Visualization of Theory Graphs

Master's Thesis in Computer Science

submitted
by

Marcel Rupprecht

born 21.07.1994 in Forchheim

Written at

Professur für Wissensrepräsentation und -verarbeitung
Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg.

Advisor: Michael Kohlhase

Started: 12.11.2018

Finished: 13.05.2019

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Erlangen, den 13th May 2019

Abstract

Many mathematical software systems are based on the explicit representation of knowledge and organize that modularly. Even though there are a few tools focusing on giving the user a direct way to manipulate and interact with the represented knowledge using graph viewers like TGView [Rup17] and TGView3D [RM19], none of these tools describe the underlying requirements. However these requirements would set a strong future research direction and every theory graph exploration tool would benefit greatly from basing them on these requirements. Therefore I introduce in this work several requirements for a theory graph exploration tool, which were collected by doing qualitative interviews and analyzing use cases. I evaluate the current implementation state of TGView based on the requirements condensed in this work. This evaluation shows, that TGView in the current version meets many of the outlined requirements, but still suffers from big limitations due to the browser, in which TGView runs. These browser limitations seem to be not circumventable. That is why I think the future research direction will be guided by technologies like C# and Unity, which still run in the browser but allow multi-threading and GPU boosted rendering.

Contents

Chapter 1

Introduction

Many mathematical software systems are based on the explicit representation of mathematical knowledge and organize them modularly. Computer algebra systems like GAP [Gap] or Sagemath [Sag] organize mathematical objects in a class hierarchy and dispatch computational methods along that. Axiom [Ope] even adds axiomatizations to the object classes to encode system knowledge. Theorem prover systems often organize axioms, definitions, theorems and proofs into theories, which are connected by inclusions and other theory morphisms (theory graphs). IMPS [Far93] has pioneered the ”little theories paradigm”, and the dominant provers like Coq [Coq], Isabelle [Pau94], and PVS [Owr92] follow its intuitions; Also the Mizar [Miz] system has a system of articles that can be understood as a lightweight theory graph. But the use of modular theory-graph like knowledge organization principles is not restricted to computational and formal systems: OMDoc-based active documents [Koh11] are generated from a theory graph of document fragments, and the SMGloM terminology [Koh14] uses theory graphs to model both domain concept dependencies as well as multilinguality. Linguistic relations like hypernymy or antonymy between terms can be derived from the theory graph structure.
Also Wikipedia is often seen as a large ”citation graph” of knowledge and books sometimes end the preface with a graph-like diagram of chapter dependencies to allow readers multiple paths through the book. Surprisingly, most of the systems discussed above do not give the user a direct way to interact with the represented knowledge via its graph structure. Many of the systems do allow to generate a static graph via the GraphViz system [Ell04]. Even though one can add limited interactions by adding links to SVG versions of the graph, any change leads to server-side re-layouting, which cannot take display requirements of the client into account. An exception to this rule is the Protégé system for ontology development, which sports numerous graph display plugins [Pro] for interaction with the ontologies. For theory-graph-based systems, understanding

the modular structure of theories and their connections is crucial for accessing the knowledge in the system and understanding its behavior. Visualizing them as graphs is the most natural way of conveying this knowledge. However, most such graphs are too large for normal screens, if presented in their full glory. Therefore, the user needs to interact pan, tilt, and zoom, the graph, colorize, hide, and cluster nodes, and drill down on the information of the theories and morphisms. It is this important interaction aspect that is not supported in the current state of the art.
Therefore I developed TGView, a flexible, interactive Theory-Graph Viewer as part of a university project at the Friedrich-Alexander-University Erlangen-Nuremberg. After several experiments with TGView, we had a tool to play with. However due to the explorational and experimantational way of developing TGView, I had no direction, requirements or anything else, which would direct the future progress of TGView and other theory exploration tools. After completing the experimentation phase, I had to define a clear direction and requirements to ensure further efficient and effective progress. So the research question for this Master thesis is:

What are the requirements for a MMT/OMDoc ecosystem exploration tool and how does the latest version of TGView solve them?

Contributions This work focuses on requirements on a high level. This allows the requirements to be applied to theoretical any other system, which is built for the MMT/OMDoc ecosystem. Therefore the contributions of this work are three-fold:

- This work defines several requirements for working with MMT/OMDoc ecosystem and exploration of this system to maximize information visualization, knowledge deduction and user experience.
- TGView is checked for fulfillment of all requirements and the corresponding solutions are analyzed.
- In the last step possible improvements and workarounds for limitations are collected and outlined.

Overview The following thesis is divided into six chapter. The first chapter is an introduction. In the second chapter the thesis will outline related work. Chapter three will explain preliminaries and how TGView in general works. The fourth chapter focuses on condensing the use cases and experience we collected with TGView into general applicable requirements for an MMT/OMDoc ecosystem exploration tool. In the fifth chapter this thesis analyzes TGView for fulfilling the

requirements defined in chapter four. The thesis ends with outlining solutions and therefore possible future directions for limitations of TGView found in chapter five.

Chapter 2

Related Work

Due to the huge demand for graph exploration tools, there exist several frameworks for drawing graphs and interacting with them. In this section I outline possible graph exploration tools and their limitations.

GraphViz The most common solution used to visualize graphs is GraphViz [Ell04] [Gra]. GraphViz is a graph visualization software developed by AT&T and Bell Labs. The description of the graph is implemented by the markup language DOT. This DOT file contains a list of all nodes and edges as well as their relations. An example and the corresponding image can be seen in Figure 2.1.

```
digraph
{
    a -> b;
    b -> c;
    c -> d;
    d -> a;
}
```

Listing 2.1: DOT Format Example

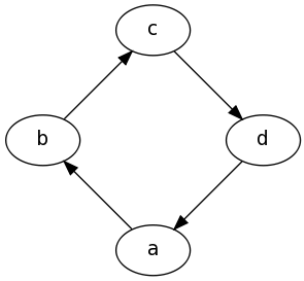


Figure 2.1: Graph generated with GraphViz

A graph generated by GraphViz has several automatic but fixed layout algorithms, so that the graph can be experienced visually well for several use cases. GraphViz generates a static output image from the input, mostly using the SVG format. Even though it is possible to change and interact with the static SVG graph via sophisticated algorithms, HTML overlays and by adding links, possible interactions are still limited. Usually the interactions with SVG graphs look like

follows.

- Show or hide nodes and edges
- Follow links by clicking on nodes
- Zoom in and out
- Move the graph as a whole
- Applying cluster and layout algorithms

Even if the above mentioned points can be implemented in a graph viewer with corresponding effort, every change to the graph requires communication with the GraphViz backend. This behavior does not satisfy the interactive applications nowadays, especially in the web area. In addition, features such as coloring individual nodes, moving single nodes, adding and deleting nodes, clustering by hand and many other interaction options cannot be implemented at all or may only be implemented with enormous effort.

Gephi Originally introduced to allow data scientists and analysts interacting with huge graphs and help them discovering patterns, Gephi quickly expanded into general graph exploration use cases [Bas09]. It is an open-source tool developed using Java for exploratory data analysis. As the authors chose Java as programming language, Gephi can be installed on every major operating system, which supports Java binaries.
Gephi can read graph data in the following formats:

- **Spreadsheets/CSV:** Each row contains two nodes. The first node is the ”from node” and the second one the ”to node”. On this way whole graphs can be constructed using table based data.
- **Database:** Additionally Gephi provides the possibility to load graphs from graph databases like Neo4j [Neo].
- **Project Files:** Of course Gephi also supports some home-made project files, which allow importing and exporting graphs.
- **DOT:** Besides the custom formats, Gephi also supports to read graphs in DOT format, which is also used by GraphViz.

In general Gephi tries to offer a wider variety of interaction possibilities with the graph, which include standard interactions like zooming in and out, moving nodes, adding/deleting nodes/edges, clustering nodes and colorizing nodes. Some of the more outstanding features are as follows:

- **Dynamic Filters:** Gephi allows to write and save complex filter queries, which shows/hides only nodes affected by this query.
- **Cartography:** The graph can be improved by using several ranking and partition data algorithms. These will also customize color and size of nodes to make the network as easy to read as possible.
- **Export for several Formats:** Gephi allows to export the final graph as PDF, SVG and PNG.
- **OpenGL Usage:** OpenGL is used to speed up the whole graph drawing and interaction procedure and therefore also allows gigantic graphs of up to 100’000 nodes and 1’000’000 edges.
- **Many Layout Algorithms:** There are many different layout algorithms implemented in the default Gephi environment and even more can be added by using a plugin system.

In general Gephi provides rich interactions, has several different layout algorithms and scales up to one million edges. The major drawbacks of Gephi lies in their programming language and sometimes overwhelming UI.
As Gephi is based on Java (not Java-Web-Apps), it cannot be used directly in a browser without major rewrites of the Gephi code. Additionally Java in the browser is still seen very sceptically as there have been several security issues in the past. A great point of Gephi is its enormous amount of settings, query options and layout algorithms. However this also means that a potential user has to dive deeply into Gephi until he is able to work with the tool efficiently. Ultimately Gephi is optimized for general graph exploration. This means, that there exist only very few algorithms optimized for exploration of the MMT/OMDoc ecosystem. Figure 2.2 shows Gephi and an example graph.

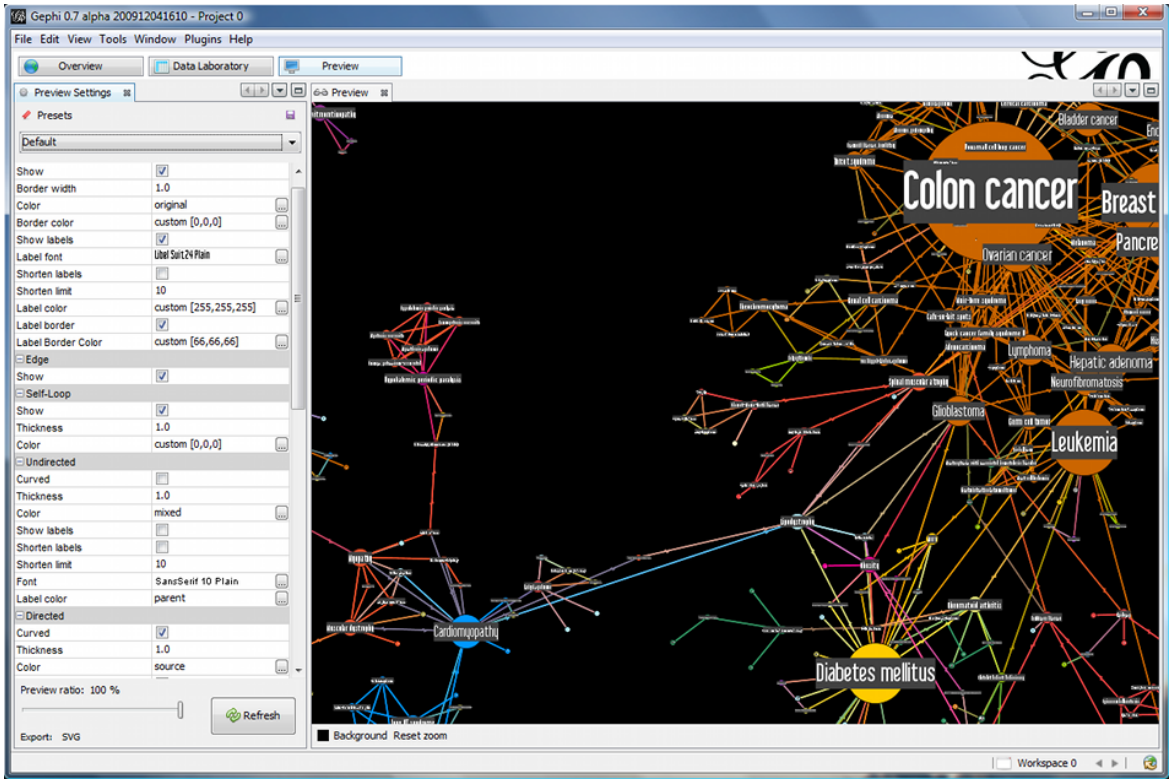


Figure 2.2: Example graph generated with Gephi

yEd Graph Editor Another big graph exploration tool is yEd, which started as UML diagram modeling software, but was recently extended to handle and layout any graphs [yEd].
Similar to Gephi, yEd has several interaction possibilities. In this point Gephi and yEd are quite similar, even though yEd lacks a few major features, which Gephi offers like not that many different layout algorithms, no dynamic filtering and only home-made and excel based graph format for import is allowed.
As yEd started as UML modeling program, it allows to combine any UML diagram with any kind of graphs. This offers the possibility to embed additional information in the graph without explicitly modeling these information as nodes or edges (e.g. add general side-notes or fixed diagrams/starting points). yEd is well-known and widely used as an UML tool. Therefore there exist rich integrations into many IDE, plenty of tutorials and source code demos. This helps getting started with yEd as a developer.
The biggest drawback of yEd is, that even though it offers a rich developer API, it is not open source. This makes directly adjusting algorithms, extending functionalities and adding new features impossible. Figure 2.3 shows yEd and an example graph.

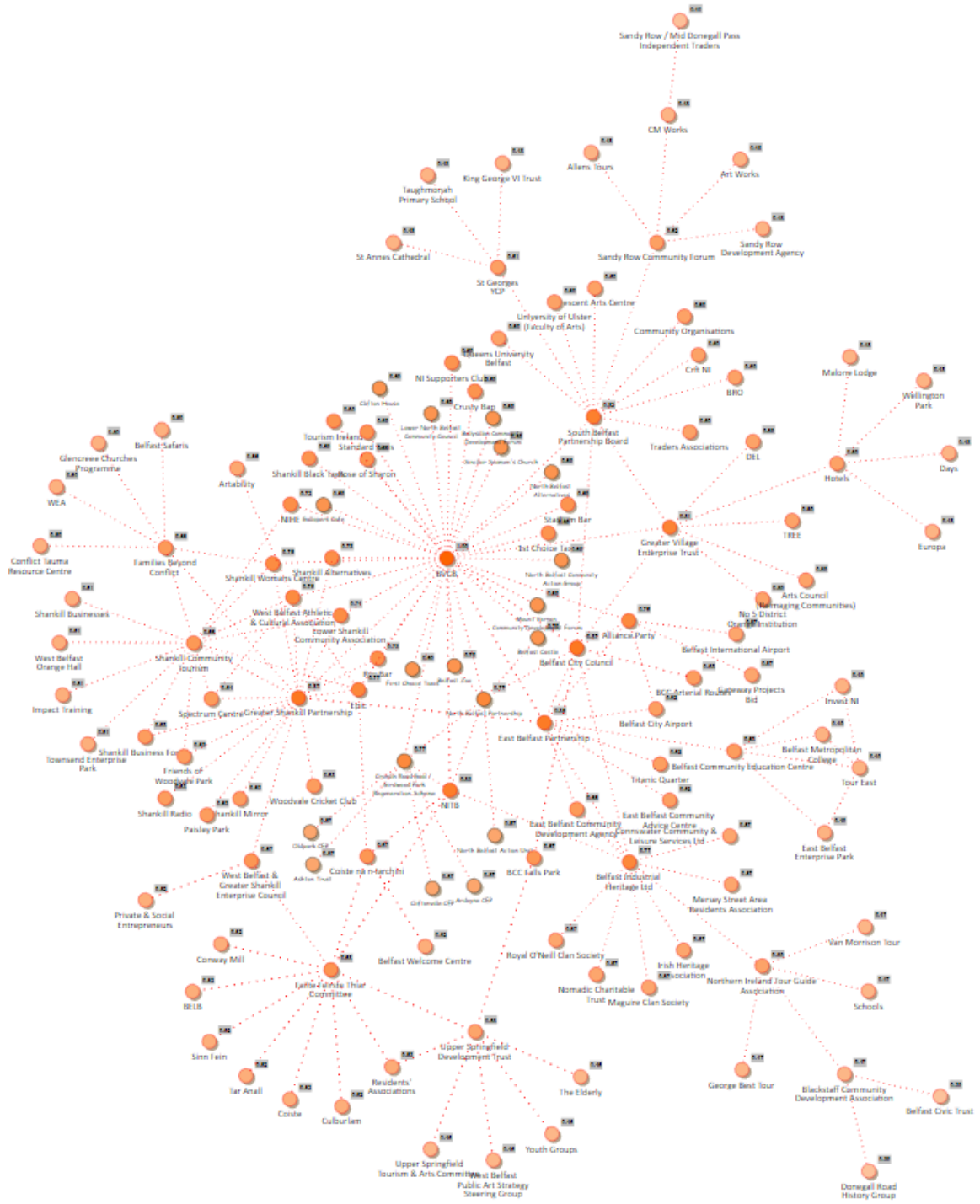


Figure 2.3: Example graph generated with yEd

Cornac The framework Cornac introduced by Perrot et al. [Per18] is no real graph framework but an approach to tackle visualization of huge graphs on an efficient and scalable way. The authors implement it by using a three-fold pipeline.

