Dhruv Makwana

Exploring the structure of mathematical theories using graph databases

Computer Science Tripos, Part II

Trinity College

May 8, 2017

Proforma

Name:	**Dhruv Makwana**
College:	**Trinity College**
Project Title:	**Exploring the structure of mathematical theories using graph databases**
Examination:	**Computer Science Tripos, Part II, 2016–2017**
Word Count:	–
Project Originator:	Dr. Timothy G. Griffin
Supervisor:	Dr. Timothy G. Griffin

Original Aims of the Project

100 words

Work Completed

100 words

Special Difficulties

100 words [hopefully "None."]

Declaration

I, Dhruv Makwana of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed: Dhruv Makwana

Date:

Contents

1 Introduction

Coq is an interactive proof-assistant [9]; it allows users to formalise mathematical theories into machine-checked proofs. Coq libraries, which encode mathematical theories, are difficult to understand. In this dissertation, I will describe a new tool I implemented for Coq users that aimed to (a) represent Coq libraries as Neo4j (graph) databases and (b) provide a library of queries with the goal of highlighting the structure and relationship between the representations of the proof-objects by using network-analysis techniques usually associated with social-networks.

1.1 Motivation

Coq proof-scripts are notoriously difficult to understand. Not only do these proof-scripts encode a mathematical theory, which can be difficult to understand in and of itself, they serve as verbose instructions on how to create proof-term whose type matches a certain proposition, rather than a statement of *why* something is true. There are currently no tools which help with the challenge of gaining a *high-level* understanding of a large Coq library.

The Feit-Thompson Odd Order Theorem [15, 2] (which has been translated into Coq [6]) is an example of a large Coq library. It is a part of larger, general trend of formalising substantial bodies of mathematics and marks the first time that we have a new way representing mathematics, different from the combination of formulas and natural-language prose usually used. Such a turning point provides an opportunity to explore the novel representations and analyses possible.

However, it is still the case that when a user approaches a large Coq library, they are left to understand and keep track of several aspects of the library (such as implicit assumptions, previously defined results and the types and conventions behind any notation) by themselves. There is very little opportunity to consider and compare different approaches for arriving at a result (i.e. number of assumptions, number of steps, some notion of the importance such as number of uses).

I obtained such an opportunity by using a query-based approach: by creating a tool that expresses Coq library as databases, then I was able query them and analyse them to gain insights such as those mentioned above.

1.2 A Database Approach

Mathematical theories are highly interconnected structures of definitions and proofs. As such, relational or a document-oriented models are not adequate representations for answering questions such as "What depends on this lemma and how many such things are there?" or "What are the components of this definition?".

Even a static, graph-based approach on its own is insufficient. Simply outputting a dependency graph is ineffective for all but the smallest of libraries. Figure 1.1 shows one such dependency graph for a medium-sized Coq library.

Graph databases have been developed to deal with highly connected data sets and path-oriented queries. That is, graph databases are optimised for computing transitive-closure and related queries, which pose a huge challenge for traditional, relational databases. Due to the emergence of massive data sets thanks to large-scale social and advertising networks, new techniques for representing and analysing such data have been developed.

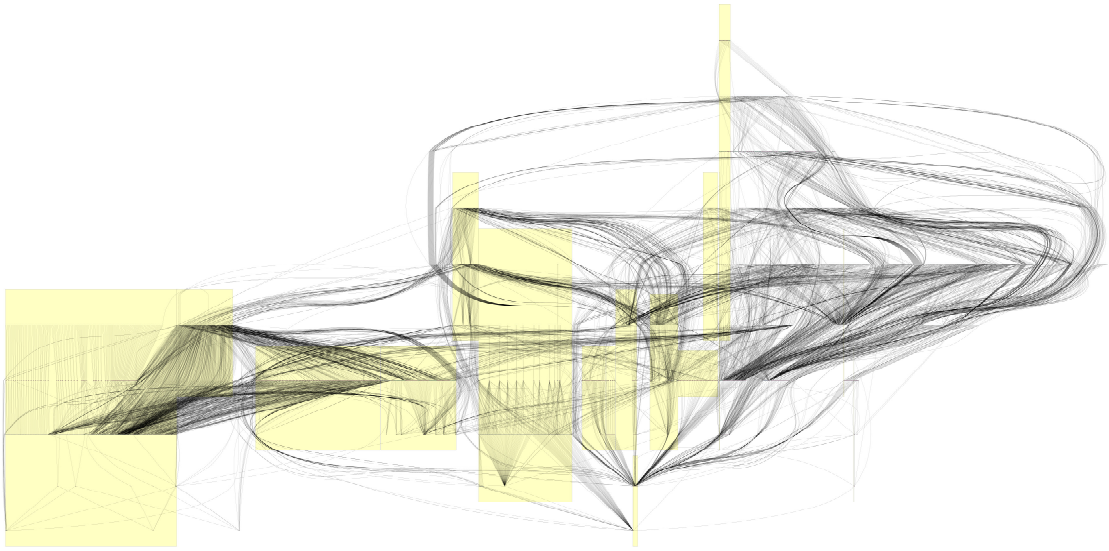

Figure 1.1 – Static graph of CAS Coq library. For all but the smallest of libraries, simply outputting a dependency graph is an ineffective way of understanding them.

For this project, I used the Neo4j [10] graph database system (and its expressive query language *Cypher*) to create a tool for Coq users which applies such network-analysis techniques to Coq libraries to aid understanding them.

1.3 Aims of the Project

In this project, I aimed to write a tool for Coq users that:

- represents Coq libraries as Neo4j graph databases, which consists of
  - exploring and choosing the correct model
  - converting and extending existing code to output CSVs
  - writing new programs to extract extra information (omitted from other, existing tools)
  - writing new programs to automate database creation.
- provides a library of Neo4j queries, intended
  - to highlight the structure of and relationship between proof-objects
  - to provide several network-analysis techniques.

Note that the aims centre around the *mechanics and details* of creating a model for Coq proof-objects and applying network analysis techniques to it and *not creating a polished, user-friendly product*. Since *usability was not a* primary *focus of this project*, I did not conduct a user-study whilst evaluating this tool.

1.4 Summary

First, I explained why Coq libraries difficult to understand. Then, I said that the trend of formalising large bodies of mathematics such as the Feit-Thompson Odd Order Theorem [15, 2] presents an opportunity to develop new representations and analyses of mathematical theories. After that, I outlined why databases, specifically graph databases, were a promising approach to this new opportunity. Finally, I listed the aims of this project.

2 | Preparation

In this chapter, I will describe how I planned this project. I will start by mentioning the development methodology and tools I used. Following that, I will present and explain the requirements I used to guide the implementation of my project. From there, I will state my choice of technologies for this project. I will then go on to elaborate on my starting point for this project. This will consist of a brief commentary on Coq, a comparison of existing tools that aim to help a user understand a Coq library and a description of Neo4j and some of its relevant plugins.

2.1 Project Planning

This project had two distinct phases: deciding how to model Coq proof-objects (an exploratory phase) and applying network-analysis techniques (a technical phase). For both, I chose a spiral software development model: think of an idea, modify the code, propagate the necessary changes, evaluate the end-result and repeat. This allowed me to experiment with ideas flexibly and easily.

I found Git (`git-scm.com`) and GitHub (`github.com/dc-mak`) invaluable during the project; they allowed me to easily track, revert and review changes. I could safely test new features on different branches before merging them in and store a copy of my work in multiple places. As it became apparent that precisely specified versioning, build-dependencies and tests were useful in spotting errors early, I added GitHub extensions such as Travis-CI for automated continuous-integration builds.

2.2 Requirements Analysis

I developed and brought together several distinct components for *modelling* and translating (from Coq to the chosen model), displaying and *interacting* (Neo4j/Cypher) and *computing* and analysing (plugins). Below is a list of required features I used throughout development to guide and provide context for implementation decisions.

- **Modelling**: The model should

  M1 include as much relevant data as possible. Here, relevant means useful to understanding a large library, but not so much so as to obfuscate any information or make learning how to use the project more difficult.

  M2 be flexible to work with and easy to translate. One could imagine different front-ends for interacting with and computing data from the model.

  M3 strike a balance between size and pre-computing too much data. Figuring out which pieces of data can be reconstructed later and which are beneficial to compute during modelling will be a matter of experimentation and weighing up ease of implementation versus ease of later processing.

- **Interacting**: Interacting with the model should

  I1 primarily, allow users to understand the data. The following two points follow from this principal goal.

  I2 support both graphical and textual modes of use. Small queries and novice users are likely to benefit from the presence of a well-designed

GUI. However, larger queries requiring more computation and flexibility will benefit from a traditional, shell-like interface.

I3 be interactive and extensible. A static presentation of data, even in a GUI, would fail to make full use of graph-databases and the ability to query, in whatever way the user desires, information dynamically.

- **Computing**: Working with the model's data should

  C1 be enabled by a core library of good defaults. Certain, common functions should be ready 'out-of-the-box' and provide users all they need to get started.

  C2 allow the user to add their own functions. It is not possible to imagine and implement all the functionality users may desire and so a way to extend the project to suit their own needs would be of great use.

2.3 Technologies Used

Choice of implementation languages was, although an important decision, almost completely dictated by the programs at the core of the project (Coq and Neo4j).

Coq and its plugins are written in OCaml (`ocaml.org`); I stuck with it for two reasons. First, it is almost always wiser to work with and modify existing systems (e.g. integrating with the build-system) and more representative of real-world work. Second, as a functional language, OCaml benefits from a strong, static (and inferred) type-system (which allows for easy experimentation and greater confidence in correctness). Hence, using OCaml for other parts of the tool which need not necessarily be in OCaml (e.g. the `dpd2` utility) was a welcome and easy decision. OCaml has several other benefits too, such as inductively-defined datatypes (useful for working with Coq's grammar) and good editing tools.

Similarly, Neo4j and its plugins are (usually) written in Java, but several languages are supported for the latter, both by Neo4j officially and by the community. As I will explain in Section 2.7, I found Java and R to be the most suitable for achieving this project's goals.

2.4 Starting Point

Both Coq and Neo4j have a rich ecosystem of tools and libraries built for them. I examined the current landscape in detail to determine the software available so as to then use the relevant programs as a starting point effectively.

2.5 Coq Proof-Assistant

The Coq proof-assistant, implemented in OCaml, can be viewed as both a system of logic – in which case it is a realisation of the *Calculus of Inductive Constructions* – and as a *dependently-typed* programming language. Its power and development are therefore most-suited and often geared towards *large-scale* developments.

On the logical side, Coq lays claim to feats of modern software-engineering such as the Four-Colour Theorem [5] (60,000 lines) and the aforementioned *Feit-Thompson* theorem (approximately 170,000 lines, 15,000 definitions and 4,200 theorems).

On the programming language side, Coq has served as the basis for many equally impressive projects. The *CompCert Verified C Compiler* [8] demonstrates the practical applications of theorem-proving and dependently-typed programming by implementing and proving correct an optimising compiler for the C programming language. *DeepSpec* [16], a recently announced meta-project, aims to integrate several large projects such as *CertiKOS* (operating system kernels), *Kami* (hardware), *Vellvm* (verifying LLVM) and many more in the hopes to provide complete, *end-to-end* verification of real-world systems.

