Dhruv Makwana

Exploring the structure of mathematical theories using graph databases

Computer Science Tripos, Part II

Trinity College

March 30, 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:

Date:

Contents

1 Introduction

This dissertation offers a solution to problems regarding the presentation of mathematics. Firstly, these problems and the specific systems involved are described. Then, the aims of the project are stated; in later chapters, details of the prepration and implementation carried out are expounded. Lastly, the chapters on evaluation and conclusion provide evidence for the success of the project, reflections on the process and suggestions for future work.

1.1 Problem

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 proof-assistants 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 of a natural-language presentation.

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.

1.2 Solution

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.

1.3 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 projects such as the Four-Colour Theorem [5] (60,000 lines) and the aformentioned *Feit-Thompson* theorem (approximately 170,000 lines, 15,000 definitions and 4,200 theorems) are feats of modern software-engineering.

On the programming language side, Coq has served as the basis for many equally fantastic 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.

1.4 Neo4j Database and the Cypher Language

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

1.4.1 Cypher: An Illustrated Example

Cypher features heavy use of pattern-matching in an ASCII-art inspired syntax. The following (slightly contrived but hopefully illuminating) example in Listing 1 illustrates some of the key strengths of graph-based modelling using Cypher.

```
MATCH path = shortestPath(
  (puppy:dog {name: "Cliff"})-[:LIKES|:KNOWS*..4]->(child:person))
WHERE puppy.age <= 2 AND child.age < 6
RETURN path,
       child.name AS name,
ORDER BY other.distance_from_clifford
```

Listing 1 – Example Cypher Query

Suppose we have a puppy named "Cliff" looking for the nearest and most familiar children (for this example, a person under the age of six) to play with.

To see how Cliff (indirectly) likes/knows this child, we bind *path* to the result of the *shortestPath* query. For the path itself, we start with a node following this structure: `(var:label {attrib: val})`. We then have a *labeled, transitive* relationship (explicitly limiting our search to paths of up to length four) expressed as an arrow with a label `-[..]->`. As such, we can discard any paths with relationships we do not want (e.g. `HATES`).

To filter based on more complex logic (than possible by pattern-matching directly on labels and attributes) we can express the requirement that the age of a dog by the name of Cliff be less than or equal to two (and similarly for the age of the child) in the `WHERE` clause.

Lastly, we return the path and order the results by proximity as a row of results, renaming the column of the child's name to simply "name".

1.5 Related Work

Some existing tools offer part of the solutions.

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-serapi) 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.

1.6 Aims of the Project

This project aims to:

- represent Coq libraries as Neo4j graph databases, which wil involve
  - 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; and to
- create a library of Neo4j queries, intended
  - to highlight the structure of and relationship between proof-objects
  - by coalescing and implementing several graph-related metrics.

1.7 Summary

An explanation of the problems which conventional presentations of mathematics suffer from was given, with *graph databases* proposed as a solution. Existing tools were mentioned and the requirements for a successful project were listed.

2 | Preparation

Before commencing implementation of the project, careful consideration was given to planning it. Current solutions were explored, studied and evaluated against the aims described in Section 1.6. The rest of this chapter will explain the initial setup, elaborate on related work and outline the project's starting point.

2.1 Project Planning

This project presents a unique idea and breaks new ground. As such, the methodology had to suit and reflect the largely exploratory nature of the process. A spiral software development model was chosen: think of an idea, modify the model (of Coq proof-objects), implement and propogate the necessary changes, evaluate the end-result and repeat. This allowed for experimentation of ideas and flexibility of implementation strategies.

Git (`git-scm.com`) and GitHub (`github.com/dc-mak`) were invaluable during the project, allowing for easy tracking, reverting, reviewing and collaborating. New features could be tested on new branches before being merged in and a copy of the work was safely backed up in one more place. GitHub extensions such as Travis-CI (continous integration) were added in later, as it became apparent that precisely specified versioning, build-dependencies and automated tests were useful in spotting errors early.

2.2 Requirements Analysis

Several components of this project needed to function correctly, both individually and in conjuction for it to work. Separate parts for modelling/translation (from Coq to the chosen model), displaying and interacting (Neo4j/Cypher) and computation (Neo4j/Cypher plugins) needed to be developed and brought together. Below is a list of required features 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 precomputing 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.

- **Interaction**: 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.

- **Computation**: 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 – specifically, dpdgraph, which was used as a starting point for extracting information about Coq proof-objects from compiled proof-scripts – are written in OCaml (`ocaml.org`). Since it almost always wiser to work with and modify existing sytems (and more representative of real-world work) and as a functional language, OCaml benefits from strong, static (and inferred) type-system (allowing for easy experimentation, greater confidence in correctness), sticking to it for other parts of the tools 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 manipulating Coq constructs) and good editing tools.

Similarly, Neo4j and its plugins are (usually) written in Java, but several lanaguages are supported for the latter, both by Neo4j officially and by the community. As will be explained in Subsection 2.4.3, Java and R were found to be the most suitable for achieving this project's goals.

2.4 Starting Point

Both Coq and Neo4j have a rich eco-system of tools and libraries built for them. Hence, it was worthwhile to examine the current landscape to determine the software available and then use it as a starting point effectively.

2.4.1 Coq

Coq is a *large* project, devloped by INRIA (France), and its size and complexity are best experienced through detangling 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 consulting the Coq developers' mailing-list, several hours were spent browsing the source code to overcome the severe difficulties in understanding the project. Prior familiarity with *using* Coq (as an introduction to tactical theorem-proving and dependently-typed programming) 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 data-model during the implementation phase.

2.4.2 Existing Tools for Coq

A number of tools were studied to learn their approaches and analyse their strengths and weakeness. A full, detailed comparison between all the tools mentioned and this project will be presented later, during the Evaluation chapter. What follows is a brief overview of each tool and the reason it was insufficient for the purposes of meeting the project aims and requirements.

I Coqdoc

Coqdoc is a documentation tool for Coq projects, includes as part of the Coq system.. It can output to raw text, HTML, LaTeX and a few other formats. 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 worked on an entirely *lexical* level, with no formal parsing, understanding or elaboration of the code structure. Some efforts were made to modify its output into a useful format (e.g. comma-separated values) for other tools; however these did not prove fruitful because tokenisation cannot infer or preserve as much information as full compilation. Hence, since it could not meet any of the modelling requirements (completeness M1, flexibility M2 and size/precomputation M3) this approach was abandoned.

II Coqdep

Coqdep is a tool which computes inter-module dependencies for Coq and OCaml programs, outputttting in a format readable by the `make` program. 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), this approach was also abandoned.

III CoqSerAPI

Coq Se(xp)rialized API 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. 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.