1. The first pipeline steps consists of an processing layer. This layer preprocesses the graph data on file system using Spark [Zah10] and GraphX [Xin13] and stores the resulting graph in an NoSQL database.

2. Afterwards the results are read from a Java-Webserver based on Apache Tomcat, which also offers an web API for the frontend.

3. The last part is the frontend, which uses WebGL and HTML5 to render the graph and apply layout and bundling algorithms.

The biggest advantage of this approach is its scalability through Spark and Hadoop distributed file system [Bor07]. Therefore the algorithms in backend, which do most of processing work, scale with the amount of machines available. Figure 2.4 shows an example of a graph generated with Cornac.

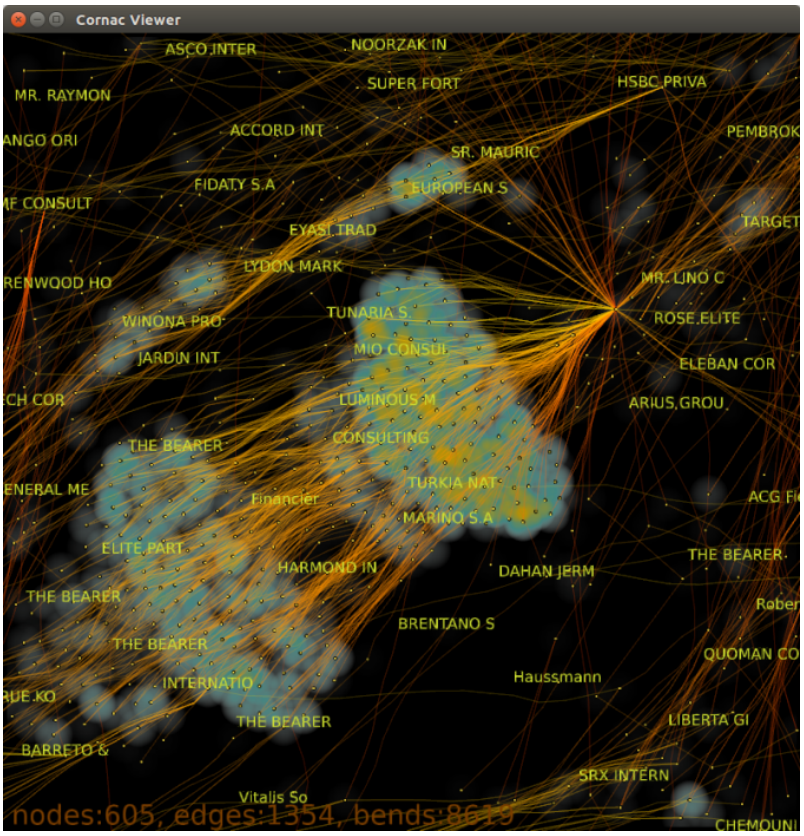


Figure 2.4: Example graph generated with Cornac

The ReGraph Toolkit ReGraph as tool offers several ways of interacting with the graph. Most of them focus on standard interaction ways like zooming in/out and moving nodes/edges [Reg]. However the most outstanding thing is a bunch of additional algorithms, which adjust the size of nodes and edges depending on several quality measurements like PageRank [Pag99] and betweenness centrality [Fre77]. On this way the importance and interconnectedness of nodes are made visually easy to grasp.
The toolkit is implemented using the Javascript library React [Rea]. This allows developers to embed ReGraph in their existing Javascript environment. Javascript is available in all modern browsers. Therefore ReGraph benefits from high cross platform availability. ReGraph also exploits WebGL to deliver high performance on large datasets.
Unfortunately ReGraph is currently only available in an early-access program. Therefore I was not able to examine details and check possible use cases. Figure 2.5 shows ReGraph and an example graph.

Figure 2.5: Example graph generated with ReGraph

ArborJS The open-source Javascript graph visualization library Arbor focuses on providing an efficient, force-directed layout algorithm and several abstractions for graph organization and screen refreshing [Arb]. It reaches this goal by extensively using web workers. Arbor itself allows to choose any visualization approach available in browser. It allows SVG rendering, drawing on canvas, using HTML elements or even WebGL rendering for high performance demands. Due to the missing drawing implementation this library is more of a starting point than an all-round framework.
Besides drawing there are also rich interactions missing. Only these two interactions are possible by default:

- Lazy loading of nodes
- Moving nodes

As a result, this means there is plenty of work left until it allows rich interactions the way Gephi and yEd does. Figure 2.6 shows an example graph of ArborJS.

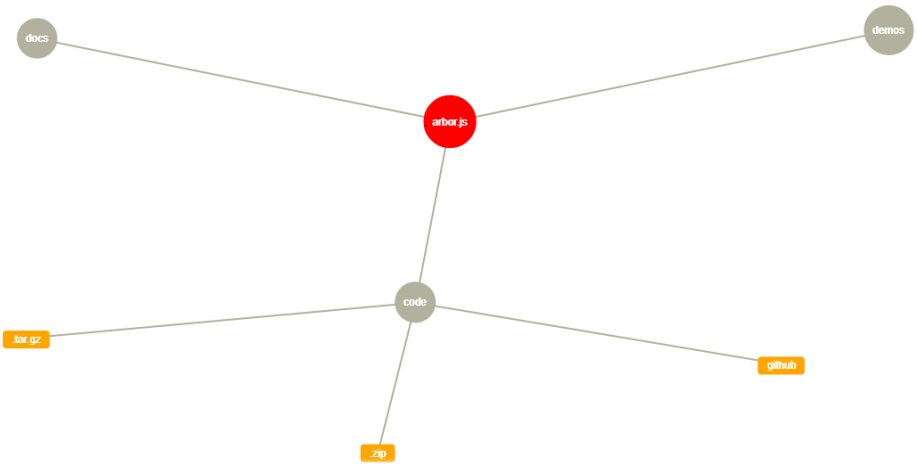


Figure 2.6: Example graph generated with ArborJS

TGView3D TGView3D is an interactive general-purpose graph viewer framework optimized to explore theory graphs in the three dimensional spatial space [RM19]. Since TGView3D was developed in parallel to TGView itself in the some research group, Marcus et al. [RM19] and me exchanged heavily ideas and code. Therefore both tools share similar approaches and algorithms, at least in their base form.
The TGView3D was built using C# and Unity and compiled to a windows binary and a Web-App, which lets the user experience TGView3D in the browser. TGView3D is, to the best of my knowledge, the only graph viewer, which also allows to experience the graphs in Virtual Reality (VR). This framework itself offers many already known standard scenarios of interactions like moving nodes, zooming in and out and navigating through the graph. However the way these interactions are implemented differ from the tools outlined so far, because the main target is VR using the Oculus Rift. The interaction ways have to fit to the natural hand movements possible in VR.
Thats why one hand of the user is a so called ”tractor beam”, which allows to push and attract nodes or even the whole graph by aiming on nodes. Basically every interaction like selecting a node and moving them is implemented using this kind of tractor beam.
The other hand has a virtual user interface attached, which allows to select different algorithms, graphs and settings. Figure 2.6 shows an example graph of TGView3D and the interaction menu.

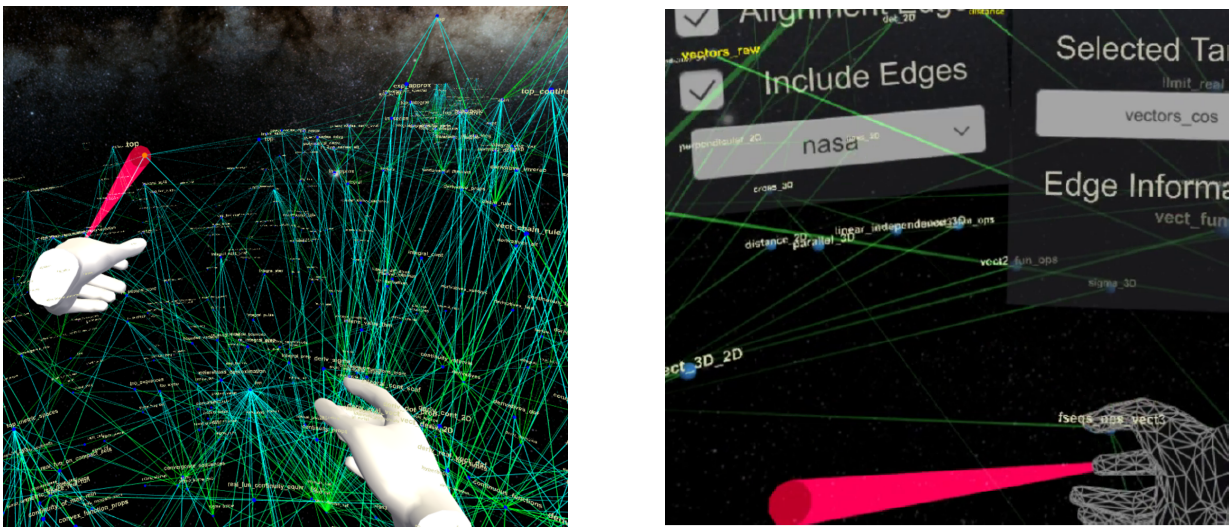


Figure 2.7: Left: Example graph generated with TGView3D; Right: The tractor beam and attached menu

Chapter 3

Preliminaries

In order to better understand theory graphs, it is important that a user can interact extensively with the graphs. However, in general, this important aspect is not sufficiently supported by state-of-the-art solutions. That is the reason why I developed as a university project TGView, a flexible and customizable graph viewer for the OMDoc/MMT ecosystem, to enable this extensive interaction with MMT. TGView is a general purpose graph framework optimized for interaction with Mathhub, MMT and theory graphs. In the following sections I introduce Mathhub, MMT and how the first version of TGView looked like.

3.1 Mathhub and MMT Ecosystem

Mathhub uses MMT/OMDoc documents to model dependencies between theories and models. The MMT language pursues two potentials at the same time, which are

- the automation potential offered by mechanically verifiable representations, as pursued in semi-automated proof assistants like Isabelle
- and the universal applicability offered by a generic meta-language, as pursued in XML-based content markup languages like OMDoc

Internally in MMT this is done by implementing foundation-independence, which avoids a commitment to any particular type theory or logic. So in more detail domain theories, logics and logical frameworks are represented uniformly as MMT theories. These theories are related by the meta-theory relation. In instance LF is the meta-theory of FOL and FOL is the meta-theory of Group.

In more detail MMT organizes its knowledge along four layers, which use URIs for connections between them [MMT]. The list of the levels looks as follows (level one is the lowest level and has no URIs for entries):

1. **Objects:** Mathematical expressions formed from applications, contexts and their bound variables and literals such as floats and strings.
2. **Symbols:** Symbols can be constants (named atomic mathematical objects e.g. axioms, theorems, ...) or structures (instantiations of modules)
3. **Modules:** Semantically relevant top-level declarations. This can be theories which encapsulate mathematical context or theory morphisms, which translate between theories either by representation (views) or by inheritance (structures).
4. **Documents:** Documents group modules together based on their semantical meaning.

Using these four layers of MMT gives the possibility to create theory graphs. Theory graphs in general are formed by nodes representing theories and edges representing the morphisms. Each node and edge is labeled with its MMT URI for interoperation with other tools. Currently MMT supports several types of morphisms, which are

- **Include Morphism:** MMT theories are often built modularly from other theories. Therefore one main declaration is the one of an import/include between two theories. MMT supports unnamed imports and named imports, called includes and structures respectively.
- **Representation Morphism:** Representation morphisms are modeled by implementing ”views” in MMT, which express translations and similar representation theorems.
- **Meta Morphism:** A meta morphism connects a theory with its meta-theory.
- **Alignment Morphism:** The alignment morphism is used when two identifiers describe the same mathematical concept.

MathHub on the other hand is a development environment for active mathematical documents and an archive for flexiformal mathematics [Rab13].
It offers a rich interface for reading, writing, and interacting with mathematical documents and knowledge. The core of the MathHub.info system is an archive for flexiformal mathematical documents and libraries in the OMDoc/MMT format. Content can be authored or archived in the source format of the respective system, is versioned in GIT repositories, and transformed into OMDoc/MMT for machine-support and further into HTML5 for reading and interaction [Ian14].

3.2 First TGView Version

Theory graphs contain multiple types of directed edges and nodes that are rich in information. Due to the fact that existing graph tools do not handle these special graphs sufficiently, I started developing TGView as part of an university project. I constructed the first version of the theory graph Graph Viewer TGView using browser technologies with client side layouting and manipulation. This offers all the flexibility and real-time manipulation of graphs mentioned earlier. Systems based on GraphViz, only allow simple interactions such as e.g. hide/show nodes, grouping/hiding partial graphics or changing the layout. However, then they still require GraphViz as backend, which needs to refresh the whole page from server. Fetching a new page from server, when interacting with the Graph Viewer reduces speed and ease of interaction dramatically. In addition, the server-side layout is subject to scalability issues and does not support responsive UI design well.

By implementing TGView fully client-side in the browser, it solves these challenges. A fully client-side solution enables TGView to scale almost infinitely and to let the user interact with graphs in a reactive way. The HTML5-based implementation allows users to interact with theory graphs using almost any device, because almost all modern devices deliver browsers, which are capable of executing Javascript. Other Graph layout libraries e.g. using Graph libraries based on Java or Flash would require browser plug-ins, which are only available for a limited period of time or limited amount of devices TGView has a graphical user interface that allows all actions to be performed at the click of a mouse without requesting any data from backend, except for lazy-loading nodes and the initial graph data. The heart of TGView is the graph library vis.js, a graph drawing library, which offers many native interactions such as node clustering, different layouts (hierarchical vs. forces driven), adding and deleting nodes/edges and efficient rendering of large graphics.