Coq is a *huge* project, developed by INRIA (France), and its size and complexity are best experienced through untangling the source code for oneself. Just for the implementation of the system (not including the standard library), Coq features approximately 3 major ASTs, 6 transformations between them, 3000 visible types, 9000 APIs and 521 implementation files containing 228,000 lines of dense, functional OCaml.

However, most of this massive project is sporadically (and tersely) documented. Even after I received some guidance via the Coq developers' mailing-list, I spent several hours browsing the source code trying to understand how the system worked. Although I had some prior familiarity with *using* Coq (as an introduction to tactical theorem-proving and dependently-typed programming), it was not useful for understanding the internals beyond context and how to compile and use programs and libraries. However, it did serve as invaluable insight for designing the model during the implementation phase.

2.6 Existing Tools for Coq

There are many tools for Coq that, like my project, *aim to help a user understand a library.* I studied a number of them to learn their approaches and analyse their strengths and weakness. What follows is a detailed account of each tool and why it did not meet the project's aims and requirements.

2.6.1 Coqdoc

Coqdoc is a documentation tool for Coq projects, included as part of the Coq system. It can output to raw text, HTML, LaTeX and a few other formats to help a user navigate and understand a Coq library. Although it supports prettifying code with syntax highlighting and Unicode characters, its most relevant feature was its hyperlinking: potentially useful for building dependency graphs.

However, the whole tool works on an entirely *lexical* level, with no formal parsing, understanding or elaboration of the code structure. Hence, since it could not meet any of the modelling requirements (completeness M1, flexibility M2 and size/pre-computation M3) this approach was abandoned.

2.6.2 Coqdep

Coqdep is a utility included with Coq that helps users understand a Coq library's *module-level* dependencies by tracking `Require` and `Import` statements. Although on first impressions, this tool seemed to offer more flexibility than coqdoc, it was even more restrictive: it simply searches for keywords (such as `Require` or `Import` for Coq and `open` or dot-notation module usage for OCaml) per file and outputs them accordingly. As with coqdoc (and for the same reasons), I abandoned this approach too.

2.6.3 CoqSerAPI

Coq Serialized (S-expression) API (`github.com/ejgallego/coq-serapi`) is a new library and communication protocol aiming to make low-level interactions easier (using OCaml datatypes and s-expressions), particularly for tool/IDE developers. It has a starting point for gathering some statistics on proof-objects in a project. While this is likely to be useful in the future, it is still far from complete and is more geared towards interactive *construction* (via a tool/IDE) rather than *analysis*. As such, tracking dependencies (critical to the modelling requirements) is not possible.

2.6.4 dpdgraph

dpdgraph (`github.com/Karmaki/coq-dpdgraph`) is a project which helps users understand the dependencies between proof-objects in a Coq library. It does so by extracting information from compiled Coq object-files to a `.dpd` file. It includes two example tools: `dpd2dot` (for producing a `.dot` file for static visualisation) and `dpdusage` (for finding unused definitions). Its developers intended it to be a starting point for other tools to build upon.

Although lots of information such as notation, the relationship between constructors and the types they construct, proof tactics, the precise kind of an object (e.g.

fixpoint, class, lemma, theorem, etc.) and which module an object belongs to was missing, I thought it unlikely that the information was not present in the compiled object files (since they are necessary for term-construction and type-checking).

Assuming that the data was already present in those files, but simply *ignored or unused*, I focused on understanding and augmenting dpdgraph to add the missing pieces to the model and output it to comma-separated values (henceforth referred to as CSVs).

2.6.5 Comparison

I have just discussed the many existing tools for Coq that *aim to help a user understand a library*. Table 2.1 summarises the main features of each. The features chosen reflect the strengths of each tool considered and are justified and elaborated upon below.

Bundle with Coq, coqdoc produces **hyperlinked source code** meaning details such as the **precise kinds** of a proof-object, the **constructors of a type** and the **type signatures** are immediately visible; hence those 5 dimensions were included. Also include in a Coq system is coqdep: a tool for modelling **module-level dependencies** that can output a `dot` file to present it **graphically**.

Coq Serialised (S-expression) API is an **interactive** IDE communication protocol with facilities for gathering some basic **statistics**. Here, interactive is used to mean that information is not presented all at-once, *statically*, but can instead be queried dynamically at run-time. An example of a static display is Figure 1.1; it shows a medium-sized Coq library as output by dpdgraph.

In principle, coqdep and dpdgraph can also support some degree of interactivity, with support from other tools (which translate `.dot` files to interactive JavaScript), although this is rarely done. What dpdgraph does do well is model **object dependencies**, with some scope for distinguishing precise kinds and displaying information graphically.

2.7 Neo4j

Neo4j is a graph database system implemented in Java. Traditional, relational database theory and systems are designed with the goal of storing and manipulating information in the form of *tables*. As such, working with highly interconnected data, such as social network graphs is best tackled with the alternative approach of *graph databases*.

Briefly, a (directed) *graph* is defined as $G = (V, E)$ where V is a set of vertices or *nodes* and $E \subseteq V \times V$ is a set of edges or *relationships* between two nodes. A *graph database* is an OLTP (online transaction processing, meaning operated upon live, as data is processed) database management system with CRUD (create, read,

	Source Code	Hyperlinks	Precise Kinds	Constr. & Types	Type Sig.	Module depend.	Graphical rep.	Interactivity	Statistics	Object depend.
Coqdoc	■	■	■	■	■	■	■	■	■	■
Coqdep	■	■	■	■	■	■	■	■	■	■
CoqSerAPI	■	■	■	■	■	■	■	■	■	■
dpdgraph	■	■	■	■	■	■	■	■	■	■

■ Has feature ■ Does not have feature ■ Can be extended to support it

Table 2.1 – Comparison of Features. There are many tools for Coq that, like my project, *aim to help a user understand a library*. This table summarises the main features of each. The features chosen reflect the strengths of each tool considered. Detailed commentary is provided in Subsection 2.6.5.

update and delete) operations acting on a graph data model. Relationships are therefore promoted to first-class citizens and can be manipulated and analysed.

Neo4j supports both graphical and textual modes of use and is easily extensible (through Cypher plugins and several language-specific bindings and libraries). It meets all the interaction requirements of helping users to understand data, being flexible in its use and extensible in its capabilities. It even includes a tool to import CSVs files containing nodes and edges into a new database. This meant I could focus on extracting and expressing in a simple format as much information as possible.

Neo4j also includes an interactive graphical interface, accessible through an ordinary web-browser. As can be seen on Figure 2.1, the tool offers

- an overview of the current labels, relationships and properties in the database
- syntax-highlighted and interactive query input
- graphical representation of query result (with options to view it as rows like a shell, or raw JSON text results) with profiling information along the bottom
- easy access to favourite queries and scripts (the star on the left)
- easy access to documentation and system information (the book on the left)
- other features such as browser sync, settings and the 'about' section.

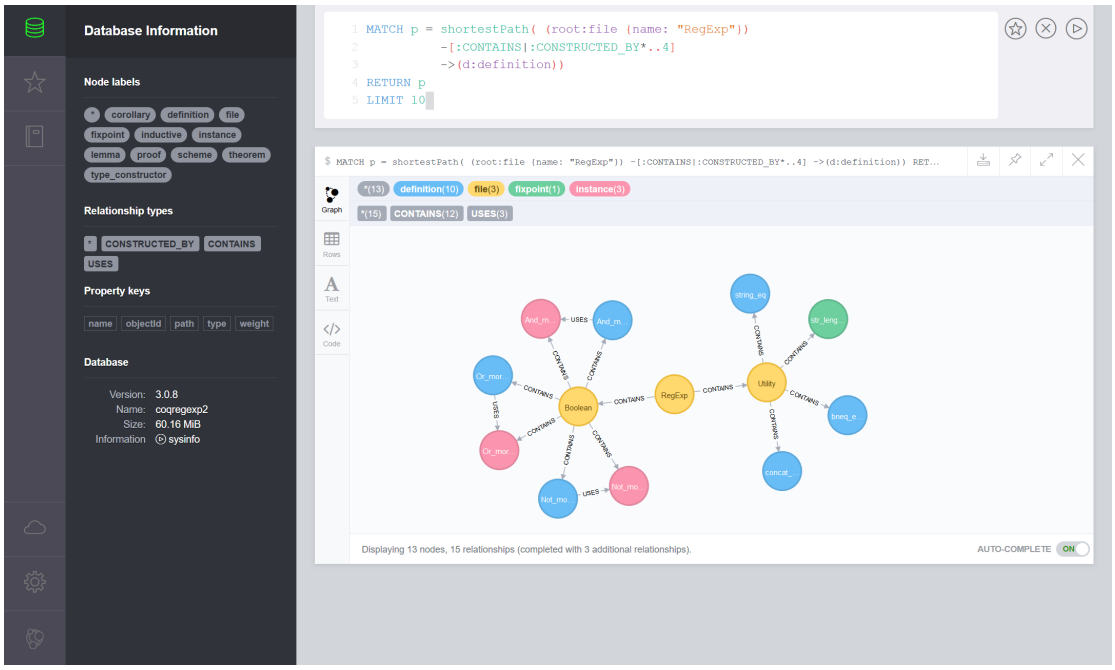

Figure 2.1 – Neo4j Interactive Browser. See Section 2.7 for a full list of features.

2.8 Existing Tools for Neo4j

Neo4j features rich integration with many languages, libraries and tools. Of those, I found to be the most relevant and useful tools for meeting the project requirements.

2.8.1 APOC: Awesome Procedures on Cypher

Awesome Procedures on Cypher, or *APOC* for short, is a community-maintained Java plugin featuring several network-analysis algorithms callable from within Cypher itself. Although there are other extension libraries (such as MazeRunner), APOC is easy to install, well-documented, up-to-date and the most comprehensive, and therefore the obvious choice as a foundation.

Since it is a Java library hooked into Cypher, it offered the potential for additional functionality to be built on top of it. For example, a library which packages some of the more complex features into *domain-specific* queries, intended for Coq users not familiar with Neo4j to get started with.

Thus, APOC helps step towards meeting the *interaction* requirements for this project by being easy to understand, flexible to use and extensible; it even goes part-way towards meeting the *computation* requirements.

2.8.2 igraph

APOC's main focus is on interacting with and combining different sorts and sources of data and so lacks graph analysis functionality *beyond* the basics. The fact that it is implemented in Java adds to its limitations: it is not well-suited to more intense analyses over large graphs of libraries and is insufficient to *fully* meet the *computation* requirements of this project.

For such tasks, igraph is ideal: it is described on its website as a *collection of network analysis tools, with the emphasis on efficiency, portability and ease of use*. Written in C/C++ (with bindings for R and Python), igraph offers a *comprehensive* set of graph algorithms without sacrificing on performance. These algorithms and their uses are described in Subsection 2.8.3.

Although igraph is not as easy to interact with as APOC, the extra capabilities afforded were indispensable towards achieving the *computation* requirements of a core library of good defaults.

2.8.3 igraph Algorithms

igraph offers a comprehensive set of network-analysis algorithms. Becoming familiar with these was a challenge and I spent several hours reading Newman's *Networks* [11] in order to understand and use them correctly. Network-analysis typically centres around two broad classes of measures: centrality and community detection; both of which are described next.

I Centrality

Centrality measures offer a way to characterise a node's *importance*.

