Visualisation of The University of St Andrews Undergraduate Modules and Requisites

Documentation and Design Process

Thomas Ekström Hansen

Summer 2019

Contents

1 Introduction **3**

2 Database Design and Implementation **4**
- 2.1 Entity Relationship Diagram . . . 4
  - 2.1.1 Constants . . . 5
  - 2.1.2 The `programme` table and its relationships . . . 6
  - 2.1.3 The `module` table and its relationships . . . 7
- 2.2 Views . . . 9
  - 2.2.1 Complete Modules . . . 9
  - 2.2.2 Complete Requisites . . . 9
- 2.3 Procedures . . . 9
  - 2.3.1 Creating a pre-requisite . . . 10
  - 2.3.2 Adding an alternative pre-requisite . . . 10
  - 2.3.3 Creating a co-requisite . . . 11
  - 2.3.4 Adding an alternative co-requisite . . . 11
  - 2.3.5 Creating an anti-requisite . . . 12
  - 2.3.6 Creating an ID module . . . 12
  - 2.3.7 Creating a prefix association . . . 12
  - 2.3.8 Creating a catalogue entry . . . 13
- 2.4 Web Scrapers for Data Population . . . 13
  - 2.4.1 Prefix scraper . . . 14
  - 2.4.2 Module scraper . . . 14
  - 2.4.3 Requisite scraper . . . 15

3 Visualisation Design and Implementation **16**
- 3.1 Server . . . 16
- 3.2 Drafts . . . 16
- 3.3 Column-oriented Layout . . . 18
  - 3.3.1 Idea . . . 18
  - 3.3.2 Data pre-processing . . . 20
  - 3.3.3 Visualisation using D3 . . . 20

3.3.4 Styling and Interactivity 21

4 Conclusion and Future Work **22**

5 Acknowledgements **24**

A Relational Model **25**

A.1 School-relevant 25

A.2 Programme-relevant 25

A.3 Module-relevant 26

A.3.1 Requisite-relevant 26

A.3.2 Teaching-relevant 27

Chapter 1

Introduction

The aim of this project was to facilitate navigation and understanding of the University of St Andrews undergraduate course catalogue. In its current form, the course catalogue is a collection of PDF-documents split up by school and by sub-honours and honours level. This makes it difficult for students and advisers alike to figure out what modules students should be taking. Further complicating the matter is that the list of tables in the PDFs give no indication of what consequences module choices will have. For example not taking the 2000-level networking module in Computer Science leads to not being able to take CS3102, which further leads to not being able to take CS4103, CS4105, CS4302, and CS5022, due to the "chaining" of pre-requisites. This consequence of not taking a second-year, sub-honours module is cumbersome to figure out based on the PDFs, as one would need to look through all the honours modules in second year, well beyond the modules relevant for that year or even the next one, which are listed in a separate document from the one the student would be looking at, i.e. the sub-honours document(s).

With joint honours, the difficulty increases even more, due to the large variety of required modules for the programme in combination with the requirements of other modules the student might be interested in.

This project aims to solve these issues, or at least facilitate the planning process, by visualising the modules and their requisites as a network of nodes and edges respectively.

Chapter 2

Database Design and Implementation

It was necessary to design and implement my own database instead of using one by the university, as they never responded to Dharini's request for either access or a dump of the database.

2.1 Entity Relationship Diagram

The database underwent several re-designs and iterations. The current version, which the underlying MariaDB database is implemented on, is version `13.c`. Note that in ER-diagrams the relationship quantifiers are read opposite to UML-diagrams, i.e. following a path out of a table, the quantifier relevant to the table is on the side *going into* the relationship, and *not* on the side coming out of the relationship.

