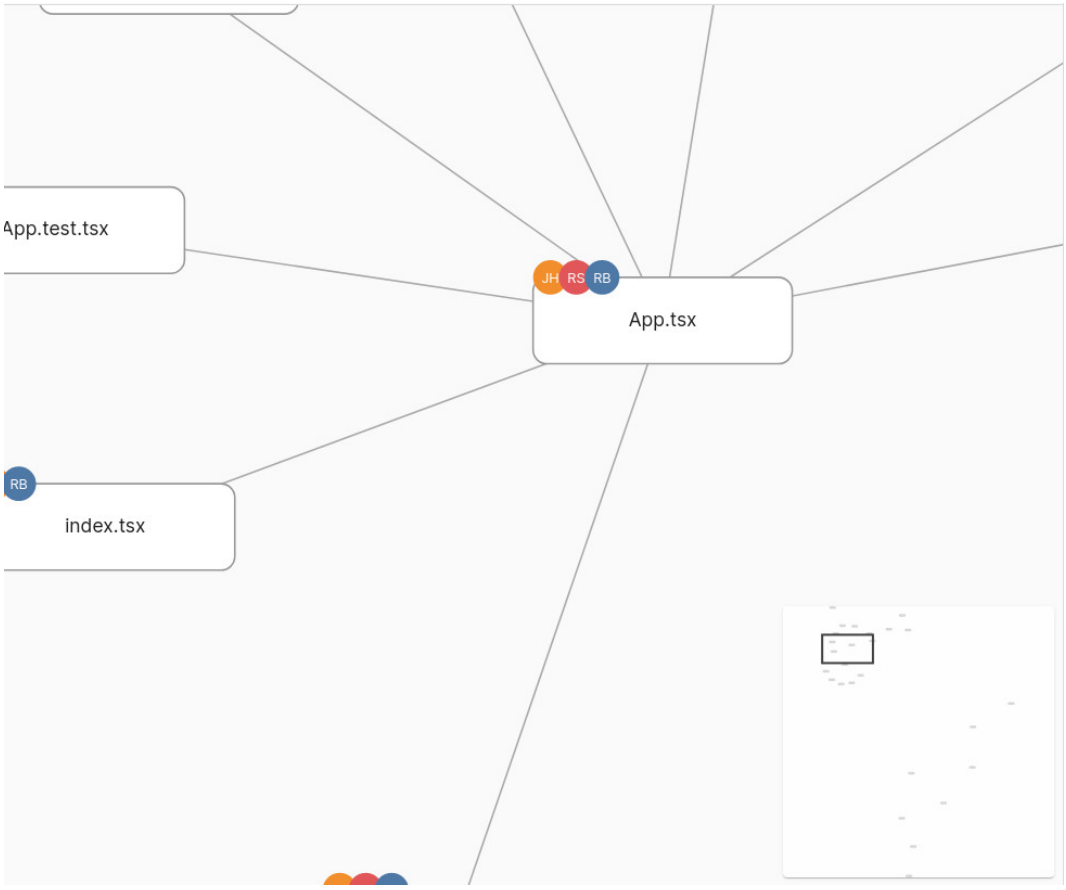


Figure 34: The minimap.

- **dagre:** Dagre is a discrete layout that places the nodes hierarchically, while minimizing the number of crossing links. The general implementation is inspired by *"A Technique for Drawing Directed Graphs"* [18] and the algorithms for minimizing the number of crossing links is based on *"2-Layer Straightline Crossing Minimization"* [23]. In my experience, this layout performs quite well, even with relatively large graphs, while proving one of the most organized and aesthetically pleasing results. For this reason, Dagre is the default layout of CodeGraph. The only negative aspect of dagre is that, for large graphs, it tends to create very elongated graphs. For example, we have configured dagre to place the nodes hierarchically from left to right. In this case, it tends to create five or six columns in which all the nodes are placed. This makes the entire graph very long along the y-axis and fairly short along the x-axis. It would have been better if it had created graphs with a more square bounding box. Dagre does provide a parameters, which allows the user to increase the space between the columns. This improves the result, but it would have been even better if dagre was able to spread the nodes out over more columns. Another parameter that does help is the rank computation algorithm. Through this parameter, Dagre allows the user to choose between three different algorithms that dagre uses to compute the hierarchy of the graph. This helps somewhat

as some algorithms create a more elongated graph than others, but is still a fairly limited parameter.

- **force:** This layout is a continuous layout based on a velocity Verlet numerical integrator [38]. It simulates physical forces between the nodes. More specifically, it defines an attracting force between connected forces and a repulsing force between disconnected nodes. This nicely groups together tightly connected clusters of nodes, while spreading them out. Additionally, there is also a central gravitational force. This keeps the dependency graph central in the canvas. This layout also gives nice results, although it does require a more careful choice of parameters than dagre. For example, if the strength of the repulsing force between nodes is set to a constant value, the nodes will either be too far apart in small dependency graphs or too close to each other in a big dependency graph. In order to get the best results, the parameters should be set dynamically based on, for example, the size of the dependency graph. One aesthetical problem that I have experienced with the force layout is the fact that disconnected nodes tend to get pushed very far away from main dependency graph.

- **radial:** The radial layout extends the force layout by additionally placing the nodes in rings around a root node. First, the user must define the root node. This is the first file that is executed in the JavaScript project, usually called index.js or App.js. This node is placed in the center of the canvas. Then the shortest distance between every node and the root node is computed. Rings of increasing radius are defined around the root node. For each node, then node is then placed on the n'th circle counted from the center, with n being the distance from the node to the root node. All disconnected nodes are placed on the k+1 circle, with k being the longest shortest distance to the root node among all the connected nodes. For the best performance, the radius of the circles should be dynamically set depending on the number of nodes in the graph. This layout provides an aesthetically pleasing and well-organized result.

- **tree:** The tree layout is very similar to the circle layout, but instead of having the root node in the center and all the other nodes around it, the root node is placed at the top and and all the other nodes are placed in a tree-like structure underneath the root node. Similar to the circle layout, there are a number of layers underneath the root node and each node is placed on the n'th layer counted from the top with n being the distance of that node to the root node. This layout suffers from the fact that the nodes on a layer tend to be group together too much. Ideally, the nodes in a specific layer would all be placed next to each other on a straight line, however, this was often not the case. Usually, tightly connected clusters of nodes on a single layer are placed on top of each other, rather than next to each other. This is because the strength of the attracting forces between connected nodes is greater than the attracting force that pull each node to the correct layer. I attempted to solve this by fixing the y-component of each node to the y-position of the correct layer, however this cause errors in D3's force algorithm. I also attempted to fix the problem

by making the attracting force of the layer far greater than the attracting force between connected nodes. However, this results in connected nodes on two sequential layers not being underneath each other, which results in many crossing lines between layers, which looks chaotic. Ultimately, it is quite hard to make this layout behave appropriately, especially since the layout has to work well with both small and large graphs.

- **multi-focal:** The multi-focal layout creates multiple gravitational points, which the nodes are drawn to based on the labels associated with the nodes. The user can define a list of labels to consider. For each of these labels a gravitational point is defined and these gravitational points are placed in a circle. Each node is then attracted to each gravitational point, for which it contains the gravitational point's label. This means that if the node only contains a label of one point, then the node will be placed very close to that point, while if it contains multiple labels associated with different points, then the node will be located in the middle of the different points. If not too many different labels are selected, then this provides a nice way of grouping nodes together based on labels. I have only implemented this with labels, but this could, of course, be extended to also allow the gravitational points to be defined based on other properties, such as which users worked one the node in the past.

- **circle:** The circle layout places all the nodes in one big circle. It is quite difficult to see any structure with this layout. Additionally, the circle quickly becomes extremely big with increasing number of nodes. This makes it very hard to get an overview of all the files at the same time. However, this layout works extremely well for quickly seeing, which are some of the most important nodes as it is very easy to identify nodes with many incoming and outgoing edges in this layout.

- **concentric:** The concentric layout works similarly to the radial layout. The radial layout is based on the continuous D3 force library, while the concentric layout is implemented using a discrete cytoscape extension. But they both place the nodes in circles around the root node with the distance of each node to the center being defined by the distance of that node to the root node. The concentric layout, being a discrete layout, it far more precise. For example, each circle around the center is evenly filled with the occupying nodes. The radial layout, on the other hand, is based on a physical simulation, and as a results feel far more organic. Additionally, because of the attracting forces between connected nodes, there also tend to be fewer crossing links with the radial layout than with the concentric layout. The concentric layout, on the other hand, has a smart way of defining the radius of the circles. This makes it scale particularly well with increasing number of nodes. Both layouts work very well and it will mostly come down to personal preference.

- **breadth first:** The breadth first layout is similar to the tree layout. Both place the root node at the top, with all other nodes in rows underneath the

root node, with the row that a node is placed in being defined by the distance of the node to the root node. The comparison between breadth first and the tree layout is similar to the comparison between the concentric layout and the radial layout. The tree layout is continuous and based on a physical simulation, while breadth first is a discrete layout. This makes it more precise, while the tree layout feels more organic. Once again, the tree layout tend to have fewer crossing edges. However, the fact that the breadth first layout is able to place all the nodes evenly over the length of a layer makes it gain my preference. The only thing to keep in mind is the fact that the distance between the layers should be dynamically set.

- **cose:** The cose layout uses a physical simulation similar to D3's force layout. It replicates a spring-like force between connected nodes. It is based on *"A layout algorithm for undirected compound graphs"* [16]. The layout provides parameters to define the ideal edge length, the node repulsion strength, the elasticity of the edges, etc. This allowed me to get very good results for small graphs. However, I had a very hard time to get equally good results for big dependency graphs. Nodes tend to be close together, to the point where they even overlap. There also tend to be more crossing edges compared to, for example, D3's force graph. However, it is very snappy, even for big graphs.

- **cose-bilkent:** The cose-bilkent layout is an extension of the cose layout created by the i-Viz lab at Bilkent University. Compared to the cose layout it aims to improve the result at the expense of some increased computation time. In my experience, the cose-bilkent generates substantially better results than the cose layout. The nodes are spread out far better. However, there still seem to be more crossing edges than with D3's force layout, which makes it a little bit more chaotic. One very nice advantage of cose-bilkent over the force layout is the fact that cose-bilkent places all the disconnected nodes in a small contained group, while with the force layout, disconnected nodes tend to wander off in different directions. In theory, the cose-bilkent computation time should be worse than the cose layout, however, I noticed very little difference between the two layouts. Both were very snappy. Maybe this difference becomes more apparent with greater graphs than are taken into consideration for CodeGraph.

- **fcose:** The fcose layout is an extension of the cose-bilkent layout. It claims to achieve the same results as cose-bilkent twice as fast. It does this by using techniques from spectral graph layouts described in *"SSDE: Fast graph drawing using sampled spectral distance embedding"* [8]. Spectral layouts use the eigenvectors of the LaPlace matrix of the graph to compute the layout very fast [19]. Cytoscape recommends fcose as the first layout to choose if the user want a layout based on a physical simulation. From my experience, these claims are accurate. Fcose and cose-bilkent achieve near identical results and fcose seems to be faster. Although dependency graphs are rarely sufficiently large to really notice a big difference between the two layouts.

- **cola:** Cola is a layout based on physical simulation. It is designed for small graphs and this is apparent. For small graphs, it provides really nice results very quickly. For large graphs, however, it tends to take a very long time to compute. For large graphs, the nodes are spaced apart very nicely, but there are many crossing edges. This makes this layout one of the less useful layouts.

- **avsdf:** The avsdf layout places the nodes in a circle similar to the circle layout. However, it does it a bit smarter than the circle layout as it also minimizes the number of crossing edges. It is based on *"New circular drawing algorithms"* [20]. It works very well and has the same advantages as the circle layout.

- **cise:** Cise is a spring-based physical simulation layout. Furthermore, it groups all the nodes in a cluster together by placing them in a circle. This creates a graph consisting of groups of nodes placed in a circle connected together. It also performs different operations to minimize the number of crossing edges. It is based on *"CiSE: A Circular Spring Embedder Layout Algorithm"* [15]. The results are very aesthetically pleasing and well-organized even for very large graphs. The only caveat is that it requires meaningful clusters to function optimally. CodeGraph uses MCL, which results in aesthetically pleasing and structurally good clusters, but more research is required to discover whether the clusters are also functionally meaningful.

- **klay:** The klay layout is similar to the dagre layout. They are both discrete layouts that generate similar results. It also places the nodes in a hierarchical structure. It even uses many of the same underlying algorithms. Klay does provide more parameters than Dagre, which allows the user more control over the result. On the other hand, Dagre gives the edges a very nice shape, based on the placement of the connected nodes, while klay simply connected nodes through a straight line. The choice between klay and dagre comes down to personal preference.

- **random:** The random layout simply places all the nodes randomly within a predefined bounding box. This layout has very little benefits. I use it mainly as a benchmark to compare other layouts.

There are some additional cytoscape layout extensions that are not mentioned here, including: polywas, Euler, Elk, springy and spread. I did try to implement these, but they did not work and they gave me varying errors. Since there are already a significant number of effective layouts I decided not to spend any time to get these layouts working.

In conclusion, all the layouts provide a unique result and they all have their own strengths and weaknesses. However, it is my opinion that dagre and klay provide the most well-organized results. This is no surprise as they are both discrete layouts that minimize the number of crossing edges. Of all the continuous layouts, I like the force layout the best as it is reasonably snappy for large graphs, spreads out all the nodes very well and gives an overall organic and aesthetically pleasing result. The cose layout and the derivatives cose-bilkent and fcose also provide decent results, but

for larger graphs they tend to have more crossing edges than the force layout, which makes it more chaotic. The cise layout also gives a very unique, well-organized and aesthetically pleasing result. However, since it puts so much emphasis on clusters it is only useful if the clusters are defined meaningfully. The circle layout and the radial layout can also provide unique, useful insights in specific situations.

However, I do want to stress that the best strategy for a user working with a large JavaScript project is to first filter out any nodes the user is not interested in and then choose the appropriate layout. A layout can only go so far in the organization of a large graph. Reducing the graph is a far more effective strategy.

When implementing layouts using the cytoscape library, I did face one challenge. In CodeGraph, cytoscape is generally run in headless mode, meaning that we only use cytoscape for performing computations, not for the actual visualization of the graph. Instead we use our own custom D3 visualization, which allows us greater flexibility. However, through trial and error, I discovered that cytoscape does not update the position of the edges when running in headless mode. In order to remedy this, I implemented a hidden svg canvas, in which cytoscape performs the rendering invisible to the user. This fixed the issue.

Currently, all the graph computations are performed in the main thread. This means that all the processes, including user-interactions with the GUI, the computation of the placement of the nodes and edges, the computation of shortest paths and the rendering of the dependency graph in the canvas, all have to wait for each other. This works fine for small graphs (+-100 nodes), but for bigger graphs, CodeGraph does become a bit sluggish. In the future, I would like to implement web workers that each take care of a different task, such that the computation is spread out over multiple threads. This should considerably improve the performance of CodeGraph.

Each node in the dependency graph is represented by a rectangle with some basic information. Figure 35 shows a node. In the center of the rectangle, the name of the file is displayed. Underneath the filename, in a more subtle style, the path of the file relative to the root directory of the project is mentioned. To the left, the rectangle contains an icon specifying the file type, for example, a JavaScript file, an HTML file, etc. In the top left corner, the node contains small profile pictures of up to 3 users. These are the users who most recently edited the file. Finally, on the right side, the rectangle contains up to two labels associated with the node. The edges are simply represented by a line with an arrow indicating the direction in the center.

Finally, pressing the settings button in the navigation bar opens a settings page as shown in Figure 36. The settings page contains three tabs: The project tab, the layout tab and the permissions tab. The project tab contains the settings that allow the user to link the project with a git repository, the layout tab contains all the parameters of all the different layouts. This allows the user to modify the different layouts. Finally, the permissions tab allows the user to add or remove users to the project. At this time only the layouts tab has been implemented.

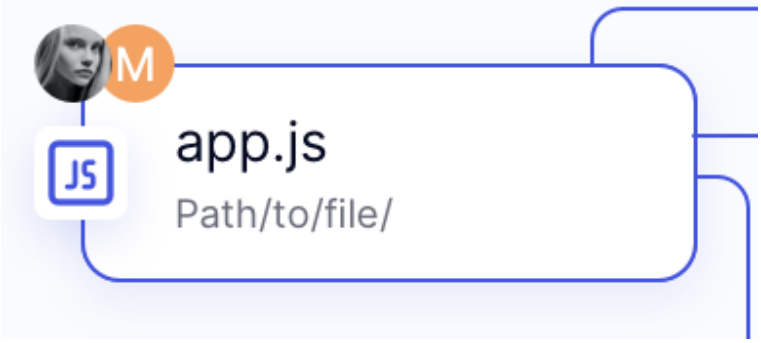


Figure 35: The representation of a node in the dependency graph.

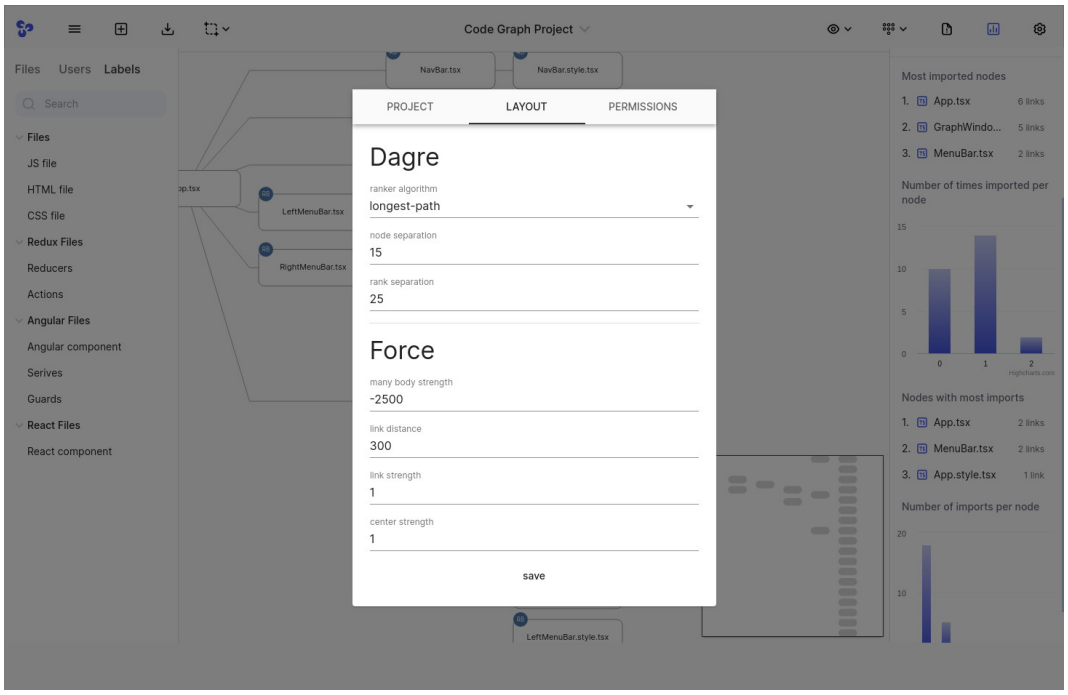


Figure 36: The settings

3.6 Development

The development of CodeGraph has taken place over a period of five months, from January 2021 up to and including May 2021. At first, the aim of the project had not been to develop CodeGraph. Rather, Nebulae was working on their Gloow application, a 2D graph-based knowledge organizer, and I was asked to create a 3D application that could be used to extend Gloow. I spent all of January creating a first prototype with the goal of experimenting with different features. To this end, I created an 3D graph application using the 3D Force Directed Graph library[20]. Under the hood, this library uses Three.js[21], a JavaScript 3D rendering library, for the

[20] https://github.com/vasturiano/3d-force-graph
[21] https://threejs.org/

visualization of the 3D graph and either D3-force-3D[22] or nGraph[23] for the layout of the 3D graph. Screenshots of this first prototype can be found in appendix D.1. It included some features that were later directly copied over to CodeGraph, including the graph statistics, the color-coded clustering of nodes and the ability to take a screenshot of the graph and download the raw data. However, many features also did not get included in CodeGraph including the ability to expand and contract clusters such that an entire cluster is represented by a single node, the ability to add or remove nodes and links on the fly and RDF tags. Resource Description Framework (RDF) tags [24] is a web standard that is specified by the World Wide Web Consortium (W3C)[24]. It allows resources to be described in a subject - predicate - object form. For example, in the sentence *"the sky is blue"*, *"the sky"* is the subject, *"is"* is the predicate and *"blue"* is the object. Each predicate, each object and many subjects as well are defined by a URI that is publicly available on the web. This way resources on the web can describe themselves using external predicates and subjects existing somewhere else on the web. This allows very powerful cross-document integration over the web. In the first prototype, I experimented with defining the links and nodes using these RDF tags. This allowed the application to automatically detect what the nodes and links meant and retrieve related information from the web. Figure 37 show how the application automatically displays a description and a photo of Brad Pitt when the user selects a node with an RDF tag specifying that the name of the node is Brad Pitt. This is accomplished by querying DBpedia for the specified RDF tag. DBpedia [25] is a database with the all the RDF data underlying wikipedia. So, when querying DBpedia for a specific RDF tag, it will provide the wikipedia data that is related to that RDF tag. This is a very flexible and powerful feature that allows the application to give extra context no matter what kind of information the user is working with. If there is a wikipedia article for it, then the application can provide the information. Since many documents across the web use the same RDF tags, DBpedia can even provide information retrieved from sources outside of wikipedia. The application also features a 3D graph visualization, in which that user can pivot, pan and zoom in 3D space.