IV dpdgraph

dpdgraph is a tool which extracts dependencies between Coq objects 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 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, it seemed unlikely that the information was not present in the compiled object files. Assuming that the data was already present in those files, but simply *ignored or unused*, implementation of the modelling aspect of this project focused on understanding and augmenting dpdgraph to add the missing pieces to the model and convert the whole thing to comma-separated values (henceforth referred to as CSVs)

2.4.3 Neo4j

Neo4j is one of the most popular graph database systems. It 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 requirement 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, which meant modelling could be focused towards 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

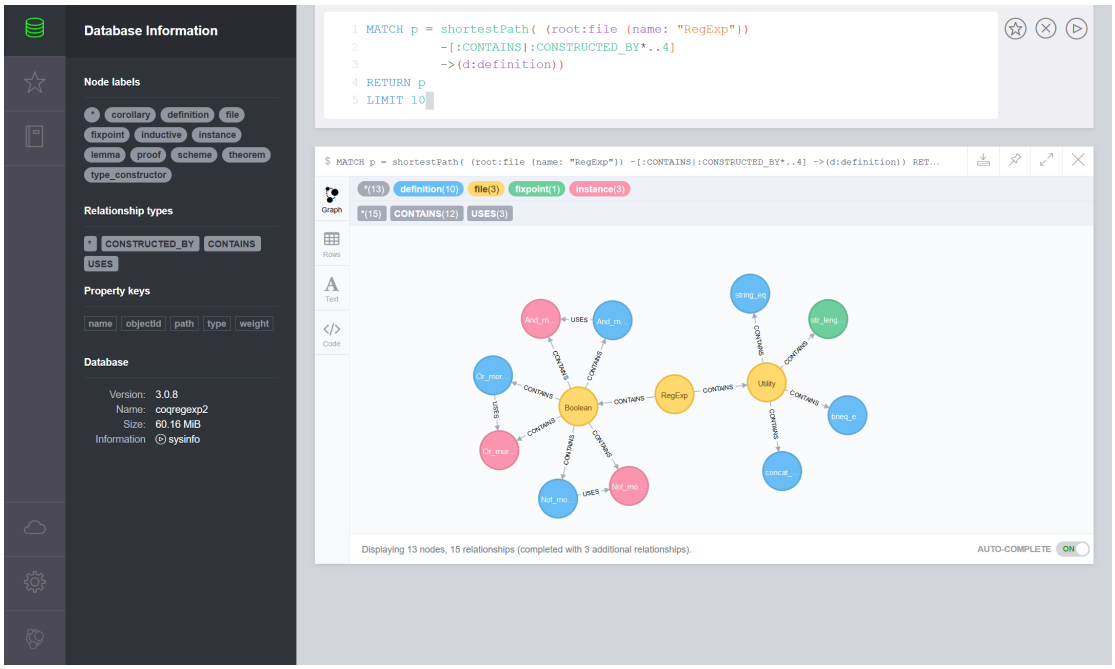

Figure 2.1 – Neo4j Interactive Browser

- an overview of the current labels, relationships and properties in the database
- syntax-highlighted interactive-editing box
- 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)
- and many more features such as browser sync, settings and the 'about' section.

2.4.4 Existing Tools for Neo4j

Neo4j features rich-integration with many languages, libraries and tools. Of those, the following were the most relevant and useful tools for meeting the project requirements.

I APOC: Awesome Procedures on Cypher

Awesome Procedures on Cypher, or *APOC* for short, is a community-maintained Java plugin featuring several graph algorithms callable from within Cypher it-

self. Although there are other extension libraries (such as MazeRunner), APOC is well documented, up-to-date and the most comprehensive, and therefore the obvious choice as a foundation. By being a Java library hooked into Cypher, it offered the potential for additional functionality to be built on top of it which packaged-up 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, flexibile to use and extensible; even going part-way towards meeting the *computation* requirements.

II igraph

APOC has some key strengths that made it a good choice: it is easy to install and use and has some basic graph algorithms to get started with. However, its 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 further adds to its limitations: it is not well-suited to more intense analyses over large graphs of libaries 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 alogrithms and their uses will be described later, in the Implementation chapter. For now, it suffices to surmise that although igraph is not as easy to interact with (via the statistics-oriented programming language R, as detailed in the next paragraph) as APOC, the extra capabilities afforded were indispensible towards achieving the *computation* requirements of a core library of good defaults.

III visNetwork

With igraph and APOC providing starting points for the computational aspects of the projects, and the interactive Neo4j browser providing a well-polished, graphical mode of interaction with basic, but useful, visualisation, the last piece of the project was to incorporate the extra information gained from *executing* the graph algorithms.

Several visualisation programs exist for Neo4j; however, many are for commercial, industrial use and offer the features/complexity (and pricing) to match. All tools which 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, visNetwork, (an R library exporting to JavaScript

which can be rendered inside a browser) was chosen due to its simplicity and ease of integration with previous tools mentioned above.

IV R

R is a statistics-oriented programming language, part of the Free Software Foundation's GNU poject. It is relevant for this project because it offers an easy way to tie together Neo4j (through official bindings), igraph and visualisation using visNetwork. This convenience came at the price of having to learn R for this project, having been unfamiliar with it prior. Nonetheless, it is a well-documented, relatively easy to pick-up language and offered even more opportunities for learning during the course of this project.

2.5 Summary

A detailed account into the planning of this project was given. The choice of development methodology (spiral) and development tools (Git, GitHub, Travis-CI) were noted. Requirements on modelling, interaction and computation were explained and the choice of technologies and tools used as statrting points (Coq, OCaml, dpdgraph, Neo4j, APOC, igraph, R and visNetwork) were justified *in relation* to *which* requirements they satisfied.

3 | Implementation

Upon completion of the project's plan, its execution commenced. What follows is an account of the programs written, problems encountered, solutions implemented and tests conducted using the project-structure shown in Figure 3.1 as a guide. *Reasons* for design decisions (arrived at using *formative* evaluation techniques) are detailed in Chapter 4 on Evaluation.

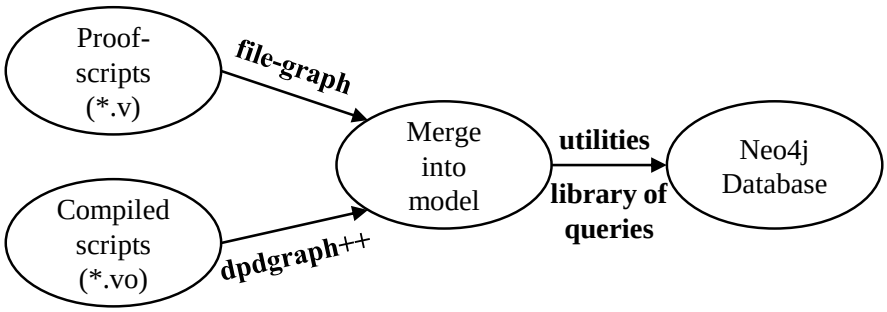

Figure 3.1 – System Components

3.1 Coq object-files to CSV

This section of implementation corresponds to "dpdgraph++" on Figure 3.1: modelling the data contained in and the structure of Coq object-files (`*.vo`) as comma-separated values (CSVs). The initial model (inherited from the open-source "dpdgraph" tool) and subsequent changes to it will be described.

3.1.1 Modelling

Initially, each edge was assigned a *weight* representing the number of (directed) uses of one node by another. Each node was assigned four *properties*:

- *body*, a boolean representing whether a global declaration was either transparent or opaque;
- *kind*, a ternary value representing whether a *global reference* (a kernel side type for all references) was a reference to either the environment, an inductive type or a constructor of an inductive type;
- *prop*, a boolean value representing whether a term is a `Prop` (a decidable, logical property about a program, as opposed to a general `Type`);
- *path*, a string value represent the module an object is in.

These properties were difficult to understand: they are not in the vocabulary of a Coq programmer (e.g. `Definition`, `Inductive`, `Theorem`) and could not represent the richness of the AST appropriately. It was not documented how to translate these constructs back to familiar terms and thus, it quickly became clear that the these properties needed to be replaced by more general and descriptive ones (expounded below).