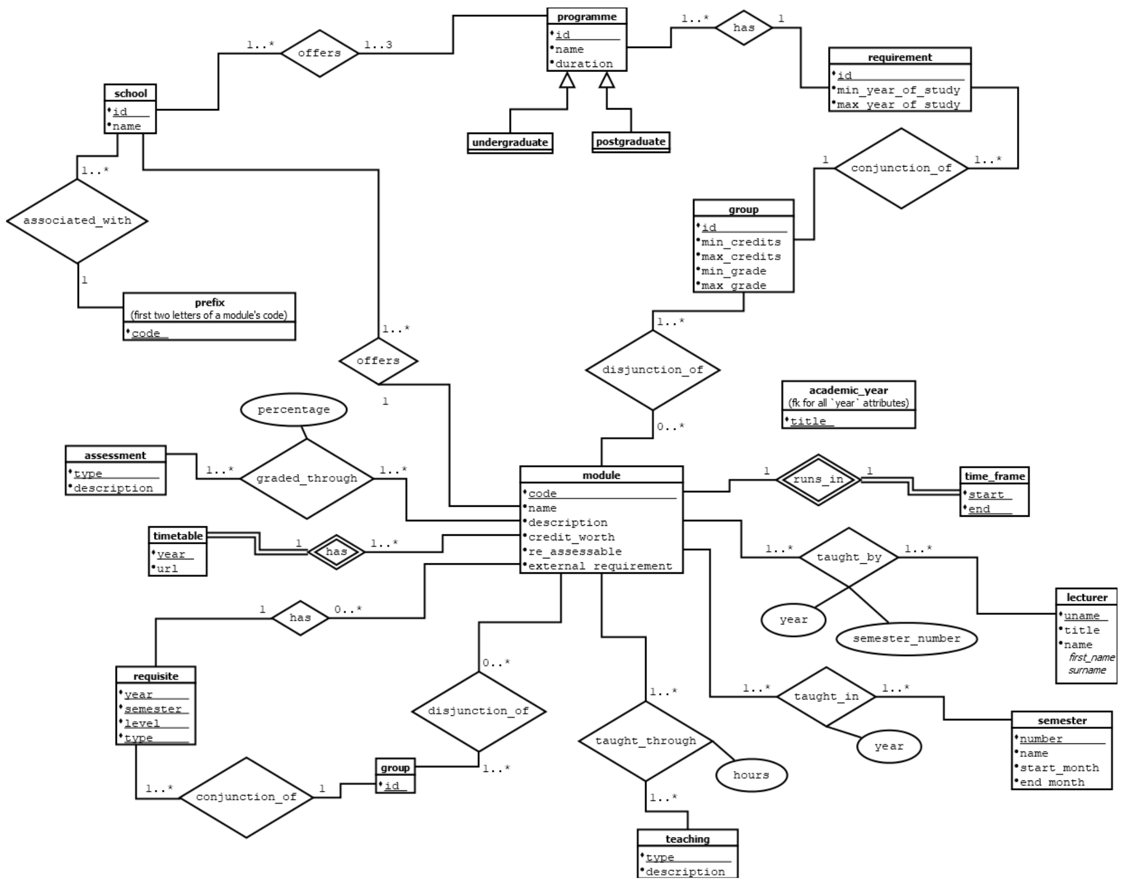

Figure 2.1: The ER-diagram which the underlying database is based upon.

Note that two tables cannot have the same name. As such, there are slight inconsistencies between the names in the ER-diagram and the names used in the database: the `group` table used for programme requirements is called `disjunctive_group` and the `group` table used for module requisites is called `requisite_group`. For the full details, see the relational model (Appendix A).

2.1.1 Constants

There are 5 tables which contain "constants". These are:

`assessment` Contains the 3 types of assessment that the university acknowledges/use on their website. These are: "Written Examination", "Practical Examination", and "Coursework". The first two entries have a description pulled from the university's website. There is no official description for coursework.

`semester` Contains the 4 time periods that a module can run in: “Semester 1” (numbered 1), “Semester 2” (numbered 2), “Summer” (numbered 3), and “Whole Year” (numbered 4).