During the development of the first prototype in January and for the remainder of the entire development process, I had a meeting with Joshua from Nebulae about every week. During these meetings, I presented the work that I had accomplished that week, he provided feedback and we discussed the continuation of the project and, more specifically, what I would work on the subsequent week. This process is in line with the agile software development process [5]. In fact, we utilized multiple techniques from the scrum development process [32] such as having weekly sprints ending in a meeting with the product owner during which we discuss the newest version of the application. We also maintained a kanban board [2] using jira[26], on which we maintained a list of tasks that had to be done, tasks that were currently in

[22] https://github.com/vasturiano/d3-force-3d
[23] https://github.com/anvaka/ngraph.forcelayout3d
[24] https://www.w3.org/
[25] https://www.dbpedia.org/
[26] https://www.atlassian.com/software/jira

Figure 37: By using RDF tags, the application can automatically retrieve related information.

progress, tasks that were finished and had to be be reviewed and tasks that were completed. Of course, scrum is a process mostly aimed at teams, while I was the only person who worked continuously on CodeGraph. Therefore, some parts of the scrum process were not applicable in this case.

Near the end of January, Joshua and I discussed the further development during a weekly meeting. We both felt that the application was too general, and could therefore not provide powerful features. Therefore, we decided to shift focus and instead create CodeGraph, an application focused on the visualization of the dependency graphs of JavaScript projects. We discussed possible use-cases, requirements that the application should fulfill and features that it should contain. Then, we created a rough planning and time frame.

I started by creating a paper prototype of CodeGraph. This prototype can be found in appendix C.1. It has a few differences compared to the final design. For example, all the navigational icons are placed on the left side as opposed to the top. It also includes more interactivity features, such as a pencil tool that allows the user to create drawing on the canvas and a comment tool that allows the user to place comments. The placement of the minimap and the project files viewer is also different. Furthermore, I envisioned that the node information would be accessed by clicking a node, which would then open a pop-up window in the canvas. In the final design, this was changed to having all the node information be placed in the right menu bar. The benefit of the original design is that you can easily compare information from different nodes by opening multiple pop-up windows, while the benefit of the final design is that it is more consistent and well-organized. The original design also included a commit tree window in the bottom. This window would show all the git commits in all the different branches. This would allow the user to quickly change what commit the dependency graph in the graph window would be based on. Apart from the default mode it would also feature a difference mode, in which the

user could see the difference between commits. The user would first set the mode to *difference* through a dropdown menu and then he would select two commits in the commit tree window. Each node that was changed between the two commits would then have a small bubble that shows the number of lines of code that were added in green and the number of lines that were removed in red. Nodes that were added or removed in their entirety would be shown in green or red respectively. Finally, the original design also featured a singular node view. In this view, the entire canvas would be occupied by a rectangle that would represent a single node. This view would give detailed information on the node such as a list of all the functions and classes that are included in the node, a set of labels, a set of associated resources and a link to the entire file content. On the left side of the node rectangle it would also feature all the nodes that import the selected node and on the right side a list of all the nodes that the selected node imports.

I presented this prototype to Joshua, after which we made a list of alternations to the design. This design and the list of alterations were then sent to Alex Prokhorov, the UX/UI design expert at Nebulae, to create a new and improved design. This design can be viewed in appendix C.2. It features a far more modern, sleek look with a more well-organized and sensible placement of all the components. The design of CodeGraph aims to replicate this paper prototype as closely as possible.

After a few weeks of finalizing the design with Joshua, Alex, I started developing CodeGraph. Once again I worked in weekly sprint with a meeting with Joshua at the end of every week. I was the main developer on CodeGraph, but Joshua Hulpiau, Robby Prima Suherlan and Ega Kamalludin also contributed considerably to the codebase. Throughout the process, we maintained a Continuous Integration (CI) process[17]. Before starting development, I set up a CI process. This consisted of a framework that automatically runs all the Jest test[27], whenever a new commit gets pushed to the git repository. It also includes a linting process which automatically checks the code style of any new commits using eslint[28]. Finally, it also includes a google cloud[29] integration which automatically builds a docker image[30] and pushes it to a google cloud docker repository. A google cloud compute instance then pulls the docker image and starts a server hosting the newest version of CodeGraph. This whole process allows us to immediately notice if something is wrong with a commit and it provides us a live version of the software that can easily be tested and shared.

Whenever I implemented a new feature, I would first put the corresponding task on the kanban board in the *in progress* column. Then, I would create a new branch with a name corresponding to the ID of the kanban task. I would then work on it until it was ready to be integrated into the main branch. Then, I would merge it with the main branch. If I had worked in a larger team, then we would have had a more strict checking process, in which each merge to the master branch would first have to be reviewed by a different developer. However, since I was the only developer who continuously worked on the project, we opted to forgo this checking process. I

[27] https://jestjs.io/
[28] https://eslint.org/
[29] https://cloud.google.com/
[30] https://www.docker.com/

also tried to keep the tasks relatively small so that they could be implemented fast, making the chance of branches being out of sync as small as possible. Nevertheless, there were a couple of times that Joshua and I had modified the same file in different branches. This had to be resolved through the tedious process of manually resolving merge conflicts.

I developed the CodeGraph prototype from approximately Mid-February till Mid-May.

3.7 Testing

In order to test the usability of CodeGraph, I have performed a series of usability tests. These usability tests are conducted in person as follows: The test subject is seated in front of a computer. I present the user with the form of consent shown in appendix E. This form informs the subject of the purpose and nature of the study, what data will be collected and what will happen to the data. This includes the fact that the session will be recorded. It also mentions the fact that the test subject will remain completely anonymous. Once signed, I verbally inform the test subject that this session will be recorded. I turn on the recorder, which captures both audio and video of the test subject through a webcam, and simultaneously captures a screen recording. These recordings are captured using OBS[31]. These recordings are later analyzed to gain insights into how users interact with CodeGraph. Once the results have been gathered, these recordings are destroyed. At this point, I introduce the user to CodeGraph and give a brief demonstration. If the user is unfamiliar with the file explorer nautilus[32], the text editor gedit[33], the terminal gnome terminal[34], the IDE Webstorm[35] or the git repository bitbucket[36], then I also give a brief demonstration of these tools to the extend that the subject might use these tools during the usability test. If the test subject so desires, he is granted the opportunity to play with CodeGraph or any of the general applications. Then, the user is presented with the actual test as an online google form. This form can be viewed in appendix F. It consists of 5 sections:

1. **Start:** The first section of the form describes the setup of the test to the subject.

2. **General Information**: The second section asks for some general information about the test subject, including his age, gender, occupation, and whether he has experience programming JavaScript or programming in general. This information is used to analyze the data based on traits of the participants.

3. **Test Session with General Applications:** This section and the test contains a set of questions about a provided JavaScript codebase. The test subject is

[31] https://obsproject.com/
[32] https://linux.die.net/man/1/nautilus
[33] https://help.gnome.org/users/gedit/stable/
[34] https://help.gnome.org/users/gnome-terminal/stable/
[35] https://www.jetbrains.com/webstorm/
[36] https://bitbucket.org

supposed to find the answer to these questions using the software at hand. In this section, the user is allowed to use a file explorer, a text editor, a terminal, an IDE and an online git repository. Question 1 asks the participant to find how many import statement a certain file contains. Question 2 asks how many files import a module from a certain file. Question 3 asks how many lines of code a certain file contains. Question 4 asks how many files the project contains. Question 5 asks which file contains the most import statements. And finally, questions 6 asks for the initials of someone who has edited some file. The participant is allowed to skip a question if he is confident that he cannot find the correct answer. The participant is also timed when performing this test.

4. **Test Session with CodeGraph:** This section contains nearly identical questions as the previous section, only the specified file is different for each question. This way, the participant cannot simply recall the answers of the previous sections. Question 4 is replaced with a question asking how many import statements the entire project contains and question 5 is replaced with a questions asking, which file has been imported the most. For these questions, the participant is only allowed to use CodeGraph to find the correct answers. Since the questions in this section and the previous section are nearly identical, the participants' performance in both instances can easily be compared. Once again, the participant is timed while performing this test.

5. **Experience Survey:** Finally, the last section of the form contains a set of questions polling the user's subjective experience working with CodeGraph. The first question asks the participant to rate how enjoyable it was to use CodeGraph on a scale from 1 to 5. The second question asks the participant to rate the efficiency of CodeGraph from 1 to 5. The next two questions ask the participant for the positive aspects and the negative aspects of CodeGraph respectively. Finally, the last question gives the participant the opportunity to add any comments.

Once the test has been completed, I engage with the test subject in an informal conversation. During this conversation I try to gauge the user's experience when performing the tests and using CodeGraph. Once all the tests have been completed, I correct all the answers and analyze the results. The results are based on two metrics: the accuracy (how many questions did the participant answer correctly in both sessions) and speed (how long did it take the participant to answer all the questions in both sessions). These results of all the test subjects are compiled to form one single statistic. Then, the recordings are also analyzed to see what features of CodeGraph the participants used in order to answer each question. The results of these tests are presented in Section 3.8.

Under normal circumstances, I would have been able to conduct these tests on campus with fellow computer science students as my main test subject. However, due to the COVID pandemic, and the necessity to conduct these tests in-person, I was severely limited in my ability to gather test subjects. In fact, due to the Belgian

COVID measures that were in place at the time that I was conducting these tests, I was only allowed to be in close contact with close family. Because of this, I was only able to conduct the usability tests with three test subjects, of whom only a single subject has experience with programming. Because of this small number of test subjects, the results of these tests are by no means statistically significant. However, they can serve as an indication of the usability of CodeGraph.

3.8 Results

The results are shown in table 2 and the answers of the participants are included in appendix G.

	score general tools	speed general tools (min.)	score CodeGraph	speed CodeGraph (min.)
Participant 1	50%	17:55	100%	15:04
Participant 2	33.34%	6:10	83.34%	9:45
Participant 3	50%	14:30	100%	7:37
average	**44.45%**	**12:51**	**94,45%**	**10:48**

Table 2: Results of the usability test with three participants.

Using CodeGraph, the test candidates completed the tasks 2.03 minutes faster, on average, compared to using the general text-based tools. This corresponds to a 16.05% speed increase. Additionally, they also got 3 more correct answers on average, which corresponds to a 50% increase in accuracy. As stated before, three test subjects is far too few to be considered statistically significant. Therefore, these results should be considered nothing more than a general indication. Also, the tasks were specifically designed to make use of specific features of CodeGraph and by no means represent a general cross-section of all the tasks that a developer performs on a regular basis. There are still many tasks for which the general tools would be far more appropriate than CodeGraph and there are many developer tasks that a developer would not be able to complete at all with CodeGraph. However, for the kinds of tasks for which CodeGraph is specifically designed, the results clearly indicate an improved workflow over general, text-based tools. This is especially the case for some specific information that CodeGraph provides which would be very hard to obtain using regular, text-based tools.

When observing the test subjects while they performed the tasks and when analyzing the screen recordings, some observations became apparent. For example, users heavily gravitate towards graphical tools. During the test with the general tools, users were provided a set of tools, including a terminal. However, none of the test subjects used the terminal, favoring the file explorer and a graphical text editor instead. Also, the two tasks that were only solvable using advanced terminal commands, none of the candidates were able to solve. This is no surprise as the test

subjects had no to limited programming experience. For experienced developers, this might have been a different story entirely.

Some flaws in CodeGraph's design also became painfully apparent. For example, users often had trouble finding a particular node in the dependency graph. First they tried to click a file in the left menu bar. They seemed to expect that the viewport would move to the selected node, placing it in the center. When this did not happened, they seemed to be confused and tried clicking the file a couple more times. Then, they shifted strategies and hovered over the file instead, which highlights the corresponding node in the dependency graph and in the minimap. Since the participants were zoomed in when they did this, not all the nodes were visible in the viewport and they had to find the highlighted node through the minimap. This was not an easy task as the dependency graph was very large and, consequently, the nodes in the minimap very small. This meant that the participants had to find a small blue rectangle of only a couple of pixels in width and height. Once found, the participant would manually drag the viewport to the location of the node. This process often took up to 30 seconds to complete. This whole, painstaking process could have been avoided by placing the node in the center of the viewport when selecting a node.

Another confusion of the participants was that they expected the node properties to appear as soon as they clicked a node. This is not the case as the user still has to open the node properties view in the right menu bar. After a couple of seconds of retrying to select a node, the participant would remember the I showed them how to open the node properties during the CodeGraph demo. As the participants were not yet familiar with all the icons, the participant would then proceed to try pressing a bunch of different icon buttons, before finally opening the properties window. This could be solved by opening the properties window in the right menu bar whenever the user selects a new node. However, this could also be annoying if the user is not interested in node properties. It would be best to let the user decide how to handle this situation through the settings window.

The participants were at times also underwhelmed with the amount of information in the node properties window. For example, most of the participant would first look at the properties of a particular node in the right menu bar when asked how many incoming / outgoing links a particular node had. Once they discovered that the properties window did not contain this information, they would manually count the number of incoming / outgoing edges in the dependency graph. The properties window should contain all the information that a user could be interested in, to avoid the need for such manual, time-consuming work.

There was also confusion about the function of different icon buttons. For example, when the participants tried to locate a particular node in the graph, they first tried selecting the file in the file explorer and when that didn't work, they would try to click the eye icon next to the file name, assuming this button would move the viewport to that node. However, the eye icon hides the node in the dependency graph, which caused even more confusion as the participant was then not able to find the node at all since it was hidden. Another example includes the fact that when participants tried to open a particular window, for example the properties window, rather than

being able press the correct button based on the icon, they simply pressed all the different buttons until they found the right one. In order to solve this confusion, more attention should be given to the icons, making their meaning instantly recognizable.

Another point of confusion related to the filtering functionality was that, when the user filtered out a single node using the eye icon, the effect did not immediately become apparent. This is because there were so many nodes that hiding a single node became visually unnoticeable. It is important that each action the user performs is followed by clear visual feedback showing the result of the action.

One of the tasks required the participant to count the number of incoming edges and another task the number of outgoing edges. Incoming and outgoing edges can be recognized by the direction of the arrow on the edge. However, counting these edges was a surprisingly difficult task. To begin with, the arrow is placed in the middle of the edge. Therefore, if the edge is long, the participant had to navigate all the way to the middle of the edge in order to identify whether the edge defined an incoming or an outgoing edge. Furthermore, since there were many nodes and edges, the arrows were sometimes covered by other nodes or it wasn't clear which node a particular edge belonged to. In order to solve the first problem, it would be beneficial if the arrow was not only placed in the middle of the edge but also at the beginning and at the end of the edge. This would make the nature of each edge instantly clear when looking at a node. To solve the second problem of visual clutter, it would be useful if not only the selected node itself was highlighted, but also the edges connected to the selected node. This would make all the relevant information of the selected node a lot more clear.

Another observation was that participants often tried to right click the different elements. What's more, when they tried to get information on a particular node, they would often right click the node. This would display the general browser context menu. Then they would click "inspect element", which would open the developer menu of the browser which would then show the HTML code defining the element. This was, of course, not the right course of action for what the participants tried to accomplish. These actions make it clear that participants are trained to look for additional options and information using the right mouse click. This habit probably originates from native applications in which the right mouse button click is indeed a common action. However, in web applications such as CodeGraph, this is far less common. In order to solve this confusion, CodeGraph would need a custom context menu with CodeGraph-specific options.

Another actions that participants often tried to perform without success is navigating to a particular location in the dependency graph by clicking on that location in the minimap. This did not work. Instead, users should drag the viewport rectangle in the minimap to the desired location. However, none of the participants tried to do this. Therefore, the user interaction with the minimap should be reviewed to make it more intuitive.

The last observation is the fact that participants barely used the filtering functionality. This is problematic since the optimal workflow is to first filter out any nodes that the user is not interested in and then complete any other actions needed to accomplish the goal. If the user does not first filter the graph, then any further actions

will become far more difficult to perform because of the overload of information. In order to solve this problem, the GUI of CodeGraph should be redesigned to put far more emphasis on the filtering. Currently, it is viewed as an optional side-feature, while it should almost be considered an essential, main feature. That is not to say that the user should not be able to view the entire graph at once, but this should be conscious decision rather than a default behavior.

The participants were also asked how they experienced using CodeGraph. When asked how enjoyable their experience using CodeGraph was compared to the general, text-based tools on a scale from 1 to 5, the participants rated CodeGraph a 4.0 on average. When asked how efficient their experience with CodeGraph was on a scale from 1 to 5, they rated it a 3.66 on average. So, their experience was more enjoyable and more efficient with CodeGraph than with the general tools. When asked what the participants liked about CodeGraph, the responses were: *"All you need in one central place"*, *"It is well organized and you can find things easily."*, *"It's like a puzzle. It looks good graphically."*. So participants seem to appreciate the information that CodeGraph makes available and the application's aesthetics. When asked what they did not like about CodeGraph, they replied: *"editor is not easy to use; file info box might show more"*, *"It is hard to see in which direction the arrow goes"*, *"At the start, it's hard to get an overview. Also hard to figure out how everything works, but once you understand it, it's logical."*. So the navigation should be better, the information should be more visible, more information should be included and it should be more intuitive. Finally, when asked for any final comments, participants gave the following replies: *"very good tools but users need some training before they can use its full potential"*, *"I like it more then other general tools."*, *"sometimes, when there are many edges close to each other, it becomes confusing and the direction of the arrows are sometimes hard to tell"*. So positive comments, with another indication of some visual flaws. These results are in line with my observations of the participants completing the tasks.

4 Discussion

The current version of CodeGraph achieves the goals it set out to accomplish. However, there is still a lot of room for improvement. For example, the visualization of the dependency graph could be improved with the following improvements: currently, the selected node is highlighted, and when the user hovers over a file or folder in the file explorer, the corresponding node(s) are also highlighted. However, it would be even more powerful if all the nodes connected to the selected node and the connecting edges would also be highlighted, albeit in a different color. This allows the user to quickly see all the selected node's neighbors. This strategy is adopted by some other applications discussed in Section 2.1, and it provides a very pleasant user experience.

Currently, all the nodes in the dependency graph are the same size. It might be worthwhile to explore strategies for making the size of a node reflect the importance of the node. Of course, the importance of a node is subjective. Strategies that have been adopted by other applications include making the importance of a node correspond with the number of lines of code that the file contains, or making it correspond to the number of outgoing and/or incoming edges. Testing would have to be done to determine the best strategy. It would also be possible to let the user decide if dynamic sizing is enabled, and if so, what strategy needs to be applied.

Large graphs, which dependency graph often are, can easily become chaotic. In order to reduce the chaos, CodeGraph encourages users to first filter out any unnecessary nodes. Then CodeGraph offers many different layouts, some of which are capable or organizing a large graph orderly. In order to minimize the chaos, other applications have also implemented the ability to collapse groups of nodes into a single node and expend them again. This is similar to filtering in the sense that it reduces the number of nodes and links displayed in the dependency graph. However, when filtering nodes, they are simply removed as if they did not exist, whereas with collapsed nodes, these nodes are still part of the dependency graph, albeit in a more abstract form. Both strategies have their own advantages and disadvantages. CodeGraph could definitely benefit from the ability to collapse groups of nodes. In fact, this feature was implemented in the first prototype, but was removed in the final version because of lack of time.