I Precise Kinds

Apart from *path*, all the properties were removed and replaced by two *labels*: labels are 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. Implementation of this was simply a matter of looking up and expanding the abstract syntax tree (AST) starting from the type `global_reference`.

The two labels are *kind* and *subkind*. A *kind* is a string that can take one of the following values, each directly corresponding to an AST term: module, class, type_constructor, inductive_type, definition, assumption or proof.

Optionally, some terms have *subkinds*, for distinguishing different constructs more precisely. 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.

II Recursive Modules

What remained from the initial model was the *path* property. The issue here was that an inherently *hierarchical* structure (of inclusion) was represented *flatly* as a string attribute of a node. This made modules second-class citizens, not subject to the same analyses and manipulations as proof-objects, excluding the possibility of expressing *module-level dependencies* (as possible in the coqdep tool).

Implementing this feature was difficult; there were two major phases. Firstly, the type of a node was expanded to include modules (using a variant datatype). Finishing this involved locating and fixing the resulting type-errors. This meant modules were in the model, but as a flat structure: modules could be related to objects but not to other modules.

Thus, the second major phase was inferring and adding all the "ancestors" of a module with the correct relationships (parent as source, child as destination, repeatedly up to the root module). The initial attempt resulted in stack-overflow for larger examples. A stack-trace did not highlight any point of error so it was *assumed* to be due to the naïve, but easy-to-write (and check) non-tail-recursive implementation. So, the code was rewritten in a space-efficient, tail-recursive manner (which allows for deeply nested module hierarchies to be handled). Surprisingly, the problem persisted; further investigation discovered the fault lay in the separate `dpd2` tool (the details of which can be found on page 20).

III (Co-)Inductive Types and Constructors

Now that the properties of the original dpdpgraph had been superseded by more general and flexible alternatives, it was time to implement new features. One of the most obvious and frustrating omissions from the initial model was the inability to relate an (co-)inductive type to to its constuctor(s). Expanding the

AST term for type-constructors showed which type it constructed (however, types have no information about which constructors construct them). Hence, since dependencies were constrcuted in a depth-first manner *down* the AST, they had to be built in reverse. Therefore, in order to correct this when outputting edges, each pair of nodes has to be checked 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, then the direction of the edge output was swapped.

IV Types

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.

Following the functions called for the Coq command `Check` <*expression*> (for printing the type of a given expression) led to the algorithm for *getting* the type; all that was left was converting the output to an OCaml string. It was *using* the output which was problematic. Newlines, quotation marks, and commas had to be replaced by hash signs, single-quote marks and underscores respectively, so as to not interfere with the `dpd` and CSV encoding of data.

V Relationships

Modelling relationships was the most interesting aspect of deciding how to represent information. Of primary concern was 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 (co-)inductive type, they should be able to expand the type and see its constructors. This leads to the basic modelling structure of `(src)-[:USES]->(dst)` for dependencies. For (co-)inductive types, this is refined to a `(type)-[:CONSTRUCTED_BY]->(constr)` and for modules this becomes `(module)-[:CONTAINS]->(object)`.

Two issues arose whilst implementing these relationships. First, finding and matching types and constructors (details of which are described earlier). Second, balancing expressivity against simplicity. Relationships of the following format, `X_USES_Y` were considered for kinds X and Y. Although this was useful for fewer kinds, 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, this aspect was simplified to the model presented above.

3.1.2 Translation

Once a model is constructed (in the form of a graph), it is translated by the `dpd2` tool to a CSV file for use by Neo4j's import tool to create a database. An overview of the `dpd` and CSV formats, as well as the `dpd2` tool itself, is presented next.

I `dpd` Format

The graph representing the model is output 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

Upon output as as `dpd` file, a model can be translated to various other formats. As an example, dpdgraph includes a tool to output a `dot` file, used extensively for *visualising* graphs by many tools, from a `dpd` file. However, for the purpose of this project, the `dpd` file is translated (using the `dpd2` tool described below) to *two* CSV (comma-separated values) 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), refactored into `dpd2csv` which output CSVs instead. Later, both were combined into a more general, `dpd2` tool which could accept the file-type as a command-line argument. As mentioned in II Recursive Modules on page 17, a stack-overflow error occured because of this tool. 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, the `-keep-trans` flag was passed on subsequent uses to avoid the issue altogether.

3.2 Coq source-files to CSV

This section of implementation corresponds to "filegraph" on Figure 3.1: modelling the data contained in and the structure of Coq source-files (`*.v`). First, deficiencies with the model so far (relying only on information from `*.vo` files via "dpdgraph++") are mentioned. Then, attempts to address those deficiencies are described. Finally, conclusions regarding the practicality, necessity and contribution of "filegraph" to the overall project are drawn.

3.2.1 Deficiencies in the Model

Some small issues remained with the model as presented so far: *apparent* absence or duplication of some modules and a lack of notation and tactics. Oddities with modules arose from the use of *functors*: modules that take other modules as arguments (used to abstract over agruments, tactics, defintions 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).

3.2.2 Exploring Solutions

Notation, tactics and functors can all be dealt with by simply parsing source-files directly with the Coq parser. However, *several issues* stop this from being a pragmatic solution, least of which are the size of the AST and complexity of parsing Coq files.

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. Futher 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 progress had been made tackling these issues, a simpler solution was desired. *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. A shell-script was used to prototype a tool that converted glob files to CSVs. Although promising for tactics and notation, glob files suffer from the same inability to distinguish between modules and functors.

3.2.3 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 *typeclasses* 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 is possible to *reconstruct* generalisations *representing functors* once the database is created. As such, extracting information directly from Coq source-files contributes little to the overall project, but presents clear ways forward for future-work.

3.3 CSV to Neo4j

As part of the "utilities" in Figure 3.1, Neo4j comes with a command-line import tool that – 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 – constructs a database. As explained in II CSV on page 19, the headers determine the labels, properties, IDs of nodes as well as the start- and end-points, properties and type of edges.

3.4 Query Library

Finally, this last section of the implementation corresponds to the "library of queries" shown in Figure 3.1. Extensions, written (in Java and R) to meet the interaction (I1, I2, I3) and computation (C1, C2) requirements on page 8, and their uses are outlined.

3.4.1 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 propogation (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.

I Additions

APOC is used as a foundation for simpler, domain-specific query library. As an example, a user can write `coq.assumptions(<node>, <depth>)` to examine the n^{th} level assumptions behind a given node. More convenience functions are defined for executing some of the listed algorithms on all nodes in a given module or of a given type. For more complex, efficient and scalable/scriptable analyses however, igraph and R must be used.

3.4.2 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, several example programs were written which automatically processed data from a new database, stored the information again for later use (to avoid recomputation) and output appropriate visualisations. Since processing could take on the order of minutes, 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. commiting a transaction to the database) were provided.

As stated previously, igraph provides a comprehensive set of graph algorithms with an *emphasis on efficiency, portability and ease of use*. The metrics these algorithms compute and an interpretation of them in the context of mathematical theories is given next. Although a wealth of algorithms are available in igraph, visualisation and evaluation was focused on only the most common and well-known ones.

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 (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]

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3.1}$$

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]

$$C(x) = \frac{N}{\sum_y d(y,x)} \tag{3.2}$$

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 defintion.