Betweenness centrality is a measure based on shortest paths. For each node (v), the number of shortest paths (from source s to target t, σ_{st}) which pass through it ($\sigma_{st}(v)$) is its betweenness centrality. When applied to mathematical theories, how "unavoidable" a given object is for the results which mention it. [4]

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.1}$$

Closeness centrality is also measure based on shortest paths. For each node, the sum of the length of all shortest paths to every other node is its closeness centrality. In a directed, dependency graph of mathematical theories, this corresponds to how "foundational" a node is. It is typically calculated as the reciprocal of *farness* ($\sum_y d(y, x)$), scaled by the number of nodes in the graph (N) to allow comparisons with graphs of different sizes. [1]

$$C(x) = \frac{N}{\sum_y d(y, x)} \tag{2.2}$$

PageRank is a variant of **eigenvector** centrality. For an adjacency matrix $\mathbf{A}$, the v^{th} component of the eigenvector $\mathbf{x}$ (whose entries must all be non-negative, corresponding to the greatest eigenvalue λ) is the v^{th} node's *relative, eigenvector* centrality. Normalising the eigenvector provides the *absolute eigenvector* centralities. [11]

The principle behind such a measure is that connections to high-ranking nodes contribute more to a node's rank than connections to low-ranking nodes. As such, for mathematical theories, it is like a *weighted in-degree/number-of-uses* which takes into account the importance of the nodes which is using a given type, proof or definition.

PageRank is similar; instead of the vector $\mathbf{x}$ such that $\mathbf{Ax} = \lambda\mathbf{x}$, for number of nodes N, random-jump probability $1 - d$, stochastic adjacency matrix $\mathbf{L}$, we want $\mathbf{r}$ which satisfies equation 2.3. [14] Hence, it can be viewed as the probability of an easily-confused, randomly-perusing mathematician coming across a given proof or definition, perhaps upon a first reading.

$$\mathbf{r} = \frac{(1-d)}{N}\mathbf{1} + d\mathbf{Lr} \tag{2.3}$$

II Community Detection

Complex networks can exhibit community structure; that is, the graph can be (roughly) divided into sparsely-connected dense groups. Although mathematical theories are often divided into sections, chapters and books, the following algorithms provide scope for re-evaluating these groupings.

Label propagation is a simple, near linear time method for determining which community a node belongs to. Each node starts with a unique label, after which, on successive iterations, it adopts the label held by most of its neighbours, until a consensus is reached. This whole procedure is repeated a few times and an aggregate result constitutes the output. [17]

Edge betweenness (like betweenness centrality), is also based on shortest paths, with the idea that edges separating communities are likely to have high edge betweenness (since all shortest paths must pass through them). Removing the edge with the greatest betweenness value and recomputing over the remaining edges successively will result in a rooted tree, a hierarchical map (called a dendrogram) where the root represents the whole graph and the leaves represent individual nodes. [12]

Modularity is a measure of how well network can be divided. Formally, it is the fraction of edges that fall within a given grouping (across the whole graph) minus the expected number of those which could have fallen within the group by chance (and so is a real number between -0.5 and 1). Calculating this in an optimal manner is an NP-complete problem, and so a fast and greedy version of it was used. [3]

2.8.4 visNetwork

Several visualisation programs exist for Neo4j; however, many are for commercial, industrial use and offer the features/complexity (and pricing) to match. All tools that offer live visualisation with built-in Cypher query execution (e.g. KeyLines, TomSawyer, Linkurious) are proprietary, requiring a fee to use and offering more granularity than required. Offline (and open-source) solutions (which require data to be exported in some manner before visualisation) such as Gephi or Alchemy.js offer similarly many features, but at the cost of a steep learning curve.

Ultimately, I chose visNetwork, (an R library exporting to JavaScript which can be rendered inside a browser) because of its simplicity and ease of integration with the previous Neo4j extensions mentioned above.

2.8.5 R

R is a statistics-oriented programming language, part of the Free Software Foundation's GNU project. It is relevant for this project because it offers an easy way to tie together Neo4j (through official bindings), igraph and visualisation using visNetwork.

To take advantage of this convenience, I had to learn R for this project, not having used it before. It is a well-documented, relatively easy to pick-up language and I learnt a lot from using it over the course of this project.

2.9 Summary

In this chapter, I gave a detailed account of how I planned this project. At the start, I mentioned the choice of development methodology (spiral) and development tools I used (Git, GitHub, Travis-CI). Following that, I presented and explained the requirements my project should meet for modelling, interacting and computing with Coq libraries. From there, I stated how the choice of technologies was dictated by the decisions to use Coq and Neo4j.

I then elaborated on my starting point for this project. I gave a description of Coq, its uses and explained why (complexity and poor documentation) it is a difficult system to work with. I also compared existing tools that aim to help a user understand a Coq library (coqdoc, coqdep, CoqSerAPI, dpdgraph) in relation to my project's requirements and showed that although none satisfied all of the requirements of my project, dpdgraph provided a limited platform to build upon.

Similarly, I gave a description of Neo4j and some of its plugins (APOC, igraph and visNetwork), explaining their features, advantages, disadvantages in relation to which of the project's requirements they met. I discussed how I needed to learn the R programming language to tie together Neo4j, igraph and visNetwork in order to meet the project's interacting and computing requirements.

3 Implementation

In this chapter, I will describe how I implemented my project. What follows is an account of the programs I wrote, problems I encountered, solutions I implemented and tests I conducted using the project structure shown in Figure 3.1 as a guide.

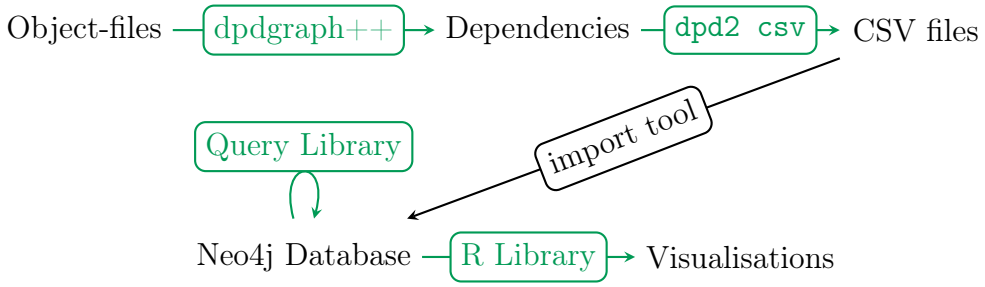

Figure 3.1 – Structure of Project. Nodes show the format of the information. Edges show the transformations. Edges in green show my contributions.

3.1 Coq object-files to CSV

This section of implementation corresponds to "dpdgraph++" and "`dpd2 csv`"on Figure 3.1: modelling the data contained in and the structure of a Coq library (of `.vo` compiled proof-scripts) as CSVs. First, I will briefly describe the algorithm I used to construct a dependency graph constructed from a compiled Coq library. Then, I will elaborate upon how I modelled the Coq library by going through the attributes of each edge and node in the dependency graph. Finally, I will describe the output format of the `.dpd` file containing the graph constructed from the compiled Coq library.

3.1.1 Algorithm

This is a high-level overview of the algorithm I used in "dpdgraph++" to construct a dependency graph from a compiled Coq library.

Input: A Coq script containing a list of files in the library (I wrote a shell-script to generate this file automatically) and the path to the compiled Coq library.

Output: A `.dpd` file containing a description of a graph whose nodes represent Coq proof-objects and whose edges represent various relationships between the proof-objects.

1. For each module, for each proof-object in the module:
   (a) if the proof-object is not in the set of nodes, add it.
   (b) for each module in the chain of modules from the proof-object's parent to the root of the library:
      i. if the module is not in the set of nodes, add it.
      ii. if there is not an edge between a module and its child in the set of edges, add it.
   (c) collect the proof-object's dependencies by recursing over its AST.
   (d) for each dependency:

i. if the dependency is not in the set of nodes, add it.
ii. if there is not an edge between the proof-object and its dependency in the set of edges, add it.
iii. otherwise, retrieve the edge and increment its *weight attribute* (representing the number of uses)

2. For each node in the set of nodes, output all of its attributes (attributes described in Subsection 3.1.2; output format described in Subsection 3.1.3).

3. For each edge in the set of edges, output all of its attributes (source, destination and weight; output format described in Subsection 3.1.3).

3.1.2 Modelling

Although I used dpdgraph as a starting point, I rewrote all of the attribute-assigning code because the meaning of the attributes it initially collected was not obvious and terminology it used did not correspond to recognisable Coq constructs.

I added the following attributes and improved the dependency-collection for inductive types and their constructors (as I will describe in III Types and Constructors on page 20).

I Precise Kinds

A proof-object can be one of the following things, corresponding to an AST term: module, class, type_constructor, inductive_type, definition, assumption or proof.

Optionally, some terms have more precise terminology, for distinguishing different constructs. For example, when writing Coq, there is no `Proof` keyword; instead `Theorem`, `Lemma`, `Fact`, `Remark`, `Property`, `Proposition` and `Corollary` all are *synonyms* for proofs. Full details can be found in Appendix A.

So to model this categorisation, I assigned each node a "kind" *label*: labels are strings used to group nodes into subsets; since a node can belong to more than one subset, it can have more than one label assigned to it. To model the more precise distinctions, I optionally assigned some nodes a "subkind" label.

II Recursive Modules

Every proof-object is contained within a module and every module is either contained in a file, is a file or is a directory. To model this inclusion relationship, analyse module-level dependencies (like coqdep) and contrast the results community detection algorithms with the module structure of a library, I augmented "dpdgraph++" to include modules in the graph. Using the Coq API, I could get the fully-qualified path (path from the library's root) of a proof-object. Using a variant datatype, I expanded the type of a node in the graph to include modules and propagated the changes throughout the project.

However, because the Coq API returned module paths as strings, modules were in the model as a flat structure: modules could be related to objects but not to other modules. I fixed this by inferring (splitting the fully-qualified path) and adding all the "ancestors" of a module with the correct relationships (parent as source, child as destination, repeatedly up to the root module).

III Types and Constructors

One of the most glaring omissions from dpdgraph's initial model was the inability to relate a type to its constructor(s). To fix this, I improved the dependency-collection code.

Expanding an AST term for type-constructors shows which type it constructed (note that types have no information about which constructors construct them). Since dependencies were constructed in a depth-first manner *down* the AST, I had to store the type and type-constructor relationship reversed but output it in the correct order.

To do so, I compared each pair of nodes to see if it (a) was a type and a constructor and if so, (b) the constructor's fully-qualified type matched the fully-qualified name of the type. If both these criteria were met, I swapped the direction of the edge output.

IV Types

Another glaring omission from dpdgraph's initial model was the inability to see the type of a proof-object. Type theory is central to a Coq user's work and being able to include them in the model, would, along with kinds, subkinds and modules, help towards meeting the modelling requirement M1 of including as much relevant data as possible.

Coq's type-checking algorithms is complex. To replicate it within "dpdgraph++", I followed the functions called for the Coq command `Check` <*expression*> (for printing the type of a given expression). This led to the algorithm for *getting* the type, which I then implemented.

A subtlety I overcame was *using* the output in `.dpd` and CSV files. I replaced newlines, quotation marks, and commas with hash signs, single-quote marks and underscores respectively, because the latter were used to delimit data in `.dpd` and CSV files and would have otherwise caused errors when they were being parsed into subsequent programs.

V Relationships

Modelling relationships was the most interesting aspect of deciding how to represent information. I wanted to keep consistent the notion of expanding a node to see more details: if a user is looking at an object, they should be able to expand

the object to see what the object depends on; if a user is looking at a module, they should be able to expand the module to see the objects contained within that module; if a user is looking at a type, they should be able to expand the type and see its constructors.