When exploring a large dependency graph, they user can easily get lost if the application does not provide sufficient visual clues. To this extend CodeGraph has implemented highlighting of the nodes, in which the user has expressed interest. However, this could be further emphasized using animations. For example, in the first prototype, whenever the user selected a node, the viewport would "fly" to the selected node, placing it in the center of the viewport. This worked especially well since it was visualized in 3D space. Similar strategies could be explored for CodeGraph.

CodeGraph currently only visualizes the dependencies between files, i.e., files which are imported in other files. However, in Section 2.1, we saw that other applications have implemented many different types of dependencies. Some applications have implemented high-level dependencies, such as graphs depicting how entire projects depend on each other, while other applications have implemented low-level dependencies such as which functions call which other functions, and everything in

between. I believe CodeGraph could benefit from the implementation of lower-lever dependencies such as dependencies between functions and classes instead of strictly between files. Different levels could even be combined. For example, you could have a dependency graph with large nodes representing files with edges connecting them. Then, within these large file-nodes, you could place a group of smaller nodes, which represent functions and classes, which once again are connected with each other. These function and class nodes could be connected to other nodes within the same file node, but also to the function and class nodes in other file nodes. Higher level dependencies, such as dependencies between entire projects, would be outside of the scope of CodeGraph as CodeGraph is focused on the exploration of a single JavaScript project, not an entire ecosystem.

CodeGraph could also benefit from a more specific handling of different node types. Currently, the application already creates visual distinctions between nodes representing different types of files, but they are handled more or less identically. For example, a node representing a JavaScript file will show the exact same properties in the right menu bar as an SVG file. This could be improved by having the properties be more suitable to the specific file type. For example, for JavaScript files, the number of lines of code is an important property, while for an SVG file, this is far less the case. Whereas for an SVG file, the result of the rendering is again far more important. This specific handling of different node types is especially important for external library files. Currently, CodeGraph does not create nodes for external libraries. However, it would be greatly beneficial if an entire external library would be represented by a single node. For example, an NPM[37] package could be represented by a single node and, when selecting the node, the right menu bar could fetch and show the corresponding README file of the package, instructing the user how to use the library.

CodeGraph simply displays the dependency graph. It does nothing to analyze the quality of the dependency graph. This is in contrast to some applications in Section 2.1. It would be useful if CodeGraph also detected bad dependencies. These include circular dependencies, conflicting dependencies and dependencies that are imported into the file, but are then never used within the file. These bad dependencies could then visually be brought to the attention of the users, for example, through color coding.

The last major improvement that I envision for the dependency graph visualization of CodeGraph is the detection and visualization of many different types of dependencies. Currently, the only dependencies that are detected are the JavaScript imports and exports. However, in web development, there are many different ways files can rely on other files and external sources. For example, JavaScript files can fetch external data using fetch() functions, HTML files can import external stylesheets and fonts, CSS files import local and external images and variables. It would be beneficial if these other types of dependencies are also detected and visualized.

Apart from the dependency graph itself, CodeGraph could also benefit from some other additional features. Currently, CodeGraph only displays the dependency graph

[37] https://www.npmjs.com/

based on the newest commit in the master branch. However, it would be useful if the user is able to select, which commit from which branch to base the dependency graph on. This feature was designed in the first paper prototype as shown in appendix C.1, but was not implemented in the actual prototype. Another useful feature is the ability to compare the difference between commits through visual cues in the dependency graph. This feature was also included in the first paper prototype, but was not implemented in the real application. This functionality could even be enhanced with some features that show the evolution of the codebase over time as we saw in some other applications. For example, CodeGraph could include an animation of how the dependency graph has changed over time. Yet another view that could be beneficial to the user is the single node view. This view would show all the details of a single node while removing all the clutter of the rest of the graph.

CodeGraph includes many different properties of the selected node in the right menu bar. However, one thing that currently is not included as of yet is the file contents. For a JavaScript file, this would be the code, for an image file, this would be the actual image, for a data file, this would be the data, nicely formatted.

CodeGraph allows the user to filter the nodes using various metrics. For example, the user can filter specific files and folder or he can filter based on labels, associated users, etc. However, in Section 2.1, we saw that some applications have integrated very rich filtering systems that allow the user to create custom filtering queries using regular expression or using custom JavaScript functions. They also allow the user to save these filters for later use. CodeGraph would also benefit from these features as filtering is an essential part of the CodeGraph workflow.

A last candidate for future improvement is concerned with performance. Currently, the entire CodeGraph application runs on a single, main thread. This includes the GUI, the user interactions, the data analysis, the layout computation, the statistics generation and the rendering of the dependency graph. This results in a noticeable slowdown when working with large dependency graphs. For example, for a graph with about a thousand nodes, it takes a couple of seconds to switch between layouts. Since everything is computed on the main thread, the GUI also becomes less responsive while CodeGraph is computing the new layout. This could be solved by spreading the different computations over multiple threads using Web Workers. This would not only make the computations faster, leading to less waiting time for the user, it would also keep the GUI snappy, even while CodeGraph is performing computations.

To conclude, the results indicate that a visual tool like CodeGraph is definitely useful for developers. At the moment, CodeGraph is not sufficiently polished to release as a commercial product, but with enough additional refinement and the inclusion of some additional features, this could definitely become the case.

5 Conclusion

In this thesis, I have presented CodeGraph, a tool that visualizes the dependency graph of JavaScript projects. This is useful for new developers learning the architecture of a project, existing developers analyzing the project for maintenance and refactoring purposes and project managers who require an overview of the project in order to manage the development and assign resources. The usability tests have indicated that users are able to work more accurately and faster with CodeGraph than with traditional tools.

CodeGraph is my attempt at answering the main research question: **How can dependency graphs be used to provide a better understanding of JavaScript projects?**

I also set out to answer the following questions: *What requirements should an application providing a dependency graph fulfill in order to provide the user with as many useful insights as possible? How can dependency graphs be visualized in order to optimize the usability for the user? How can such a tool be integrated in the current project development workflow as seamlessly as possible?*

I discussed the motivation for the development of CodeGraph and the requirements of CodeGraph in Sections 3.2 and 3.4 respectively. In Section 3.5, I discussed the visualization of the dependency graph at length. This consists of an informative and clear design of the nodes and edges, an extensive set of filtering tools to remove any superfluous data and different layouts defining the placement of the nodes and edges, each layout being suited for a specific situation. Finally, in Section 3.5, I discussed how CodeGraph is seamlessly integrated in the developer's workflow by linking it with the project's remote git repository after which it retrieves the needed information automatically.

References

[1] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin. On the shape of circular dependencies in java programs. In *2014 23rd Australian Software Engineering Conference*, pages 48–57. IEEE, 2014.

[2] D. J. Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[3] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 46(6):e33–e33, 2018.

[4] M. Balzer and O. Deussen. Exploring relations within software systems using treemap enhanced hierarchical graphs. In *3rd IEEE international workshop on visualizing software for understanding and analysis*, pages 1–6. IEEE, 2005.

[5] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[6] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *2006 13th Working Conference on Reverse Engineering*, pages 199–210. IEEE, 2006.

[7] Y. Chen, Z. Guan, R. Zhang, X. Du, and Y. Wang. A survey on visualization approaches for exploring association relationships in graph data. *Journal of Visualization*, 22(3):625–639, 2019.

[8] A. Civril, M. Magdon-Ismail, and E. Bocek-Rivele. Ssde: Fast graph drawing using sampled spectral distance embedding. In *International Symposium on Graph Drawing*, pages 30–41. Springer, 2006.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[10] D. T. Daniel, E. Wuchner, K. Sokolov, M. Stal, and P. Liggesmeyer. Polyptychon: A hierarchically-constrained classified dependencies visualization. In *2014 Second IEEE Working Conference on Software Visualization*, pages 83–86. IEEE, 2014.

[11] C. De Roover, R. Lämmel, and E. Pek. Multi-dimensional exploration of api usage. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 152–161. IEEE, 2013.

[12] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[13] M. Dias, D. Orellana, S. Vidal, L. Merino, and A. Bergel. Evaluating a visual approach for understanding javascript source code. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 128–138, 2020.

[14] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 91–94, 2008.

[15] U. Dogrusoz, M. E. Belviranli, and A. Dilek. Cise: A circular spring embedder layout algorithm. *IEEE transactions on visualization and computer graphics*, 19(6):953–966, 2012.

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

[17] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[18] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[19] C. Godsil and G. F. Royle. *Algebraic graph theory*, volume 207. Springer Science & Business Media, 2001.

[20] H. He and O. Sykora. New circular drawing algorithms. 2004.

[21] K. B. IIBA. *A Guide to the Business Analysis Body of Knowledge*. International Institute of Business Analysis, 2009.

[22] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. *Readings in Information Visualization: Using Vision to Think*, pages 152–159, 1999.

[23] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. In *Graph Algorithms And Applications I*, pages 3–27. World Scientific, 2002.

[24] O. Lassila, R. R. Swick, et al. Resource description framework (rdf) model and syntax specification. 1998.

[25] H. Liu, Y. Tao, W. Huang, and H. Lin. Visual exploration of dependency graph in source code via embedding-based similarity. *Journal of Visualization*, pages 1–17, 2021.

[26] H. Liu, Y. Tao, Y. Qiu, W. Huang, and H. Lin. Visual exploration of software evolution via topic modeling. *Journal of Visualization*, pages 1–18, 2021.

[27] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[28] Y. Park and C. Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 3–10. IEEE, 2009.

[29] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A Design Science Research Methodology for Information Systems Research. *JMIS*, 24(3), 2007.

[30] D. Reniers, L. Voinea, and A. Telea. Visual exploration of program structure, dependencies and metrics with solidsx. In *2011 6th International workshop on visualizing software for understanding and analysis (VISSOFT)*, pages 1–4. IEEE, 2011.

[31] W. Scacchi. Understanding the requirements for developing open source software systems. *IEE Proceedings-Software*, 149(1):24–39, 2002.

[32] K. Schwaber and J. Sutherland. The scrum guide. *Scrum Alliance*, 21:19, 2011.

[33] D. Seider, A. Schreiber, T. Marquardt, and M. Brüggemann. Visualizing modules and dependencies of osgi-based applications. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 96–100. IEEE, 2016.

[34] R. Tamassia. *Handbook of graph drawing and visualization*. CRC press, 2013.

[35] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. Extraction and visualization of call dependencies for large c/c++ code bases: A comparative study. In *2009 5th IEEE International workshop on visualizing software for understanding and analysis*, pages 81–88. IEEE, 2009.

[36] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[37] H. C. Vázquez, A. Bergel, S. Vidal, J. D. Pace, and C. Marcos. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and software technology*, 107:18–29, 2019.

[38] L. Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[39] T. Würthinger, C. Wimmer, and H. Mössenböck. Visualization of program dependence graphs. In *International Conference on Compiler Construction*, pages 193–196. Springer, 2008.

A Layouts

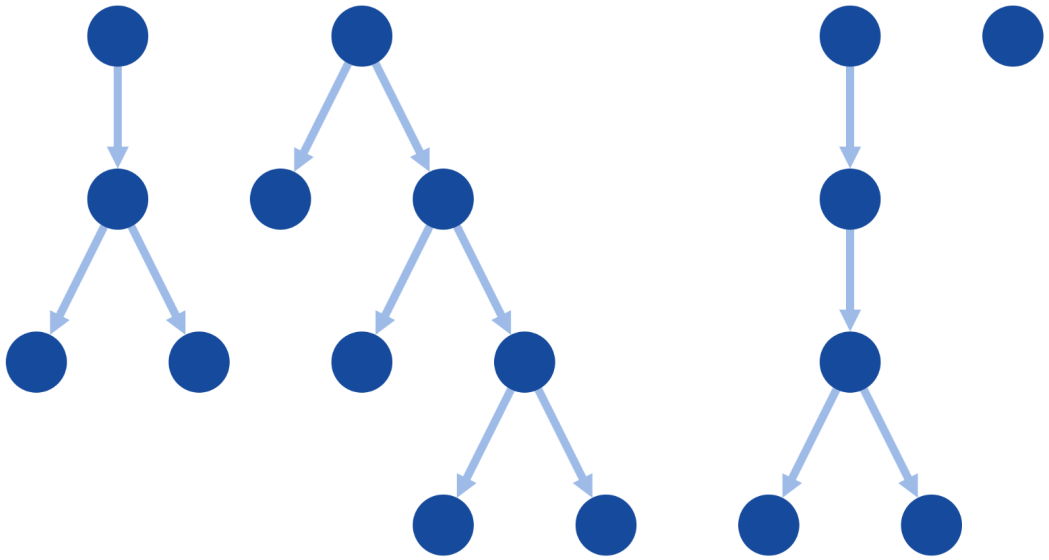


Figure A1: Dagre example from cytoscape.

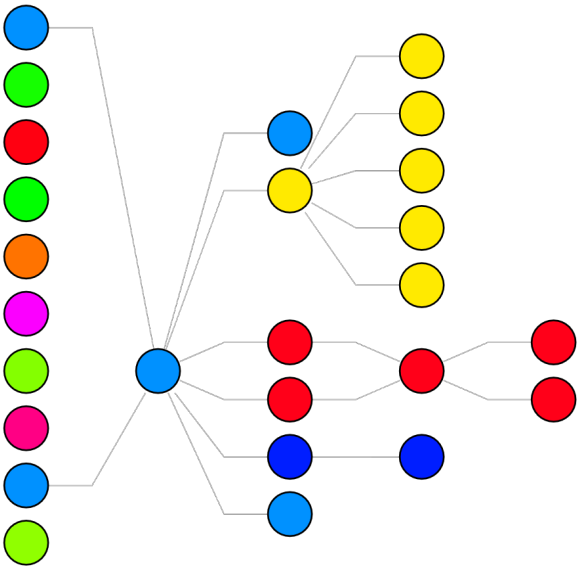


Figure A2: Dagre on small dependency graph.

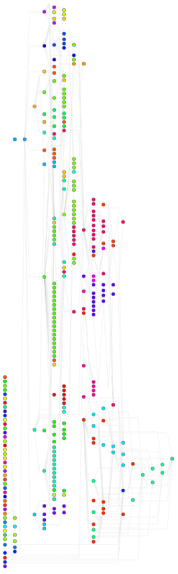


Figure A3: Dagre on large dependency graph.

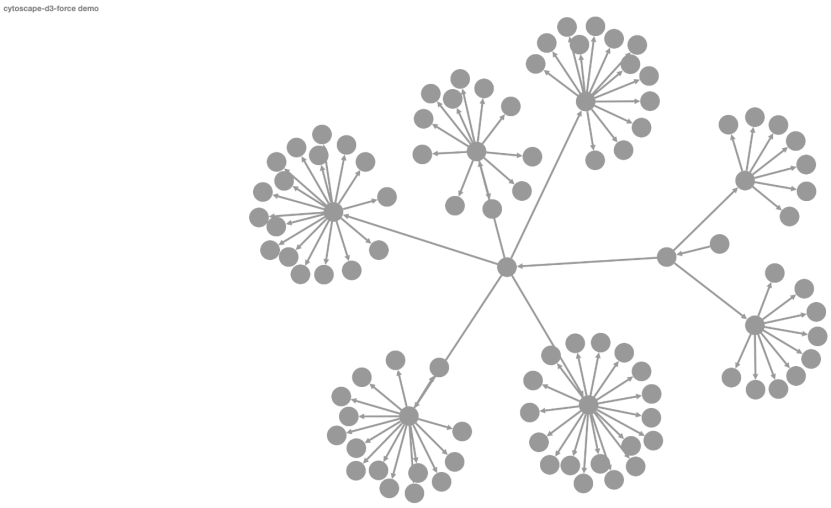


Figure A4: Small D3-force example from cytoscape.

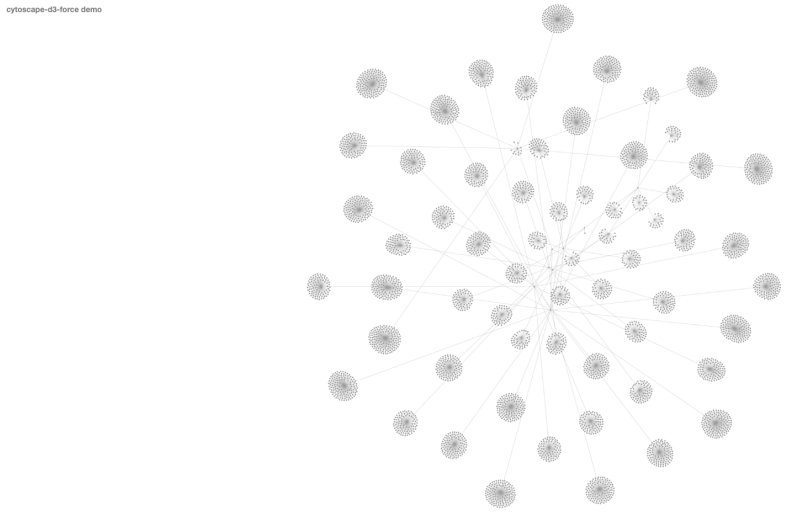


Figure A5: Large D3-force example from cytoscape.

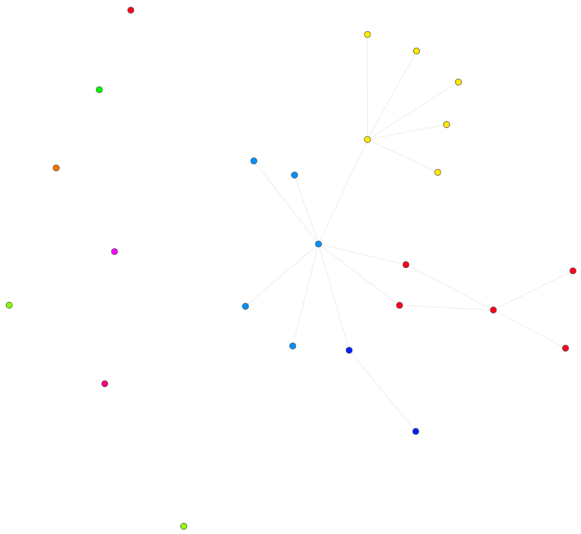


Figure A6: Force layout on small dependency graph.

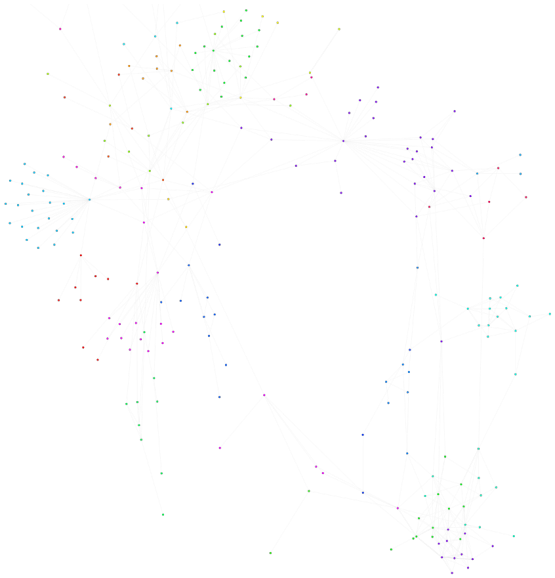


Figure A7: Force layout on large dependency graph.

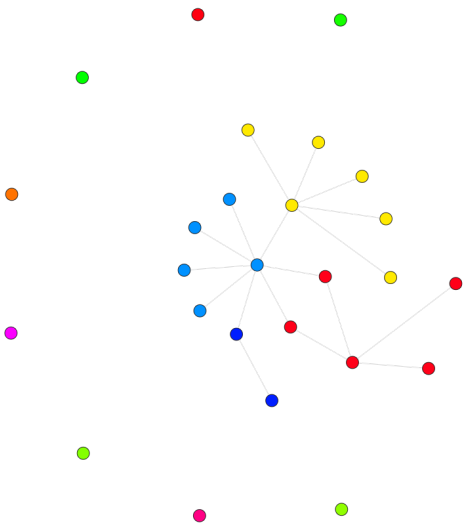


Figure A8: Radial layout on small dependency graph.