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 3.3. [14] Hence, it can be viewed as the a probability of an easily-confused, randomly-perusing mathematician coming across a given proof or definition, perhaps upon a first reading.

$$\mathbf{r} = \frac{(1-d)}{N}\mathbf{1} + d\mathbf{L}\mathbf{r} \tag{3.3}$$

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]

III Visualisation

There are many interesting ways to visualise the plethora of data that analysing large mathematical theories 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 to rapidly explore different ideas.

Another, more direct, way of displaying infomation is by displaying the graph itself. As will be shown 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). Thanks to an (experimental) integration with igraph (specifically, igraph's layout mechanisms) graph layouts could be pre-computed in fast, native C/C++.

3.5 Project Related

During implementation, several skills and lessons were learnt 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, the following areas received particular focus.

3.5.1 Testing

For most of the project, testing was done by 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, automated tests were used (which came in particularly useful when the project had to be bifurcated to support two different versions of Coq). Small scripts – of problems solved when learning Coq – were used to check output on a small scale (where every single aspect of a project was known, and testing turnaround was quick). Coq's Standard Library was used as a large scale stress-test, to ensure all constructs were translated correctly. When problems were found, they could usually be traced back to debug output produced during execution.

3.5.2 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). Setting up Travis-CI for continuous-integration builds made version dependencies precise and explicit, removing much of the hassle and uncertainty surrounding builds on other machines.

3.5.3 Tooling

At first, the project was run half on Windows and half on a Linux VM (virtual machine), using shared folders. There were a number of reasons for this: Coq and Neo4j were already set up on Windows, both are easier to interact with in a graphical environment and starting up a VM just for some experimentation incurred quite an overhead. Eventually, this set-up became confusing and time-consuming and a leap to running the project fully on a Linux VM was made. Nevertheless, issues arose when working on R integrations: running a Java database inside a Linux VM was insufferably slow, requiring a SSH reverse-port-forwarding to be set up (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, editor integrations for OCaml were set up, dramatically reducing the edit-compile cycle (especially for a strong, statically-typed programming language as OCaml). Coq's use of non-standard OCaml features, extensions and build-systems became particularly frustrating at this point. For example, 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 the complexity of the build-system became apparent.

3.6 Summary

Just presented was an in-depth account of the programs written (dpdgraph++, dpd2, Java library, R library), problems encountered (first-class modules, relating types and constructors, incorporating type signatures, CSV translation, extracting information directly from Coq source-files, dependency and version tracking, impenetrable build-systems), solutions implemented and tests conducted. Where relevant, experiments were mentioned to illustrate lessons learnt. Throughout, project requirements were referenced to justify important decisions.

4 | Evaluation

Here, the project's resounding success, in relation its aims (as listed in Section 1.6), will be justified. A comparison with existing tools will demonstrate the advantages of this project over existing tools (those mentioned in Subsection 2.4.2). Sample output will be shown and the interesting insights they provide will be explained.

This project aims to:

- represent Coq libraries as Neo4j graph databases, which wil involve
  - 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; and to
- create a library of Neo4j queries, intended
  - to highlight the structure of and relationship between proof-objects
  - by coalescing and implementing several graph-related metrics.

Figure 4.1 – Aims of the Project

4.1 Features

For convenience, the aims of the project, as listed in Section 1.6, are reproduced above, in Figure 4.1. Within the first major bullet-point (representing Coq libraries as Neo4j databases), the latter three aims (of adapting, extending and writing new programs to output CSV, extracting extra information and automating database creation) were exposited in the Implementation chapter.

So, to assess the first aim of the project (exploring and choosing the correct model), all of the programs listed in Subsection 2.4.2, Existing Tools for Coq, are compared side-by-side in Table 4.1, in which this project comes out favourably.

4.1.1 Other Tools

The features chosen for comparison reflect the strengths of each tool considered.

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 dispaly is Figure 4.2; it shows a medium-sized Coq library as ouput by dpdgraph.

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

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

Table 4.1 – Comparison of Features

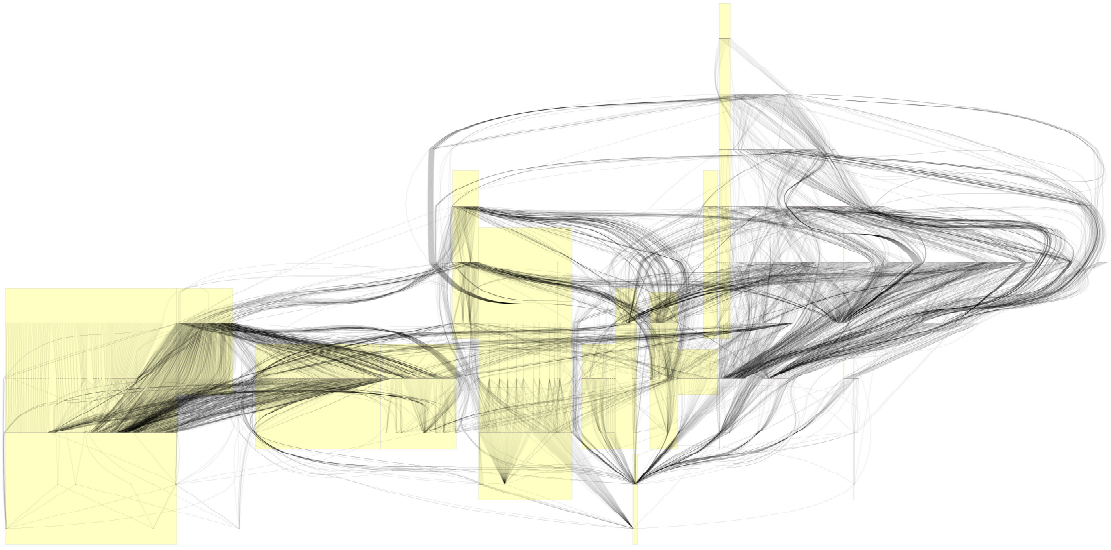

Figure 4.2 – Static Output of dpdgraph (using `dpd2dot`) on CAS

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.

4.1.2 This Project

It is clear from the table that this project either supports, or can support, every feature supported by other tools. However, it is important to note, that in many cases, this project does not simply match the feature, but exceeds it in ways described next.

Linking to source code could be supported by modifying either the model, database or JavaScript visualisations (outuput 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. For a proof object, its type is a statement of the actual theorem being proved, hence this feature is incredibly important when coming to grips with a mathematical theory. Crucially, it is a *fully expanded* type signature, making explicit any assumptions introduced (perhaps hunderds of lines prior in the source code) into the environment.

Module dependencies are set with the query in Listing 2 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 this project: by using a Neo4j graph database, we can understand and manipulate this relation in a much more flexible and scalable manner than any visualisation can manage.

4.2 Performance

This project could have all the features any user could ever want but if it ran so slowly that nobody would have the patience to use it, then it would be wrong to consider it as having met its aims. So, here, this project's execution time is compared to most of the tools from the previous section. Although it 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, the Coq (8.6) Standard Library was used, 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 it was not measured. For dpdgraph, seprate measurements were taken for outputting a `dpd` file and converting that file to a dot Format. A similar approach –of measuring database creation (model output, CSV translation and data import) and library analysis separately –was taken 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 Results

Figure 4.3 shows the results of the comparison, as broken-down by bash's time command into time spent in user-code, system calls and overall. Note that the data is presented on a *logarithmic* scale. Immediately we see coqdoc takes very little time to run, approximately 10 seconds and coqdep even less at around 4 seconds, which, considering their purely lexical approaches, is to be expected.