To give the user access to the math archives on MathHub, I use a tree-structured menu, where we can select theories for drawing. I decided to use the Javascript library jsTree with its rich API for constructing the tree-menu. Figure 3.1 shows the version of TGView after completing the university project.

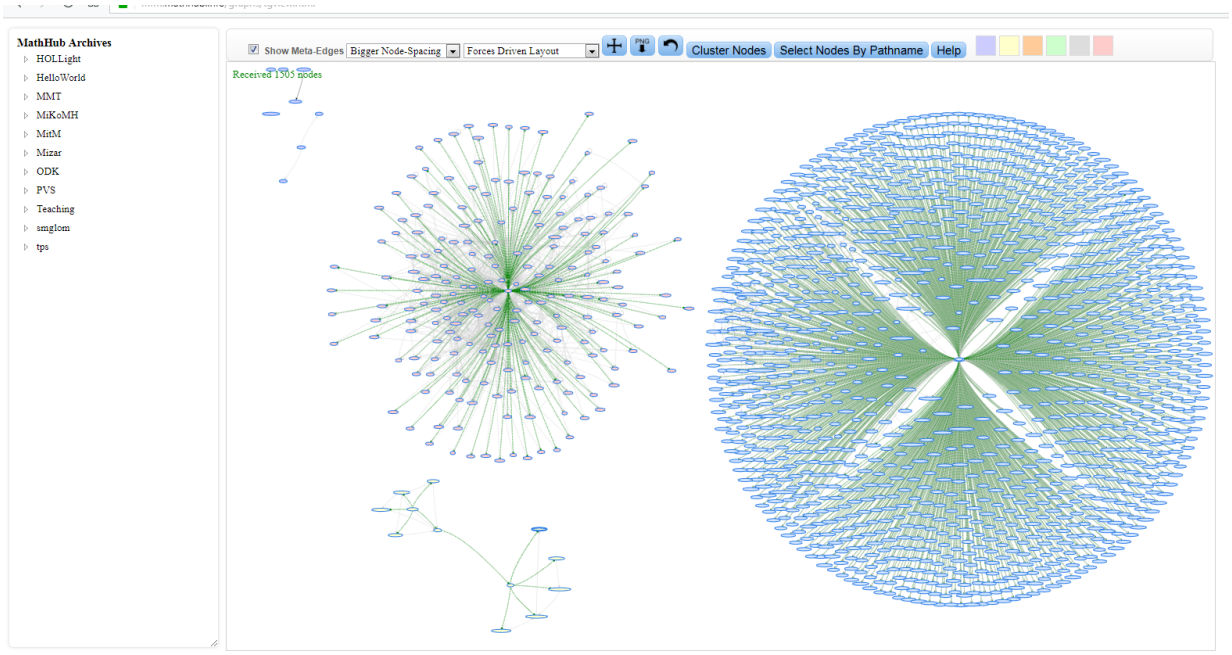


Figure 3.1: First TGView version

TGView communicates with MMT based on JSON and AJAX-Requests using HTML5. Initially TGView sends a graph request to the server. The server sends the requested graph data to browser. This data is processed by TGView, which then updates the canvas and HTML-DOM. TGView only needs to communicate with the server, when the user switches graphs, extends graphs or loads child nodes. To keep our system modular I divide functionality between client – which is responsible for graph layout and user interaction – and the server, which has the task of providing graph data in JSON. Figure 3.2 shows the architecture behind the communication of TGView and the server.

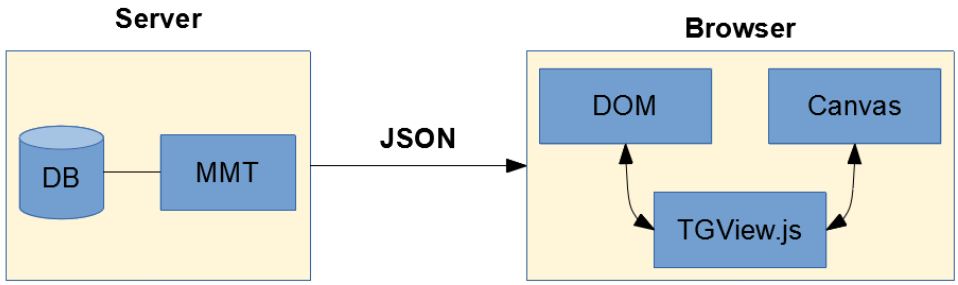


Figure 3.2: Architecture of TGView communication with server

3.3 Libraries for Graph-Drawing in the Browser

For graph visualization, TGView uses Vis.JS and typescript. Typescript is an extension of vanilla Javascript, which adds types similar to Java and C++ types to code. This provides compile time type safety for Javascript code. It makes the code more robust regarding user errors, because now the compiler can check whether types are correctly used. Besides this compile-time-checking, types also help to understand the code better than without types. Therefore types support the documentation of code. For Javascript there exist only a few other graph drawing libraries. In Table 3.1 I listed the features beneficial for TGView based on the different libraries. The features are as follows:

- **Graph-Functions:** For our original purpose, which was experimentation, it was important to have a starting point, which has plenty of predefined graph functionalities like clustering, layouting, manipulating nodes and edges and interacting with the graph and its components.
- **Easy Integration:** As the main goal of TGView is to integrate it into Mathhub, it is really beneficial, when the graph library can be easily integrated in other sites by using Javascript.
- **Export SVG:** For papers and other works it is good to have some high quality images to share. A library which supports SVG exports therefore is very beneficial to the whole research task.
- **WebGL:** Using WebGL speed up the whole network drawing a lot and allows huge graphs to view (up to ten thousands of nodes), whereby canvas based implementation already starts struggling with a few thousand nodes.
- **License:** Besides all technical requirements and wishes, there are also licensing requirements. In general the library should be possible to use in the already existing environment of licenses in MMT.

Library	Graph-Functions	Easy Integration	Export SVG	WebGL	License
ThreeJS	No	Yes	Yes	Yes	MIT
RAWGraphs	Yes	No	No	Yes	Apache 2.0
D3.js	No	Yes	Yes	Yes	BSD 3-Clause
Vis.js	Yes	Yes	No	No	Apache 2.0

Table 3.1: Core functions TGView benefits from; Red: Not met requirements; Green: Met requirements

Chapter 4

MMT/OMDoc Graph Exploration Tool Requirements

While working with TGView I condensed several requirements, which should be fulfilled for a MMT/OMDoc exploration tool. The goal of this tool is to gain insights into Mathhub and the MMT/OMDoc ecosystem and retrieve information regarding the theories, its structures and the connection between them. To reach this goal several requirements have to be met or at least should be kept in mind. The requirements are defined on a general way. Therefore they hold true for any MMT/OMDoc exploration tool, but especially for TGView. The requirements are deduced from general use cases, interviews and observations of users interacting with TGView.

Understanding Structure of Theory Graphs When asking the KWARC users for the main requirement, TGView needs I experienced always the same answer:
”The tool should help to understand the structure of theory graphs”
This requirement also fits to my initial goal of TGView, which is to understand the theory graphs on a structural way. Therefore one of the main requirements is to enable the user to understand what theories are actually include which theories and make the hierarchies visually recognizable. Furthermore I noticed, that sometimes the user does not solely want to understand the dependencies, but also want to understand the information flow and deduce new theories and knowledge. Therefore the tool should also assist the user by deducing knowledge.

Dealing with Complexity Another requirement, which I deduce from the requirement before is, that the tool should not overwhelm the user with details. But it should still allow him to dive into details, if wanted. Ignoring this requirement would make it tough to fulfill the requirement

of understanding theory graphs. In general the user should be able to switch between high-level information and low-level information overview. This would allow an intuitive, but powerful way of experiencing all libraries and its dependencies without loosing focus. Therefore the user should be able to filter the library for information intelligently.

Understanding Mathhub After I implemented a prototype of TGView it became quickly obvious by user feedback, that the user does not just want to understand the theory graphs, but also how they are embedded in the whole Mathhub ecosystem. Therefore it is not solely important to understand the structure behind the mathematical libraries and theories, but also to understand the structure of Mathhub as archiving software. The tool should allow to browse through the archives and request information belonging to a given archive, library and theory. Mathhub also allows to view the theories and math expressions in browser by using HTML5. These things should be also able to see or be incorporated into the exploration.

Manipulation of Theories As I was thinking of several use cases, I also thought about, whether someone would need the possibility to manipulate theories directly in the theory graph viewer tool. Since the resonance, when asking this question to the KWARC group users, was very positive, I made it to be a requirement. Because whenever exploring theories at some point there comes the wish to add, delete and edit theories itself in real-time to fix wrong connections, extend the theory or just to build another theory based on some old theories. In best case the theory would be created as OMDoc/MMT file and sent to the server. This supports the user when writing new theories and libraries.

Collaboration After the first version of TGView was implemented, TGView itself got also known to other universities. However the first version had a really big problem. There was no way to collaborate on layouting or structuring the graph. Even though the users used screen-shots and printed graphs for sharing, this was not comfortable. That is why another important use case of an MMT/Mathhub exploration tool is the editing and subsequent sharing of the results. This ensures that even time-consuming or special editing steps can be divided among several people. On this way sharing knowledge or even pair-programming is possible. This helps especially beginners to get used to the graph exploration tool. However also experts benefit from a real-time collaboration as on this way discussing while adjusting libraries and theories becomes easily possible even when working remote.

Highly flexible Framework As stated earlier, there are plenty of use cases, which naturally can be best experienced using graphs. Even though in this work the focus lies on a theory graph exploration tool, keeping the framework itself flexible helps to extend it to other uses cases and even completely new research areas. An exploration tool, through which the user gains information and knowledge, may also be usable for other tasks. Therefore also other disciplines should profit from this research done and should be able to use the graph exploration tool, without too much of code rewriting.

Good General User Experience Over time I noticed, that when the whole team focuses only on the functional requirements, the non-functional are getting worse. This especially happened to TGView, I focused in the beginning mainly on implementing as many use cases as possible but did not take care of general user experience. Until I received a question, which was: ”Why do I have to at first do these 5 clicks until I am able to hide nodes? I also cannot remember which node types belong to which color”
When I heard this complain, I just noticed, that one really important requirement is a good general user experience. A good general user experience is very important to ease working with the tool. It helps the user to focus on their main goals. So there is no distraction from the essential, which are the requirements already outlined.

Chapter 5

TGView Evaluation

In the next chapter TGView is checked against the seven requirements defined earlier. However before I start evaluating the current version of TGView, I will outline the actual differences between the first version of TGView and the current version. Figure 5.1 shows the final first version of TGView. Figure 5.2 shows the current version of TGView.
The main differences lie in user interface design and the algorithms behind TGView.
The first version only had three layout algorithms (forces driven, strict hierarchical and semi hierarchical). TGView now has five different layout algorithms, which will be explained in more detail later on.
Another important change was to give the graph more space by making the two menus collapsible.
I also extended the hide/show nodes checkbox, so that now every node and edge type can be hidden. I added different possibilities of sharing and uploading graphs to allow collaboration on remote.
Additionally the feature of caging nodes, which groups the selected nodes together by underlaying them with a colored rectangle was added.
I also implemented a dynamic legend for the graph to always remember what which kind of node/edges means.
Furthermore these small changes were done:

- Added additional levels of node spacing (before there were five now there are seven for more fine granular selection)
- Added possibility to not just undo things but also redo them
- Added automatic cluster algorithms
- Expanded help and documentation

- Added button for showing all hidden nodes again
- Added possibility to lazy load child nodes
- Converted TGView to typescript

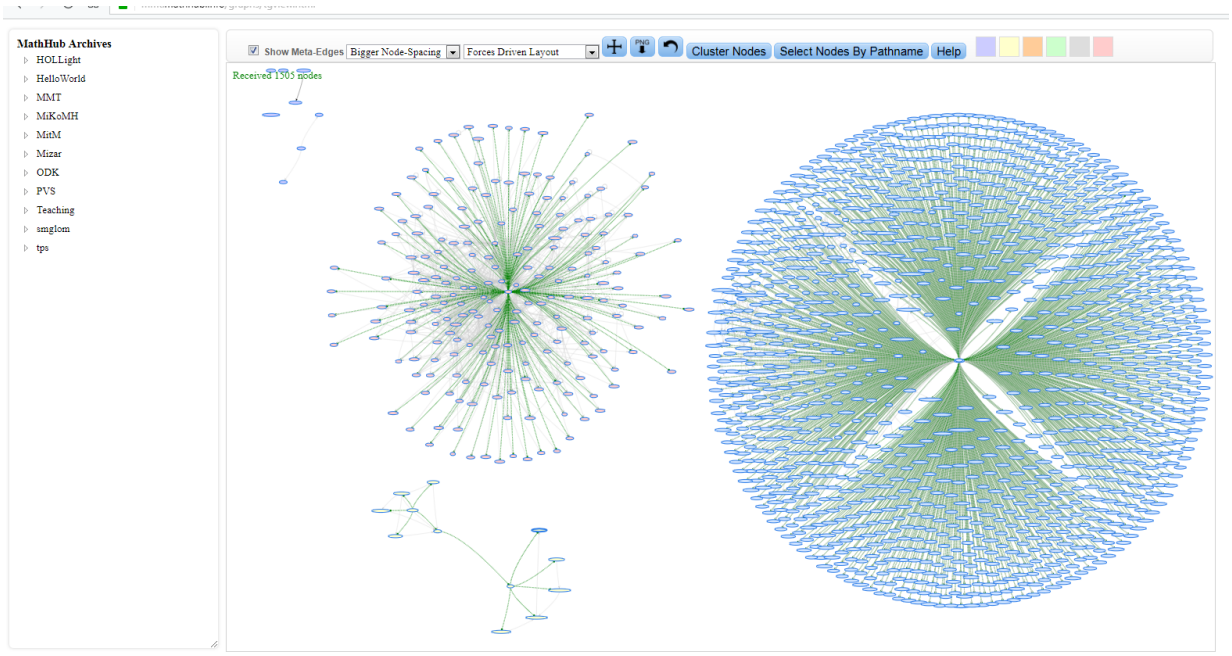


Figure 5.1: First final version of TGView

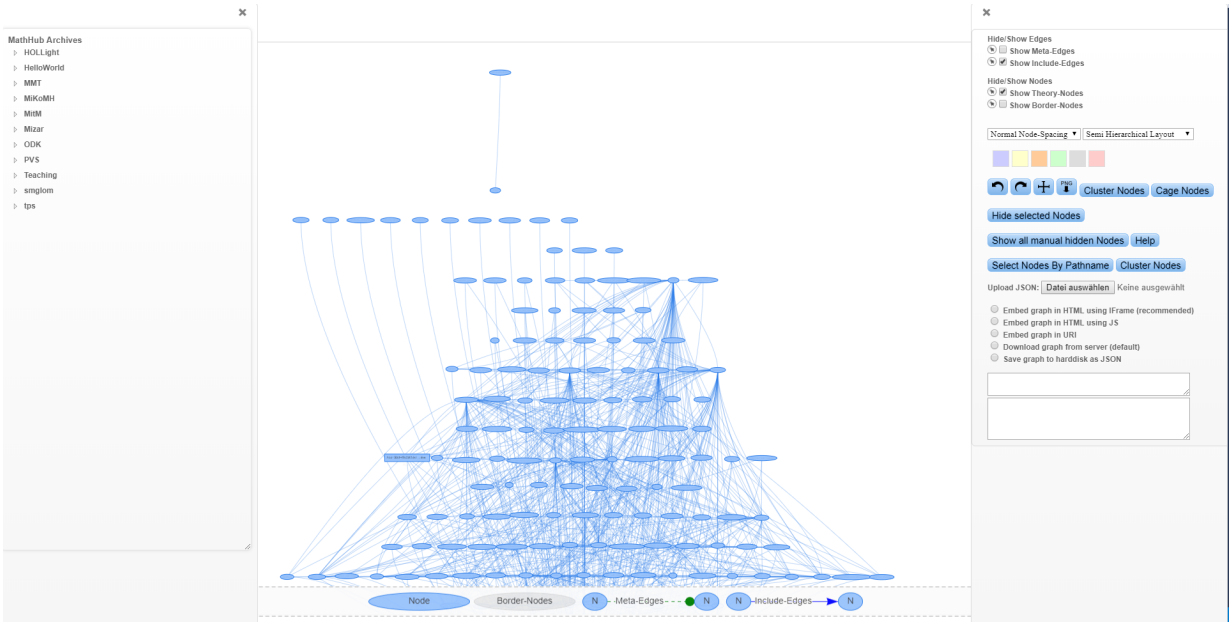


Figure 5.2: Current version of TGView

Chapter 6

Understanding Structure of Theory Graphs

TGView offers two main ways to experience the structure of theories.