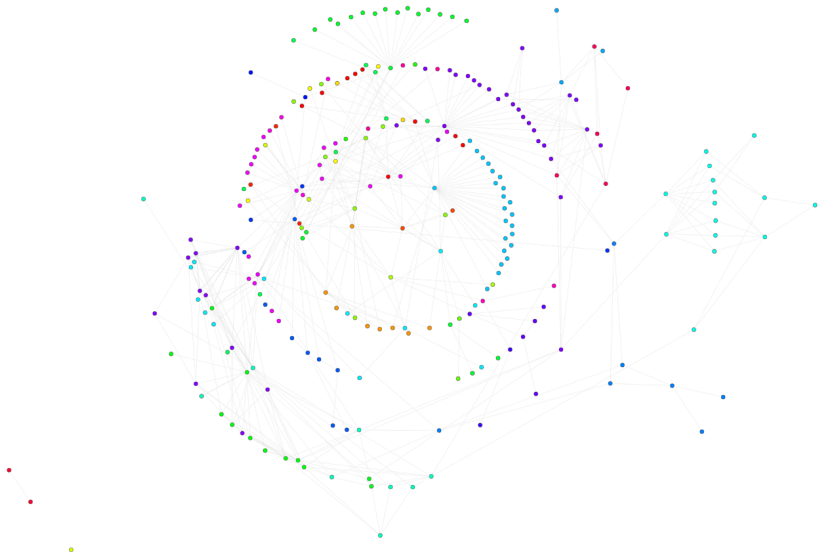


Figure A9: Radial layout on large dependency graph.

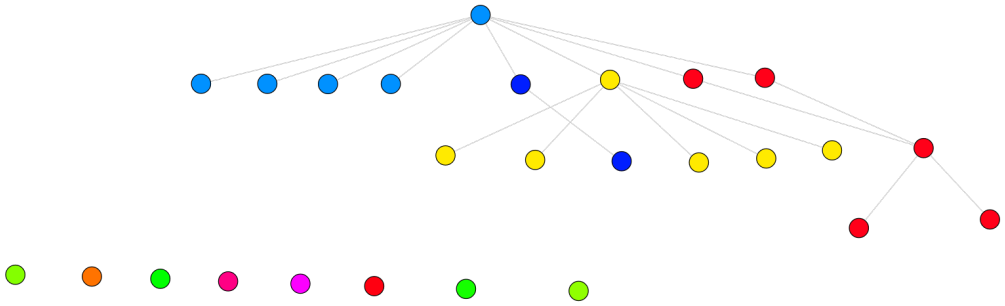


Figure A10: Tree layout on small dependency graph.

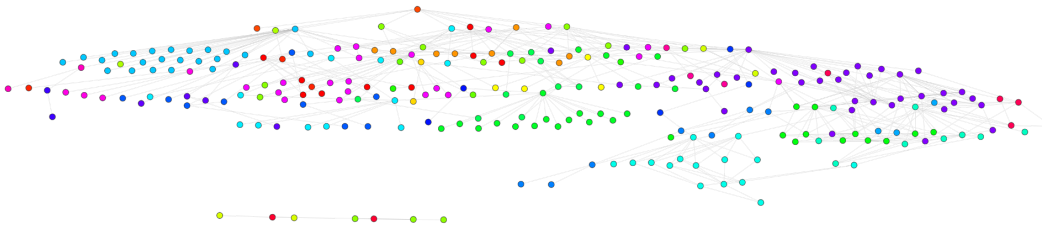


Figure A11: Tree layout on large dependency graph.

Figure A12: Circle layout example from cytoscape.

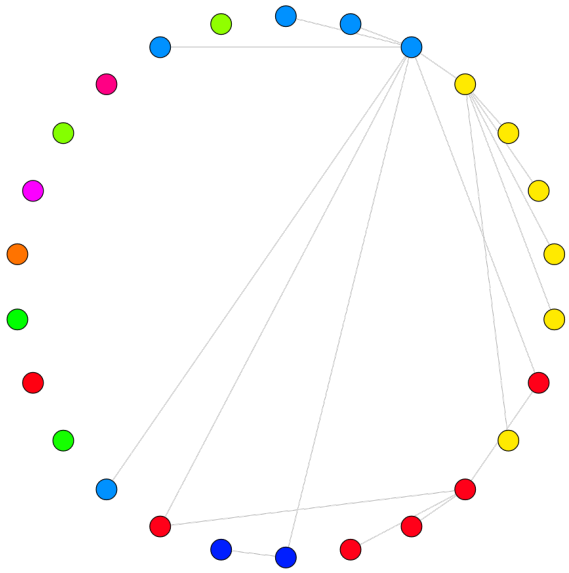


Figure A13: Circle layout on small dependency graph.

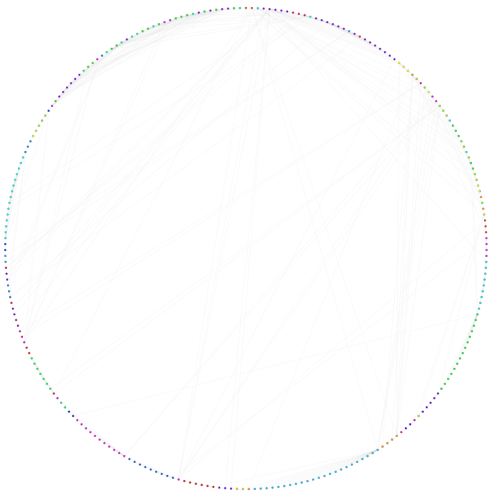


Figure A14: Circle layout on large dependency graph.

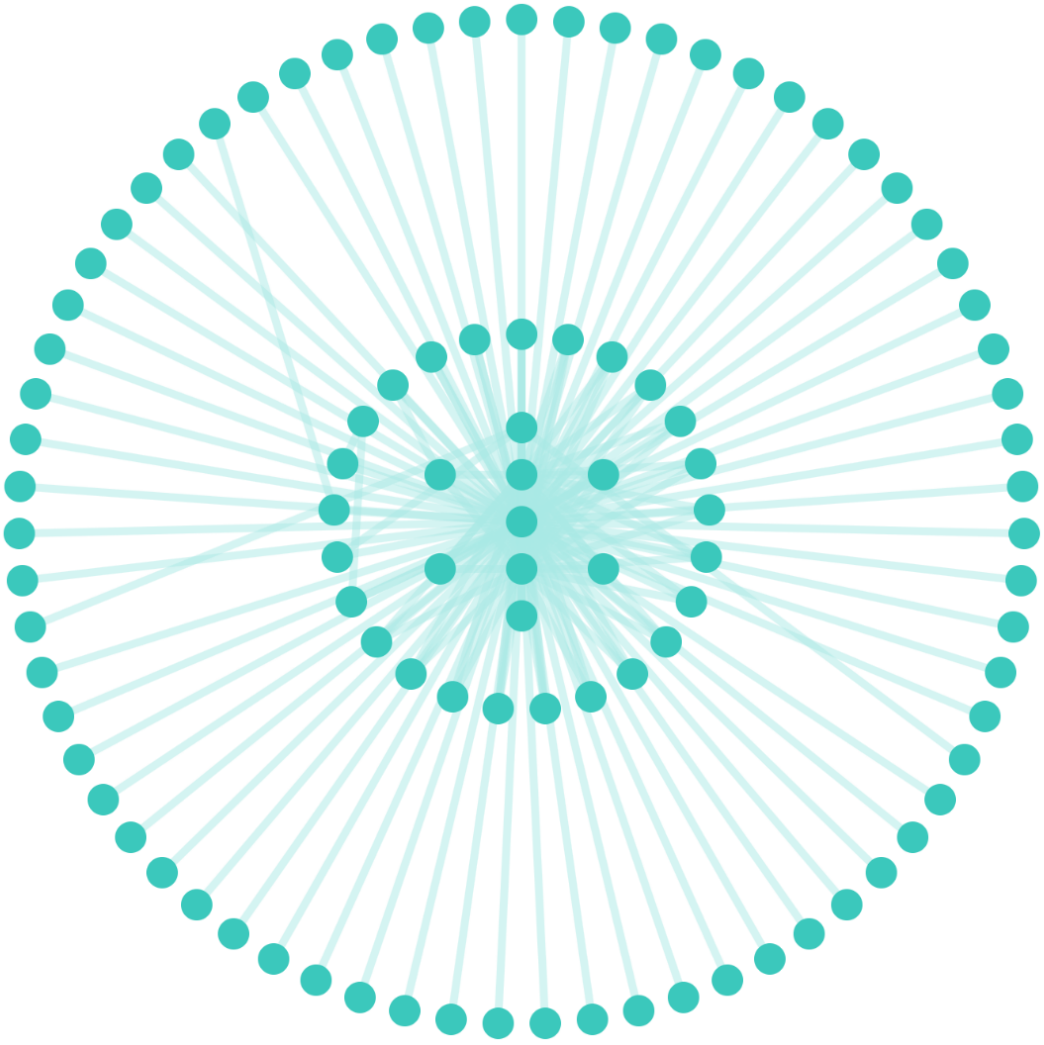


Figure A15: concentric layout example from cytoscape.

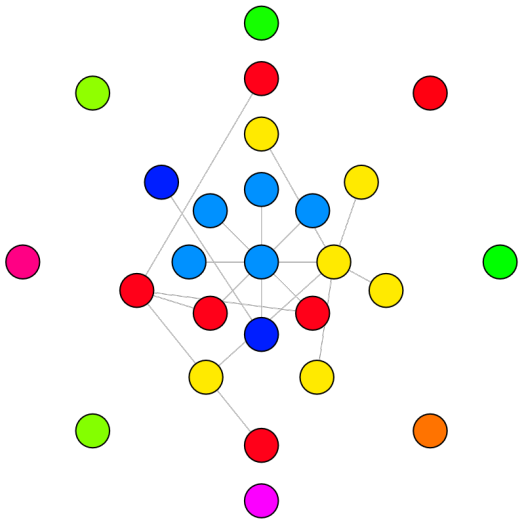


Figure A16: concentric layout on small dependency graph.

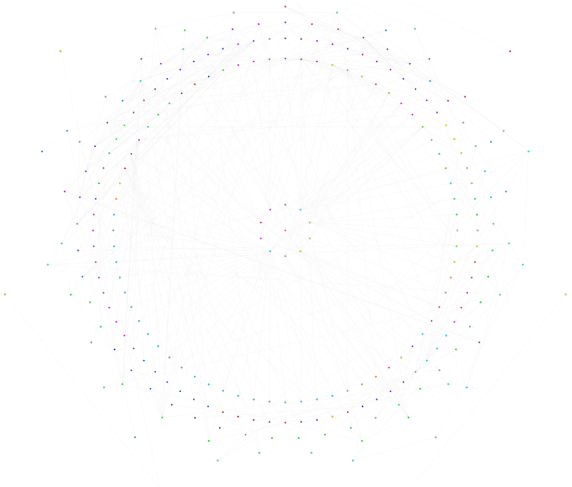


Figure A17: concentric layout on large dependency graph.

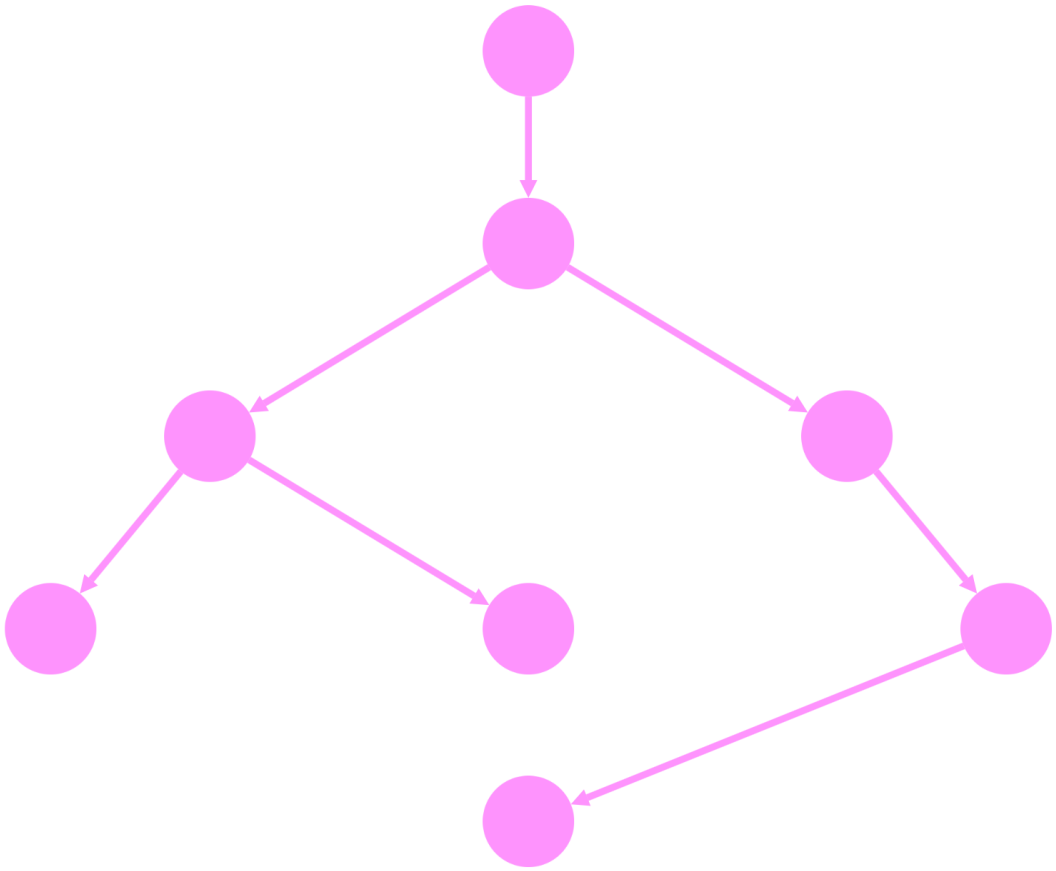


Figure A18: breadth first layout example from cytoscape.

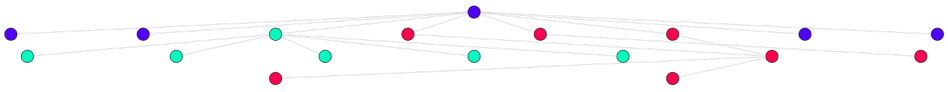


Figure A19: breadth first layout on small dependency graph.

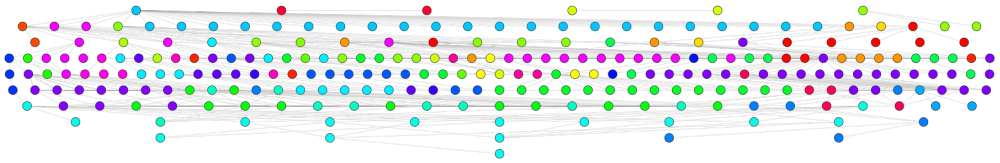


Figure A20: breadth first layout on large dependency graph.

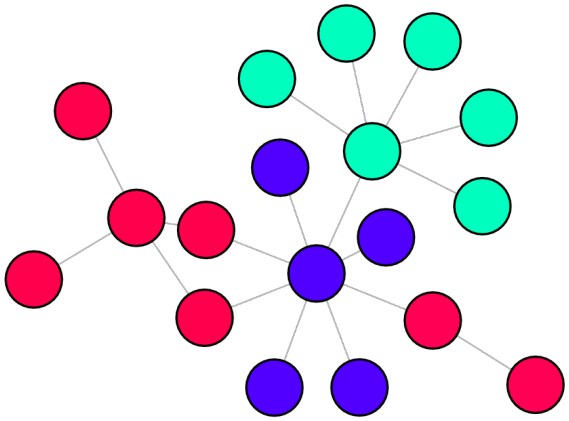


Figure A21: cose layout on small dependency graph.

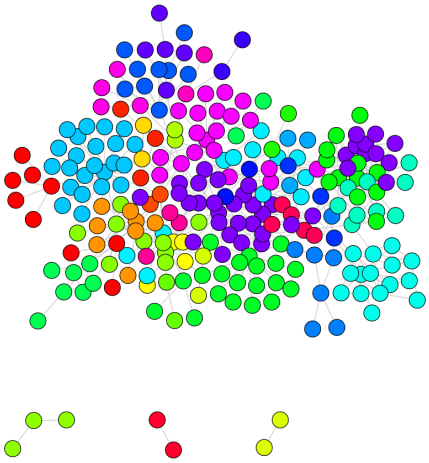


Figure A22: cose layout on large dependency graph.

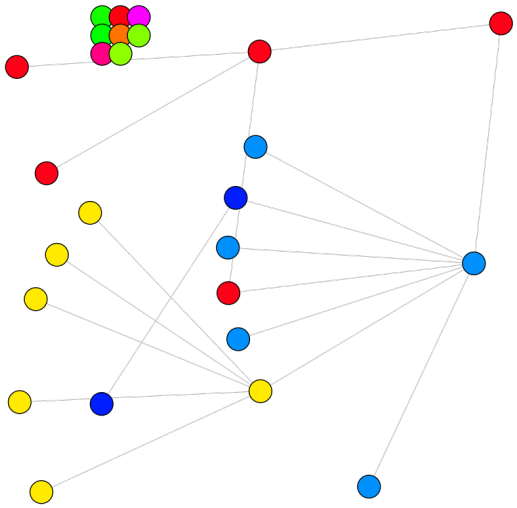


Figure A23: cose-bilkent layout on small dependency graph.

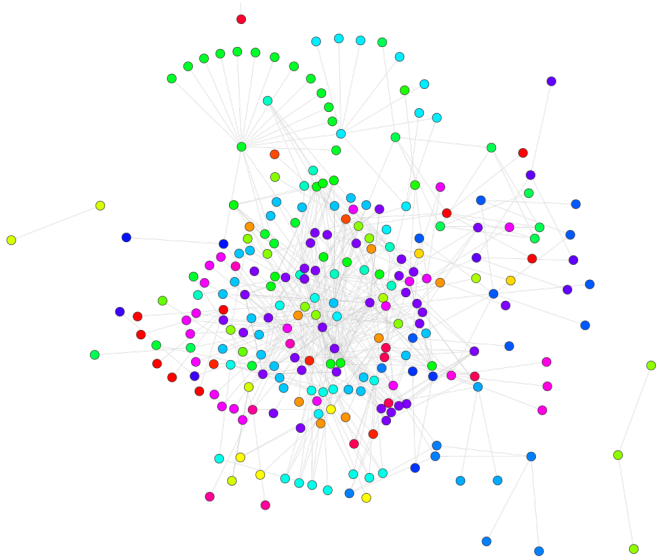


Figure A24: cose-bilkent layout on large dependency graph.

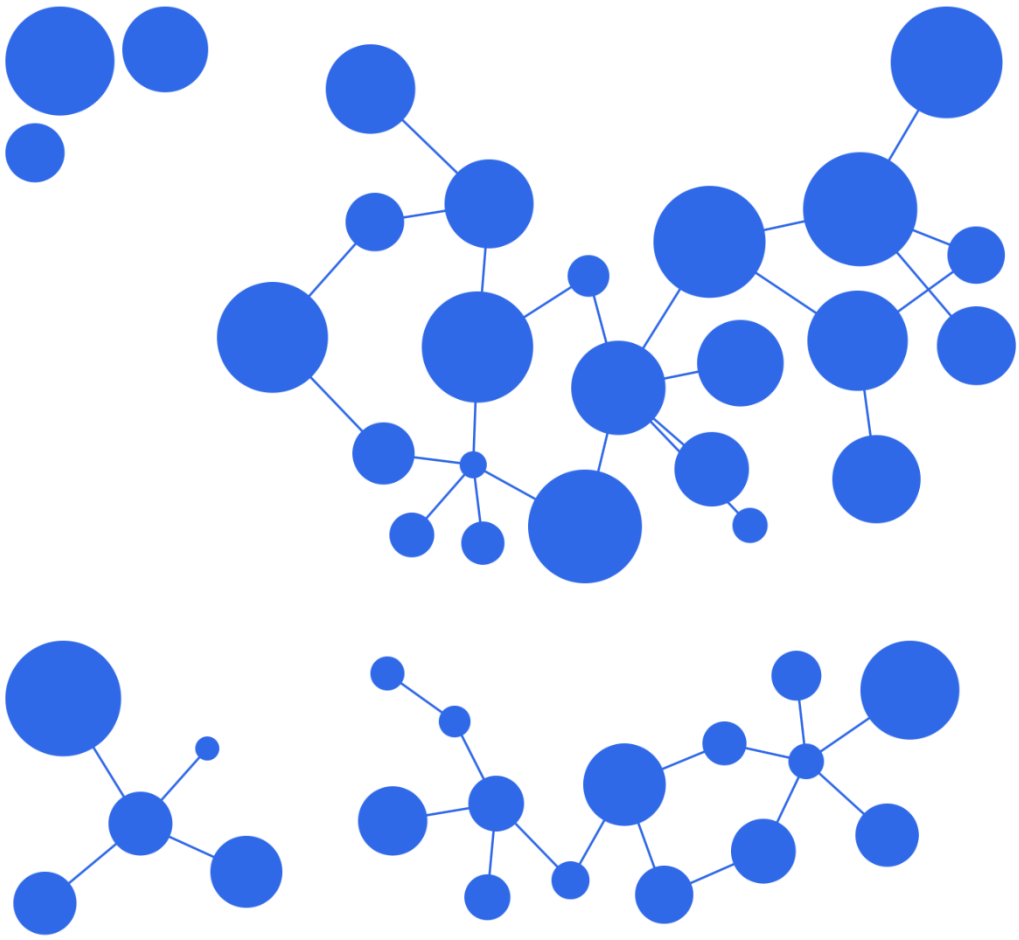


Figure A25: fcose layout example from cytoscape.