`teaching` Contains the types of teaching that I was able to come up with, it is possible that there are more. Currently, the table contains the values “Lecture”, “Tutorial”, “Seminar”, “Supervised laboratory work”, “Field work”, “Exercise class”, “Independent study”.

`school` Contains all the schools/departments of the university with a numeric ID to quickly identify them. Because they also offer modules, the following are also considered schools: “The Music Centre”, “English Language Teaching”, “The Graduate School”.

`prefix` Contains two letters which uniquely identify what school a module is offered by, e.g. modules starting with ‘CS’ are offered by the School of Computer Science. This is useful when adding modules to the database, as it allows the procedures to look up the relevant school given a module code to add, which greatly facilitates creating modules by scraping them from the university’s website. Being able to do this is largely the reason why the three ‘non-schools’ are in the `school` table. The inter-disciplinary modules, code ‘ID’, can be offered by any school and so do not have an entry in this table.

2.1.2 The `programme` table and its relationships

The `programme` table contains a unique ID, a name (e.g. “Bachelor of Science (Honours) Computer Science”), a duration (e.g. “4 years”), and a type (“Undergraduate” or “Postgraduate”). Looking to the left of the table, we can see that a programme is offered by at least 1 and at most 3 schools, modelling single- to triple-honours. Triple-honours are rare, but are offered by some language programmes, e.g. the “Master of Arts (Honours) French, German and Italian” programme. Going back that path, we can see that a school offers at least one programme.

Looking to the right of the `programme` table, things get a bit more complicated. This is due to requirements being expressed in conjunctive normal form (CNF), which is tricky to model in a database. A `programme` has at least one `requirement` consisting of a conjunction of at least one `group` which itself consists of a disjunction of at least one `module`. This ‘chain’ of tables models CNF in the following way: each `group` represents a compulsory part which must be fulfilled to satisfy the `requirement`; each `module` in the group then represents several options of which *at least one* must be taken to fulfill the `group` sub-requirement. Written out in pseudo-logic, it could look as follows:

$$requisite := (module_{a1} \vee module_{a2} \vee ... \vee module_{an}) \wedge \; ... \; \wedge (module_{z1} \vee ... \vee module_{zm})$$

Each `requirement` has a unique ID, and a minimum and maximum year of study. The latter is because whilst most STEM-programmes have the yearly requirements matching the module-level (e.g. second year means 2000-level modules), some other departments (e.g. the School of Philosophy) have requirements along the lines of "*0 to 80 credits from PY1000-PY2000 level modules across first and second year*". A `requirement` belongs to exactly one `programme` and contains at least one `group` of modules.

Each `group` in a `requirement` has a unique ID, a minimum and maximum number of credits, and a minimum and maximum grade to achieve. This is to model things similar to the School of Philosophy example given above, and also the case where certain programmes require an 11 or higher across all modules. A `group` belongs to exactly one `requiremnt` and contains at least one `module`. Modules do not have to be part of any requirement but can exist as purely optional. As previously stated, modules in a `group` are alternatives to each other, where at least one of the modules in the `group` must be taken. There is bound to be some duplication of module-groups across certain programmes. However, this model was the best way Dr. Ruth Letham and I could come up with to capture the CNF-structure of programme requirements.

2.1.3 The `module` table and its relationships

The `module` table contains a unique code (e.g. "CS1002"), a name, a description, a credit worth, whether the module is re-assessable or not, and optionally some external requirements. The external requirements are free text, as they are intended for requirements like "*Additionally, you must have mathematics (either higher or A-level) at grade A or better*", which are not a module but still a requirement. Since it is rare for modules beyond 1000-level to have these, external requirements may be `NULL`.

Going clockwise, the first table around the `module` table is the `academic_year` table. This table is not directly linked to the module table, but affects nearly all the attributes/details of a module, represented by the other surrounding tables. These may vary from year to year, and so instead of duplicating the year across multiple tables, they refer to an entry in the `academic_year` table. The '`title`' of an academic year is its string-representation, e.g. '2019-2020'. This is the way the website refers to academic years and is arguably easier to read, and model, than having two primary keys `start_year` and `end_year`.