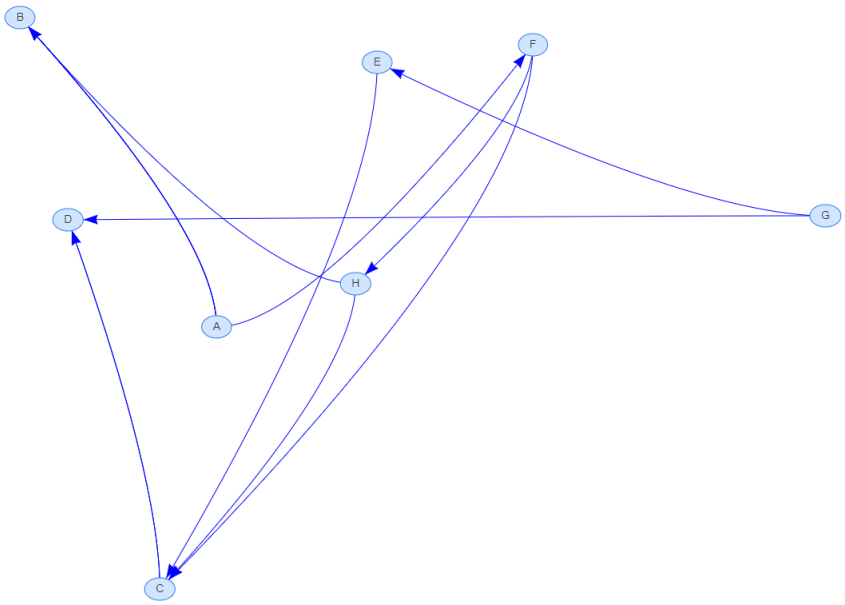
- **Graph Layouting:** TGView tries to layout the graph on such a way that the details can be seen, but the graph still feels not overwhelming.
- **Clustering:** The second approach tries to cluster by using proximity and color to highlight relations and dependencies.

6.1 Graph Layouting

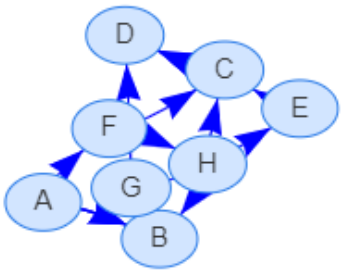
The goal of graph layout is to display graphs in such a way that the information the user tries to read out of the graph is as simple and visual as possible. Graph layouting tries to optimize the geometric arrangement of the nodes and edges of a graph in order to maximize user-friendliness and comprehensibility at the highest possible information density. The optimization problem becomes more acute when the graph changes over time by adding and deleting edges (dynamic graphs), while preserving the user's mental map as much as possible.
Different layout algorithms prepare graphs differently, so that usually different information can be obtained. For this reason, TGView has five different layout algorithms that highlight all other types of information and make it accessible to the user. In this way, the user can cover different use cases without leaving TGView.

In general, the usability and aesthetics of graphs are captured by the following quality measurements according to Purchase et al. [Pur97] (see Figure 6.1 for examples of bad layouts):

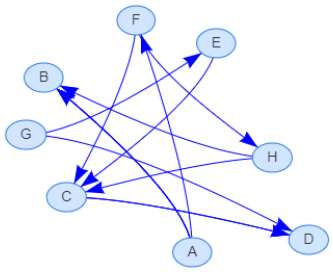
- As few edges and nodes as possible overlap.
- The area of the resulting graph corresponds to the smallest possible bounding box for the problem. Small graphs are preferred because local dependencies are clearly visible.
- Since humans prefer symmetries, the graph should also be somehow symmetric, if reasonable.
- Edges and nodes should be kept relatively simple visually so that the focus is not unnecessarily distracted from the essential, which is understanding the structure and topology of theories and their dependencies.
- The length of edges should be as minimal as possible, but should not fall below a certain threshold, as otherwise a chaotic structure may be formed, where edges are no longer clearly visually assignable to nodes.



a) Too big node spacing



b) Too small node spacing



c) Too many crossing edges

Figure 6.1: Examples of badly layouted graphs

6.1.1 Strict Hierachical Layout

The strictly hierarchical layout is a layout algorithm that is implemented by default in VisJS, which I did not adjust on any way. Nodes and edges are arranged in such a way that all successors of a node are placed in nodelevel+1 and all predecessors are placed in nodelevel-1. The graph

A –> B
A –> C
B –> C
C –> D

creates the graph on the left in Figure 6.2. Because strict hierarchies are assumed and therefore no dependencies between nodes on the same level are taken into account, unintuitive graphs can occur if the graph itself is not strictly hierarchical. Additionally the algorithm implemented in VisJS generates strange graphs even though they are strictly hierarchical (see Figure 6.2 for an example). Due to these limitations and strange behaviors, I had to implement several other algorithms on my own.

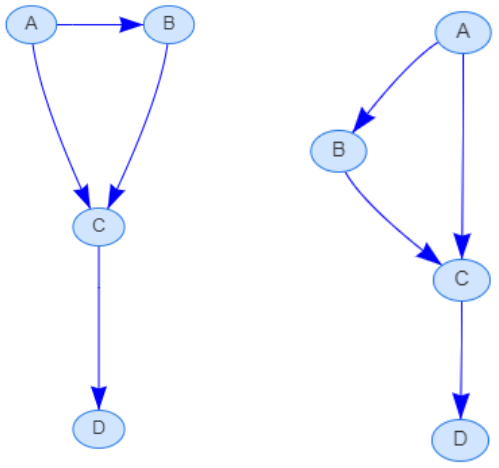


Figure 6.2: Left: Generated graph; Right: Actually expected graph

6.1.2 Semi-Hierachical Layout

To tackle the problems of strictly hierarchical layouts in general, I introduce the semi-hierarchical layout, which tries to highlight the hierarchical structure but does not expect a perfectly hierarchical graph. The algorithm works by first finding the nodes with the most outgoing and most ingoing edges. These nodes are expected to be the very top nodes, which mainly include other nodes and the most bottom nodes, which are only included by other nodes. After finding the nodes of the first level and the nodes of the last level, the algorithm calculates the level of the nodes lying between the two initially given node types. This level is calculated by sampling random paths from the top nodes to the bottom nodes, whereby only directed connections are used for traversing. If undirected connections occur in the graph, I allow the algorithm to traverse the connection, too. However undirected edges are chosen with low probability to always focus on directed connections and keep the hierarchical information visible. Sampling is used to approximate a good layout while keeping the waiting time for the user low. An optimal algorithm would calculate every path from every top node to every bottom. After traversing paths, every node on paths get assigned a number depending on the amount of nodes in path until this node. In a path looking like this:

A->B->C->D

A would become number 0, B number 1, C number 2 and D number 3. If a node occurs in multiple paths, the median of the assigned numbers is used. After all levels of all nodes are calculated, the algorithm rearranges the nodes with their corresponding level on such a way, that no level is unused between maximum level and minimum level. This is done by simply mapping the levels calculated to ascending levels beginning from zero. The levels are in a last step multiplied with a given spacing amount for vertical spacing in pixels. Figure 6.3 shows a graph layouted with the described semi-hierarchical approach.

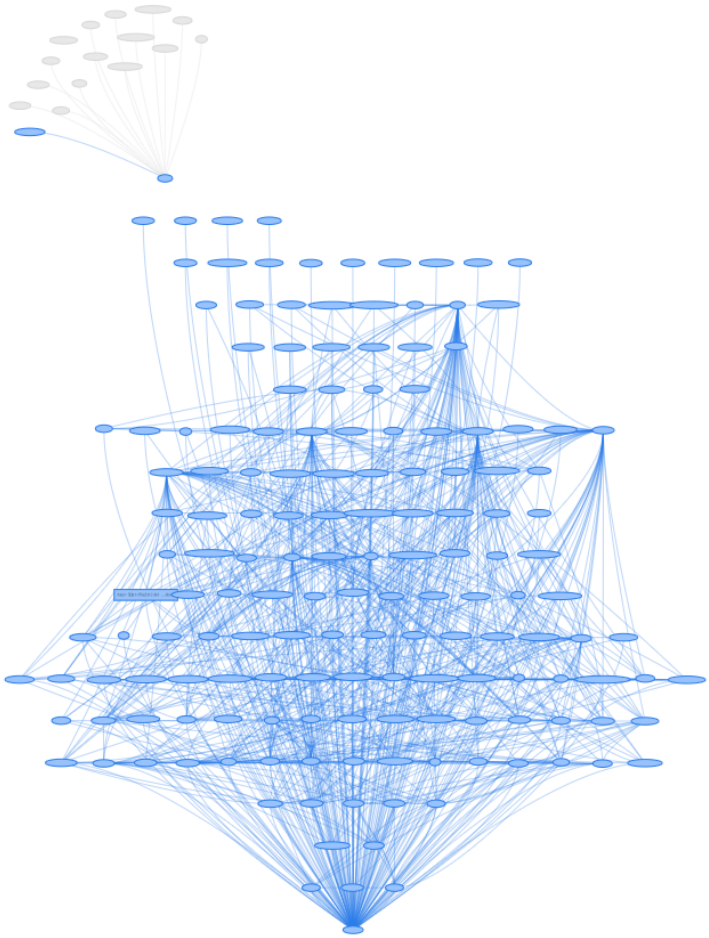


Figure 6.3: Semi-Hierachical layouted graph

6.1.3 Forces Driven Layouting

Hierarchical graphs can be nicely viewed with hierarchical approaches. However graphs without any global hierarchic ordering needs to be layouted differently to extract useful information. The group of forces driven algorithms handles graphs, which lack any global hierarchical groups.
The basic idea behind forces driven approaches is to define some spring-like attractive forces per node similar to Hooke's law and repulsive forces per node based on Coulomb's law for charged particles. The attractive forces keep the graph together and brings connected nodes near to each other, whereby repulsive forces drive all nodes away from each other. In general the attractive and repulsive forces grow/decrease with their proximity between several nodes. These forces together form very natural layouts and highlights, by design, communities by grouping them together.
In general forces driven algorithms offer plenty of advantages compared to classic layout algorithms, which are:

- Forces driven algorithms are quite easy and fast to implement. The algorithm in TGView consists of exactly 120 lines of code, where our semi-hierarchical algorithm consist of 250 lines of code.
- The family of forces driven algorithms in in general very intuitive, because they are based on real world physics.
- They approximate an optimal solution for uniform edge length, symmetry of graph and uniform node distribution well.

TGView implements the forces based algorithm known as "Spring electrical model with adaptive cooling" [Hu05]. Forces driven algorithms in general do not preserve hierarchies. In Figure 6.4 a layouted model path diagram (MPD) of the stationary van Roosbroeck system is shown. A model path diagram is like a flow-chart that uses edges to show direct and indirect causal links between variables and models. As MPDs are not structured on any hierarchical way, they highly benefit from forces driven algorithms.

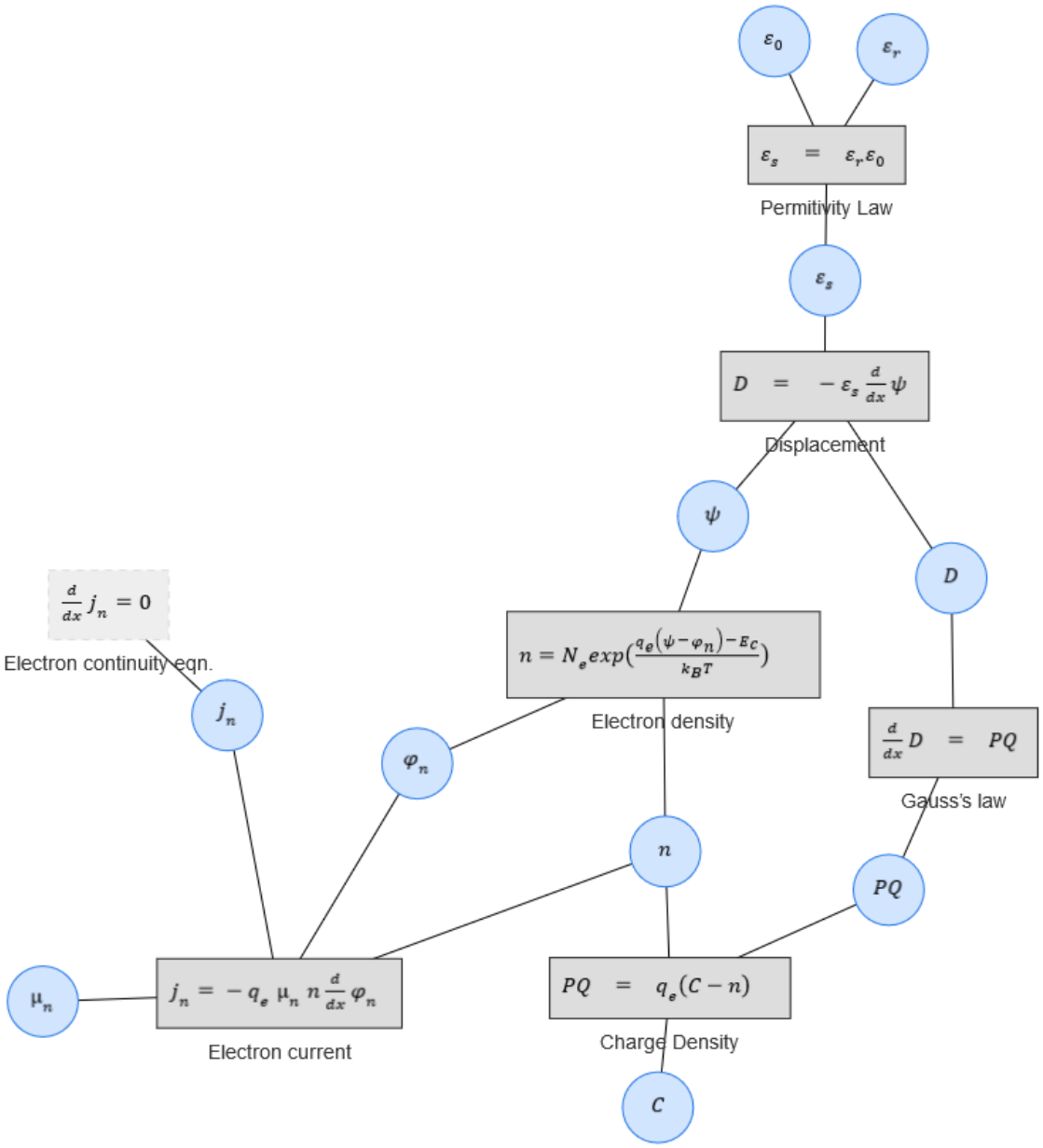


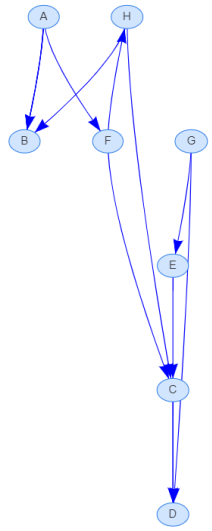
Figure 6.4: Forces driven layouted MPD graph