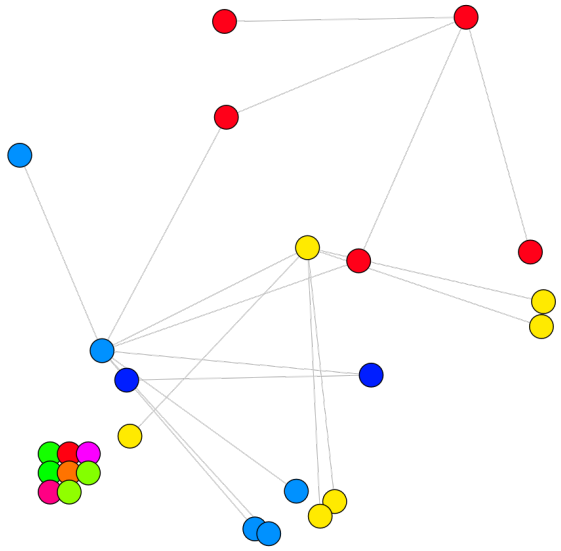


Figure A26: fcose layout on small dependency graph.

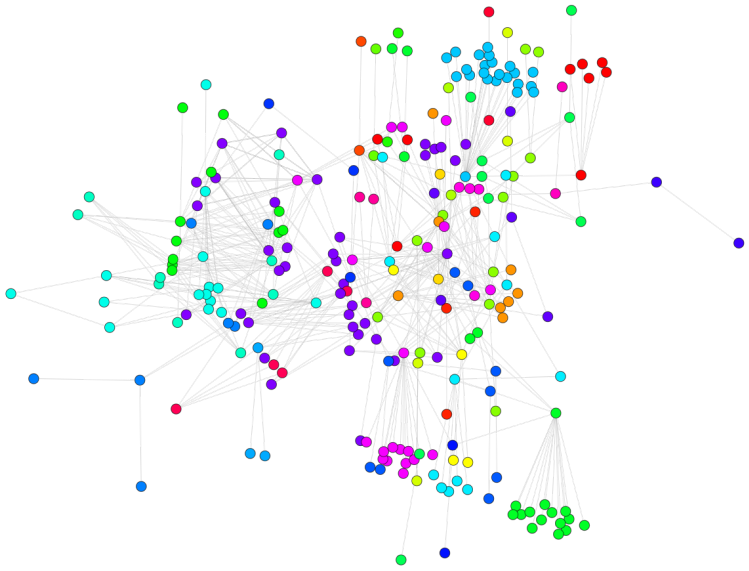


Figure A27: fcose layout on large dependency graph.

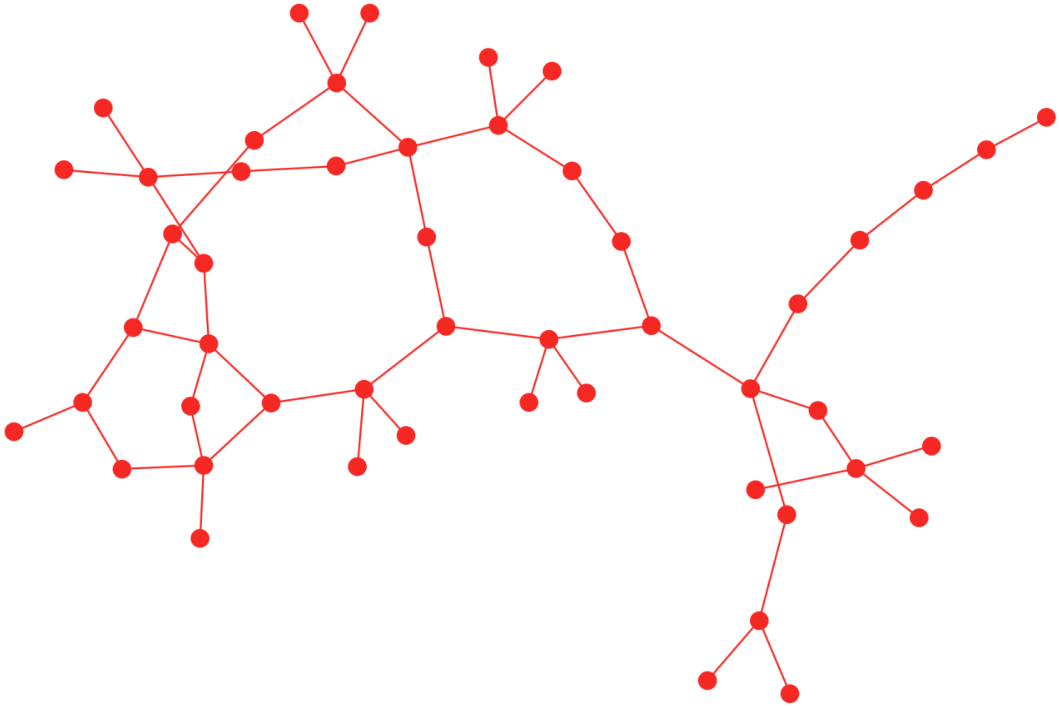


Figure A28: cola layout example from cytoscape.

Figure A29: cola layout on small dependency graph.

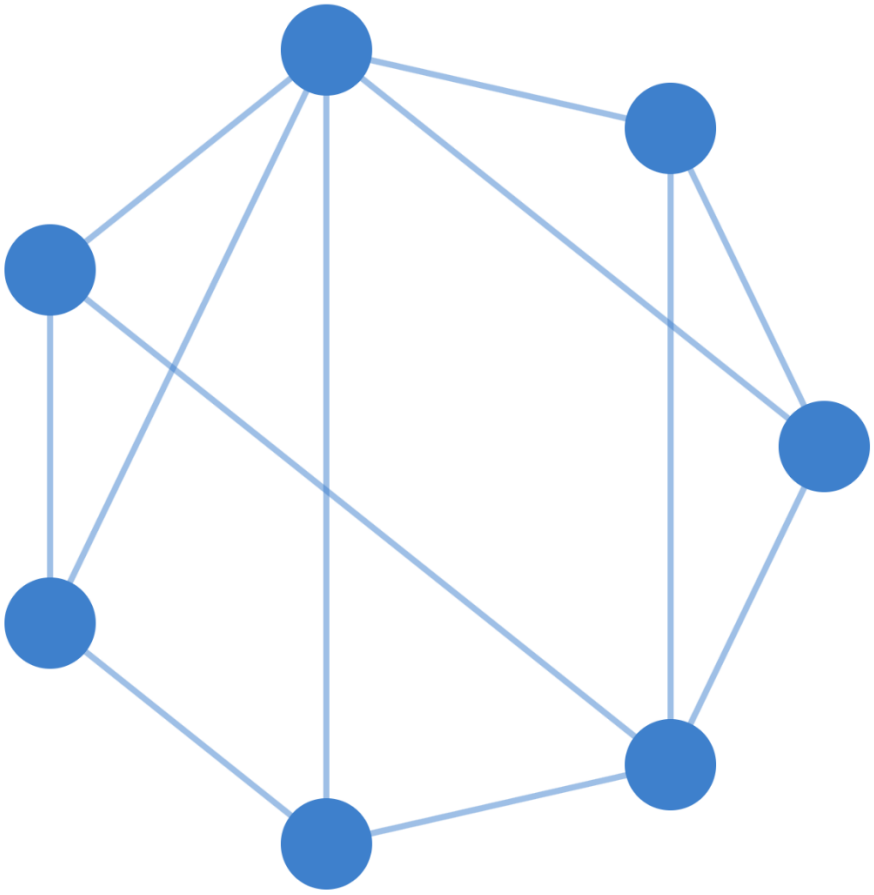


Figure A30: avsdf layout example from cytoscape.

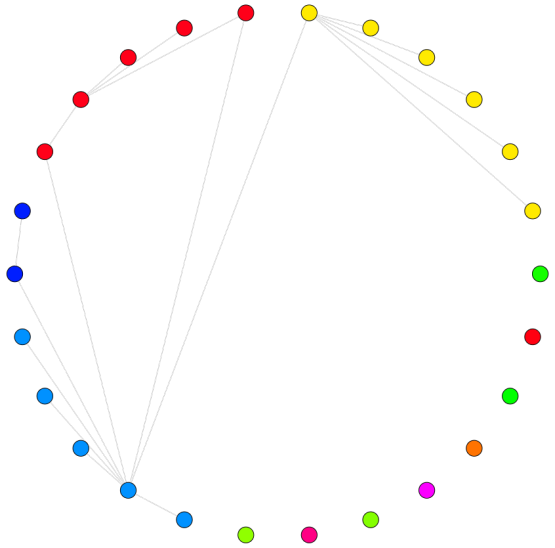


Figure A31: avsdf layout on small dependency graph.

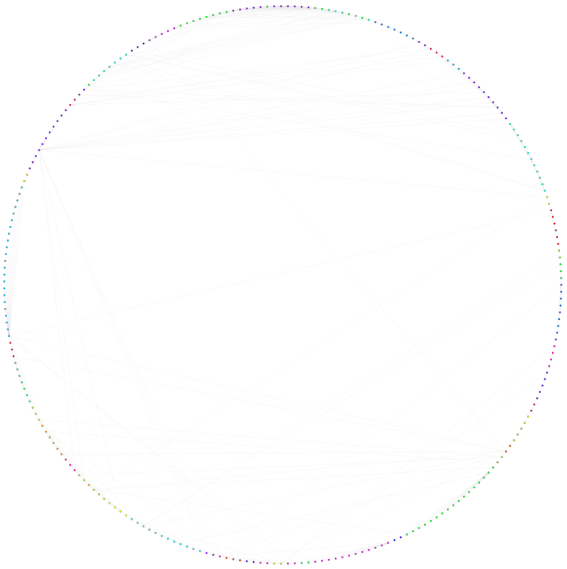


Figure A32: avsdf layout on large dependency graph.

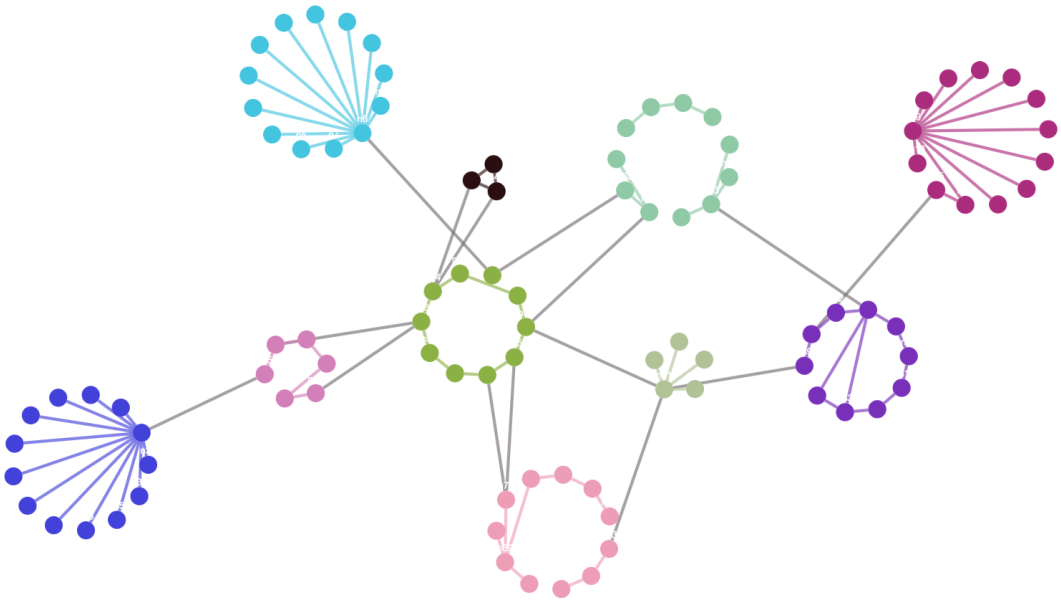


Figure A33: cise layout example from cytoscape.

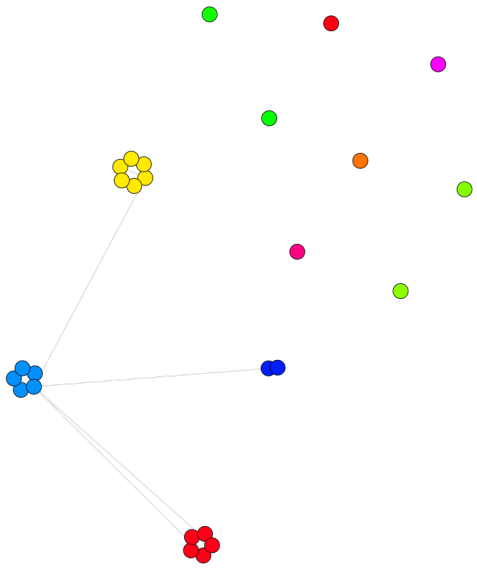


Figure A34: cise layout on small dependency graph.

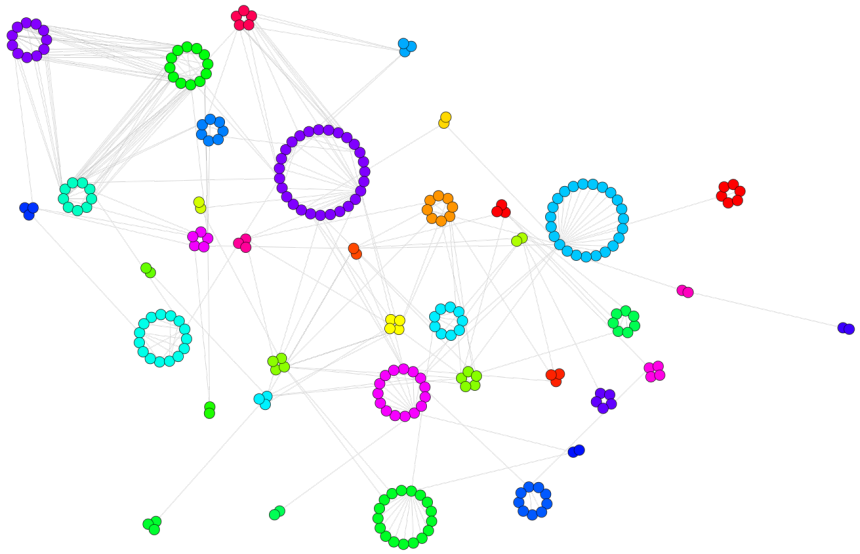


Figure A35: cise layout on large dependency graph.

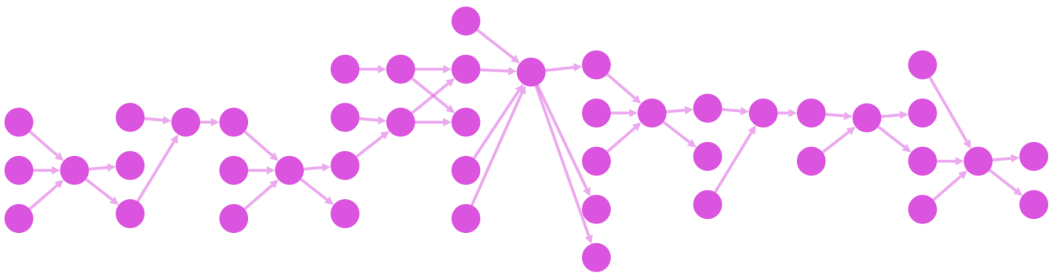


Figure A36: klay layout example from cytoscape.

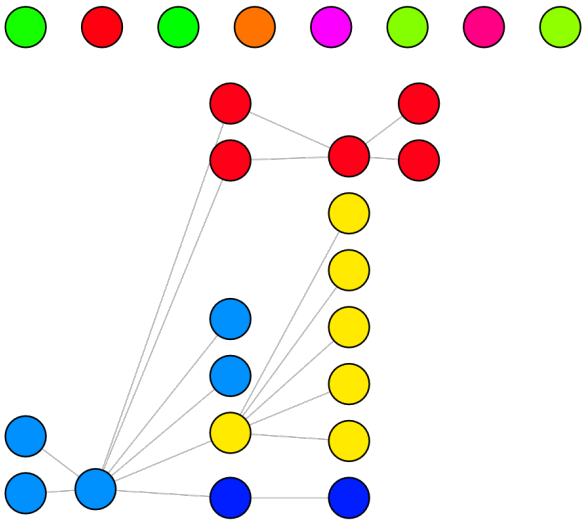


Figure A37: klay layout on small dependency graph.

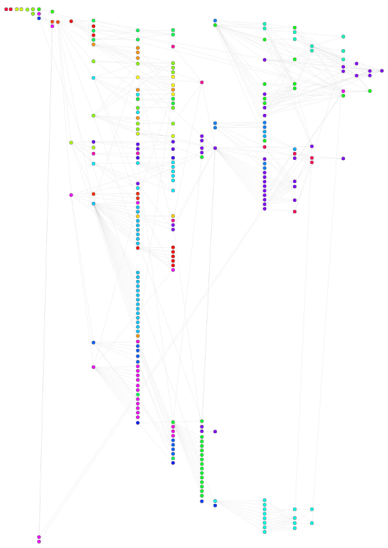


Figure A38: klay layout on large dependency graph.

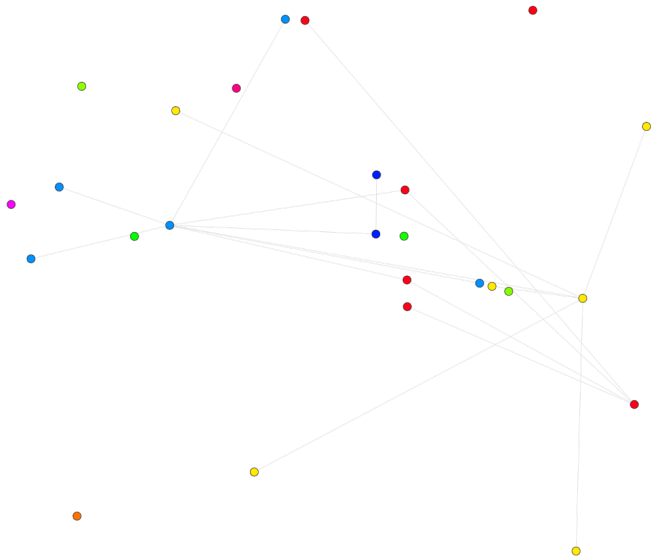


Figure A39: random layout on small dependency graph.

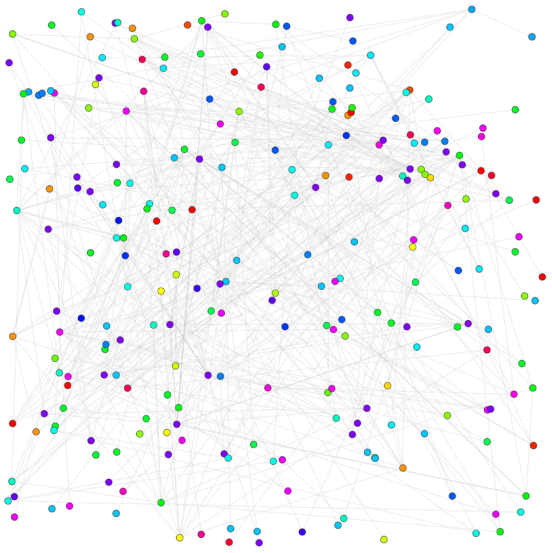


Figure A40: random layout on large dependency graph.