So it is somewhat surprising that dpdgraph runs just as quickly as coqdoc. The 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 explains precisely *what* occurs during database creation and where ineffeciencies lie.

4.2.3 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),

```
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 2 – Query to set Module Dependencies

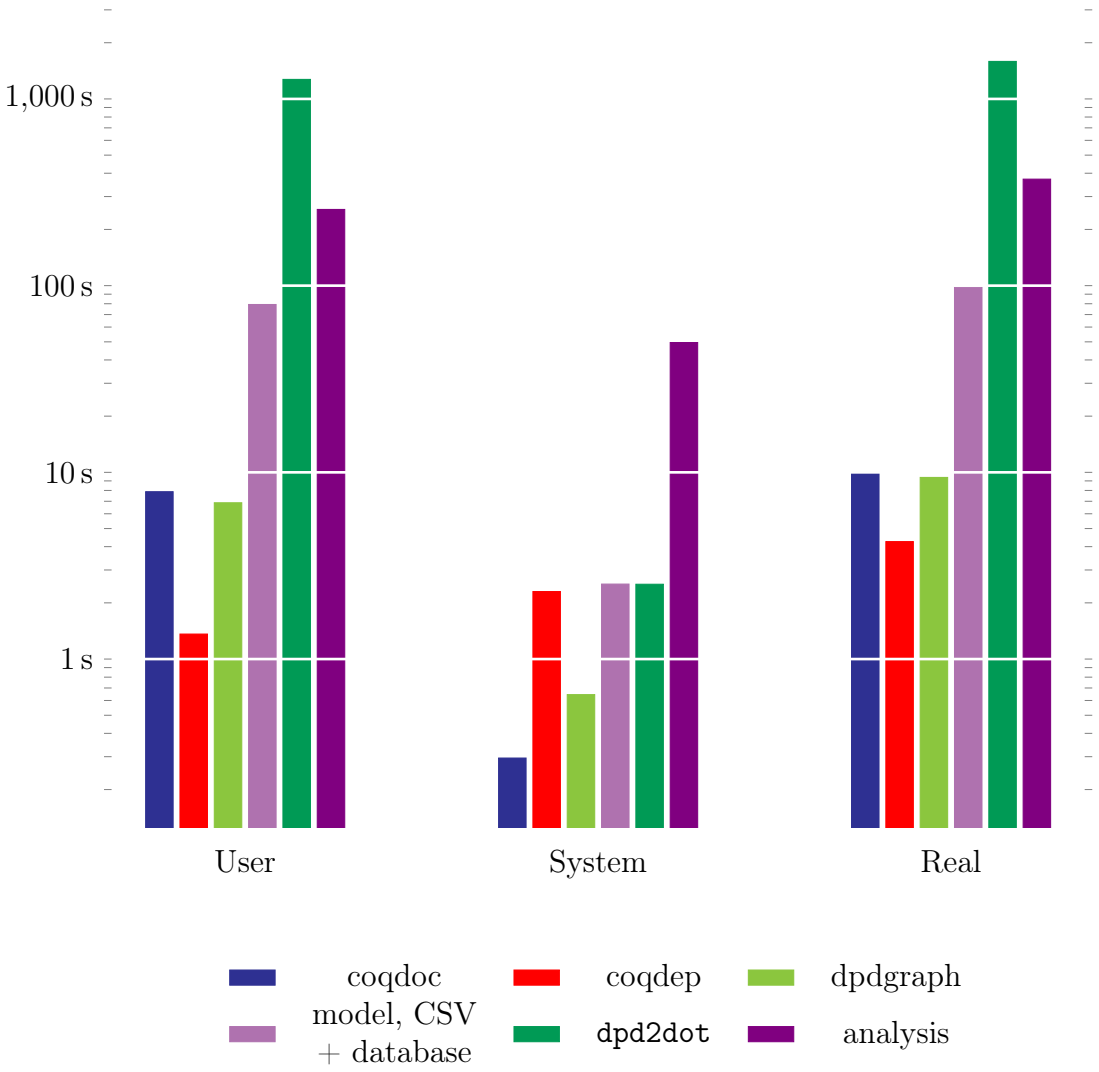

Figure 4.3 – Comparison of Execution Times

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 accomodate 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. Outputting to `dpd` was done to facilitate easy extension with other tools. It is likely that outputting a CSV directly would have resulted in being able to bypass this phase altogether, though, from a software-engineering point-of-view, the trade-off there is increased coupling.

4.2.4 Graph Analysis & Visualisation

Once a graph is created (whether it be in the form of `dpd` file or a dabtabase) 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 defintions 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` did not do graph *layout*: it just split the graph into subgraphs (based on modules) and assigned 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, the command `dot -Tsvg` to produce an SVG visualisation had to be cancelled after failing terminate within a few *hours*).

4.3 Library of Queries

Within the second major bullet-point (creating a library of Neo4j queries), the second aim (of coalescing and implementing graph-related metrics) was dealt with in the Implementation chapter. To assess the first aim (of highlighting the structure of and relationship between proof-objects), the library will be shown to work on the small case of a Coq Regular-Expression package and on the large case of the project's moonshot, the Odd Order Theorem.

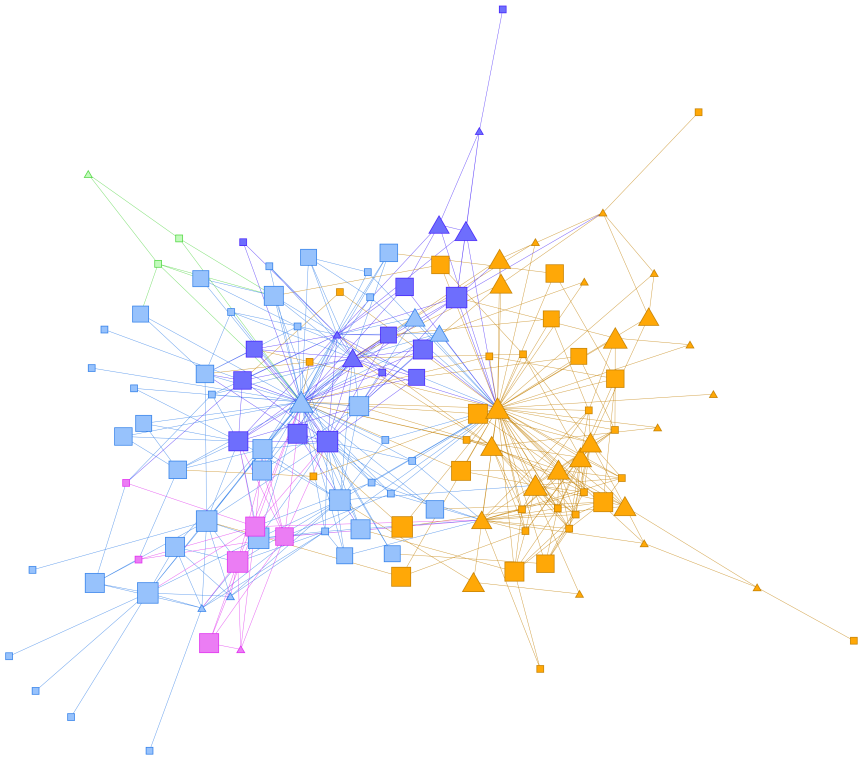

Figure 4.4 – Force-directed with Betweenness Centrality