The next table is the `time_frame` table. It represents the duration that a module is offered. If this is not known, the `end` entry could be set to 100 years in the future. The idea behind this table was to avoid students, or advisers, planning for modules in 2 years time only to then find out that the module was due to be discontinued 1 year later. A module runs in exactly one time-frame, and each time-frame is associated with exactly one module.

The next three tables are related to the teaching of the module. A module is taught at least once, in a year and a semester, by at least one `lecturer`. A `lecturer` has a user-name (e.g. teh6), a title (e.g. 'Dr.'), and a name which is composed of a first name and a surname. A `lecturer` teaches at least one module. Each module is also taught in at least one `semester`, which may vary from year to year. A `semester` has a number, a name, and a start and end month. There is at least one module per semester per year. The final table related to the teaching of a module is the `teaching` table. A module is taught in a certain number of hours, using a certain teaching type, e.g. lectures. Each teaching type also has a description, explaining what that teaching type is.

Continuing clockwise, there is a loop of tables. This loop uses the same 'chaining' of tables used to model programme requirements, to model module requisites. Similar to programme requirements, module requisites are expressed in CNF: a `module` may have 0 or more `requisites`, each of which consists of at least one `group`. All the `groups` must be satisfied to satisfy the `requisite`, and each `group` contains at least one `module`. If a group contains multiple `modules`, at least one of them must be taken to satisfy the `group` sub-requisite. The main differences to programme requirements are that module requisites can vary across academic years, semesters, and levels ("UG", "PGT", or "PGR"), and have a type, i.e. "Pre", "Co", or "Anti". A further difference is that the `group` table serves purely to provide a group ID to identify alternatives, and that these alternatives are themselves entries in the `module` table. A `module` does not have to be part of a requisite `group`, but each `group` must be associated with exactly one requisite.

The final two tables, going clockwise, contain additional information about a module. A `timetable` is identified by its associated `module` and year, but due to timetables not being agreed far in advance, I simply store the URL to the timetable information, if exists. If the information does not yet exist, the URL is `NULL`. Finally, each module is graded through at least one type of `assessment`, e.g. "Written Examination". There is a percentage associated with this, which must not be greater than 100%. A typical distribution for CS-coded modules is to have 40% coursework and 60% written examination.

This concludes the various attributes and relationships associated with the `module` table. It turns out modules are fairly complex, but I believe the current data-model captures all that is necessary for the project, and possibly a bit extra which could be useful for querying about details of a specific module.

2.2 Views

There are two views in the database: `complete_modules` and `complete_requisites`.

2.2.1 Complete Modules

The `complete_modules` view displays information similar to what can be found on the university's module search with the name of the school offering the module also being displayed. Similar to the university's portal, there is an entry per module per taught instance, e.g. if a module is taught this year and the next, there will be 2 entries. This is also true if the module is taught twice per year. The view is constructed by joining the `module` table with the `taught_in` table, matching on the module codes, and then further joining with the `school` table, matching the module's `school_id` to the `id` of the `school`.

2.2.2 Complete Requisites

The `complete_requisites` view displays all details of all the requisites, i.e. what module has the requisite, what type of requisite it is, the disjunctive group IDs, and the modules involved. The view is constructed by first joining the `requisite` table with the `requisite_group` table, matching the requisite IDs, and then joining with the `disjunctive_group` table, matching the disjunctive group IDs. The resulting view is quite big, as it duplicates rows for each requisite, for each alternative in a group. This could potentially be improved by only duplicating rows for each group and coalescing all the module codes in a group into a list-like structure. The potential disadvantage to this, is that it might be more difficult to deal with on the machine-readable side of things, e.g. a server.