B MoSCoW requirements

M = must have, S = should have, C = could have, W = won't have

Project

1.1	CodeGraph can work with JavaScript projects.	*M*
1.2	CodeGraph can work with projects based on other programming languages	*W*
1.3	CodeGraph projects are automatically built from a git repository.	*M*
1.4	CodeGraph projects automatically stay up to date with the git repository.	*M*
1.5	CodeGraph projects can be configured using config files in the project.	*C*
1.6	CodeGraph projects are accessible through an online webpage.	*M*
1.7	CodeGraph projects are accessible through an Android application.	*W*
1.8	CodeGraph projects are accessible through an IOS application.	*W*
1.9	CodeGraph projects are accessible through a native Windows application.	*W*
1.10	CodeGraph projects are accessible through a native Mac application.	*W*
1.11	CodeGraph projects are accessible through a native Linux application.	*W*
1.12	Upon first accessing a CodeGraph project, the user is asked questions about the project.	*C*
1.13	Users can provide CodeGraph with file-specific information through CodeGraph-specific comments.	*C*

GUI

2.1	CodeGraph has a navigation bar granting access to all the features of CodeGraph.	*M*

Dependency Graph

2.1	CodeGraph visualizes the dependency graph.	*M*
2.2	Every file in the javascript project is represented as a node.	*M*
2.3	A file importing another file is represented by a line connecting two nodes.	*M*
2.4	If file A imports file B, then the line connecting nodes A and B has an arrow pointing from node B to node A.	*C*
2.5	A node displays the name of the file.	*M*

2.6 A node displays information about the users who edited the file. *S*

2.7 A node displays information about the labels associated with that file. *C*

2.8 A node displays information about how the node was changed in the last git commit. *C*

2.9 Users can choose between different layouts, which define where the nodes are placed. *M*

2.10 Users can specify what is the root node/file of the javascript project. *S*

2.11 The user can interactively change the position of the nodes, e.g. by dragging them. *C*

2.12 The user should be able to view the dependency graph in 3D *W*

2.13 Every layout has a clear description which allows the user to choose the most appropriate layout in any situation. *S*

Statistics

3.1 CodeGraph displays statistics about the javascript project. *M*

3.2 The statistics include the number of nodes in the dependency graph. *M*

3.3 The statistics include the number of edges in the dependency graph. *M*

3.4 The statistics include the number of clusters of nodes in the dependency graph. *M*

3.5 The statistics include the number of connected components in the dependency graph. *M*

3.6 The statistics include the ratio between the number of links to the number of nodes. *M*

3.7 The statistics include the diameter of the dependency graph. *M*

3.8 The statistics include a list of nodes with the most incoming edges, corresponding to the files with the most imports. *M*

3.9 The statistics include a list of nodes with the most outgoing edges, corresponding to the files which are imported the most. *M*

3.10 The statistics include a histogram of number of incoming edges, i.e. the number of imports, per node. *M*

3.11 The statistics include a histogram of number of outgoing edges, i.e. the number of times imported, per node. *M*

Filtering

4.1 Users have the ability to filter which files are included in the dependency graph. *M*

4.2 Users can filter individual files. *M*

4.3 Users can filter based on the project file structure, i.e. based on folders. *S*

4.4 Users can filter based on labels. *S*

4.5 Users can filter based which users have edited the files. *S*

4.6 Users can filter based on outgoing and incoming nodes, e.g. only showing a node and all its incoming or outgoing edges. *C*

4.7 Users can filter based on clusters, e.g. only showing the nodes in a specific cluster. *C*

4.8 Filter based on whitelisting, i.e. all files are shown by default and the user specifies which files to hide. *M*

4.9 Filter based on blacklisting, i.e. all files are hidden by default and the user specifies which files to show. *C*

Data

5.1 CodeGraph can smoothly handle projects containing up to 10 files. *M*

5.2 CodeGraph can smoothly handle projects containing up to 100 files. *M*

5.3 CodeGraph can smoothly handle projects containing up to 1.000 files. *M*

5.4 CodeGraph can smoothly handle projects containing up to 10.000 files. *S*

5.5 CodeGraph can smoothly handle projects containing up to 100.000 files. *C*

5.6 CodeGraph can smoothly handle projects containing up to 1.000.000 files. *W*

5.7 Users can add labels to individual nodes / files. *M*

5.8 A node has a list of which users edited the file. *M*

5.9 A node has a list of labels associated with it. *M*

5.10 A node has the number of lines of code that the file contains. *M*

5.11 A node has a description. *M*

5.12 A node has a list of resources associated with it. *S*

5.13 Resources are defined using RDF. *W*

Settings

6.1 Users can modify a CodeGraph project using settings. *M*

C Paper prototype

C.1 Paper Prototype 1

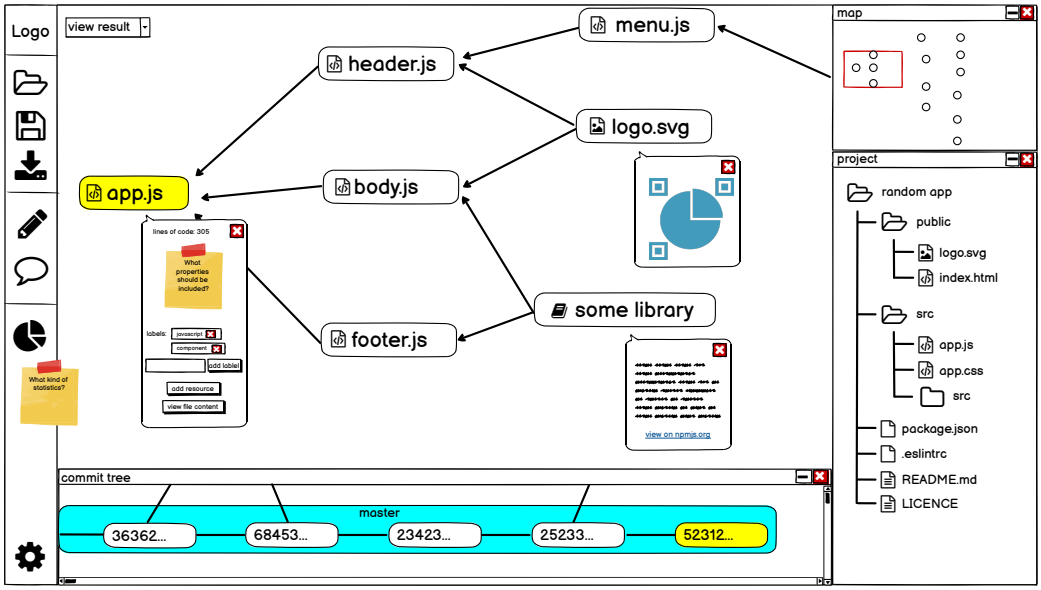
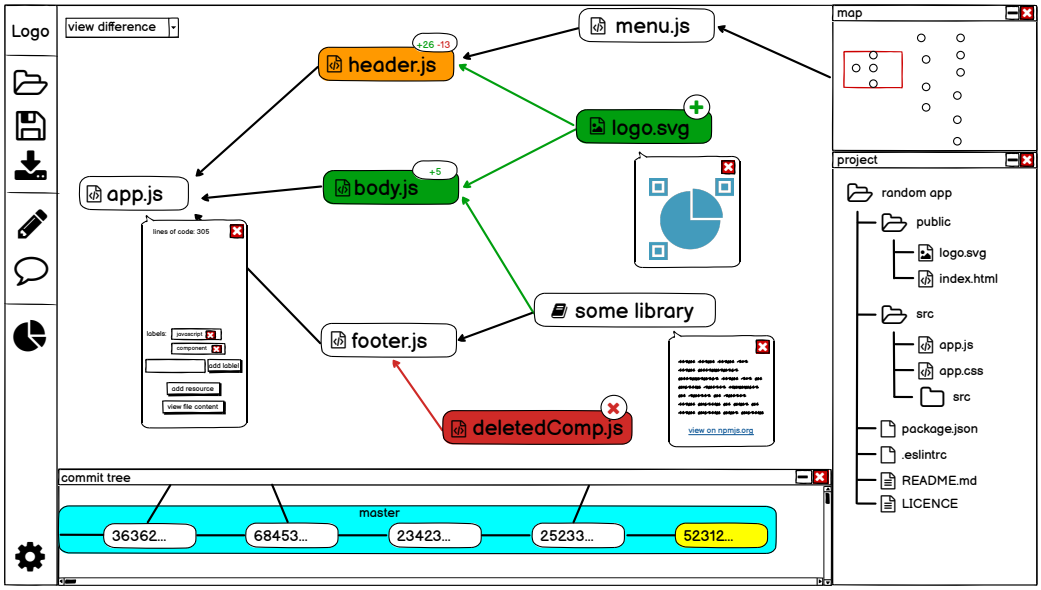
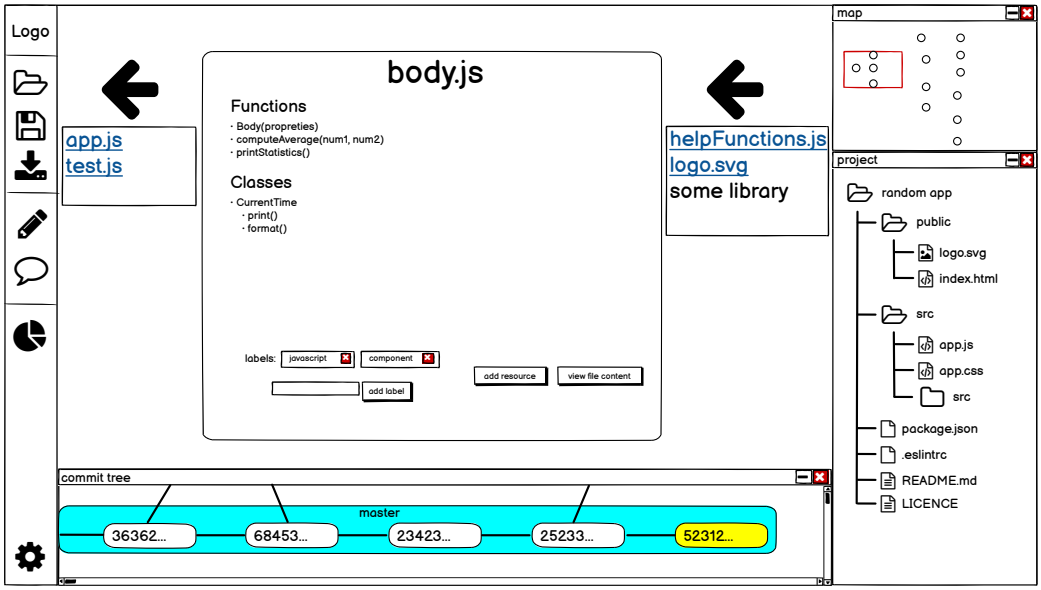
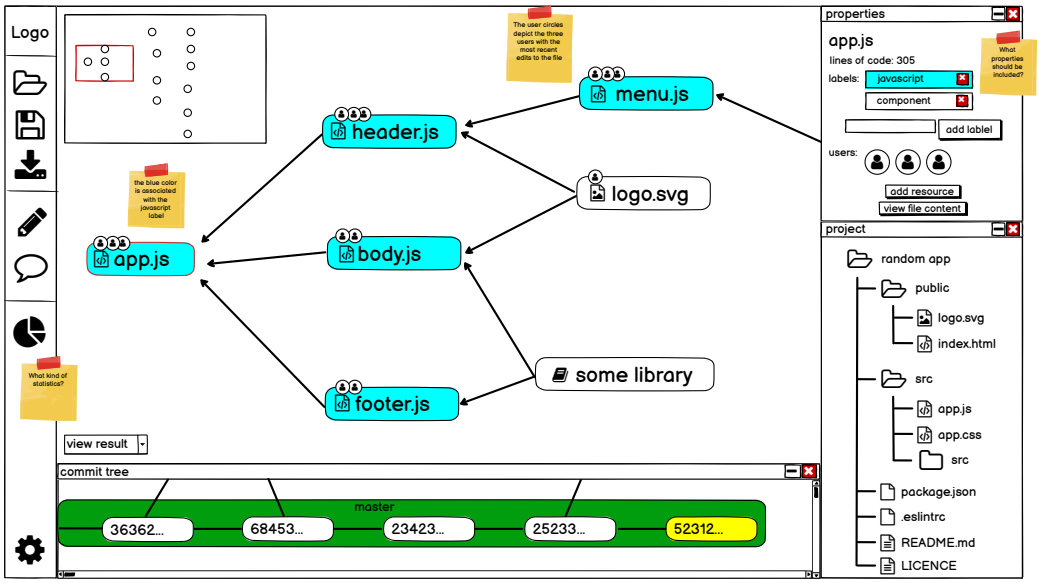


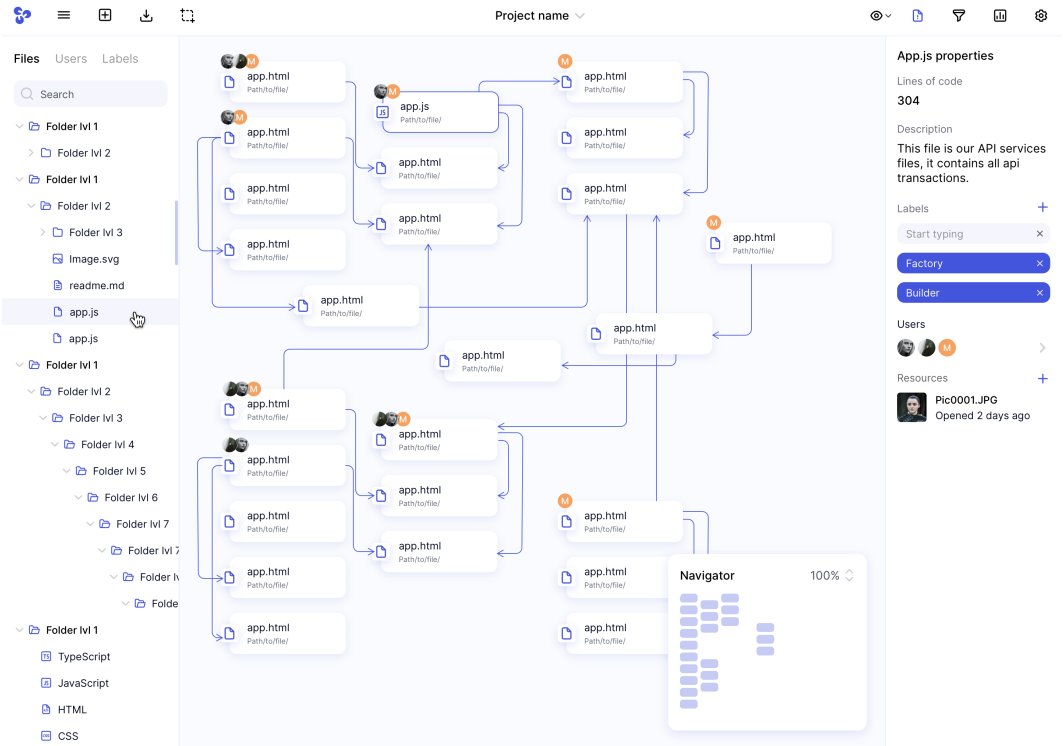
Figure C1: Version 1

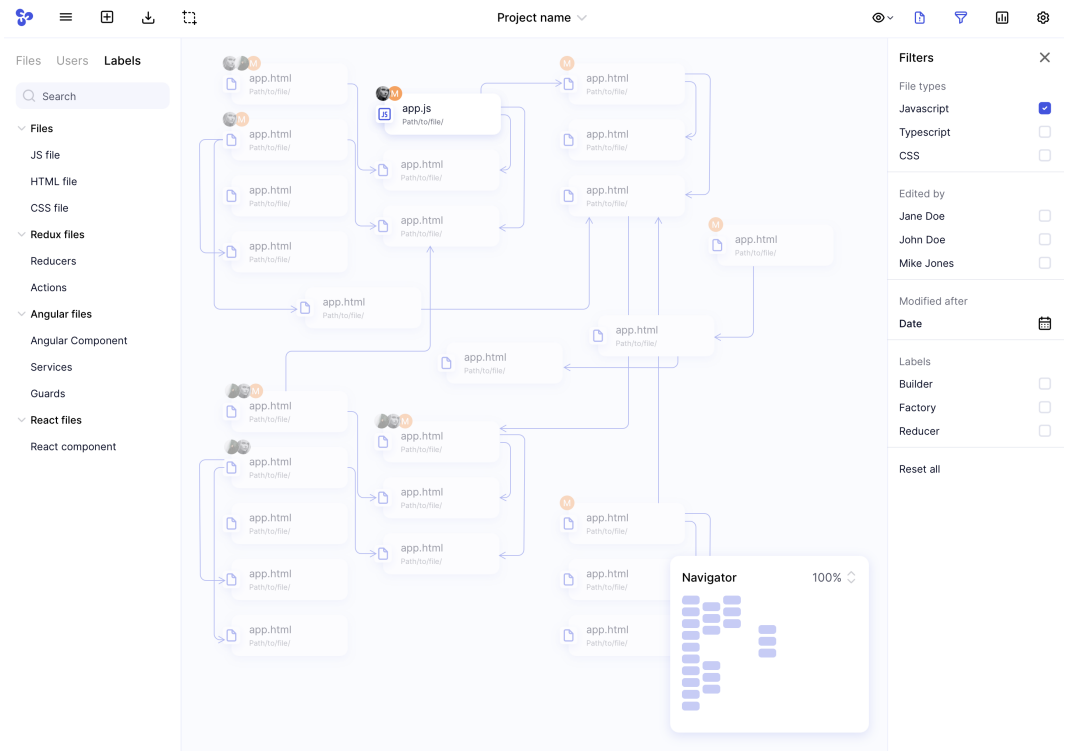
C.2 Paper Prototype 2

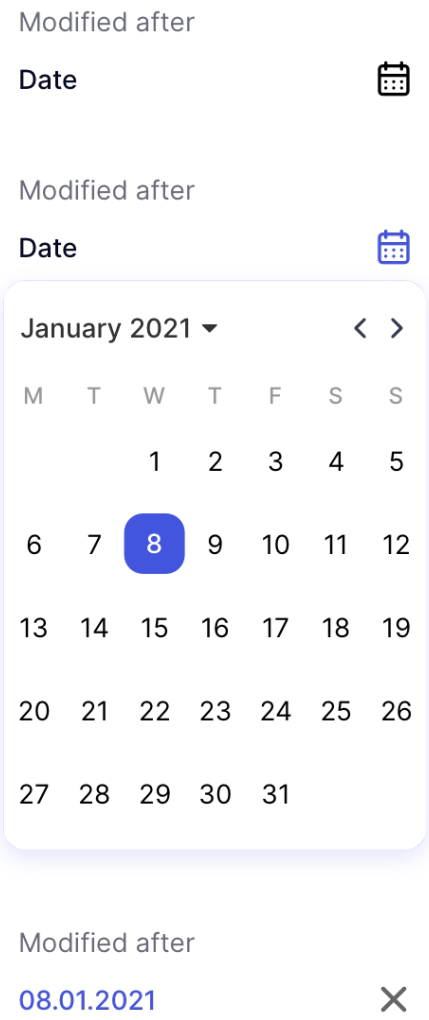

Logo
view difference
menu.js
header.js
+26 -13
logo.svg
app.js
body.js
+5
some library
footer.js
deletedComp.js
lines of code: 305
labels
javascript
component
add label
add resource
view file content
view on npmjs.org
map
project
random app
public
logo.svg
index.html
src
app.js
app.css
src
package.json
.eslintrc
README.md
LICENCE
commit tree
master
36362...
68453...
23423...
25233...
52312...