I modelled this idea with the following relationships: `(src)-[:USES]->(dst)` for dependencies, `(type)-[:CONSTRUCTED_BY]->(constr)` for (co-)inductive types, and `(module)-[:CONTAINS]->(object)` for modules.

I had two problems whilst implementing these relationships. First was finding and matching types and constructors (details of which are described in III Types and Constructors on page 20).

Second was balancing expressiveness against simplicity. I considered relationships of the following format, `X_USES_Y` for kinds X and Y. Although this was useful for fewer kinds, I decided its specificity when subkinds are included in the model made the model too large and complex (on the order of n^2 relationships). Since Cypher allows pattern-matching and filtering based on kinds and subkinds anyway, I chose the simplified model presented above.

3.1.3 Translation

This subsection corresponds to "`dpd2 csv`" on Figure 3.1. Once a model is constructed (in the form of a graph) by "dpdgraph++" and output to a `.dpd` file, it is translated by the `dpd2` tool to a CSV file for use by Neo4j's import tool to create a database. I will now present an overview of the `.dpd` and CSV formats, as well as the `dpd2` tool itself.

I `dpd` Format

"dpdgraph++" outputs the graph representing the model as a `.dpd` file with the following format for nodes (one per line):

`N: <id> "<name>" [<property>=<value>];`

for example (full type elided for brevity),

```
N: 76 "matches" [type="...  -> bool", subkind=fixpoint,
kind=definition, path="RegExp.Definitions", ];
```

and likewise for edges:

`E: <src id> <dst id> [<property>=<value>];`

for example,

`E: 150 145 [type=CONTAINS, weight=1, ];.`

II CSV

Once it is output as a `.dpd` file, a model can be translated to various other formats. For example, dpdgraph includes a tool to output a `.dot` file (a format used extensively for *visualising* graphs by many tools) from a `.dpd` file. This is what was used to generate Figure 1.1.

However, for the purpose of this project, I translated the `.dpd` file (using the `dpd2` tool described below) to *two* CSV files: one for nodes and one for edges, for use with Neo4j's import tools with the following headers.

```
objectId:ID(Object), name, kind:LABEL, subkind:LABEL, path, type
```

Here we see name, path and type declared as properties, kind and subkind declared as labels and the objectId field declared as unique identifier (or in relational terms, a key) for the nodes (in this schema, called "Objects") in the graph.

```
:START_ID(Object), :END_ID(Object), weight:int, :TYPE
```

Similarly, here we see relationships (between "Objects" as declared previously) named according to the value under the `:TYPE` column (e.g. `CONTAINS`, `USES` or `CONSTRUCTED_BY`), each with an integer property, weight.

By using a CSV format, adding extra properties, labels and relationships becomes very straightforward and allows for easy integration (with external tools) and extension for new features.

III `dpd2` Tool

Initially, this tool started out as the `dpd2dot` utility (bundled with dpdgraph). I refactored it into a more general, `dpd2` tool which could accept the file-type (`.dot` or CSV) as a command-line argument.

In using this tool, I discovered it had a bug. By default, `dpd2` attempts to remove reflexive and transitive dependencies (i.e. $a \to a$ and removing $a \to c$ if $a \to b$ and $b \to c$) in a depth-first manner. For large graphs, doing so is stack-intensive, and thus causes the aforementioned error. Since this "feature" was unnecessary (because it *removed* useful information) and appeared time-consuming to fix, I passed `-keep-trans` flag on subsequent uses to avoid the issue altogether.

3.2 CSV to Neo4j

The "import tool" in Figure 3.1 is a command-line program that is included with Neo4j. When given a target directory, a CSV file containing the nodes of the graph and a separate CSV file containing the edges of the graph, this import tool constructs a database. As explained in II CSV on page 22, the headers determine the labels, properties, IDs of nodes as well as the start- and end-points, properties and type of edges.

3.3 Query Library

This section of the implementation corresponds to the "Query Library" (including the "R Library") shown in Figure 3.1. I will now outline the extensions I included to meet the interaction (I1, I2, I3) and computation (C1, C2) requirements on page 6.

3.3.1 Java Library

Briefly, APOC (Awesome Procedures on Cypher) provides, out of the box (by placing a jar file into a database's plugins directory), many convenient, utility functions to use. Of note are:

- the ability to examine a *meta-graph* showing which labels and relationship-types are available in the database and how they are connected, allowing a user to see an overview of the model;
- a few key graph algorithms, such as node and path expansion, spanning tree, Dijkstra's shortest paths, A*, label propagation (for community detection), centrality measures (betweenness and closeness) and PageRank;
- some utility functions for regular-expressions and mathematics, useful for selecting nodes based on statistics (e.g. PageRank) or selecting nodes based on their path.

Each of these can be called directly from within Cypher; for example, writing `call apoc.meta.graph()` would return the meta-graph of the database.

3.3.2 R Library

For the R side of the library of queries, to meet requirement C1 for a core set of good, out-of-the-box defaults, I wrote several example programs written which automatically processed data from a new database, stored the information again for later use (to avoid re-computation) and output appropriate visualisations. Since processing could take on the order of minutes, I also provided status updates, informing the user of the task being executed, the time it took to execute once complete, as well as progress-bars where possible and relevant (e.g. committing a transaction to the database).

I Visualisation

There are many interesting ways to visualise the plethora of data that analysing large mathematical theories in the form of Coq libraries using graph databases makes available.

Simply plotting the density/spread of and correlation between metrics such as in- and out-degrees, PageRank, centrality measures is a good start and can occasionally yield useful information. Here, R's strength as a statistics-oriented programming language shone through, making it easy for me to rapidly explore different ideas.

Another, more direct, way of displaying information is by displaying the graph itself. As I will show in Section 4.3 Library of Queries, Library of Queries, there are a number of different layouts possible (layered, circular and force-directed), each contributing to a more comprehensive, overall, picture of understanding the theory.

Surprisingly, igraph was able to help with this aspect of the project as well. Whilst visNetwork – with its own JavaScript, force-directed, physics rendering – produced more aesthetically pleasing results for smaller graphs, the webpages it output for larger graphs took intolerably long to render inside a typical web-browser (Firefox/Opera). Thanks to an (experimental) integration with igraph (specifically, igraph's layout mechanisms), I could pre-compute graph layouts in fast, native C/C++ *before* rendering graphs in a web-browser.

3.4 Project Related

During implementation, I learn several skills and lessons about correct project management. Small things, such as grep-ing a code base or keeping track of time and a log of work done, proved to be useful. However, to ensure the project ran smoothly on a larger-scale, I focussed more on the following areas.

3.4.1 Testing

For most of the project, I conducted testing manually inspecting output. Though this was tedious, it was the only way to do so when the model was undergoing continual development. As soon as the model was settled and focus shifted to visualisation, I used automated tests (which came in particularly useful when the project had to be *bifurcated* to support two different versions of Coq).

I used some of my older Coq proof-scripts – of problems I solved when learning Coq – to check output on a small scale (where I knew every single aspect of the library, and testing turnaround was quick). I used Coq's Standard Library as a large-scale stress-test, to ensure all constructs were translated correctly. When I did find problems, they could usually be traced back to output from debug statements I had embedded into "dpdgraph++".

3.4.2 Continuous-Integration Builds

Although the project never failed to build locally, there were occasionally problems when trying to build it on the project supervisor's machine. The problem was compounded when it became apparent that smaller libraries (such as CoqRegExp and the solved problems) relied on a version of Coq (8.5.2) older than the one Mathematical Components (needed for the project's moonshot, the Odd Order Theorem) relied on (8.6). I set up Travis-CI for continuous-integration builds to make version dependencies precise and explicit, and removed much of the hassle and uncertainty surrounding builds on other machines.

3.4.3 Tooling

At first, I ran the project half on Windows and half on a Linux VM (virtual machine), using shared folders. I had a number of reasons for this: I had already set up Coq and Neo4j on Windows, both are easier to interact with in a graphical environment and starting up a VM just for some experimentation took too long.

Eventually, this set-up became confusing and time-consuming and I made the leap to running the project fully on a Linux VM. Nevertheless, I had serious issues when working on R integrations: running a Java database inside a Linux VM was insufferably slow and switching databases was not easy. I solved this problem by setting up SSH reverse-port-forwarding (to let R inside the Linux VM connect to a Neo4j instance running directly on Windows) for decent performance.

To be the most productive during longer sessions of work, I also set up editor integrations for OCaml, dramatically reducing the edit-compile cycle (especially for a strong, statically-typed programming language as OCaml). At this point, Coq's use of non-standard OCaml features, extensions and build-systems became particularly frustrating. For example, I spent a considerable time was spent untangling the Makefile inherited from dpdgraph to have cleaner, out-of-source builds and reduce the mess in the current working directory, all to no avail as I realised how complex of the build-system for Coq plugins is.

3.5 Dead-ends

I have now finished describing the whole project; all parts of Figure 3.1. I will now explain why I did not further pursue a couple of seemingly obvious, alternative strategies to modelling Coq libraries. Both centre around the same theme: analysing the Coq source-files directly instead of compiling them and then analysing the `.vo` files.

3.5.1 Coqdoc

I tried to modify coqdoc's output into a useful format but this did not prove fruitful because the purely lexical tokenisation coqdoc does cannot infer or preserve as much information as full parsing.

3.5.2 Coq source-files to CSV

So, why not try parsing the source-files directly? This is an appealing idea, because with my current approach, there is still some information being lost during compilation: there is an *apparent* absence or duplication of some modules and a lack of notation and tactics.

The former arises from the use of *functors*: modules that take other modules as arguments (used to abstract over arguments, tactics, definitions and proofs). A concrete example (from the Coq Standard Library) is the theory of total orders, minimums and maximums, which is applied to naturals, integers and rational numbers (as well as any further ordered types). Such functors cannot be compiled unless fully applied (explaining the *absence* of some modules); when fully applied, they essentially copy their *structure* into each instance (explaining the *duplication* of some module names and structures).

I Exploring Solutions

However, *several* issues stop parsing from being a pragmatic solution, least of which are the size of the AST and complexity of parsing Coq files (Coq uses an *extensible grammar* by using non-standard OCaml tools, making the whole situation quite ugly to work with).

1. *Modules and functors are represented identically* in the AST, with the former as a special case of the latter, making it very difficult to distinguish between the two on an AST-level. By far, this was the biggest roadblock.
2. *Module types*, or *signatures*, must be incorporated into the model for functors to make sense. Modules and signatures share a many-to-many relationship: a signature can be satisfied by multiple modules and a module can satisfy multiple signatures. To express this correctly would require *signature-matching*, a notoriously difficult task (and the reason why few languages support ML-style modules).
3. Further issues involve resolving objects into a global namespace and knowing which compiler flags were given during compilation (to match physical directories to logical modules), all of which complicate matching and merging with the compiled proof objects.

Even though I had made some progress in tackling these issues via parsing, I kept looking for a simpler solution. *Glob files*, produced during compilation, retain

much of the information in the source code (as a list of globally-resolved names and paths for each file) in a simpler format. I prototyped a tool (in the form of a shell-script) that converted glob files to CSVs. Although it was promising for tactics and notation, glob files suffer from the same inability to distinguish between modules and functors, so I abandoned it.