6.1.4 Water Driven Layouting

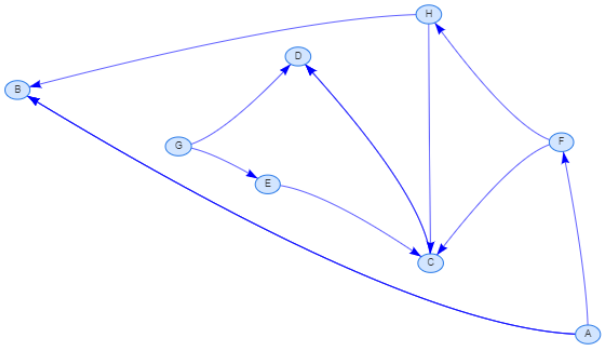
The Forces-Driven Algorithm hides many hierarchical dependencies, which could be easily incorporated into the graph layout. On the other hand, the strictly hierarchical layout and the semi-hierarchical layout destroy the ”natural structure” behind the graphs. For this reason, TGView has a compromise between all of these algorithms, the Water-Driven Layout. The name comes from the basic idea that the nodes in the graph are treated like balloons and weights of lead in the water. The balloon nodes pull the graph upwards, the lead weights pull the graph downwards. In addition, all nodes that are not connected to each other repel each other and all nodes that are directly connected to each other attract each other. The buoyancy or weight of a node can be calculated on different ways, which are:

- By counting the number of incoming and outgoing edges during initialization
- By calculating the betweenness centrality for every node, which describes the centrality of a node given the shortest paths from every node to every other node [Fre77]. The more shortest paths contain this given node, the higher the node will be in the hierarchy.
- By calculating the hub size, which puts the nodes with the highest amount of edges on the top and nodes with the lowest amount at the bottom [Kle99]

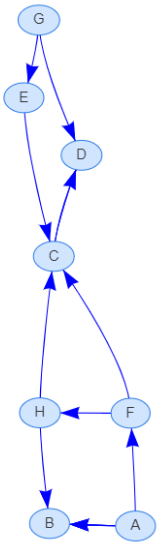
I implemented the algorithm of calculating initial weights by counting incoming and outgoing edges during initialization as this method runs fast and is quickly implemented. Each node with only incoming edges has the weight 1 and each node with only outgoing edges has the buoyancy 1 (= weight: -1). In this way, the corner nodes of the hierarchy are already defined. So that the remaining nodes are now also arranged hierarchically, the weight of the nodes is evenly distributed among the direct neighbors. This means that a node with weight 1 and two neighbors passes the weight 0.5 on to each neighbor. If there were 5 neighbors, $\frac{1}{5}$ of the weight would be passed on. If several nodes have the same neighbor, the average of all neighbor weights is taken as the final weight. The weight of the nodes initialized with 1 or -1 must not be changed.
This procedure is carried out iteratively for all nodes until either there is no more change in the weights or a specified maximum number of iterations has been reached. At the end, each node has a weight between 1 and -1. All nodes with a positive weight strive downwards and all nodes with a negative weight strive upwards. Because the nodes additionally attract and repel each other, the graph cannot expand infinitely, but converges towards a final layout at some point (see Figure 6.5 for comparison of layouts with the water driven layout).



a)
Strictly
Hierarchical
Layout



b)
Forces-Driven Layout



c)
Water-Driven
Layout

Figure 6.5: Graph generated with different layout algorithms

6.1.5 Manual Forces-Driven Layouting

The described algorithms are all great for handling the whole graph at once and extract the maximum of information from the graph. However there are use-cases, which needs to hide all nodes and edges except all nodes/edges connected to one selected node. Due to this requirement TGView has another layout algorithm, which lets the user traverse through the graph by clicking on nodes. Figure 6.6 compares the whole graph with water-driven layout to the graph after selecting one node in manual focus layout.

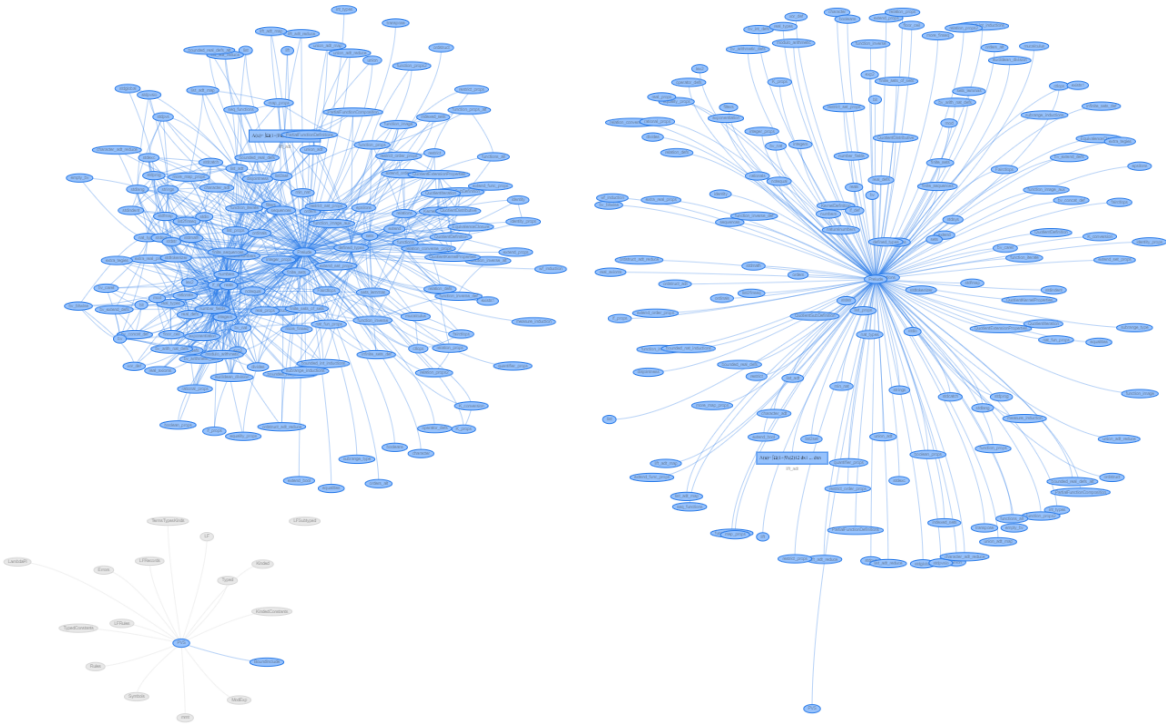


Figure 6.6: Left: Original graph; Right: After applying manual focus for the center node

6.2 Node Coloring

Another tool, besides the spatial arrangement of nodes and the graph, to understand the structure of theories is color. Color enables the user to highlight single nodes but also to group several nodes together by assigning the same color to these nodes.

6.2.1 Cluster based Node Coloring

Even though manually coloring nodes is nice to have, an automatic way of clustering is more comfortable and can deal with huge graphs in seconds. TGView implements a cluster algorithm, which is based on minimizing modularity of graphs [Bra08]. Modularity measures the strength of division of a graph into the given clusters. In general modularity is a measure for how interconnected are nodes in a cluster compared to the interconnectedness between clusters. Modularity is defined as the fraction of the edges that fall within the given cluster minus the expected fraction, if edges were randomly distributed.
So given an amount of n nodes and their corresponding clusters and edges, one can calculate the modularity of a graph as follows:

$$Q = \sum_{n_1 \in \text{nodes}} \sum_{n_2 \in \text{nodes}} \text{cluster}(n_1, n_2) \cdot (\text{linked}(n_1, n_2) - \frac{\text{links}(n_1) \cdot \text{links}(n_2)}{2m}) \tag{6.1}$$

$$\text{cluster}(n_1, n_2) = \begin{cases} 1, & \text{if } n_1 \text{ and } n_2 \text{ are in the same cluster and } n_1 \neq n_2 \\ 0, & \text{otherwise} \end{cases}$$

$$\text{linked}(n_1, n_2) = \begin{cases} 1, & \text{if } n_1 \text{ and } n_2 \text{ are connected} \\ 0, & \text{otherwise} \end{cases}$$

Where
$\text{links}(n)$ equals the amount of connected nodes to node n and
m equals the total amount of edges in graph.

By maximizing this modularity, the graph is separated into several clusters. TGView uses an evolutionary algorithm to approximate the optimal solution in a few seconds. After assigning

each node to a cluster TGView recalculates the forces driven layouts (if selected) and assign to nodes of the same cluster higher attractive forces. This highlights communities. Additionally TGView colorizes nodes of the same cluster with the same color (see Figure 6.7 for an example).

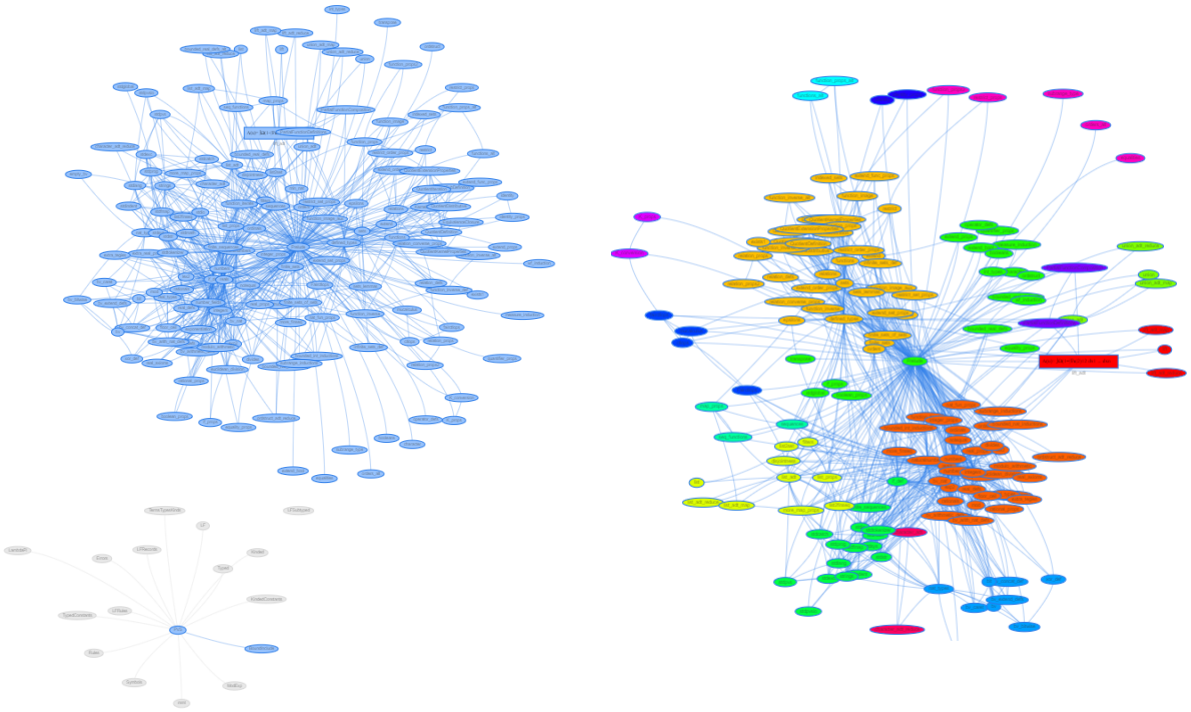


Figure 6.7: Left: Original graph; Right: Graph after modularity based clustering

Chapter 7

Understanding Mathhub

TGView offer the possibility to browse through archives in Mathhub in the sidemenu (see Figure 7.1 for an example). The archives can be expanded on left-click and on right click the graph types can be selected. On this way a Mathhub archive can be loaded with different use cases in mind. Currently TGView supports six different graph types, which are similar in their basic structure and graph algorithms, but use different node and edge styles.

- **Archive Graphs:** Shows all modules in given archive (see Figure 7.5 for HOLLight archive as example)
- **Theory Graphs:** Shows the immediate periphery (direct dependencies) of selected theory (see Figure 7.2 for PVS theory as example)
- **Document Graphs:** Shows all theories declared in given document (looks visually similar to archive graphs)
- **Path Graphs:** Shows all modules residing in a given path (looks visually similar to archive graphs)
- **MPDs:** Shows special graphs with respect to models (see Figure 7.3 for the stationary van Roosbroeck system MPD as example). A model path diagram is similar to a flow-chart that uses edges to show direct and indirect causal links between variables and models.
- **Argumentation Graphs:** Shows graphs representing an argumentation chain and their attacks (see Figure 7.4 for Tweety argumentation graph as example)

Additionally one selected theory can be examined in more detail by right-clicking on the corresponding node in the graph. This opens a context menu and allows rich interactions (see Figure 7.1

for an example).