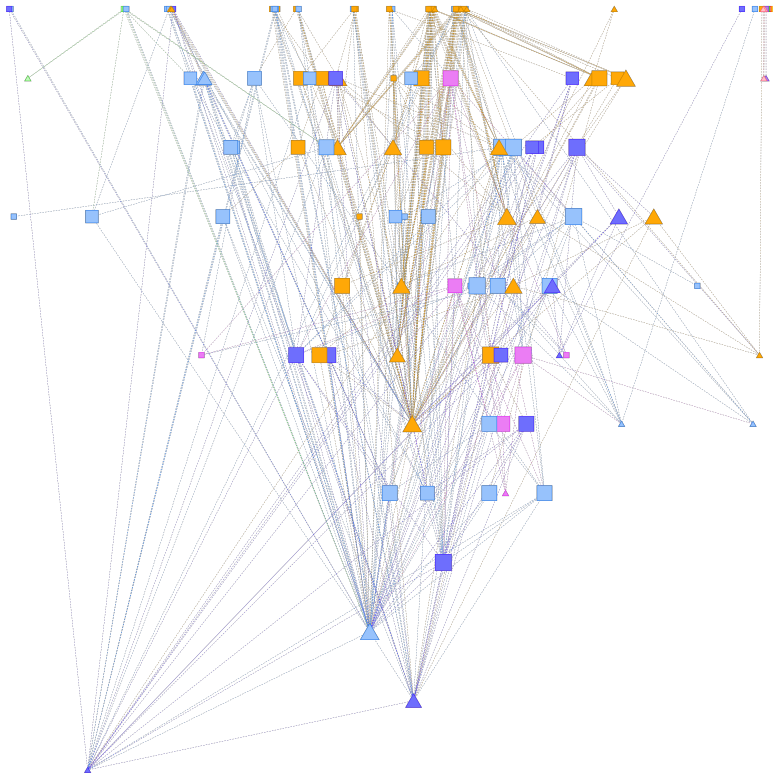

Figure 4.5 – Hierarchical with Betweenness Centrality

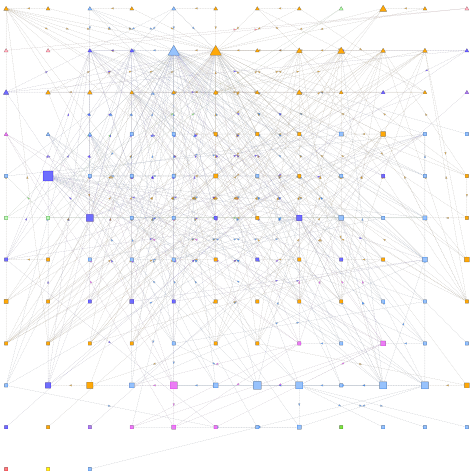

Figure 4.6 – Grid Layout with PageRank

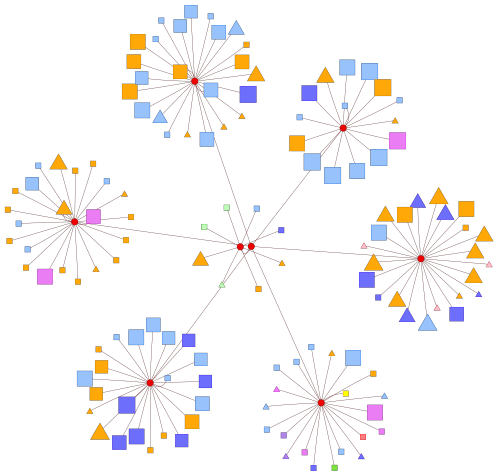

Figure 4.7 – Force-directed with Modules

4.3.1 Small: CoqRegExp

For this package, PageRank, betweenness centrality, modularity clustering, hierarchical layout, grid layout and force-directed layout produced the most aesthetically pleasing and insightful results. All graphs 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 Figure 4.7 which also includes modules (as circles).

I Direct

Figure 4.4 shows a force-directed visualisation. The size of the nodes corresponds to betweenness centrality scores (split up into 10 logarithmically equal-width buckets). Colours correspond to groups assigned by modularity clustering. Edges represent the `USES` relation; edge-directions (source-uses-destination) are omitted for clarity since they generally point toward the the center of a cluster.

We see that modularity clustering largely corresponds to the how a human might group nodes visually: two major clusters with a few, smaller clusters. Intuitively, 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 center of the light-blue cluster is a function which computes whether a given regular-expression matches a given string; the node at the center of the yellow cluster defines what it means for two regular-expressions to be equal.

II Hierarchical

As far as insights are concerned, we see that this enables us to get a high-level view of the threads of thought in the theory. We can see how these threads develop by looking at Figure 4.5. Here, a Sugiyama layout is used (the direction of edges always points downwards). If nodes were partially ordered according the `USES` relation (destination $<$ source if and only if source `USES` destination) then this layout ensures that if a node (destination) is *below* another node connected to it (source) then the source `USES` destination.