2.3 Procedures

To facilitate data insertion, and by extension the web scraping, there are several stored procedures defined in the database:

- `create_pre_requisite`
- `add_alt_pre_req_by_existing_pre_req`
- `create_co_requisite`

- `add_alt_co_req_by_existing_co_req`
- `create_anti_requisite`
- `create_id_module`
- `create_prefix_association`
- `create_catalogue_entry`

The functionality and details of each of these is explained in this given order in the sub-sections below.

2.3.1 Creating a pre-requisite

To create a pre-requisite, the user must supply a source module code, a target module code, the academic level that the requisite applies to, the academic year the requisite concerns, and the semester the requisite concerns. These arguments must be provided in this order. The **source** module is the module that **requires** something, and the **target** module is the module that **is required** by the source.

The procedure checks that both the modules exist and, if they do, creates a new entry in the `requisite` table, fetches the newly created requisite's auto-incremented ID, uses that ID to create a new entry in the `requisite_group` table and fetches the new requisite group's auto-incremented ID, and then uses *that* ID to create an entry for the target module in that group (through the `disjunctive_group` table). If either the source or the target module does not exist, the procedure errors, reporting which module did not exist.

Each pre-requisite created using this procedure is considered to be conjunctive, i.e. each must be satisfied to be able to take the source module. To create disjunctions, add alternatives, use the procedure described in the next sub-section.

2.3.2 Adding an alternative pre-requisite

To add an alternative module there must already exist a pre-requisite to add it to. An alternative is created by knowing the source (i.e. the module which requires the other module(s)) module code; an existing target module code, as well as the academic level, academic year, and semester of the existing pre-requisite; and finally, the module code of the alternative target module.

The procedure checks that the alternative to add exists and, if it does, uses the existing details in combination with the `complete_requisites` view to find the group ID that the alternative should be added to, and then adds it to the group with that ID (through the `disjunctive_group` table). If the alternative module does not exist or there is no group matching the given details of the supposedly existing requisite, the procedure errors, reporting what went wrong.

Modules added using this procedure are considered alternatives, i.e. at least one of them must be taken to satisfy the group requirement. To add another module which must *also* be taken, use the procedure described in the previous sub-section.

2.3.3 Creating a co-requisite

To create a co-requisite, the user must supply the same arguments as for creating a pre-requisite, i.e. a source module code, a target module code, the academic level that the requisite applies to, the academic year that the requisite concerns, and the semester the requisite concerns. The procedure checks that both the modules exist and, if they do, creates the co-requisite through the same steps as creating a pre-requisite. If either the source or the target module does not exist, the procedure errors, reporting which module did not exist.

Each co-requisite created using this procedure is considered to be conjunctive, i.e. each target must be taken along with the source module if one is to do the source module. To create disjunctions, add alternatives, use the procedure described in the next sub-section.

2.3.4 Adding an alternative co-requisite

To add an alternative module as a co-requisite, there must already exist a co-requisite to add it to. An alternative is created by knowing the same parameters as for adding an alternative pre-requisite, i.e. an existing source module code; an existing target module code; the academic level, academic year, and semester of the existing co-requisite; and the module code of the alternative target module. The procedure uses the same steps as the procedure for adding alternative pre-requisites, with the only difference being modifying requisites of `type` 'Co'. If the alternative module does not exist or there is no group matching the given details of the supposedly existing requisite, the procedure errors, reporting what went wrong.

Modules added using this procedure are considered alternatives, i.e. at least one of them must be taken along with the requiring/source module if one wants to take the source module. To add another module which must `also` be taken along with the source module, use the procedure described in the previous sub-section.