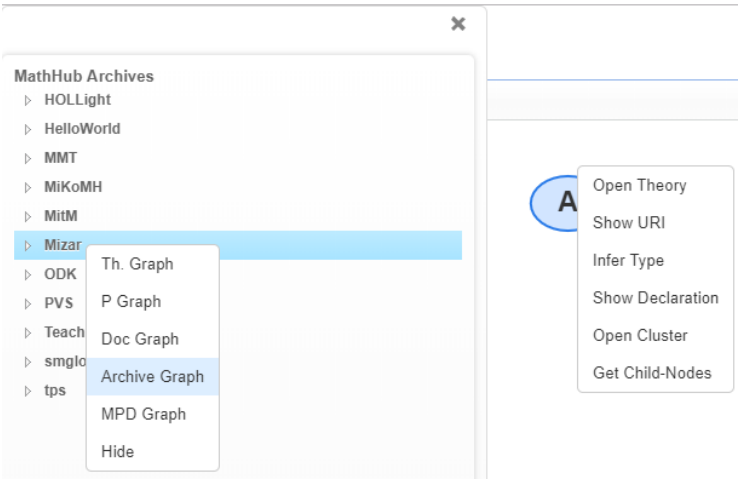


Figure 7.1: Side menu and right click on node in TGView

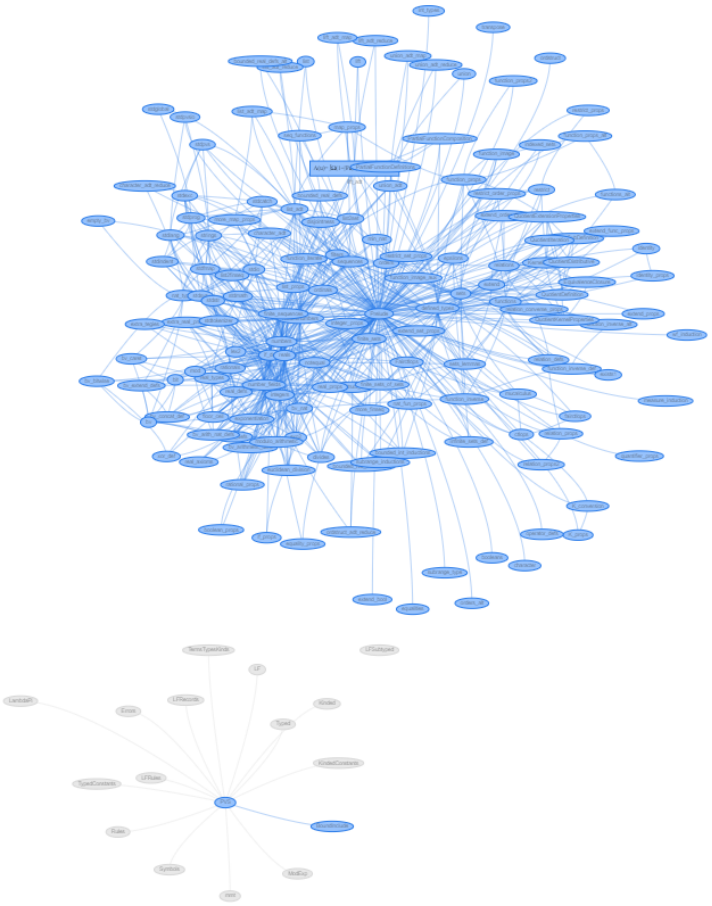


Figure 7.2: PVS Theory graph

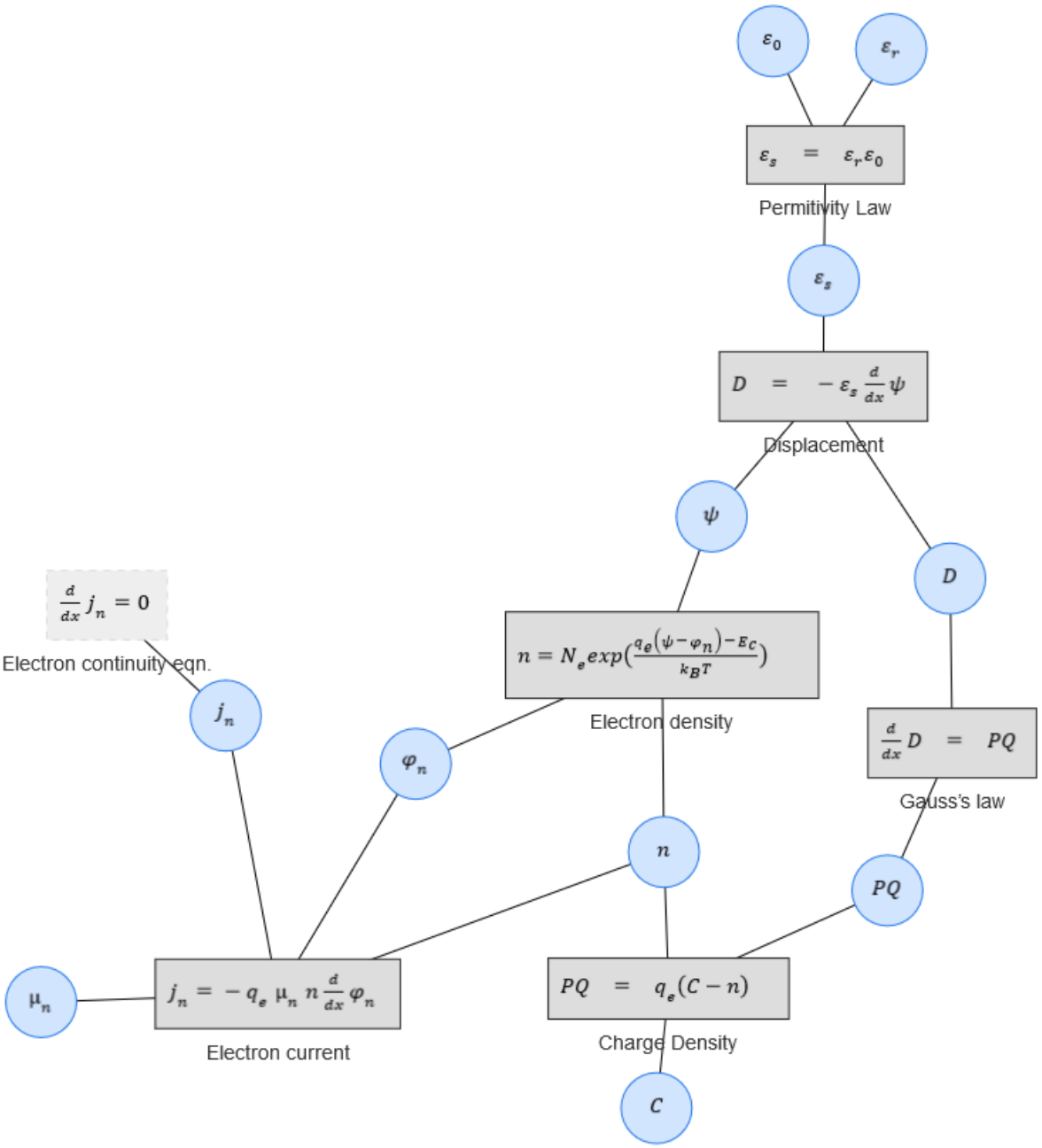


Figure 7.3: Stationary van Roosbroeck system MPD graph

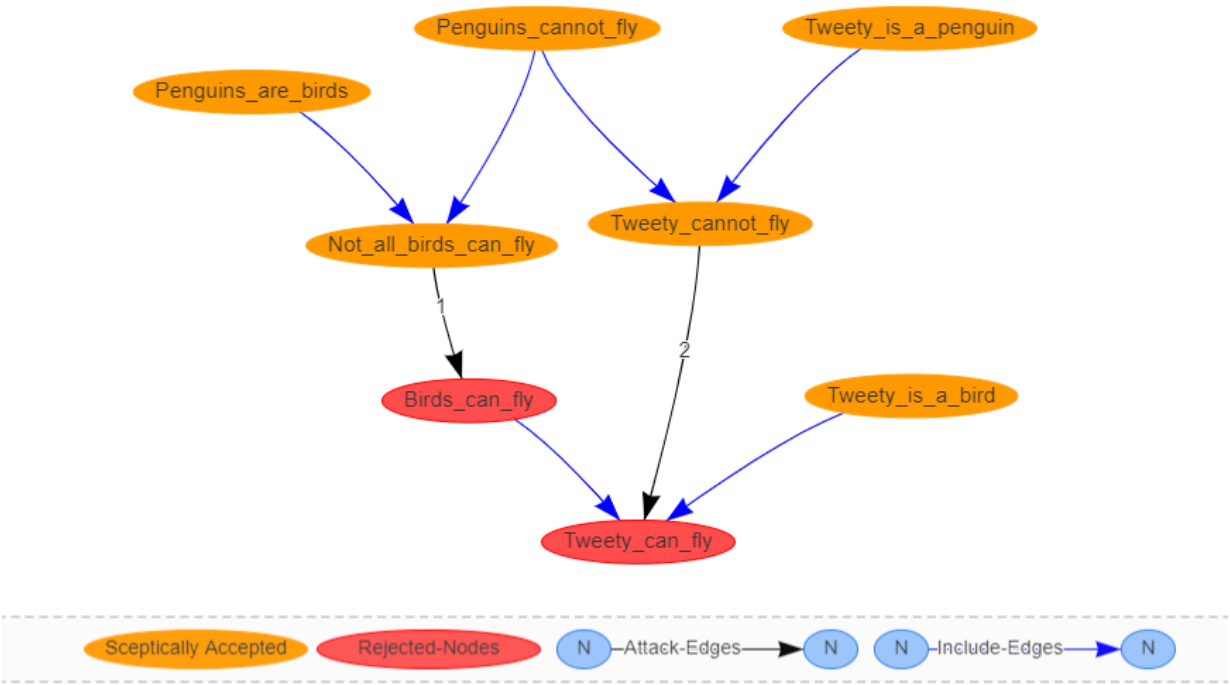


Figure 7.4: Argumentation graph with legend explaining nodes and edges

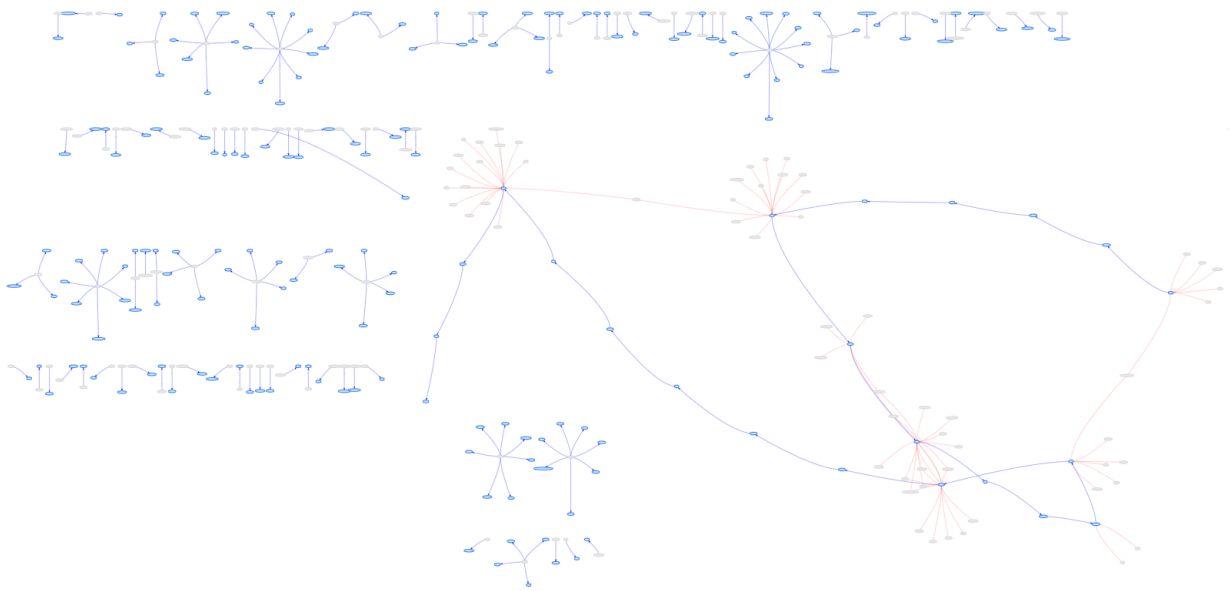


Figure 7.5: HOLLight Archive graph

Chapter 8

Dealing with Complexity

A challenge when dealing with big theories and their corresponding graphs is the amount of information visible at the same time. On one hand the user may want to see the overall structure of a graph but on the other hand he also wants to see details, when needed. Showing the whole graph with all its edges and nodes and the corresponding labels assigned to the objects, may overwhelm the user. Therefore TGView allows the user to filter the graph intelligently. On this way TGView is able to show very detailed dependencies but also provides a feeling for the overall graph structure. This filtering in TGView can be done on several ways, which include clustering, zooming in/out and caging nodes.

8.1 Zooming

The most intuitive way to interact with a graph and filter the amount of simultaneously shown information is by zooming into the graph and out of the graph. When zooming in, the user is able to focus on a small amount of nodes. While when zooming out the whole structure of the graph becomes visible. Additionally to zooming in and out, Vis.JS hides the labels of nodes and edges when a certain zoom out level is reached. This fights clutter of the graph produced by too many too small labels, which would disturb the edges and nodes overall structure (see Figure 8.1 for comparison of zoomed out and in graph parts).

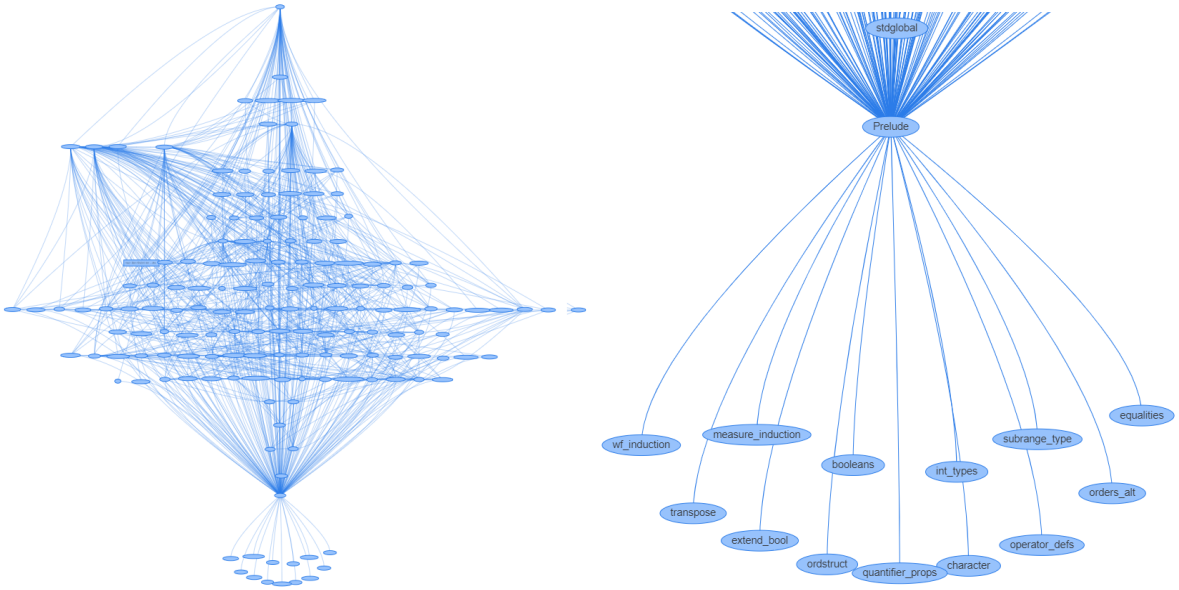


Figure 8.1: Left: Zoomed out graph; Right: Zoomed in graph

8.2 Caging Nodes

Whenever nodes belong together, there is a need to also group them together by proximity. This grouping by proximity can be done using manual drag and drop actions in TGView. However after a several nodes were arranged into a specific corner, it would be great to also be able to select and move every node belonging to the same group simultaneously. TGView provides for this use case ”node caging”. Node caging highlights all nodes belonging to one group by underlaying a colored semi-transparent area. By selecting this area all nodes in this ”cage” can be moved at the same time. These cages will also guarantee that nodes belonging together will not be split apart (see Figure 8.2 for an example of this functionality).