Applied to mathematical theories, this layout is essentially a visual study-guide, a plan for how to best tackle the material. It can be applied to subgraphs, refined by declaring the maximum depth or the lowest layer to include and filtered based on the kind or importance of a node.

III Grid and Modules

Given a strategy for how to approach the material, knowing how important each section is can be useful for gauging how much time to spend on understanding it. As indicators of importance, measures of centrality are a useful way to do precisely that. Figure 4.6 represents PageRank through node size on uniform grid, useful for visually determining relative importance.

And finally, Figure 4.7 shows the proofs and defintions with modules, where edges represent the `CONTAINS` relation. We see that the modules at the right and near the center (named Utility, Includes and Char respectively) have very few important nodes (as determined by betweenness centrality). Each module has 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.

IV Insights

Without studying even *one line* of source code, we have been able to gleam an intuitive and impressively detailed conceptual-scaffolding of this theory of regular-expressions. A user can use the library of queries to futher explore the theory or jump into reading the source code with a much better idea of how the different pieces of the puzzle come together.

4.3.2 Large: Odd Order Theorem

For the Odd Order Theorem, betweenness centrality, modularity clustering, hierarchical layout, grid layout, circular layout and force-directed layout produced the most aesthetically pleasing and insightful results. As before, only definitions (shown as triangles/$\triangleq$) and proofs (shown as squares/$\square$ symbol) are shown, except for Figure 4.9 which also includes modules (as circles).