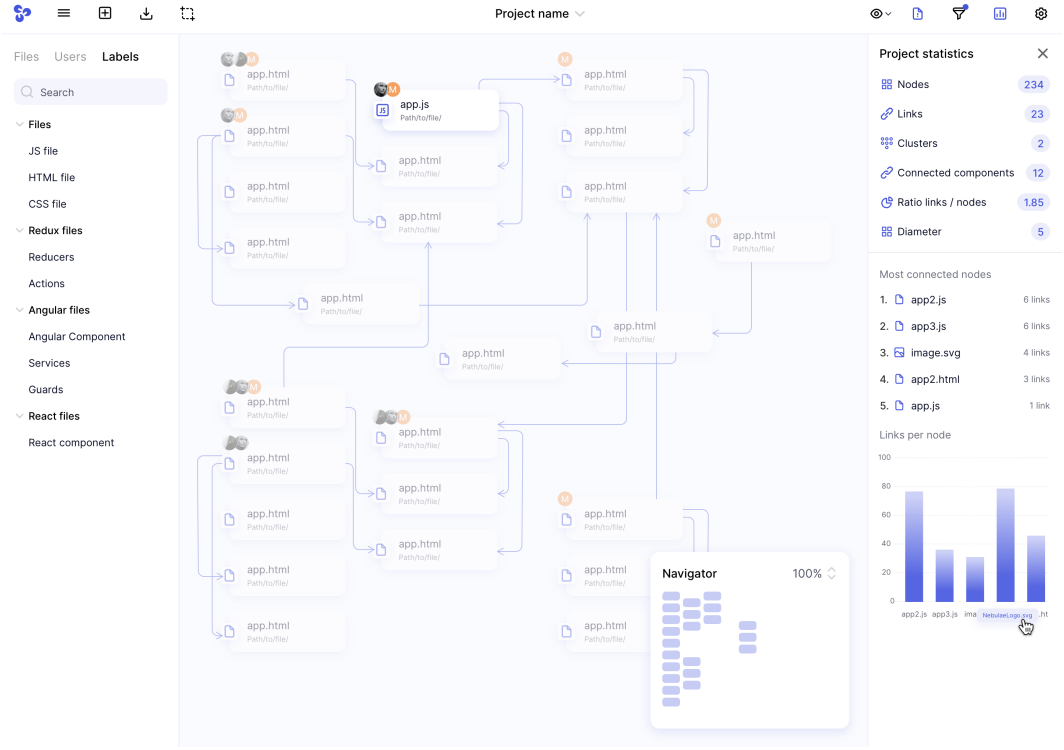

Logo
app.js
test.js
body.js
Functions
· Body(propreties)
· computeAverage(num1, num2)
· printStatistics()
Classes
· CurrentTime
· print()
· format()
labels:
javascript
component
add label
add resource
view file content
helpFunctions.js
logo.svg
some library
map
project
random app
public
logo.svg
index.html
src
app.js
app.css
src
package.json
.eslintrc
README.md
LICENCE
commit tree
master
36362...
68453...
23423...
25233...
52312...

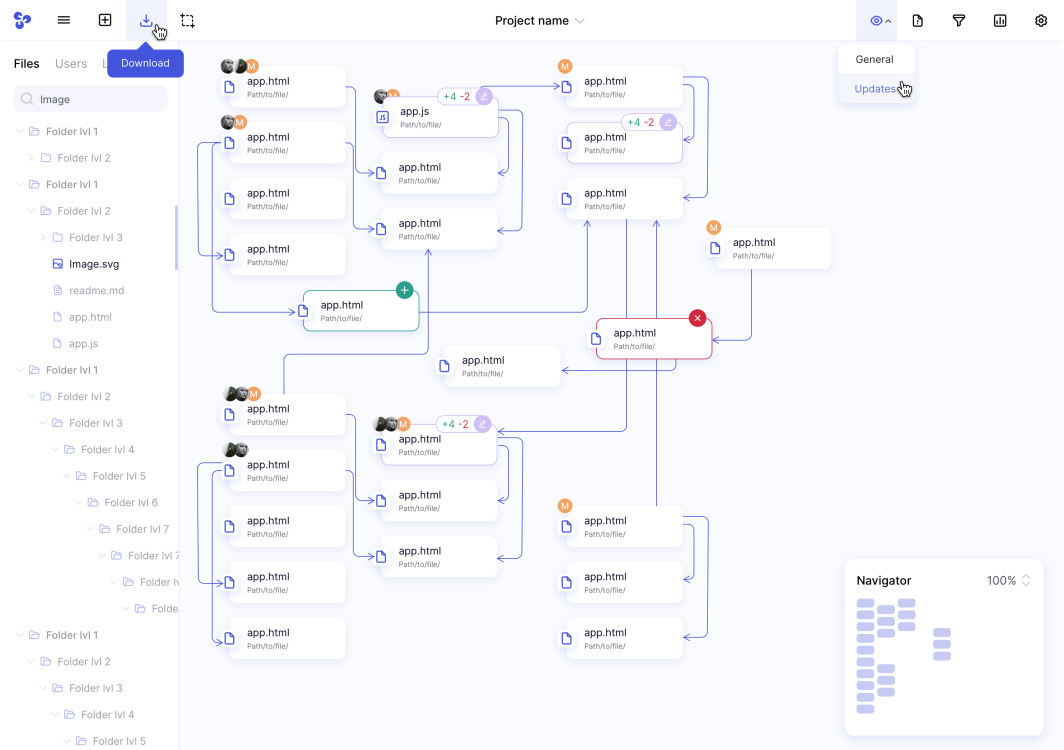

Logo
menu.js
header.js
logo.svg
app.js
body.js
some library
footer.js
properties
app.js
lines of code: 305
labels:
javascript
component
add label
users:
add resource
view file content
project
random app
public
logo.svg
index.html
src
app.js
app.css
src
package.json
.eslintrc
README.md
LICENCE
view result
commit tree
master
36362...
68453...
23423...
25233...
52312...

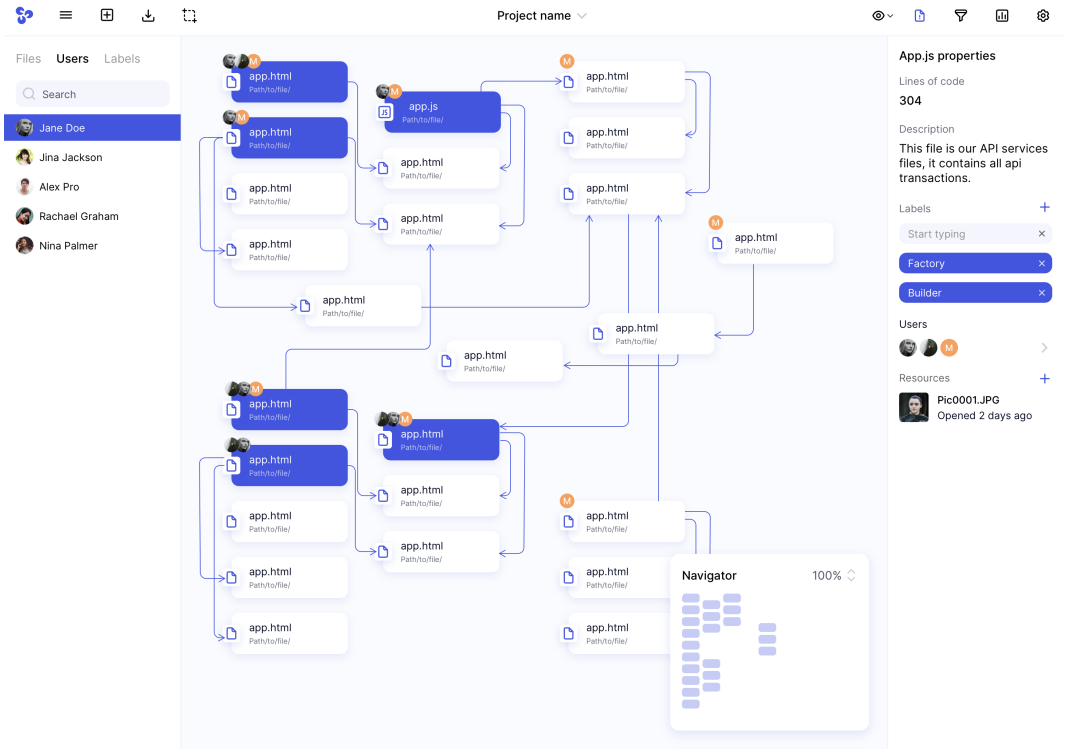

Project name
Files
Users
Labels
Search
Folder lvl 1
Folder lvl 2
Image.svg
readme.md
app.js
TypeScript
JavaScript
HTML
CSS
app.html
app.js
Navigator
100%
App.js properties
Lines of code
304
Description
This file is our API services files, it contains all api transactions.
Labels
Start typing
Factory
Builder
Users
Resources
Pic0001.JPG
Opened 2 days ago


Project name
Files
Users
Labels
Search
Files
JS file
HTML file
CSS file
Redux files
Reducers
Actions
Angular files
Angular Component
Services
Guards
React files
React component
app.html
Path/to/file/
app.js
Path/to/file/
Filters
File types
Javascript
Typescript
CSS
Edited by
Jane Doe
John Doe
Mike Jones
Modified after
Date
Labels
Builder
Factory
Reducer
Reset all
Navigator
100%


Modified after
Date
Modified after
Date
January 2021
M T W T F S S
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
Modified after
08.01.2021

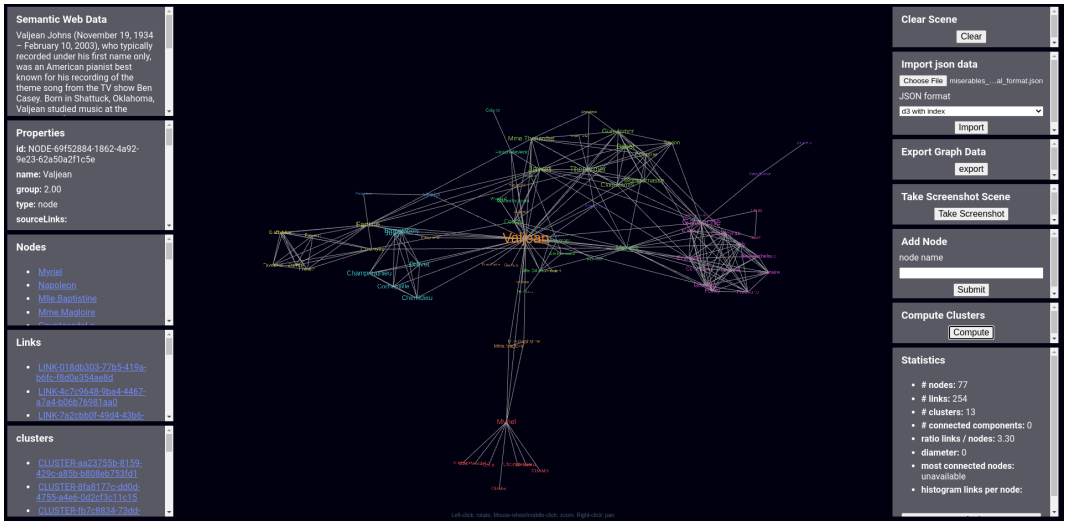

Project name
Files
Users
Labels
Search
Files
JS file
HTML file
CSS file
Redux files
Reducers
Actions
Angular files
Angular Component
Services
Guards
React files
React component
app.html
app.js
Path/to/file/
Project statistics
Nodes 234
Links 23
Clusters 2
Connected components 12
Ratio links / nodes 1.85
Diameter 5
Most connected nodes
1. app2.js 6 links
2. app3.js 6 links
3. image.svg 4 links
4. app2.html 3 links
5. app.js 1 link
Links per node
Navigator 100%

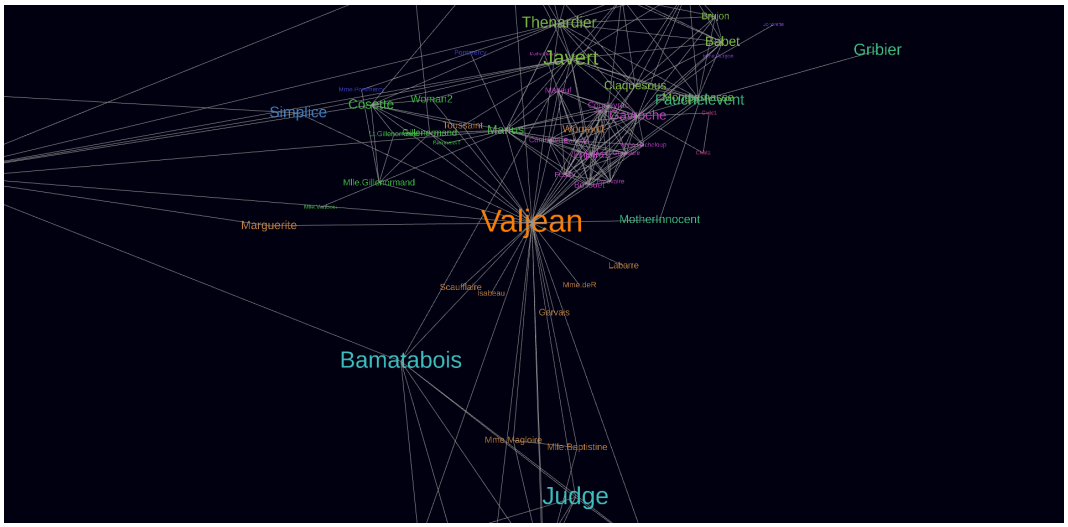

Project name
Files
Users
Download
Image
Folder lvl 1
Folder lvl 2
Folder lvl 3
Image.svg
readme.md
app.html
app.js
Folder lvl 4
Folder lvl 5
Folder lvl 6
Folder lvl 7
General
Updates
app.html
app.js
Path/to/file/
+4 -2
Navigator 100%


Project name
Files
Users
Labels
Search
Jane Doe
Jina Jackson
Alex Pro
Rachael Graham
Nina Palmer
app.html
Path/to/file/
app.js
App.js properties
Lines of code
304
Description
This file is our API services files, it contains all api transactions.
Labels
Start typing
Factory
Builder
Users
Resources
Pic0001.JPG
Opened 2 days ago
Navigator
100%

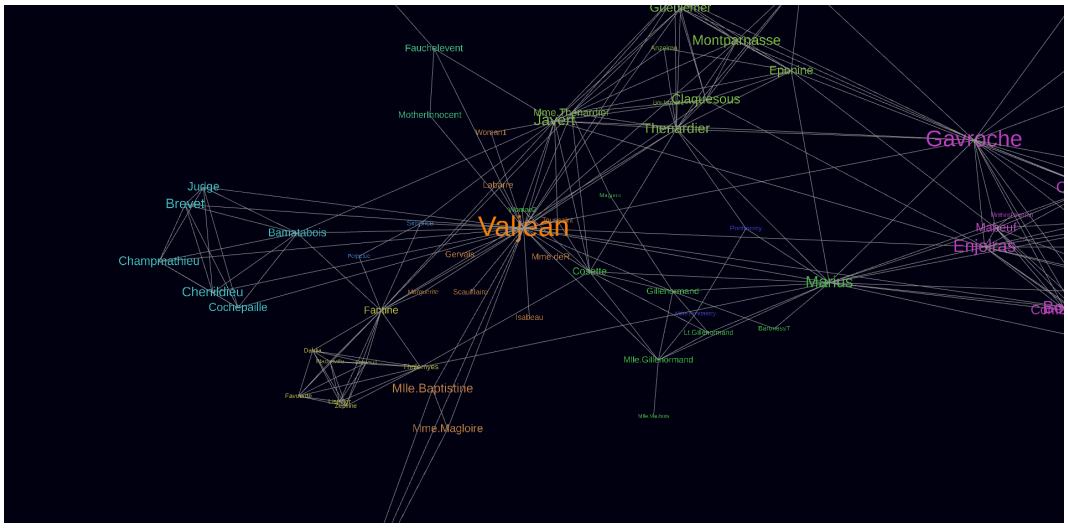
D Prototype

D.1 Prototype 1





D.2 Prototype 2


Montparnasse
Fauchelevent
Claquesous
Mme Thenardier
Javert
Thenardier
Gavroche
Mother Innocent
Labarre
Valjean
Bamatabois
Champmathieu
Brevet
Judge
Cochepaille
Cheminladieu
Gervais
Fantine
Marius
Enjolras
Mabeuf
Isabeau
Mlle.Gillenormand
Mlle.Baptistine
Mme.Magloire

E Test Subject Consent Form

CodeGraph Usability Study - Consent Form

You are being invited to participate in a usability study of CodeGraph, a JavaScript dependency-graph visualization application. This study is being conducted by Robert van Barlingen, ICT Innovation master student at Aalto University in Finland. This study is conducted as part of Robert van Barlingen's master thesis.

There are no known risks if you decide to participate in this research study. There are no costs to you for participating in the study. The information you provide will be used to assess the user experience when using CodeGraph. The questionnaire will take about thirty minutes to complete. The information collected may not benefit you directly, but the information learned in this study should provide more general benefits.

This survey is anonymous. Do not write your name on the survey. No one will be able to identify you or your answers, and no one will know whether or not you participated in the study. Should the data be published, no individual information will be disclosed.
Your participation in this study is voluntary. By completing this usability test, you are voluntarily agreeing to participate. You are free to decline to answer any particular question you do not wish to answer for any reason.

This session will be recorded. This includes a video and audio recording of yourself as well as a recording of the computer screen. These videos will be analyzed by Robert van Barlingen and possibly by other researchers. They shall not be made public and once the study has been completed, these recordings shall be destroyed.

If you have any questions about the study, please ask Robert van Barlingen directly or contact him after the fact at bobby.van.barlingen@gmail.com.

By signing this form, I state that I have read and agree with the terms stipulated above

NAME + DATE SIGNATURE

F Usability Test Google Form

CodeGraph

Thank you for participating in this usability test. The purpose of this test is to analyze the user experience when using CodeGraph. CodeGraph is a tool for analyzing dependencies in a JavaScript project.
The test consists of four parts:
1. General Information
2. Test Session with General Applications
3. Test Session with CodeGraph
4. Experience Survey

For each part, carefully complete the tasks and fill out the google form. Please notify the researcher whenever you have completed a part.
This survey is completely anonymous. The results of this survey will be used in the study of CodeGraph and will be made public.

*Vereist

General Information

1. What is your age? (in years) *

_____

2. What is your gender? *

Markeer slechts één ovaal.

- ◯ Male
- ◯ Female
- ◯ Prefer not to say
- ◯ Anders: _____

3. What is your occupation? *

_____

4. Do you have experience programming? *

Markeer slechts één ovaal.

- ◯ Yes
- ◯ No

5. Do you have experience programming with JavaScript? *

Markeer slechts één ovaal.

- ◯ Yes
- ◯ No

Test Session with General Applications

In this session you will complete a set of tasks about the CodeGraph codebase using general development tools. These tools include a file explorer (nautilus), a text editor (gedit), a terminal (gnome terminal) and an IDE (Webstorm). You are free to use these applications as you see fit. These applications should already be open on this computer. If this is not the case, or if you have any questions during the test session, feel free to ask a researcher. Try to solve these tasks as independently as possible, but if you have questions concerning the functioning of the tools or the specifications of JavaScript, you may freely ask.
If there is any task that you are not able to complete, you can answer "do not know."

6. How many import statements does /src/App.tsx contain? *

_____

7. How many files import a module from /src/contexts/dependencies/dependencies.tsx? *

_____

8. How many lines of code does /src/index.tsx contain? *

_____

9. How many files does the project contain? *

10. Which file contains the most import statements? *

11. What are the initials of someone who has edited /src/utils/file-utils.ts? *

Test Session with CodeGraph

In this session you will complete a similar set of tasks as in the previous session. This time, you may only use CodeGraph to complete the tasks. CodeGraph should already be open on this computer. If this is not the case, or if you have any questions during the test session, feel free to ask a researcher. Try to solve these tasks as independently as possible, but if you have questions concerning the functioning of CodeGraph or the specifications of JavaScript, you may freely ask. If there is any task that you are not able to complete, you can answer "do not know."

12. How many import statements does /src/components/Minimap/Minimap.tsx contain? *

13. How many files import a module from /src/utils/file-utils.ts *

14. How many lines of code does /src/components/Contexts/Contexts.tsx contain? *

15. How many import statements does the project contain in total? *

16. Which file is imported the most by other files? *

17. What are the initials of someone who has edited /src/components/LeftMenuBar/LeftMenuBar.tsx? *

Experience Survey

In this final survey, we would like to hear experience working with CodeGraph.

18. How enjoyable was your experience using CodeGraph compared to general tools? *

Markeer slechts één ovaal.

	1	2	3	4	5	
unenjoyable	◯	◯	◯	◯	◯	enjoyable

19. How efficiently did you feel you could work with CodeGraph compared to general tools? *

Markeer slechts één ovaal.