II Resolution

Both direct-parsing and glob-file approaches to notation, tactics and functors were limited and time-consuming. Functors are rarely-used with many projects (especially given the popularity and ease of use of *type-classes* for expressing generalisations) and thus risked violation requirement M1 (by (a) potentially obfuscating information through their inclusion in the model and (b) making the model more difficult to use).

Also, in light of requirement M3, given that names *and structures* of instantiated modules are duplicated, it could be possible to *reconstruct* generalisations *representing functors* once the database is created. As such, extracting information directly from Coq source-files contributed little to the overall project, but presents clear ways forward for future-work.

3.6 Summary

I just presented an in-depth account of the programs written (dpdgraph++, dpd2, Query & R libraries), problems encountered (recursive modules, relating types and constructors, incorporating type signatures, CSV translation, dependency and version tracking, impenetrable build-systems and parsing information directly from Coq source-files), solutions implemented and tests conducted. Throughout, I referenced the project requirements to justify important decisions.

4 | Evaluation

In this chapter, I will show that the project meets its aims (as listed in Section 1.3) by presenting the evidence for this claim. I will do so by (a) comparing the project with existing tools that aim to help a user understand a Coq library and (b) providing sample output and explaining the insights they provide.

```
MATCH (a)-[:USES]->(b),
      (src:module)-[:CONTAINS]->(a),
      (dst:module)-[:CONTAINS]->(b)
WHERE src.objectId <> dst.objectId
CREATE UNIQUE (src)-[r:DEPENDS_ON]->(dst)
SET r.weight = coalesce(r.weight, 0) + 1
RETURN r
```

Listing 1 – Query to set Module Dependencies

4.1 Features

I talked through most of the first aim (how I represent Coq libraries as Neo4j databases) in the Implementation chapter. To asses the capabilities of my project, and thereby the suitability of my chosen model, I will compare all of the programs listed in Subsection 2.6, Existing Tools for Coq with this project against the features listed there (features chosen to reflect strengths of each tool considered).

The project *can be extended* to support **linking to source code** by modifying either the model, database or JavaScript visualisations (output by the library of queries) to link to the relevant webpages output by coqdoc. So, a user could switch between a graphical overview and a detailed inspection at will.

Whenever a node is visible, it can be expanded to see the nodes it depends on, so in that sense, it supports **hyperlinks**, though, unlike hyperlinks, such expansion is done in place, thus retaining the *context* of its use.

Thanks to the *kind* and *subkind* labels, the project supports **precise kinds**. Also, any (co-)inductive **type** can, via the `CONSTRUCTED_BY` relation, be expanded to see its **constructors**.

Due to the **type** property, each object's type is also visible. Crucially, it is a fully-expanded type signature, making explicit any assumptions introduced (perhaps hundreds of lines prior or in a different file) into the environment.

Module dependencies are set with the query in Listing 1 by use of the `CONTAINS` relation. **Interactivity** is achieved through the Neo4j browser interface and the JavaScript visualisations; **statistics** are achieved through the library of queries; **graphical representations** by both.

An important limitation of CoqSerAPI is that its **statistics** are (at the time of writing) simply three counters, whereas this project offers many sophisticated graph metrics and the ability (through a queriable database) to gain *any* sort of information a user is interested in.

And finally, **object dependencies** are at the heart of the project: by using a Neo4j graph database, a user can understand and manipulate this relation in a much more flexible and scalable manner than any visualisation can manage.

Therefore, the project either supports, or can be extended to support, every feature supported by other tools.

4.2 Performance

Now, I will compare the project's execution time to most of the tools from the previous section. Although the project is slightly slower than other existing tools, this can be directly accounted for by its larger feature set and increased flexibility.

4.2.1 Setup

To evaluate timings, I used the Coq (8.6) Standard Library, due to its sheer size (564 modules, 5823 definitions, 23,892 proofs). For coqdoc and coqdep, Coq's Makefiles were modified to measure execution time using bash's *time* command. At the time of writing, CoqSerAPI's statistics were not fully/usably implemented, so I did not include in this comparison. For dpdgraph, I took separate measurements for outputting a `.dpd` file and converting that file to a `.dot` format. I took a similar approach for this project, so that the comparison was as fair as possible. Each set of timings was repeated five times (with the exception of `dpd2dot` which, at nearly *27 minutes*, was not repeated because of time and system-usability constraints).

4.2.2 Inefficiencies

Setting up a graph database from scratch can take some time. Assuming a Coq project is already compiled, the following steps need to take place:

1. generating file with a list of all the modules to be examined (15 ms),
2. compile that file using the Coq compiler (12.6 s),
3. convert the output `.dpd` file to CSV files (72.7 s),
4. create a Neo4j database from those CSV files (12.8 s)

When steps 1 and 2 are taken on their own, we see that the changes to accommodate a more detailed model resulted in a 25% slowdown, which is acceptable given this is a stress-test and the difference is on the order of seconds. Creating a database from CSV files takes a similar amount of time, also acceptable given the size of the graph (31,088 nodes and 850,434 edges).

So, the real bottleneck is step 3: converting `.dpd` files to CSV. During execution, `dpd2` reads in a 25MB `.dpd` file and outputs two CSV files of size 7MB (nodes) and 8MB (edges), so IO is likely to be a factor, as is reconstructing the graph in memory. I output the model to a `.dpd` to make it easier to extend this project with other tools. It is likely that outputting a CSV directly would have resulted

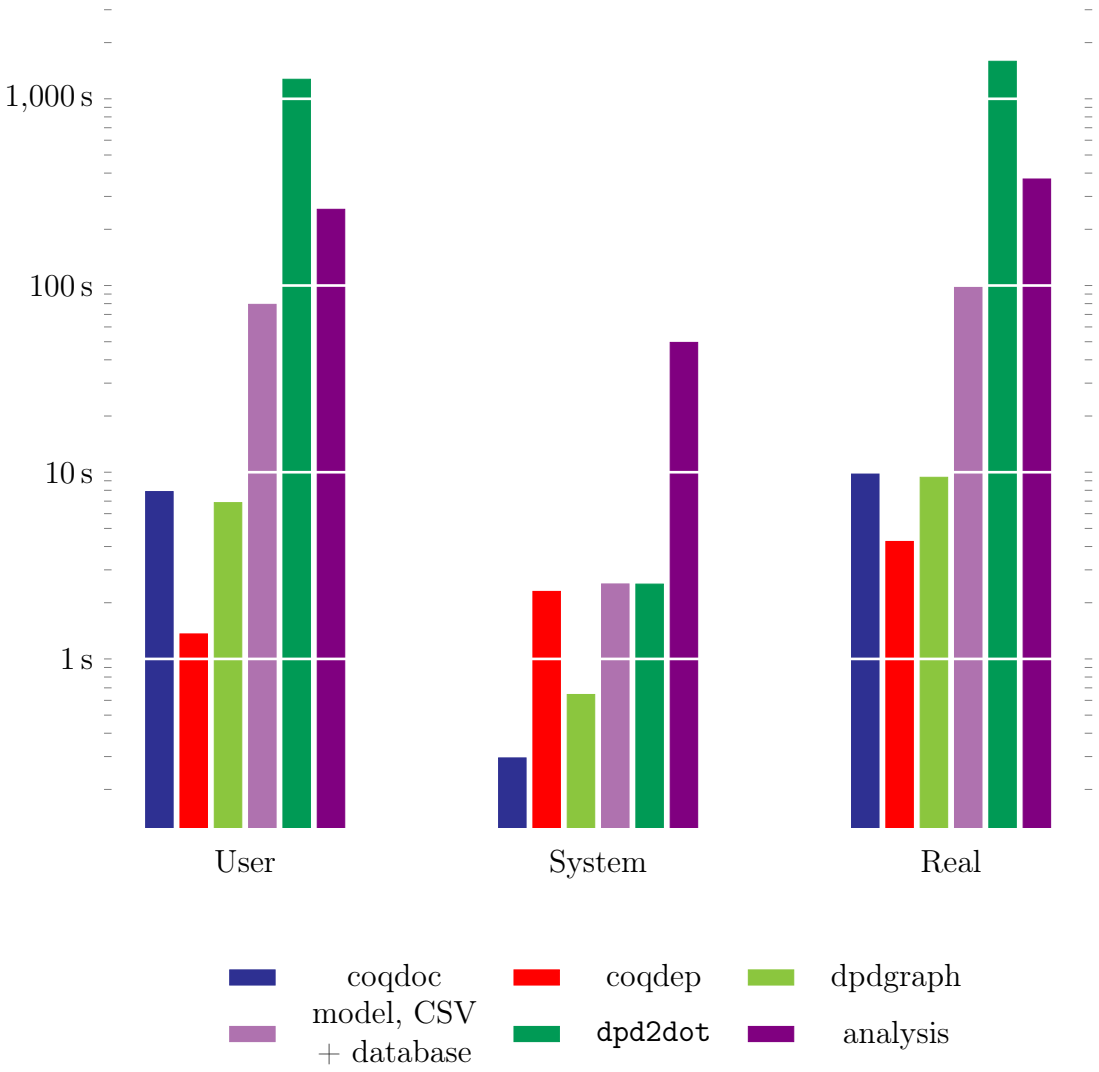

Figure 4.1 – Comparison of Execution Times. Note that the data is presented on a *logarithmic* scale. We see coqdoc takes very little time to run, and coqdep even less (which is to be expected considering their purely lexical approach). Somewhat surprisingly, dpdgraph runs just as quickly as coqdoc; its increase in system time can be explained by the 14MB dpd file output by dpdgraph. Although at first it appears that there is an order-of-magnitude slowdown with this project, more detailed examination in Subsection 4.2.2 explains precisely *what* occurs during database creation and where inefficiencies lie.

in being able to bypass this phase altogether, but from a software-engineering point-of-view, the trade-off there is increased coupling.

4.2.3 Graph Analysis & Visualisation

Once a graph is created (whether it be in the form of `.dpd` file or a database) the last step is to *use* the data by analysing and visualising it. Here, this project shows a significant improvement over `dpd2dot`.

At nearly *27 minutes*, its execution time dwarfs the analyses carried out by this project. Whereas `dpd2dot` converted the output 13MB `.dpd` file to a 24MB dot file, in about one-quarter of the time, an R script was able to run (a) PageRank and closeness centrality algorithms over all proofs and definitions and (b) output *8* different 9MB visualisations of the data. Analyses took less than a minute; visualisations ranged from 20 to 90 seconds.

It should be noted that `dpd2dot` does not do graph *layout*: it just splits the graph into sub-graphs (based on modules) and assigns a colour and label to each node (based on their properties). Converting the `.dot` file to a viewable format (e.g. a scalable vector-graphic or SVG) is up to another tool (that being said, I cancelled the command `dot -Tsvg` to produce an SVG after it failed terminate within a few *hours*).

4.3 Library of Queries

I talked through how I provide several-network analysis techniques in the Implementation chapter. To assess whether this tool highlights the structure of and relationship between proof-objects (the second aim), I will show the output of the library of queries on the small case of a Coq Regular-Expression library and on the large case of the project's moonshot, the Odd Order Theorem.

All visualisations show only definitions (shown as triangles, reminiscent of the $\triangleq$ symbol sometimes used for definitions) and proofs (shown as squares, reminiscent of the end-of-proof $\square$ symbol), except for Figures 4.4 and 4.6 which also includes modules (as circles).