2.3.5 Creating an anti-requisite

To create an anti-requsitie, the user must supply the same arguments as for creating a pre-requisite or a co-requisite, i.e. a source module code, a target module code, the academic level that the requisite applies to, the academic year that the requisite concerns, and the semester the requisite concerns. The procedure checks that both the modules exist and, if they do, creates **two (2)** anti-requisites: one from source to target, and one from target to source. Each anti-requisite is created using the same steps used to create a pre-requisite. Creating the requisites both ways guarantees that no-one can accidentally be advised onto a module they had an anti-requisite for due to someone forgetting to add the record of the anti-requisite on the other side. If either the source or the target module does not exist, or there already exists an anti-requisite with all the given specifics in either direction, then the procedure errors, reporting what went wrong.

There is no option to add an alternative anti-requisite as it does not make sense to think/talk about alternative anti-requisites; All anti-requisites are conjunctive. This can also be reasoned about by looking at the logic representation:
$\neg(a \lor b)$ is the same as $\neg a \land \neg b$

2.3.6 Creating an ID module

To create an interdisciplinary (ID) module, the user must supply the name of the school offering the module, the module code, the module name, the academic year the module runs in, the module's SCOTCAT credit worth, the number of the semester the module runs in, the module description, and a boolean indicating whether the module can be re-assessed/re-sat or not.

The procedure checks that the module is ID-coded and that there is a school with the given name. If either is not the case, the procedure errors, reporting what went wrong. If everything is okay, the procedure inserts the academic year in the `academic_year` table unless it is already present, finds the school ID based on its name, creates the module, and creates an entry in the `taught_in` table. This ensures the academic year exists and that the module is taught at least once. Using the school name to look up the ID helps prevent the user potentially misremembering or mis-typing the school ID as names are easier to remember.

2.3.7 Creating a prefix association

To create a prefix association, the user must supply the prefix (e.g. 'CS') and the name of the school to associate it with (e.g. 'School of Computer Science'). Since all schools will have some prefix associated with them, if this procedure is called with a non-existant school, the school will be created. If the school already exists, it will not. In either case, the ID of the school is found, and used to create an entry in the `prefix_association` table. Schools can be associated with multiple prefixes, but each prefix should match exactly one school.

2.3.8 Creating a catalogue entry

A catalogue entry is defined as the information that comes up when searching on the university's catalogue portal, with the exception of requisites and external requirements, as these would require the procedure to support variadic arguments. This procedure relies on the entries in the `prefix_association` table and therefore should never be used with ID-coded modules, as they cannot be uniquely mapped to a school.

To create a catalogue entry, the user must supply the module code, the module name, the academic year the module is taught in, the module's SCOTCAT credit worth, the number of the semester in which the module is taught, the module description, and a boolean indicating whether the module can be re-assessed/re-sat or not.

The procedure creates the academic year if it does not already exist, uses the `prefix_association` table to look up the school offering the module based on the first two characters in the code, creates the module if it does not already exist, and creates a `taught_in` entry for the relevant academic year and semester. Attempting to re-create an existing module with different semester and academic year inputs will succeed by creating an entry in the `taught_in` table, but this will also warn that the module already existed.

2.4 Web Scrapers for Data Population

I initially thought of entering the data manually. This might have been fine for just the Computer Science modules, although it probably would have taken at least a day of very repetitive work. The number of modules for CS is not that many. The problem arises from the fact that the website has a row for each year and semester that a module is taught, often resulting in 2 or 4 entries for the same module (the website only shows the current academic year and the previous). Alice Lynch wrote a web-scraper in Python, based on the data I thought I needed. I then modified this scraper to include the SQL-queries, and to parse and split the strings to extract the necessary information. This resulted in 3 scrapers:

- `module-magic.py`

- `prefix-magic.py`
- `requisite-magic.py`

The '`magic`' part of the name is because the scrapers are highly specialised and use a lot of magic regular-expressions and string-splitting. They can be used to scrape one thing from one website, and nothing more. If something else needs scraping, it would be better to copy and modify the original scraper rather than try to extend any of the existing ones. Each scraper takes some arguments and flags. The details of these can be found by passing the `-h` or `--help` flag when calling them.