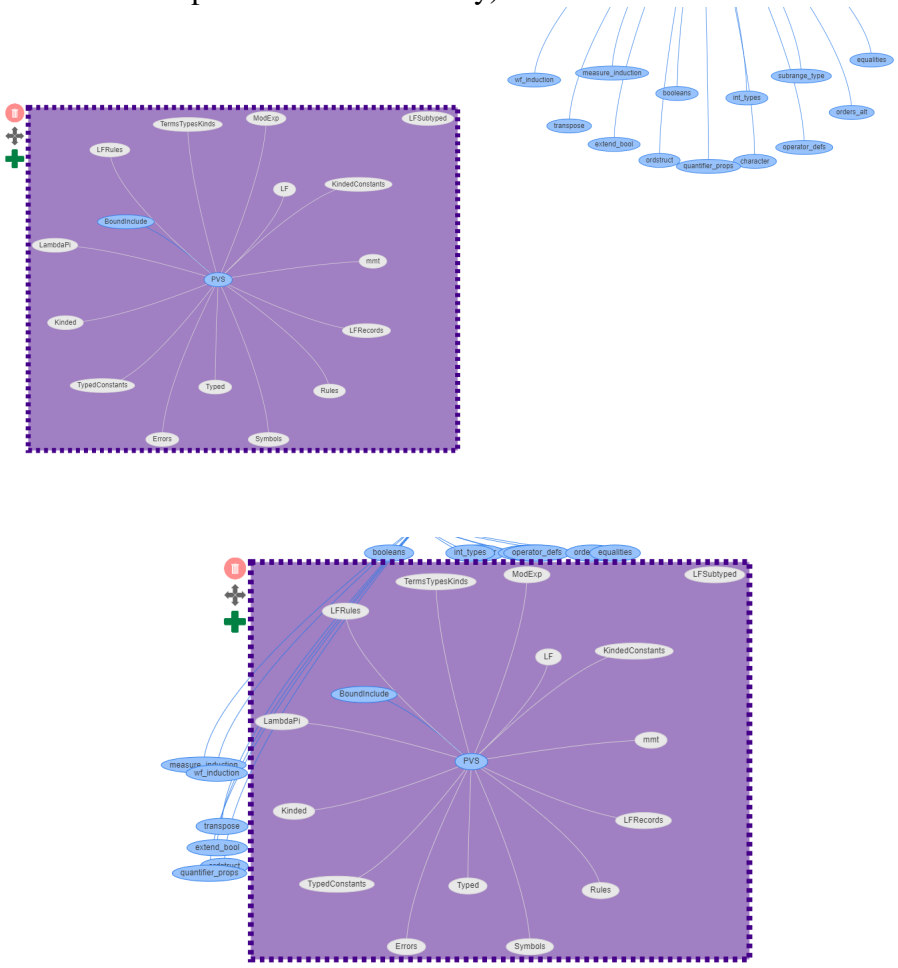


Figure 8.2: Fist image: Caged nodes in purple; Second image: The same caged nodes, which were moved to the center of the not caged nodes

8.3 Clustering

Besides caging of nodes, there exist use cases, which need the graph to lose complexity in structure by grouping nodes together. TGView supports clustering of nodes into one new node (see Figure 8.3) to fulfill this requirement. These nodes then can be moved around and edited like any other node in the graph. It is also able to open these cluster again later on and to cluster two already clustered nodes together into one new node.

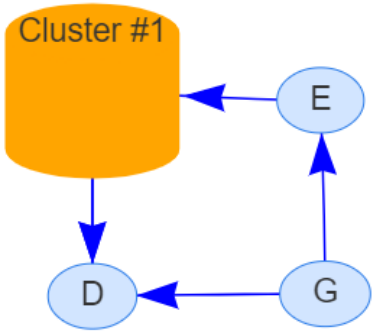


Figure 8.3: Cluster nodes into one new node

8.4 Lazy Loading of Childnodes

Lazy loading is a design pattern to delay the generation of subnodes until the nodes are really needed. This means that TGView only loads subnodes from MMT if they are really shown by the user, which explicitly triggered the corresponding action. The lazy loading of nodes is especially advantageous when dealing with huge graphs. Large graphs not only benefit from shorter loading times, but also give the user a tidy overview by displaying only those nodes that are relevant.
With the particularly large Coq graph, the differences are immediately noticeable. There the complete graph consists of about 2,000,000 edges and about 100,000 nodes. Loading all these nodes and edges at the same time would not only mean an extremely long waiting time during initial loading, but would also cause a memory overflow in Javascript, since Javascript is only allowed to use a relatively small amount of memory. In Firefox x64 the limit, according to our tests, is about 1Gb. Accordingly, it is technically not possible to load the complete graph at the same time. However, usually it is not necessary to load the complete graph, because this would overwhelm the user. Figure 8.4 shows a part of the Coq graph where only two of the 3000 expandable nodes (some expandable nodes are subnodes, so not everything is visible at the same

time) were expanded. At this image the sheer amount of dependencies and nodes is also visually visible.

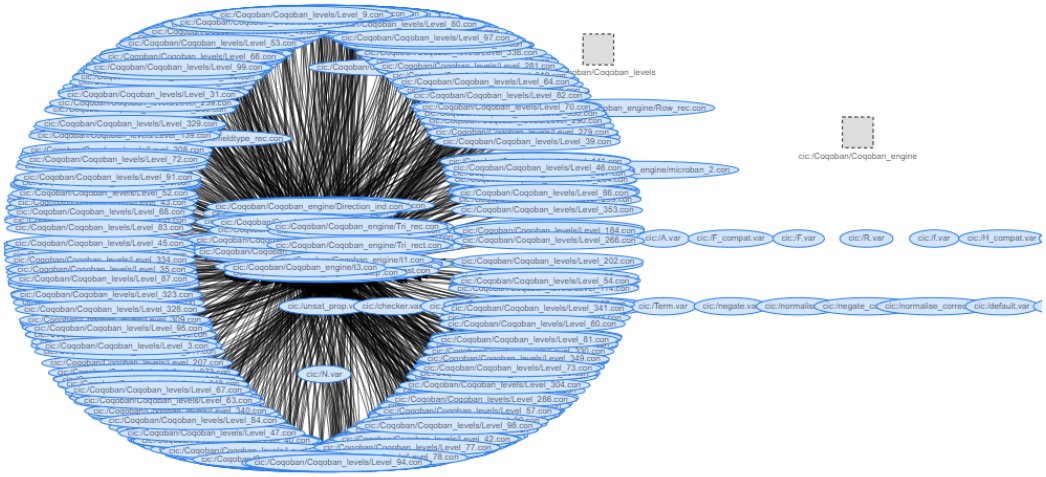


Figure 8.4: Coq-graph with two expanded nodes; grey nodes are expandable; blue nodes are not expandable

8.5 Hide and Show Nodes

Another way of reducing the complexity in TGView is by applying filters, which hide or show nodes/edges. TGView allows to hide/show nodes manually, but also allows filtering based on classes. This is implemented by checkboxes in UI next to the corresponding classes of nodes/edges. Figure 8.5 shows an example of a graph with hidden Meta-Edges and Border-Nodes in TGView.

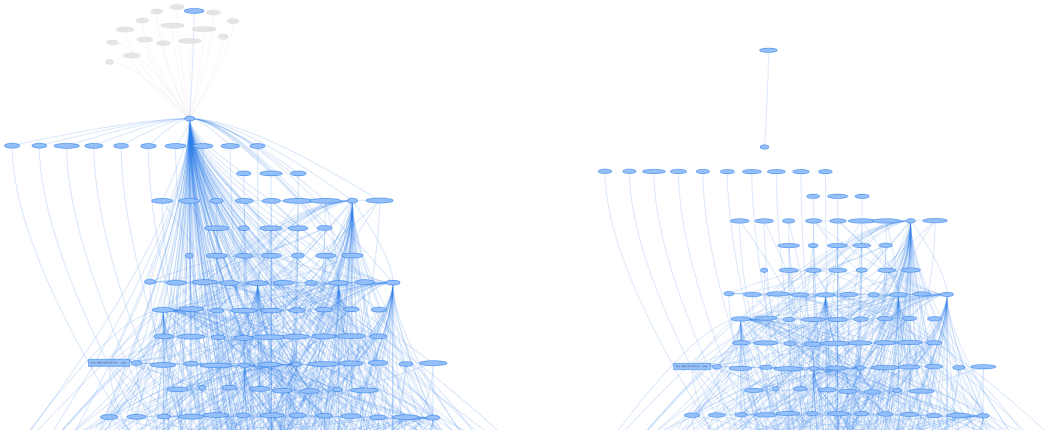


Figure 8.5: Left: Graph with Border-Nodes and Meta-Edges; Right: Graph with hidden Border-Nodes and Meta-Edges

8.6 Nodes and Edges - Legend

Remembering all node/edge styles and their corresponding meaning can be very challenging. The TGView-Legend helps to keep track of all used nodes/edges and their corresponding class by listing them at the bottom of TGView. This supports the user in not loosing the overview and focus, when searching for the class of a given theory. Figure 8.6 shows the legend for NASA graph. The legends adjusts to only show the classes, which are actually used in the graph. This helps to keep the legend clean.

Figure 8.6: Legend in TGView

Chapter 9

Remaining Requirements

As TGView mainly focused on the big three requirements described earlier, the solutions for the remaining requirements are not that rich.

9.1 Manipulation of Theories

TGView allows to add theories on a graph-based-view. Therefore the graph can be manipulated (afterwards also shared) on following ways:

- **Adding nodes and edges:** By adding nodes/edges manually, TGView allows to model whole graphs and theories in browser without writing any line of code.
- **Deleting nodes and edges:** As important as adding nodes/edges is deleting node/edges to undo wrongly placed nodes and connections.
- **Editing nodes and edges:** Editing nodes/edges is achievable by first deleting a node/edges and then re-creating it. However a shortcut for changing nodes/edges is by directly editing them.

9.2 Collaboration

Another important use case of an MMT/Mathhub exploration tool is the editing and subsequent sharing of the results. This ensures that even time-consuming or special editing steps can be divided among several people. By running TGView online and web-based, with MMT as the backend, fast collaboration across multiple locations is possible. TGView offers a variety of integration and sharing possibilities. These look like the following:

- **Image:** The graph is downloaded as an image. Unfortunately, due to framework limitations, no vectorized images are possible as output, only PNG files in a maximum resolution of 3000x3000px.
- **Javascript:** The TGView graph can easily be integrated as Javascript. Thus an integration into other websites is no problem.
- **URL:** An inclusion via URL returns a URL which can be called directly to load the corresponding graph.
- **IFrame:** Another way to share the graph is as IFrame. With this TGView can be integrated directly into any other website.
- **File:** Additionally, TGView offers to save and load the graph as a JSON file. This makes it very easy to create collabaration options and share layouted and generated graphs.

In addition to the graph itself, all layout, color and cluster changes are also saved. Thus a layouting of one graph from different persons is conceivable.

9.3 Highly Flexible Framework

Although TGView was created as part of the KWARC Group's MMT project, it still offers processing of non-mathematical graphs. Due to the general JSON structure, each graph is explorable in the corresponding JSON format. I demonstrate the extreme adaptability of TGView in Figure 9.1. It shows how TGView was used for layouting CYP enzymes and their interactions (medical field) [Ren02]. No line of code was touched, only the JSON was created in a suitable structure and imported into TGView.

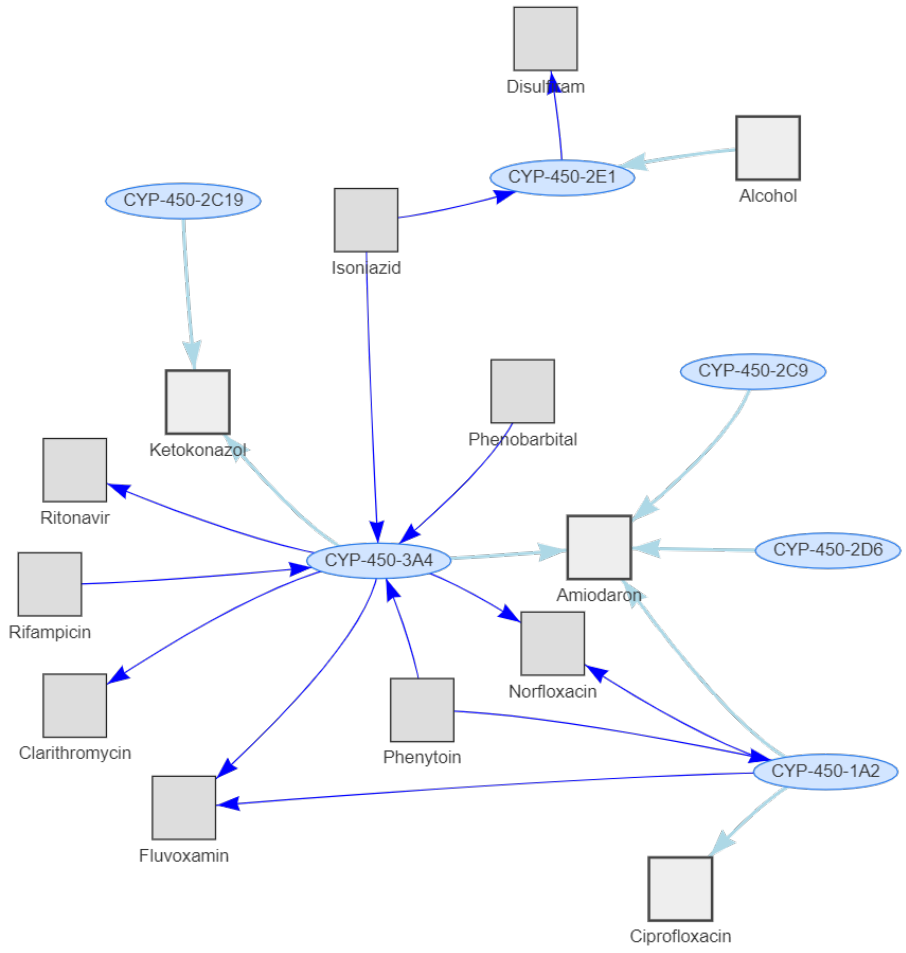


Figure 9.1: CYP-enzyme interaction in TGView

This makes TGView a generally usable graph framework, which is optimized for displaying and editing theory graphs. As shown in the previous example, however, other areas also benefit from the algorithms developed.

I ensure the maximum flexibility of TGView by working with node styles and edge styles, which can be easily defined by adding meta-information to the global option file. The general structure of such styles is shown in Listing 9.1

```
EDGE_STYLES=
{
    "include":
    {
        color:"#cccccc",
        colorHighlight:"#cccccc",
        colorHover:"#cccccc",
        dashes: false,
        circle:false,
        directed: true
    },
   ...
}

NODE_STYLES =
{
    "model":
    {
        shape: "circle",
        color: "#EAEAEA",
        colorBorder: "#DADADA",
        colorHighlightBorder: "#AAAAAA",
        colorHighlight: "#DADADA",
        dashes: false
    },
    ...
}
```

Listing 9.1: Example of defining edge and node styles

Using this easy to extend and highly modular JSON structure allows TGView to work with any kind of graph.

9.4 Good General User Experience

TGView as a tool for exploration of graphs, in particular theory graphs, is not only required to cover given use cases, but also to optimize the general user experience and user friendliness of the user interface. In the following I will therefore examine how well the general human-machine interaction at TGView performs based on the eight golden rules for a positive user experience of Shneiderman [Shn86].