4.3.1 Small: CoqRegExp

I will now show, how, without studying any code, a Coq user can use this project to understand the structure of the CoqReqExp library.

I Neo4j & APOC

As visible by the examples in Tables 4.1 and 4.2, and Figure 4.2, a user can make arbitrary queries on the CoqRegExp library database. The first gives an indication

```
MATCH (obj) RETURN LABELS(obj), count(*) AS total
ORDER BY total DESC LIMIT 5
```

LABELS(obj)	total
proof, lemma	79
definition	17
proof, theorem	16
definition, instance	14
type_constructor	8

Table 4.1 – Top 5, most common kinds of proof-objects in CoqRegExp with their frequency and the query used to obtain them above the table.

```
MATCH (n) WITH collect(n) AS nodes
CALL apoc.algo.pageRank(nodes) YIELD node, score
RETURN node.name, LABELS(node), node.path, score
ORDER BY score DESC LIMIT 5
```

node.name	node.path	LABELS(node)	score
RegExp	RegExp.Definitions	inductive_type, inductive	7.12518
matches	RegExp.Definitions	definition, fixpoint	3.12445
Or	RegExp.Definitions	type_constructor	2.35997
re_eq	RegExp.Definitions	definition	2.30212
Cat	RegExp.Definitions	type_constructor	2.04202

Table 4.2 – Top 5 proof-objects by PageRank in CoqRegExp, with the modules they are in, their kinds, their PageRank values, and the query used to obtain them above the table.

of the most common kinds of proof-objects: mostly lemmas, with definitions and proofs following. The second uses an APOC procedure for PageRank over all proof-objects to quantify the importance of each. The definition of a regular-expression and the definition of what it means for a regular-expression to match a string top the table, with other, fundamental theory components/proof-objects following. The third also uses an APOC procedure to produce a visual representation of the schema of the database.

II Visualisation

Intuitively, in Figures 4.3 and 4.4, the light blue cluster represents *executable functions*; the orange cluster represents *proofs of correctness*; the light purple cluster represents proofs using the definition of string length; the light green cluster represents proofs involving converting strings to regular-expressions; the violet cluster represents proofs relating to nullable strings. The node at the centre of the light blue cluster is a function which computes whether a given regular-expression

```
CALL apoc.meta.graph
```

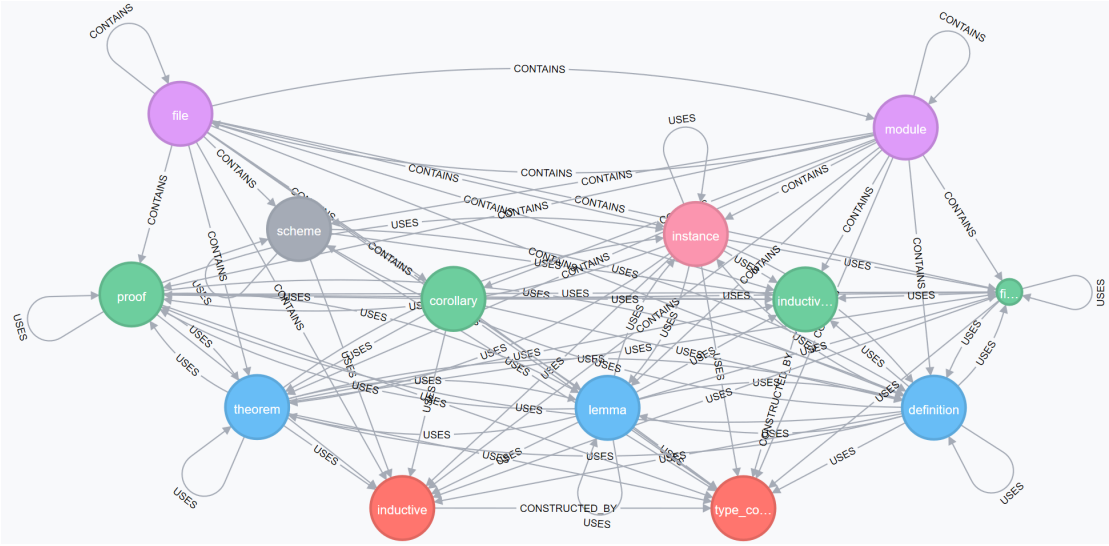

Figure 4.2 – Meta-graph of CoqRegExp representing the labels (kinds/subkinds) and relations (`USES`, `CONTAINS`, `CONSTRUCTED_BY`) between them. Above the figure is the Cypher query which calls the APOC procedure to produce this. At the bottom, red nodes represent types and constructors; above them, blue nodes represent definitions; above them, green nodes represent proofs. The grey node represents a scheme; the pink node represents an instance and the two purple nodes at the top represent files and modules.

matches a given string; the node at the centre of the orange cluster defines what it means for two regular-expressions to be equal.

4.3.2 Large: Odd Order Theorem

This Coq library closely follows the structure of the source material it encodes: Peterfalvi [15] and Bender & Glauberman [2]. Each section (chapter) in the original books is a file/module in the Coq library; each definition/lemma/proof/corollary corresponds to the same in the books. Following the convention in the Coq library, 'Bender & Glauberman' will henceforth be abbreviated to 'BG' and 'Peterfalvi' to 'PF'. For brevity, 'Odd Order Theorem' will also be abbreviated, to 'OOT'.

I will now show, how, without studying any code, or reading BG or PF, a Coq user can use this project to understand aspects of BG and PF.

New to this section, edges are coloured. Unless a figure is stated as being 'flipped', the colour of an edge is the same colour as its source (user); otherwise, it is the same colour as its destination (used).

I Neo4j

By modelling Coq libraries as Neo4j graph databases, we can answer powerful questions. For example: there are 1064 proofs in BG and PF, *does every one of*

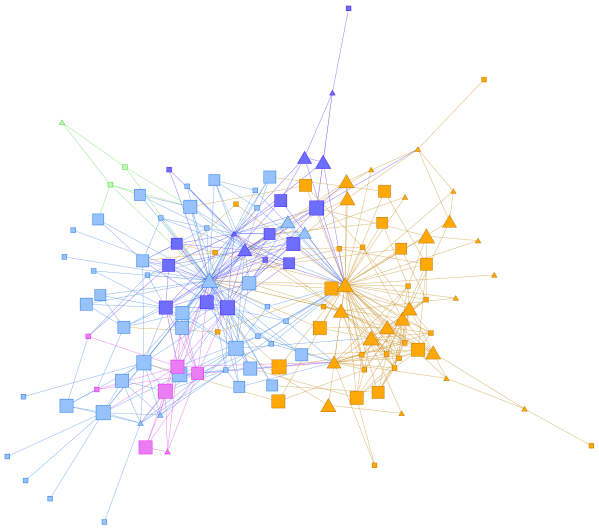

Figure 4.3 – Force-directed visualisation of definitions and proofs in CoqRegExp. Colours assigned by modularity clustering largely correspond to the how a human might group nodes visually: two major clusters with a few, smaller clusters (see II Visualisation for an interpretation of what each colour corresponds to). The size of the nodes correspond to betweenness centrality scores (split up into 10 logarithmically equal-width buckets). Edges represent the `USES` relation; edge-directions (source-uses-destination) are omitted for clarity since they generally point toward the centre of a cluster.

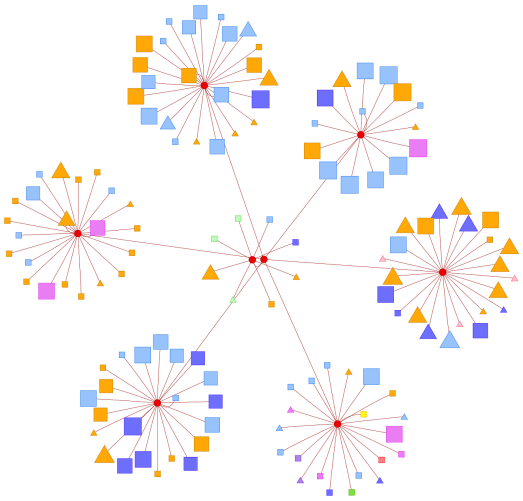

Figure 4.4 – Force-directed visualisation of definitions, proofs and modules in CoqRegExp. Setup is similar to Figure 4.3 above, except edges represent the `CONTAINS` relation. Each module tends to have two main parts: those relating to executable functions those relating to proofs of correctness, occasionally accompanied by a few definitions/proofs on nullability or string length.

```
MATCH (q:proof)
WHERE NOT (:proof {path: "mathcomp.odd_order.PFsection14",
                   name : "Feit_Thompson"})
          -[:USES*]->(q:proof)
RETURN q.name, replace(q.path, "mathcomp.odd_order.", "") AS path
```

q.name	path
pcore_Fcore	BGsection15
main	stripped_odd_order_theorem...
ell_sigma_leq_2	BGsection14
Ptype_trans	BGsection14
P1type_trans	BGsection14

Table 4.3 – Five (of 89), proofs the Odd Order Theorem Coq library which do not ultimately lead to the proof of the Feit-Thompson Odd Order Theorem. Module names (path property) have been shortened to remove redundant information. The "stripped_odd_order_theorem" is a self-contained proof relying on only basic Coq features and is not part of BG or PF.

them ultimately lead to the proof of the Feit-Thompson OOT? In general, how many "dead-ends" – proofs, definitions or types – are there in these two books, and where are they? These questions have been answered in Tables 4.3 and 4.4 respectively.

II Visualisation

In Figure 4.5, we see that modularity clustering over proofs and definitions connected by the `USES` relation is able to distinguish/explain *some* or the grouping achieved by modularity clustering but not the amalgamation of all the groups in the central cluster.

Figure 4.6 shows visually what one might expect intuitively: *proofs and definitions within the same module tend to belong to the same group*. However, it does not explain why the groups *cross* module boundaries and represent *multiple* modules.

Figure 4.7 explains why the groups cross module boundaries: they demarcate different parts/phases of BG and PF. Both the figure and the list below show this Coq library is not quite the linear chain of dependencies ranges one may expect. Instead, the overlapping ranges of these groups suggest that the material is heavily interconnected and that there are other ways of approaching the material other the linear presentation of BG and PF.

- **pink:** BG 1-6 and BG Appendices A/B/C
- **dark green:** BG 7-13 and PF 9-10 and 12-14
- **purple:** BG 14/16 and PF 3-4/8

```
MATCH (q) WHERE NOT((q:module) OR ()-[:USES]->(q))
RETURN LABELS(q),
       REPLACE(q.path, "mathcomp.odd_order.", "") AS path,
       COUNT(*) AS total
ORDER BY total DESC LIMIT 7
```

LABELS(q)	path	total
definition, scheme	stripped_odd_order_theorem	27
proof, lemma	BGsection16	6
proof, lemma	BGsection15	5
proof, lemma	PFsection8	5
proof, lemma	BGsection10	4
proof, remark	BGsection14	4
proof, lemma	PFsection5	4

Table 4.4 – Top 7 kinds of proof-objects in the OOT Coq library which are never used again (of which there are 107), grouped by module and ordered by frequency. Module names (path property) have been shortened to remove redundant information. The "stripped_odd_order_theorem" is a self-contained proof of the entire OOT relying only on basic Coq features and is not part of BG or PF. Many unused results are simply called 'lemma' instead of the more descriptive and conventional 'corollary' or 'remark'.