This Coq package closely follows the structure of the source material: Peterfalvi [15] and Bender & Glauberman [2]. Each section (chapter) in the original books is a file/module in the Coq package; each definition, lemma or proof corresponds to the same in the books. Following the convention in the Coq package, Bender & Glauberman will henceforth be abbreviated to BG and Peterfalvi to PF.

I Direct & Modules

Starting with a force-directed visualisation of nodes, Figure 4.8 highlights five major groups: dark-blue, pink, yellow, purple and dark-green, and shows that modularity clustering largely agrees with the layout. Clear connections are visible between each of the groups, with a central group showing a mix of almost all groups.

We can begin to uncover some of the meaning by looking at Figure 4.9. The seven pink circles near the center are BG sections 1-6 and appendices A and B. As the first sections, they lay the groud-work for the rest of the theory;

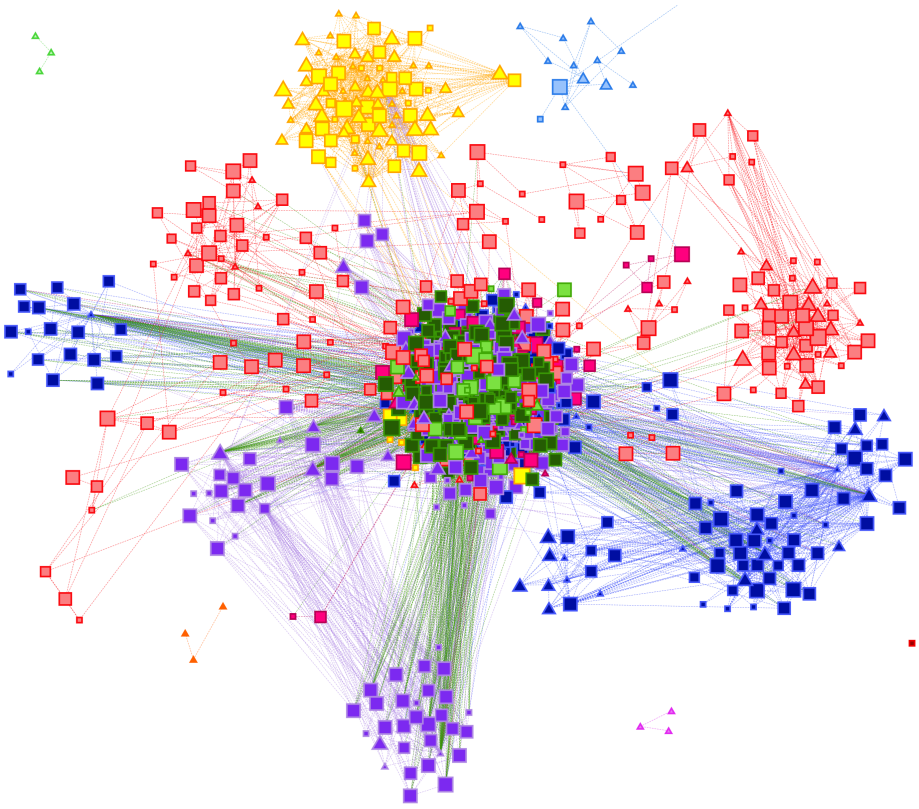

Figure 4.8 – Force-directed (some nodes omitted for clarity)

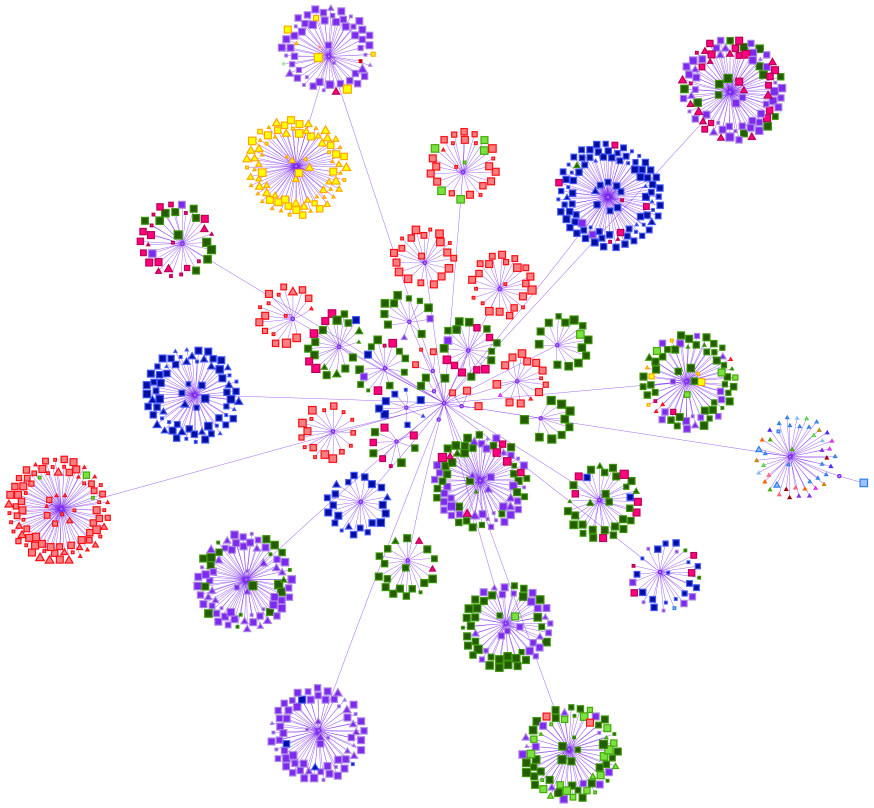

Figure 4.9 – Modules

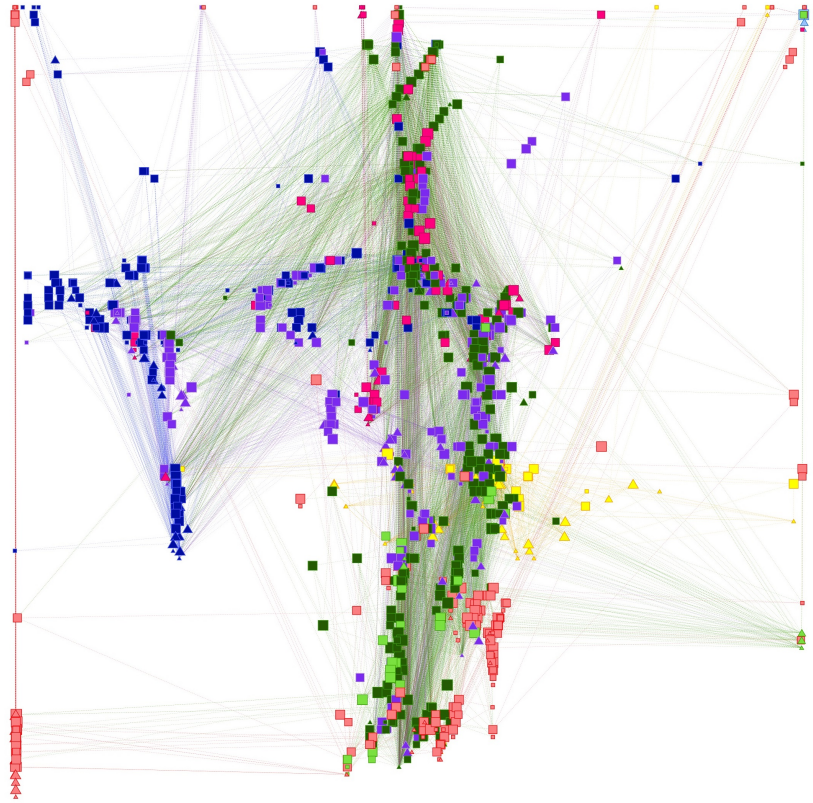

Figure 4.10 – Hierarchical

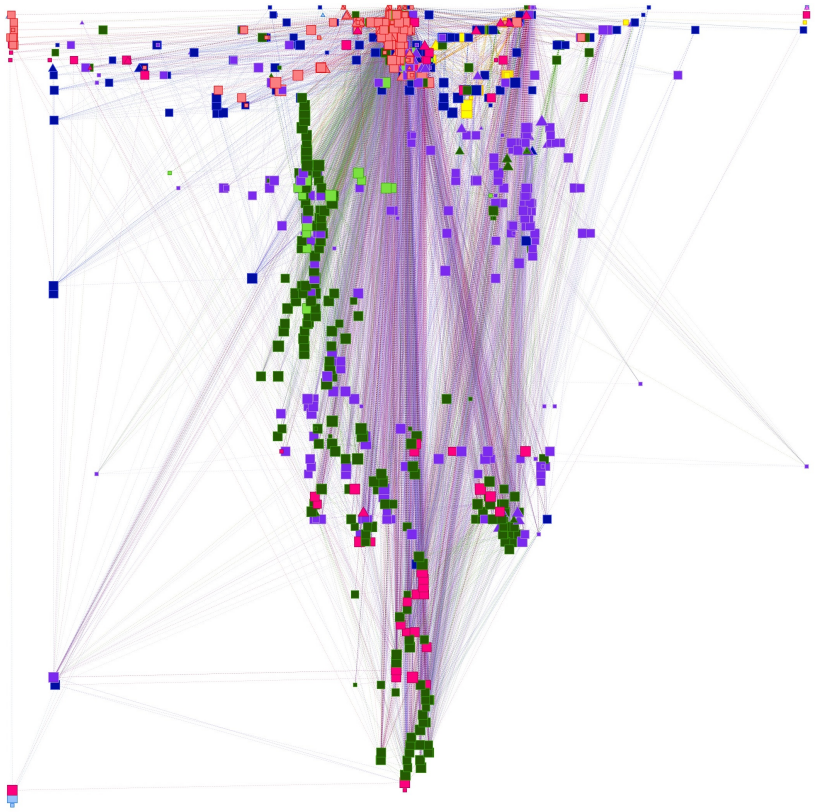

Figure 4.11 – Hierarchical Flipped

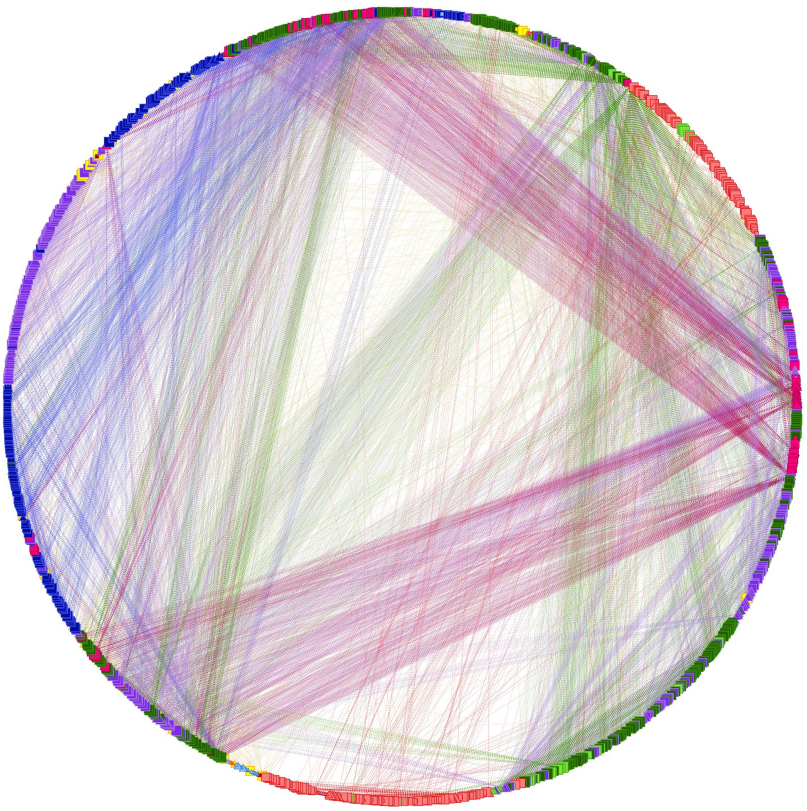

Figure 4.12 – Circular Flipped

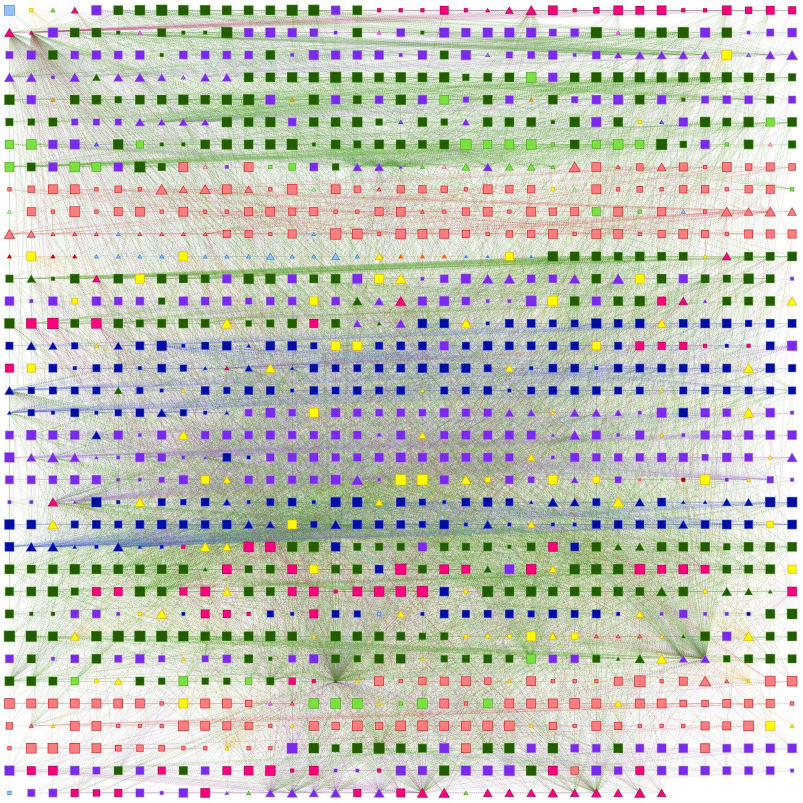

Figure 4.13 – Grid

5 | Conclusions

Witty quote.

5.1 Summary

It was difficult to spend hours staring at the entire Coq compiler code base, figuring out the types and transformations.

5.2 In Hindsight

5.3 Future work

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

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

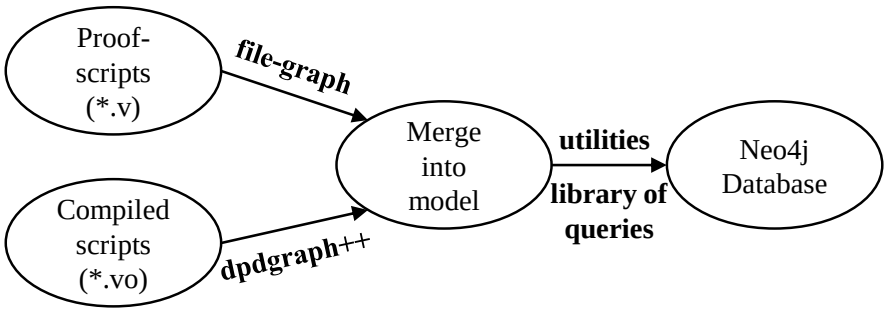

Figure B.1 – System Components (repeated from page 16)

i7-4650U 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).