	1	2	3	4	5	
unefficient	◯	◯	◯	◯	◯	efficient

20. What are positive aspects of CodeGraph? *

21. What are the negative aspects of CodeGraph? *

22. Do you have any other comments?

G Usability Test Results

CodeGraph

3 antwoorden

Analyse publiceren

General Information

What is your age? (in years)

3 antwoorden

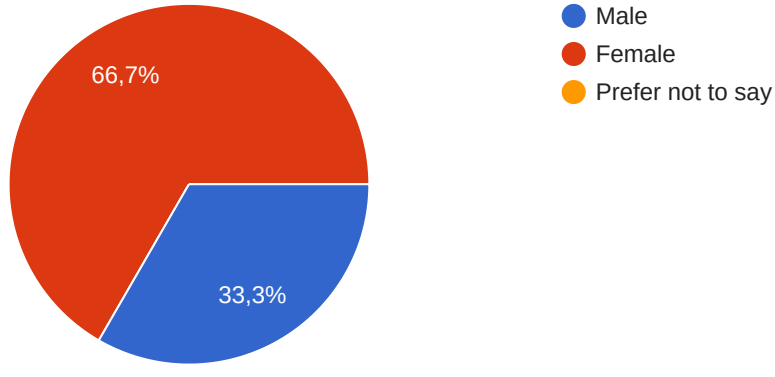
24

42

27

What is your gender?

3 antwoorden



What is your occupation?

3 antwoorden

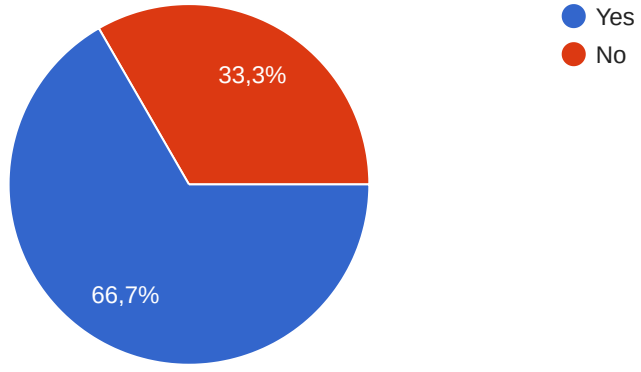
Primary school teacher P2

Techno Marketing

Artist

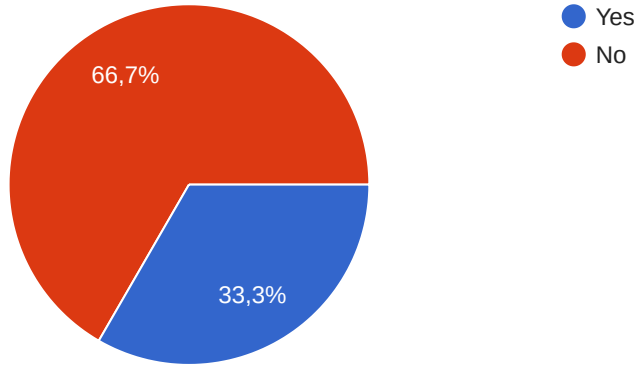
Do you have experience programming?

3 antwoorden



Do you have experience programming with JavaScript?

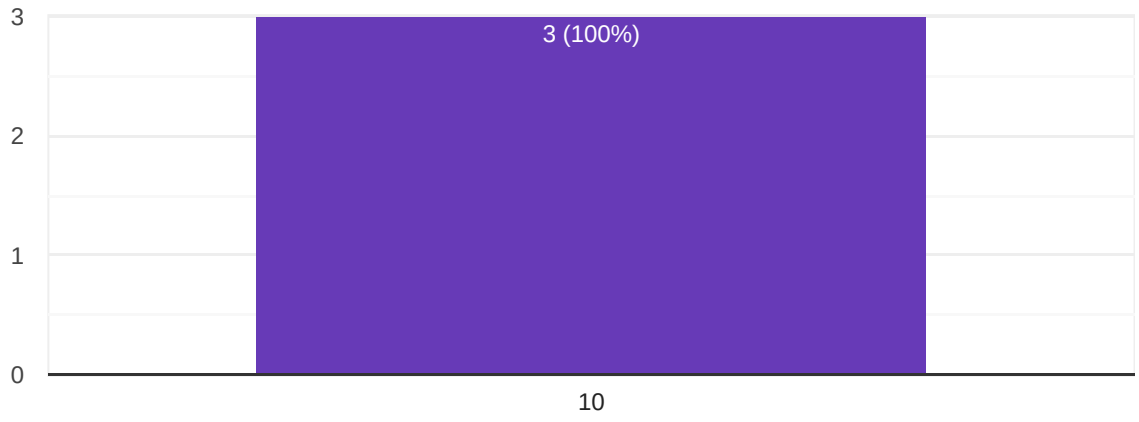
3 antwoorden



Test Session with General Applications

How many import statements does /src/App.tsx contain?

3 antwoorden



How many files import a module from /src/contexts/dependencies/dependencies.tsx?

3 antwoorden

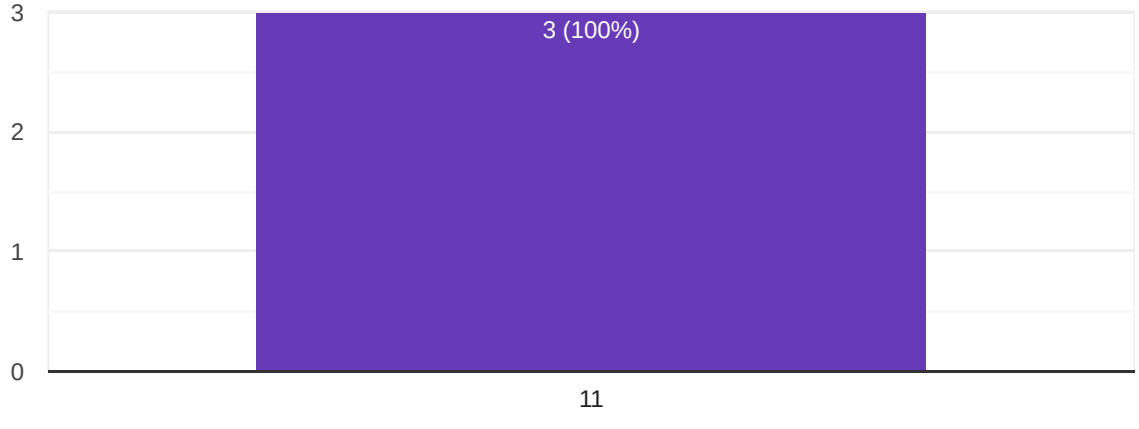
i don't know

dont know

2

How many lines of code does /src/index.tsx contain?

3 antwoorden



How many files does the project contain?

3 antwoorden

I don't know

1252

74.373

Which file contains the most import statements?

3 antwoorden

I dont't know

dont know

cytoscapehooks.ts

What are the initials of someone who has edited /src/utils/file-utils.ts?

3 antwoorden

E.K.

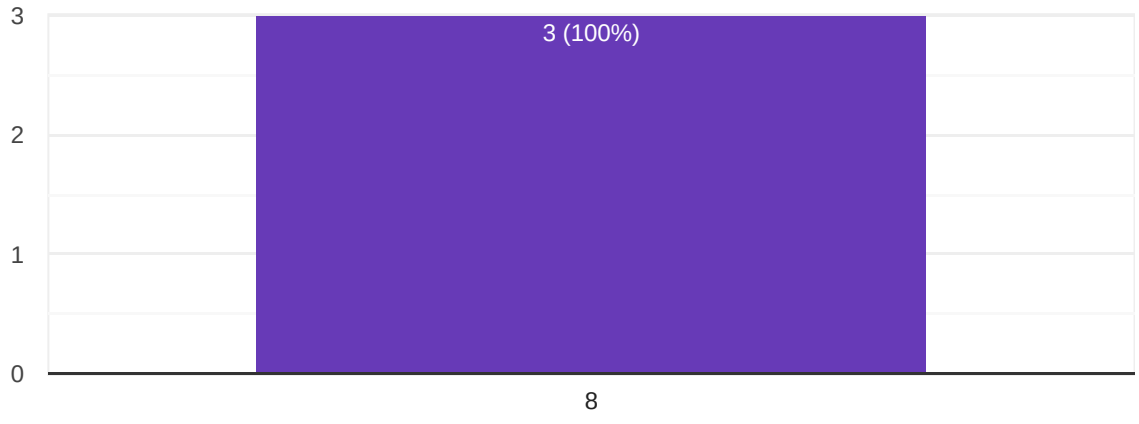
dont know

EK

Test Session with CodeGraph

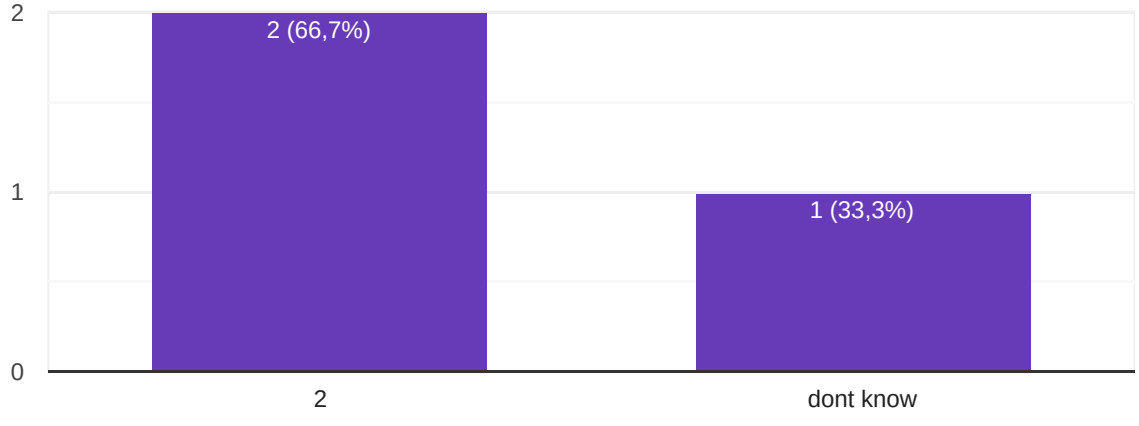
How many import statements does /src/components/Minimap/Minimap.tsx contain?

3 antwoorden



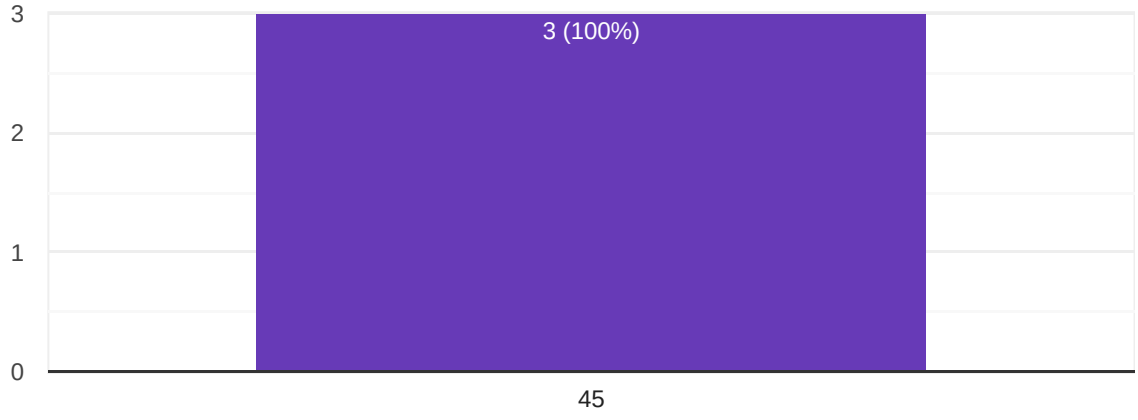
How many files import a module from /src/utils/file-utils.ts

3 antwoorden



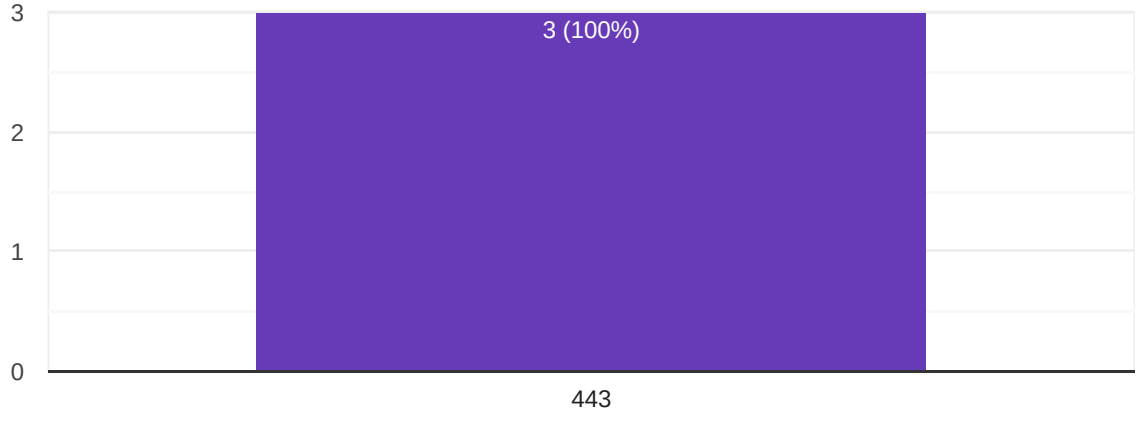
How many lines of code does /src/components/Contexts/Contexts.tsx contain?

3 antwoorden



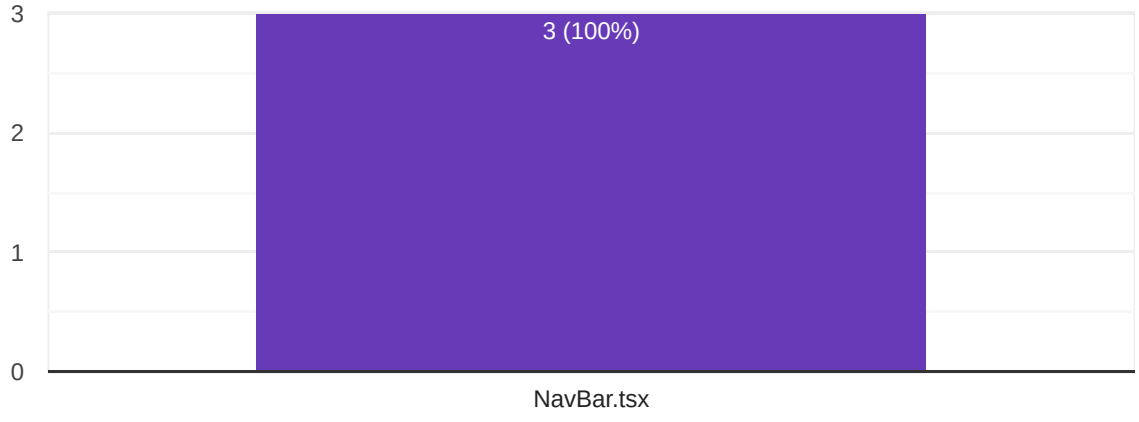
How many import statements does the project contain in total?

3 antwoorden



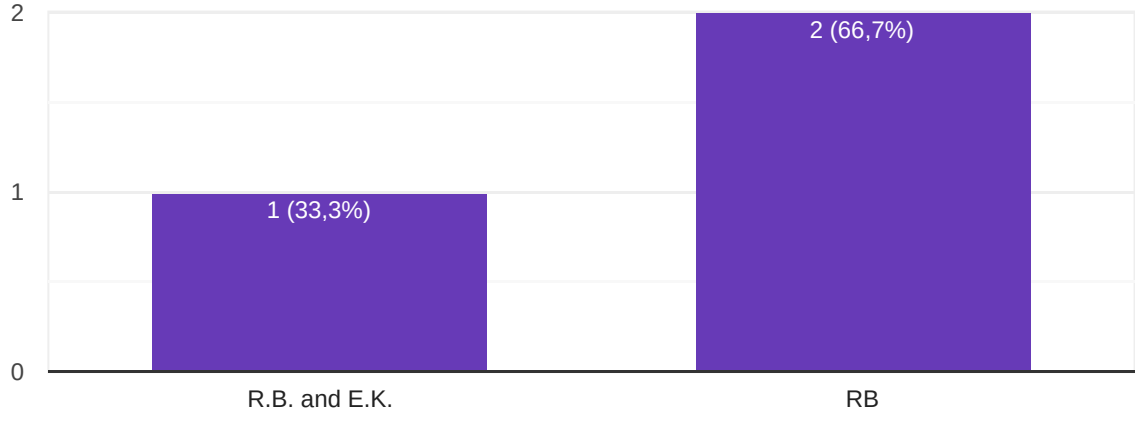
Which file is imported the most by other files?

3 antwoorden



What are the initials of someone who has edited /src/components/LeftMenuBar/LeftMenuBar.tsx?

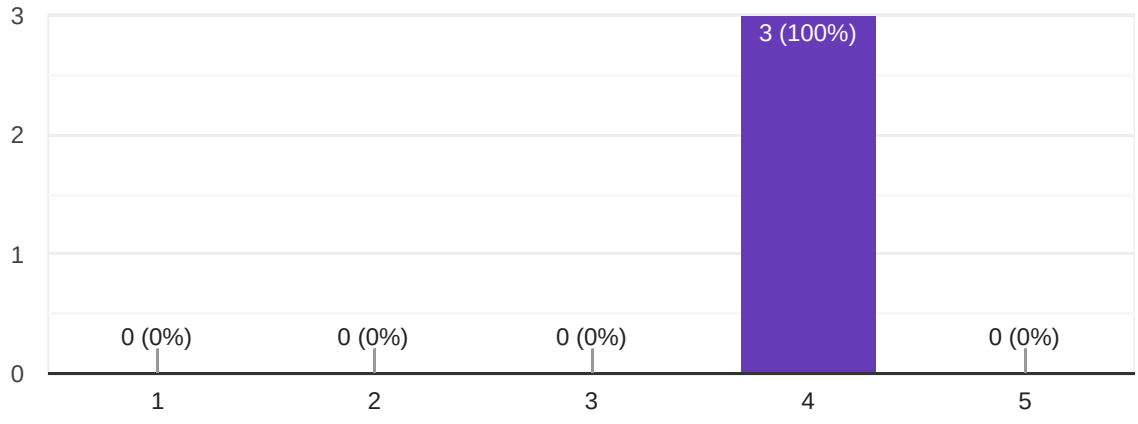
3 antwoorden



Experience Survey

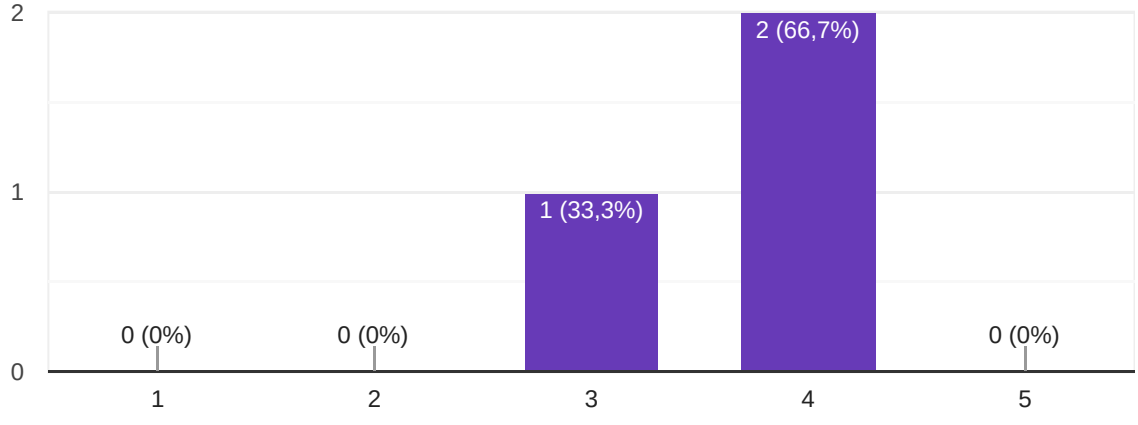
How enjoyable was your experience using CodeGraph compared to general tools?

3 antwoorden



How efficiently did you feel you could work with CodeGraph compared to general tools?

3 antwoorden



What are positive aspects of CodeGraph?

3 antwoorden

It is well organized and you can find things easily.

All you need in one central place

Het is als een puzzel. Grafisch ziet het er goed uit.

What are the negative aspects of CodeGraph?

3 antwoorden

It is hard to see in which direction the arrow goes.

editor is not easy to use; file info box might show more

Moeilijk om in het begin een overzicht te krijgen. Lang zoeken hoe het werkt maar eens je het doorhebt zit het wel logisch in elkaar.

Do you have any other comments?

3 antwoorden

I like it more then other general tools.

very good tools but users need some training before they can use its full potential

Soms zijn de vele lijntjes door elkaar een beetje onoverzichtelijk en de richting van de pijltjes moeilijk te achterhalen, als je het 100% visueel bekijkt.