- **yellow:** CyclicTIisoReflexion in PF 3
- **magenta:** BG 15 and PF 11
- **dark blue:** PF 1-2 and 5-7.
- **light blue:** self-contained proof of the OOT.

Figure 4.10 shows the hierarchical, Sugiyama layout. A consequence of this algorithm is that nodes are placed closest to their first use, usually done as a heuristic to minimise edge crossings. Intuitively, one can think of this approach as the lazy/ultra-efficient student who only studies a particular definition/proof just before it is needed by some other part of the text (or like call-by-need evaluation in a non-strict programming language).

As such, swapping the direction of the edges does not simply flip the layout vertically, as Figure 4.11 shows. One could conjecture that Figure 4.10 shows how mathematics is developed (a narrow foundation, top-down refinement, creating proofs/definitions as and when needed, several independent lines of thought) and Figure 4.11 shows how mathematics is presented (a broad base of all the groundwork that will be needed first, followed by a bottom-up, rapidly developing, linear argument which uses the base extensively).

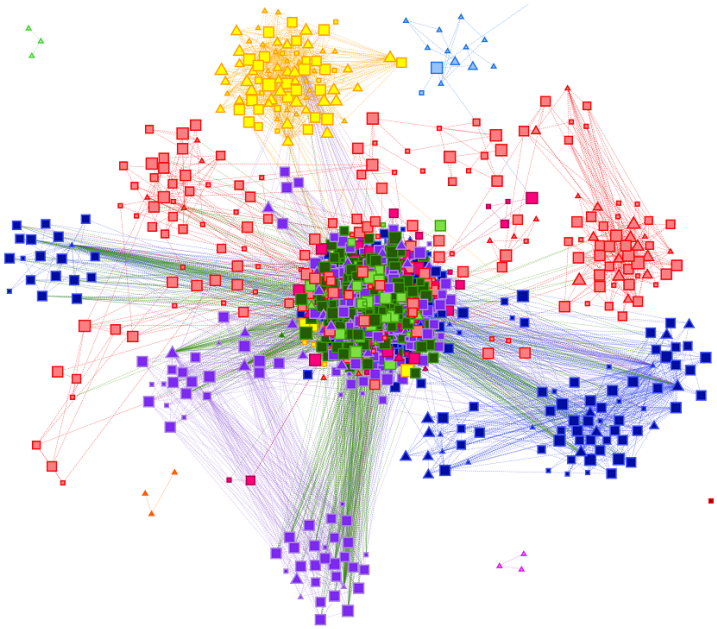

Figure 4.5 – Force-directed visualisation of definitions and proofs in the OOT Coq library (some nodes omitted for clarity). Setup is the same as that of Figure 4.3. Observe that modularity clustering is able to distinguish between nodes in the centre as belonging to different groups. This distinction is more evident in Figures 4.6, 4.7, 4.8 and 4.9; it is explained in II Visualisation.

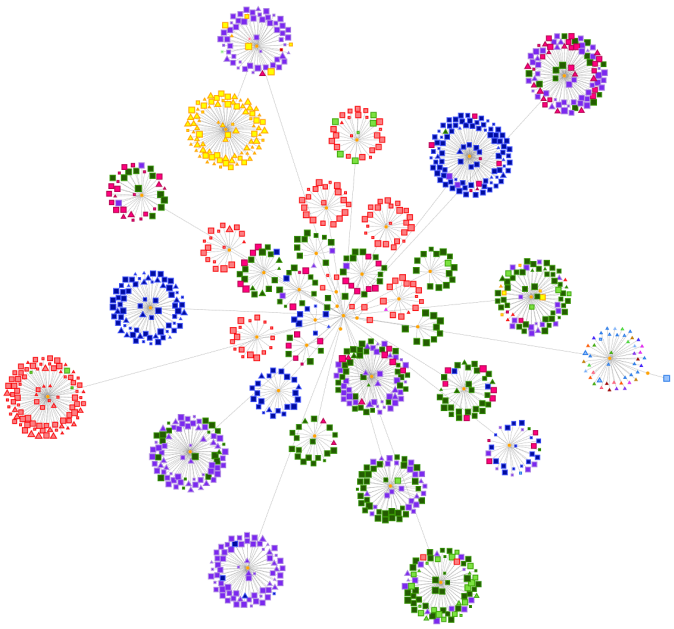

Figure 4.6 – Force-directed visualisation of definitions, proofs and modules in the OOT Coq library. Setup is the same as that of Figure 4.4. The module in yellow – a self-contained result by the name of 'CyclicTIisoReflexion' – is the only one *nested* inside another (PF 3). Unlike CoqRegExp, here, proofs and definitions in each module tend to belong to the same group. Note that groupings *cross* module boundaries; this is elaborated upon in II Visualisation.

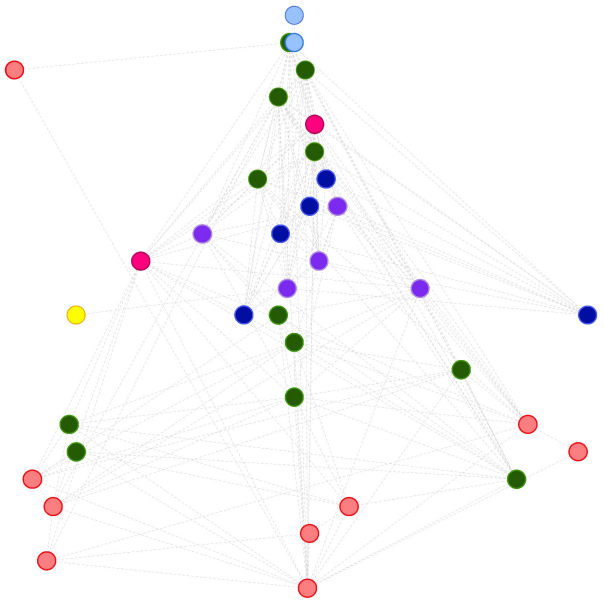

Figure 4.7 – Sugiyama (hierarchical) layout (the direction of edges always points downwards, above `DEPENDS_ON` below) of **modules** in the OOT Coq library. Modules are coloured according the most frequent (mode) colour of all definitions and proofs in it. Edges represent the `DEPENDS_ON` relation between modules defined in Listing 1. This visually represents the chapter dependencies in BG and PF, starting with BG 1 at the bottom and ending with PF 14 (containing the actual proof of the Feit-Thompson OOT) at the top.

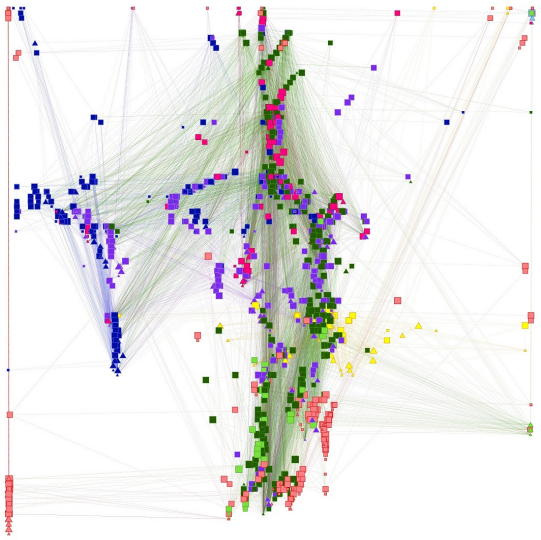

Figure 4.8 – Sugiyama (hierarchical) layout (the direction of edges point downwards) of definitions and proofs in the OOT Coq library. Colours are assigned by modularity clustering. Edges represent the `USES` relation: above *uses* below. A by-product of this layout is that nodes are are placed higher/closest to their *users*.

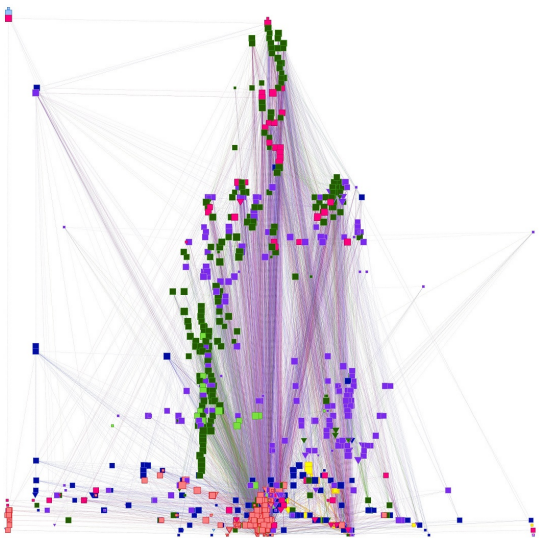

Figure 4.9 – Setup is the same as Figure 4.9, except that (a) the edge directions are *reversed* and (b) the image is reflected vertically. It is still the case that above *uses* below. However, doing this has the effect of placing a node lower/closest to the nodes *using* it. An interpretation of this is provided in II Visualisation.

4.4 Summary

To assess the two major aim of this project (choosing the correct model and being able to gain insights into mathematical theories), its features, performance and output were examined in detail. For its features, a systematic comparison with other, existing tools of a similar purpose, showed this project went above and beyond what existing tools offer. For its performance, a similar comparison of execution times showed this project to be slightly slower, but still acceptable and usable for even the most extreme of cases. For its output, two Coq libraries, each representing a mathematical theory, were explained via the insights gained from using this project on them.

5 | Conclusions

5.1 In Hindsight

A lot was learned throughout the course of this project. Working with existing systems *and* creating something novel can be difficult to juggle mentally: the cognitive load of imagining what is possible within the given frameworks and timescales is a skill, one which was refined over the course of the project. Initially, the idea, suggested by the project supervisor, seemed fascinating and intriguiging. After the preliminary preparations, the novelty, difficulty and usefulness of the concept had taken root.

However, at several instances throughout implementation, it was easy to get stuck in the details and spend hours staring at the entire Coq compiler code base just to figure out how to get the name of the module an object is contained in or how to convert some part of the AST to a string for use elsewhere; the lack of good, structure documentation made the whole endeavour even more frustrating at times. Other times it was simply the bewildering choice of how best to proceed next given many options and no clear way of comparing between them. It was rarely *writing* the code which was the issue, but *knowing what to aim for*. It was in those moments that having an experienced, focused and clear supervisor who reiterated the overarching vision of the project was the most useful.

Project management and organisation tended to be easy and helped immensely in providing a useful structure for guiding implementation. A good grasp of Unix, Git and programming languages proved to be essential to executing this project as smoothly as possible. Evaluating the project gave a chance to switch from *developing* to *using* and explore what was possible with the project in its final form, and how it could be improved and extended given more time.

5.2 Future Work

Immediate extensions to this project could come from fleshing out the model further, to include tactics, notation and other aspects of the Coq system. More ambitious extensions could involve somehow presenting some information to a user *as they are working* on something, or using data science techniques to analyse the output model as a means to providing new insights or a more helpful compiler

which can suggest how to (re)structure a project. Although very useful and easily interpretable in the context of mathematical theories, this project's core concept could also be applied to any programming language and its libraries, to provide new ways of becoming familiar with and understanding increasingly complex software projects.

Bibliography

[1] Alex Bavelas. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.

[2] Helmut Bender, George Glauberman, and Walter Carlip. *Local analysis for the odd order theorem*, volume 188. Cambridge University Press, 1994.

[3] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[4] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[5] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[6] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[7] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[8] Xavier Leroy. The compcert c verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[9] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[10] Neo4j. Neo4j. neo4j.com. Accessed: 13/10/2016.