Consistency Consistency is achieved by allowing similar interactions with designs, buttons, colors, menu structures, and user procedures to produce similar results in similar situations. For example, error texts with a red background on one side and an orange background on the other are inconsistent with the principle of consistency. Standardization of information transfer ensures that the user is able to apply knowledge from one click to another without having to learn new models for the same action. Consistency therefore plays an important role in helping users find their way through the program. So they can reach their goals faster and easier.
In TGView, this rule manifests itself in a uniform color scheme and a uniform interaction with the UI.

Informative Feedback Informative feedback ensures that the user knows what the program is doing at any time and in every situation. There should be human readable feedback for each action within a reasonable period of time. Long waiting times without an indicator of progress and actions without recognizable feedback should therefore be avoided.
TGView is a Javascript and HTML5 application, which makes the use of multi-threading difficult. Due to the single-threaded environment of Javascript, it is not possible to send appropriate feedback to the user at any time without a lot of programming effort. A detailed breakdown of the challenges and feedback from TGView is as follows:

- **Feedback when loading graphs:** Feedback during download and re-layout of graphs can be seen in TGView in the form of a small status text in the upper left corner. TGView itself regularly sets the current status during long computations, but the browser renders these results only as soon as the computations are done. Accordingly, only the first and last status text is visible, i.e. exactly those texts that do not yet or no longer fully use the browser render thread. I was able to mitigate this problem by making the mouse pointer, which is not rendered by the main browser thread, display a rotating circle or hourglass as long as TGView is processing anything.

- **Feedback when opening PopUps:** TGView uses Mathhub and redirects to the corresponding Mathhub page when opening theories behind nodes. Some settings in the browser prevent the opening of such pop-ups and the feedback from the browser is mostly hard to not visible. Therefore TGView additionally writes into the status box if such problems occur.
- **Selecting nodes:** When a node is clicked in TGView, that node and its associated edges are highlighted in color so that the user receives visual feedback about their action.

Closureness Dialogs should be designed in such a way that the user can see at any time how far away from the target he is. In addition, each action should have a defined beginning and end.
There are no dialogs in TGView that consist of several steps. Therefore this point is omitted for TGView.

Error Prevention Incorrect entries should be avoided by providing the user with sufficient information at all times. If incorrect entries nevertheless occur, the tool should recognize them and help the user to correct them.
TGView allows user input when creating/editing nodes/edges. The texts next to the input fields are provided with an information text that appears when driving over them to avoid mistakes. In addition, the entries are checked for errors. A distinction is made between the following scenarios:

- **Invalid MathML string:** When creating a node, the MathML string is parsed and displayed to the right of the input fields. This avoids errors and helps to detect syntactically or semantically incorrect formulas.
- **Invalid ID:** Each node and each edge must have a unique ID. During the input the uniqueness of the ID is determined and if it is not unique, an error is displayed.

Reversibility The reversibility of individual and group of actions is essential for a good user experience. According to Shneiderman, this reduces the fear of mistakes because the user knows that any mistake can be undone. This also encourages curiosity and the exploration of new options and actions.
In TGView there is a classic redo and undo button, which undoes the last action or executes the undone action again. Every error in TGView can be fixed quickly and easily.

User Control The user should always have control over any actions. Actions should be initiated primarily by the user and should consult the user in case of doubt. This gives users a feeling of full control over events.
In TGView, the user is the initiator of any action.

Relief of Short-Term Memory Because we humans only own limited attention, it is important to relieve short-term memory as much as possible. It is well known that people can keep about five different elements in their short-term memory at the same time. Therefore, the interaction possibilities that are visible at the same time should be limited to approximately this amount or at least should be well-structured. In addition, an appropriate information hierarchy is important.
TGView tries to relieve short-term memory by keeping the unneeded menu items on the left and right collapsible. This not only prevents the short-term memory from overflowing, but also offers the advantage of using on demand more space for the graph itself.

Universal Usability Although TGView mainly visualizes mathematical concepts as graphs, one requirement of the user interface is that both beginners and professionals can use the tool. The user interface was deliberately designed so that anyone can interact with TGView and the graph without understanding the mathematical concepts. However there still exist some expert functions like uploading a manually created JSON or by editing/adding nodes manually. Therefore same interface is practical for beginners and professionals.

Chapter 10

Conclusion and Future Work

Many fields in science benefit highly from graph visualization based on their data to gain insights into structures and dependencies.
So, there are also many different graph visualization tools out there. Ranging from GraphViz with its powerful API but fixed and non-interactive layout, over several experiments like Cornac, which was optimized for huge graphs, to Gephi offering rich interactions, which sometimes overwhelm the user.
However all these works lack important features for working with theory graphs and MMT/Mathhub. Even though many of them offer the possibility to extend them by using plugins or writing own code, they most of the time lack browser support. But this is mandatory for Mathhub and MMT. Therefore I developed the first version of TGView as part of an university project. This phase was especially driven by exploration and testing. While running this phase, I collected feedback, interviews, use cases and more especially from the KWARC group to optimize TGView. Optimizing was done by analyzing several use cases and getting feedback from users. I aggregated the results and insights to form several requirements for graph visualization as part of the MMT/OMDoc ecosystem. While aggregating these requirements, I started developing TGView version two, which main goal was to fulfill as many of the requirements. These requirements were checked in this work against the current version of TGView. The overall results is that TGView meets big parts of the requirements but lacks fulfillment of requirements especially in the field of collaboration and general user experience due to browser limitations. Therefore I will now outline possible future improvements and challenges to overcome. In detail there are five big challenges, which TGView needs to overcome to fully unleash the power of theories modeled as graphs.

Browser limitations While modern browsers offer plenty of interaction possibilities and high-computing throughput in Javascript, they still lack many beneficial features and put down a bunch of limitations. One of the biggest challenge is to overcome the single-threaded work of Javascript. Even though latest browsers offer some alternative to multi-threading, called ”web-worker”, they still add a few layers of complexity and overhead in terms of computing performance. As in Javascript every thread is a new process, it is not possible to share variables in memory within several threads or use some basic locking and awaiting structures, which other languages offer. Instead in Javascript you have to set up a message sender and a message receiver and send the data over these two methods. This adds not just overhead to the communication itself but also makes synchronizing more complex. Several algorithms in TGView would be extremely easy to parallelize (e.g. the forces driven algorithms or modularity based clustering), but the added complexity and overhead leaves the benefits of migration to multi-threading in Javascript questionable. The single-threaded character of Javascript is the biggest show-stopper. Unfortunately there are many more limitations like a maximum of about 1Gb memory usage per website, limited dimensions for downloading generated images in Javascript (about 3000 by 3000 pixels) and low GPU utilization using standard Javascript drawing methods.
The best method to overcome these browser limitations while still enabling user to interact with theory graphs without installing any software, would be to convert TGView to some web-compatible technologies like Unity and Java.

Modeling theories Modeling whole theories from the very scratch and connecting them with other theories in browser by drag and drop one of TGView visions. A user then is able to model some graphs in TGView and TGView converts the graph to MMT/OMDoc. This MMT/OMDoc file will then be sent to the MMT backend, which includes it into theory collection. For this vision to become true, not only TGView needs a lot of adjustments but also MMT needs to support this properly. Also challenges like applying type checkers of MMT while preserving responsiveness and real-time editing of TGView needs to be overcome.
This can be reached by implementing a simple type checker in frontend and apply the whole type checking procedure only once when issued by user. This would work similar to nowadays IDEs, which check the code for obvious mistakes before sending the code to the compiler, who will find more hidden problems.

Limitations of two dimensional space Marcus et al. [RM19] developed TGView3D, a 3D Graph-Viewer using Unity and C#. TGView3D offers one dimension more than TGView. This boosts the understanding of whole graphs by several magnitudes combined with the right algo-

rithms. As 3D offers by nature one more free dimension variable to adjust, it is also by nature more powerful when modeling complex structures than the two dimensional space.
However the line between powerful and overwhelming is smaller for 3D than for 2D graphs. I still think, that 3D graphs, which can be drawn planar easily, brings more benefits than solely using 2D graphs. This point may become no longer relevant if we decide to merge TGView and TGView3D, which would also ease maintenance due to one repository instead of two.

Good theory graph layouts The current layout algorithms were a good step in the right direction. But in general I still fail to find an algorithm, which highlights the unique structure of theory graphs.
Finding such an algorithm will involve a lot of observations and theoretical calculations. The currently most superior algorithm seems to be TGView's water driven layout. The results are quite structured but still look natural and symmetrical. Especially for big graphs even the water driven layout fails and produces cluttered graphs without meaningful structures. An algorithm, which is optimized on theories and their transitive connections, may be converting the graph first to a minimal spanning tree, calculating the hierarchical layout, converting the spanning tree back to original graph and then finetune using forces driven algorithms.

Extensive collabaration TGView currently provides the possibility to layout a graph and share the results by sending the downloaded graph file. This means, that if two persons edit the same graph at the same time, the other person would not see the changes until sent as file. But even then the two graphs may differ completely and cannot easily be merged. A better solution would be to enable synchronized real-time editing of graphs. This would also allow to explain and show several structures in the theories over big distances without downloading any file.

In general theory graphs are a reasonable way to interact with MMT and the theories, but the browser limitation are sometimes not circumventable. Therefore future implementations may use other technologies. A possible future TGView implementation could use Unity and C# like used by TGView3D as first steps look really promising. We still have to evaluate its potential more deeply as TGView itself was developed and tested over nearly two years. Where on the other hand TGView3D indeed uses insights of TGView, but still only exists for about one quarter of the time. Most of the problems and challenges occurred especially over time when working with Vis.js and Javascript. Therefore future research could be directed in evaluating TGView3D as alternative to TGView or even merging both systems into one.

Bibliography

[Arb] *Arbor.js*, http://arborjs.org/, [Online; accessed 06/04/2019].

[Bas09] M. Bastian, S. Heymann, M. Jacomy: *Gephi: An Open Source Software for Exploring and Manipulating Networks*, 2009.

[Bor07] D. Borthakur: *The hadoop distributed file system: Architecture and design*, *Hadoop Project Website*, Bd. 11, Nr. 2007, 2007, S. 21.

[Bra08] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, D. Wagner: *On modularity clustering*, *IEEE transactions on knowledge and data engineering*, Bd. 20, Nr. 2, 2008, S. 172–188.

[Coq] *The Coq Proof Assistant*, http://coq.inria.fr/, [Online; accessed 07/31/2010].

[Ell04] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, G. Woodhull: *Graphviz and dynagraph—static and dynamic graph drawing tools*, in *Graph drawing software*, Springer, 2004, S. 127–148.

[Far93] W. M. Farmer, J. D. Guttman, F. J. Thayer: *IMPS: An interactive mathematical proof system*, *Journal of Automated Reasoning*, Bd. 11, Nr. 2, 1993, S. 213–248.

[Fre77] L. C. Freeman: *A set of measures of centrality based on betweenness*, *Sociometry*, 1977, S. 35–41.

[Gap] *The GAP Group. GAP – Groups, Algorithms, and Programming*, www.gap-system.org, [Online; accessed 08/30/2016].

[Gra] *Graphviz – Graph Visualization Software*, http://www.graphviz.org/, [Online; accessed 05/22/2017].

[Hu05] Y. Hu: *Efficient, high-quality force-directed graph drawing*, *Mathematica Journal*, Bd. 10, Nr. 1, 2005, S. 37–71.

[Ian14] M. Iancu, C. Jucovschi, M. Kohlhase, T. Wiesing: *System description: MathHub. info*, in *Intelligent Computer Mathematics*, Springer, 2014, S. 431–434.

[Kle99] J. M. Kleinberg: *Authoritative sources in a hyperlinked environment*, *Journal of the ACM (JACM)*, Bd. 46, Nr. 5, 1999, S. 604–632.

[Koh11] M. Kohlhase, J. Corneli, C. David, D. Ginev, C. Jucovschi, A. Kohlhase, C. Lange, B. Matican, S. Mirea, V. Zholudev: *The planetary system: Web 3.0 & active documents for stem*, *Procedia Computer Science*, Bd. 4, 2011, S. 598–607.

[Koh14] M. Kohlhase: *A data model and encoding for a semantic, multilingual terminology of mathematics*, in *Intelligent Computer Mathematics*, Springer, 2014, S. 169–183.

[Miz] *Mizar*, http://www.mizar.org, [Online; accessed 02/27/2013].

[MMT] *MMT – Language and System for the Uniform Representation of Knowledge*, https://uniformal.github.io/, [Online; accessed 08/30/2016].

[Neo] *The graph database Neo4j*, https://neo4j.com/, [Online; accessed 01/04/2019].

[Ope] *OpenAxiom: The Open Scientific Computation Platform*, http://www.openaxiom.org, [Online; accessed 05/22/2017].

[Owr92] S. Owre, J. M. Rushby, N. Shankar: *PVS: A prototype verification system*, in *International Conference on Automated Deduction*, Springer, 1992, S. 748–752.

[Pag99] L. Page, S. Brin, R. Motwani, T. Winograd: *The PageRank citation ranking: Bringing order to the web.*, Stanford InfoLab, 1999.

[Pau94] L. C. Paulson: *Isabelle: A generic theorem prover*, Bd. 828, Springer Science & Business Media, 1994.

[Per18] A. Perrot, D. Auber: *Cornac: Tackling Huge Graph Visualization with Big Data Infrastructure*, *IEEE transactions on big data*, 2018.

[Pro] *Visualization - Protege Wiki*, https://protegewiki.stanford.edu/wiki/, [Online; accessed 05/22/2017].

[Pur97] H. C. Purchase, R. F. Cohen, M. I. James: *An experimental study of the basis for graph drawing algorithms*, *Journal of Experimental Algorithmics (JEA)*, Bd. 2, 1997, S. 4.

[Rab13] F. Rabe: *The MMT API: a generic MKM system*, in *International Conference on Intelligent Computer Mathematics*, Springer, 2013, S. 339–343.

[Rea] *React - A JavaScript library for building user interfaces*, https://reactjs.org/, [Online; accessed 05/05/2019].

[Reg] *The ReGraph Toolkit*, https://cambridge-intelligence.com/regraph/, [Online; accessed 06/04/2019].

[Ren02] S. Rendic: *Summary of information on human CYP enzymes: human P450 metabolism data*, *Drug metabolism reviews*, Bd. 34, Nr. 1-2, 2002, S. 83–448.

[RM19] F. R. Richard Marcus, Michael Kohlhase1: *TGView3D System Description: 3-Dimensional Visualization of Theory Graphs*, 2019, S. 5.

[Rup17] M. Rupprecht, M. Kohlhase, D. Müller: *A flexible, interactive theory-graph viewer*, in *12th Workshop on Mathematical User Interfaces, MathUI*, 2017.

[Sag] *The Sage Developers. SageMath, the Sage Mathematics Software System*, http://www.sagemath.org, [Online; accessed 09/30/2016].

[Shn86] B. Shneiderman: *Eight golden rules of interface design*, *Disponible en*, 1986.

[Xin13] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica: *Graphx: A resilient distributed graph system on spark*, in *First International Workshop on Graph Data Management Experiences and Systems*, ACM, 2013, S. 2.

[yEd] *yEd - Graph Editor*, https://www.yworks.com, [Online; accessed 06/05/2019].

[Zah10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica: *Spark: Cluster computing with working sets.*, *HotCloud*, Bd. 10, Nr. 10-10, 2010, S. 95.