[11] Mark EJ Newman. The mathematics of networks. *The new palgrave encyclopedia of economics*, 2(2008):1–12, 2008.

[12] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[13] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[14] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[15] Thomas Peterfalvi. *Character theory for the odd order theorem*, volume 272. Cambridge University Press, 2000.

[16] Benjamin C Pierce. The science of deep specification (keynote). In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 1–1. ACM, 2016.

[17] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

A | Full Model

Here is the full translation of the Coq AST to kinds and subkinds.

```
let kind_of_constref =
let open Decl_kinds in function
| IsDefinition def -> ("definition", Some (match def with
  | Definition -> "definition"
  | Coercion -> "coercion"
  | SubClass -> "subclass"
  | CanonicalStructure -> "canonical_structure"
  | Example -> "example"
  | Fixpoint -> "fixpoint"
  | CoFixpoint -> "cofixpoint"
  | Scheme -> "scheme"
  | StructureComponent -> "projection"
  | IdentityCoercion -> "coercion"
  | Instance -> "instance"
  | Method -> "method"))
| IsAssumption a ->
  ("assumption", Some (match a with
  | Definitional -> "definitional"
  | Logical -> "logical"
  | Conjectural -> "conjectural"))
| IsProof th ->
  ("proof", Some (match th with
  | Theorem -> "theorem"
  | Lemma -> "lemma"
  | Fact -> "fact"
  | Remark -> "remark"
  | Property -> "property"
  | Proposition -> "proposition"
  | Corollary -> "corollary"))

let kind_of_ind ind =
let (mib,oib) = Inductive.lookup_mind_specif (Global.env ()) ind in
if mib.Declarations.mind_record <> None then
  let open Decl_kinds in
```

```
35    begin match mib.Declarations.mind_finite with
36    | Finite -> "recursive_inductive"
37    | BiFinite -> "recursive"
38    | CoFinite -> "corecursive"
39    end
40  else
41    let open Decl_kinds in
42    begin match mib.Declarations.mind_finite with
43    | Finite -> "inductive"
44    | BiFinite -> "variant"
45    | CoFinite -> "coinductive"
46    end
47
48  let get_constr_type typ =
49    Names.KerName.to_string (Names.MutInd.user typ)
50
51  let kind_of_gref gref =
52  if Typeclasses.is_class gref then
53    ("class", None)
54  else
55    match gref with
56    | Globnames.ConstRef cst ->
57    kind_of_constref (Decls.constant_kind cst)
58
59    | Globnames.ConstructRef ((typ, _), _) ->
60    ("type_constructor", None)
61
62    | Globnames.IndRef ind ->
63    ("inductive_type", Some (kind_of_ind ind))
64
65    | Globnames.VarRef _ ->
66    assert false
67
68
69  let kind_of_obj = function
70  | G.Node.Gref gref ->
71    kind_of_gref gref
72  | G.Node.Module modpath ->
73        ("module", match modpath with
74        | Names.ModPath.MPbound _ -> Some "bound"
75        | Names.ModPath.MPdot _ -> Some "module"
76        | Names.ModPath.MPfile _ -> Some "file")
```

B | Project proposal

Computer Science Tripos – Part II – Project Proposal

Exploring the structure of mathematical theories using graph databases

Dhruv C. Makwana, Trinity College

Originator: Dr. Timothy G. Griffin

Project Supervisor: Dr. Timothy G. Griffin

Directors of Studies: Dr. Frank Stajano & Dr. Sean B. Holden

Project Overseers: Dr. David J. Greaves & Prof. John Daugman

Introduction and Description of the Work

This project aims to (a) represent Coq libraries as Neo4j (graph) databases and (b) create a library of Neo4j queries with the goal of highlighting the structure and relationship between the representations of the proof-objects.

Mathematics textbooks aimed at professionals/researchers follow a well-established rhythm: define some constructions and some properties on them and prove theorems on both, with lemmas, corollaries and notation interspersed throughout. Such a presentation is concise but limiting: it is linear; it forces the reader to keep track of dependencies such as implicit assumptions, previously defined results and the types and conventions behind any notation used; and it offers little opportunity to consider and compare different approaches for arriving at a result (i.e. number of assumptions, number of steps, some notion of the importance of a result such as number of uses by later results).

With the increasing popularity of interactive theorem-provers such as Coq [9] and Isabelle [13], many mathematical theories (such as the formidably large Feit-Thompson Odd Order Theorem [15, 2]) have been [6] or are being translated and formalised into machine-checked proof-scripts. However, these proof-scripts on

their own inherit the same disadvantages as the aforementioned textbooks, as well as some new ones: they are usually more verbose and explicit and are primarily designed for automation/computation than readability. The former (usually out of necessity to convey to the computer the intended meaning) leads to unnecessary "noise" in the proof and the latter departs from the vocabulary or flow a natural-language presentation may have.

The database world is currently experiencing a tremendous explosion of creativity with the emergence of new data models and new ways of representing and querying large data sets. *Graph databases* have been developed to deal with highly connected data sets and path-oriented queries. That is, graph databases are optimised for computing transitive-closure and related queries, which pose a huge challenge for traditional, relational databases.

A graph-based approach to the representation and exploration of the structure of proof-objects would be a far more natural expression of the complex relationships (i.e. chains of dependencies) involved in constructing mathematical theories. Questions such as "What depends on this lemma and how many such things are there?" or "What are the components of this definition?" could thus be expressed concisely (questions which are not even expressible with standard relational databases systems such as SQL). A popular graph database, Neo4j [10] with an expressive query language *Cypher* will be used for this project.

Resources Required

Software

Several components of software will be required for executing this project, all of which are available for free online.

For using the proof-scripts, the Coq proof assistant will be required, as well as the Proof General proof assistant (`proofgeneral.github.io/`) for the Emacs (`www.gnu.org/software/emacs/`) text-editor.

For writing the plug-in to access Coq proof-objects, the parser and associated modules in the source code will be required (`github.com/coq/coq`) written in the OCaml programming language (`ocaml.org`) with the OCaml's Package Manager OPAM (`opam.ocaml.org`).

For building the library of (Cypher) queries, Neo4j Community Edition will be used.

Hardware

Implementation and testing will be done on both Windows 10 and a Linux Virtual Machine as appropriate and convenient on a Surface Pro 3 (Intel Haswell i7-4650U

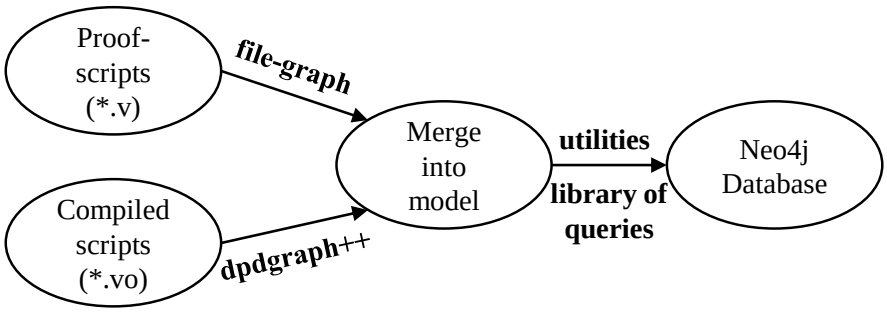

Figure B.1 – System Components (repeated from page 18)

1.7-3GHz, 8GB RAM, 512GB SSD) with a personal GitHub account and physical backup drive (Seagate 1TB) making hourly backups using Windows' File History.

Starting Point

Some existing tools offer part of the solutions: these will be used and combined as appropriate. A large part of the project will rely on my knowledge of OCaml and Coq usage and internals.

Coq-dpdgraph (`github.com/Karmaki/coq-dpdgraph`) is a tool which analyses dependencies between *compiled* Coq proofs. As such, desirable information about notation, tactics, definitions and the relationship between a type and its constructors is lost.

Coqdep is a utility included with Coq which analyses dependencies *at the module level* by tracking `Require` and `Import` statements.

Coq SerAPI (`github.com/ejgallego/coq-serapis`) is a work-in-progress library and communication protocol for Coq designed to make low-level interaction with Coq easier, especially for IDEs. It has a starting point for gathering some statistics of proof-objects in a project.

All of these tools have the same disadvantage: they present information statically, with no way to query and interact with the information available.

Substance and Structure of the Project

The project will have three major parts, as shown in Figure 3.1.

Processing Compiled Files

First, using coq-dpdgraph as a starting point, a tool which expresses a compiled proof-script as CSV files (shown as "dpdgraph++" in the diagram). Finding what information can and should be extracted will be an iterative process. Although coq-dpdgraph is functional, it is very basic with no way of even relating the relationship between a (co-)inductive type and its constructors, hence much work is to be done to even come close to utilising the full potential of compiled proof-scripts.

Processing Source Code Directly

Second, using Coq's sophisticated extensible-parser, to parse, gather and convert to CSV files the desirable but missing information coq-dpdgraph does not extract (shown as "file-graph" in the diagram). An interesting feature of Coq's parser is that it allows new constructs and notation to be defined: this is used heavily in some projects and therefore poses a great challenge for simply understanding and using the parser effectively.

Extraction and Analysis Tools

Lastly, writing utilities to automate analysis of Coq files and importing them into Neo4j and libraries of queries to run on imported data in Neo4j. Since it is not known what sort of data can be extracted and what will be useful or interesting to know, modelling the data – in this case the structure and objects of a mathematical proof – will be an non-trivial task which will be tackled iteratively.

Extensions

Extensions for this project will come from the process of adapting the project to be compatible with SSReflect [7], part of the Mathematical Components set of tools for Coq. These set of tools use low-level hooks in the Coq plugin system to significantly alter the specification and computation of proofs. As such, although they allow for large-scale projects to be formalised more easily, they are non-standard and would thus be very difficult to support fully.

Success Criteria

Alongside a planned and written dissertation describing the work done, the following criteria will be used to evaluate the success of this project:

1. A schema of attributes and relations for each proof-object is defined.
2. Programs which convert proof-scripts and compiled proofs to CSV files are implemented.
3. A library of queries in order to manipulate and explore the proof-objects is implemented.
4. These new sets of tools are shown to have more capabilities and perform comparably to existing tools for exploring mathematical theories.

Timetable and milestones

Date	Milestone
21-10-2016	Complete Project Proposal
04-11-2016	Finish a prototype compiled-to-CSV tool. Get familiar with Neo4j Cypher. Understand how to use the Coq parser.
18-11-2016	Refine compiled-to-CSV tool: tests and documentation. Explore queries possible and start the library. Begin work on translating Coq constructs from proof-scripts.
02-12-2016	Finish a prototype script-to-CSV tool.
16-12-2016	Test and document script-to-CSV tool.
30-12-2016	Begin work on integrating tools into one workflow.
13-01-2017	Stabilise and document whole project so far. Prepare presentation for CoqPL Conference.
27-01-2017	Look at SSReflect and evaluate changes to be made.
10-02-2017	Incorporate changes from feedback/new features.
24-02-2017	Test and document the new features.
10-03-2017	Write Introduction, Preparation and Implementation chapters.
24-03-2017	Fix bugs/unexpected problems.
07-04-2017	Write Evaluation and Conclusion chapters.
21-04-2017	Fix bugs/unexpected problems.
05-05-2017	Complete Dissertation (references, bibliography, appendix, formatting).