Common to all the scrapers is that they print the SQL-queries that will be executed and asks the user to confirm them. **It is highly recommended** to skim over all of these to sanity-check that none of the scraped pages contained some oddity not previously encountered. This happened on several occasions when I was using them, so whilst some oddities have been caught and adapted to, it is still likely that there might be others.

2.4.1 Prefix scraper

The prefix scraper was used to scrape the schools and the various module-code prefixes associated with them. This was done by constructing a dictionary mapping prefixes to school names. The key-value pairs are then stored in the database using the `create_prefix_association` procedure.

It is unlikely that this scraper has to be used more than once as it scrapes all the prefixes rather than a specific subset.

2.4.2 Module scraper

The module scraper takes a prefix/school code and scrapes all the modules matching that prefix. Since the table-view of the website has duplicate entries for modules taught in both semesters, compared to the page for the module which says "`Semesters: BOTH`", the semester number is scraped from this table. The remaining details, i.e. the module code, module name, academic year, SCOTCAT credit worth, module description, and whether the module is re-assessable, are all scraped from the catalogue page.

There are a couple of features useful for testing and debugging:

- The number of modules scraped can be limited by passing the `-l` or `--limit` flag followed by an integer.

- Passing in a module code or a prefix followed by a number (e.g. '`CS3`') will work and only scrape that specific module or the modules at that level respectively.

2.4.3 Requisite scraper

The requisite scraper takes a prefix and scrapes all the requisites of all the modules matching that prefix. This scraper is the largest and messiest because of the vast number of inconsistencies across the website. A few examples are: Undergraduate requisites can be denoted by the word 'Undergraduate', the abbreviation 'UG', or the abbreviation 'Ug', and similar for postgraduate requisites ('PGT'); which notation is used is usually consistent in the same description, i.e. both undergraduate and postgraduate requirements are referred to by the capitalised abbreviation, except for when it isn't; undergraduate requirements come before the postgraduate requirements, except for when they don't; it is usually only 5000-/masters-level modules that have co-requisites, except for when it isn't; etc... All of these edge-cases lead to a number of magic regular expressions, string-matching, and string-splitting. The current version seems to work, but it may be the case that when more schools' modules are scraped (beyond just CS and Biology), more edge cases will pop up. These will then have to be added to the scraper.

The scraper starts by grouping the pre-, co-, and anti-requisites for each module, for each semester and academic year. If a module has no requisites of a certain type, the variable the group would be stored in is `None`. Each of these groups is then split into 'levels', i.e. undergraduate or postgraduate, using a large number of `if-else` statements, regular expressions, and substring-matching. Finally, each of the undergraduate and postgraduate, pre-, co-, and anti-requisites are committed to the database using the appropriate stored procedures. The scraper takes alternatives into account, calling the stored procedures to create alternative requisites as appropriate.

Chapter 3

Visualisation Design and Implementation

3.1 Server

To be able to use D3.js and construct a web-page, I needed a server which could query the database and supply the data in the formats I required. This server was built using Python Flask due to its reputation for being easy to get up and running, and easy to use. Although the Flask server was initially intended to be minimal, supplying the raw data and nothing more, it eventually became more sophisticated. It currently handles data-filtration, and keeps a directed graph of the modules and their requisites for server-side finding of children or ancestors.

3.2 Drafts

To get an overview of what the network of modules looked like, I put the data into the form of a JSON-object containing two arrays: '`nodes`' and '`links`'. This form is similar to the mathematical representation of graphs, a set of vertices and a set of edges, and is widely used by the D3 examples of network-visualisations.

To get an overview of what the network looked like, I modified a D3-example of a force-directed graph.

Interestingly, the CS-modules form a more circle-layered structure, compared to the Biology-modules which form two 'clusters' and several small 'constellations' of modules. This is because CS has more core modules which then enable a lot of different topic, whereas Biology splits into two 'factions' (molecular and medicinal) early on. There are also some modules which are completely disjoint from the network. These are mostly 5000-level PGT master modules, along with

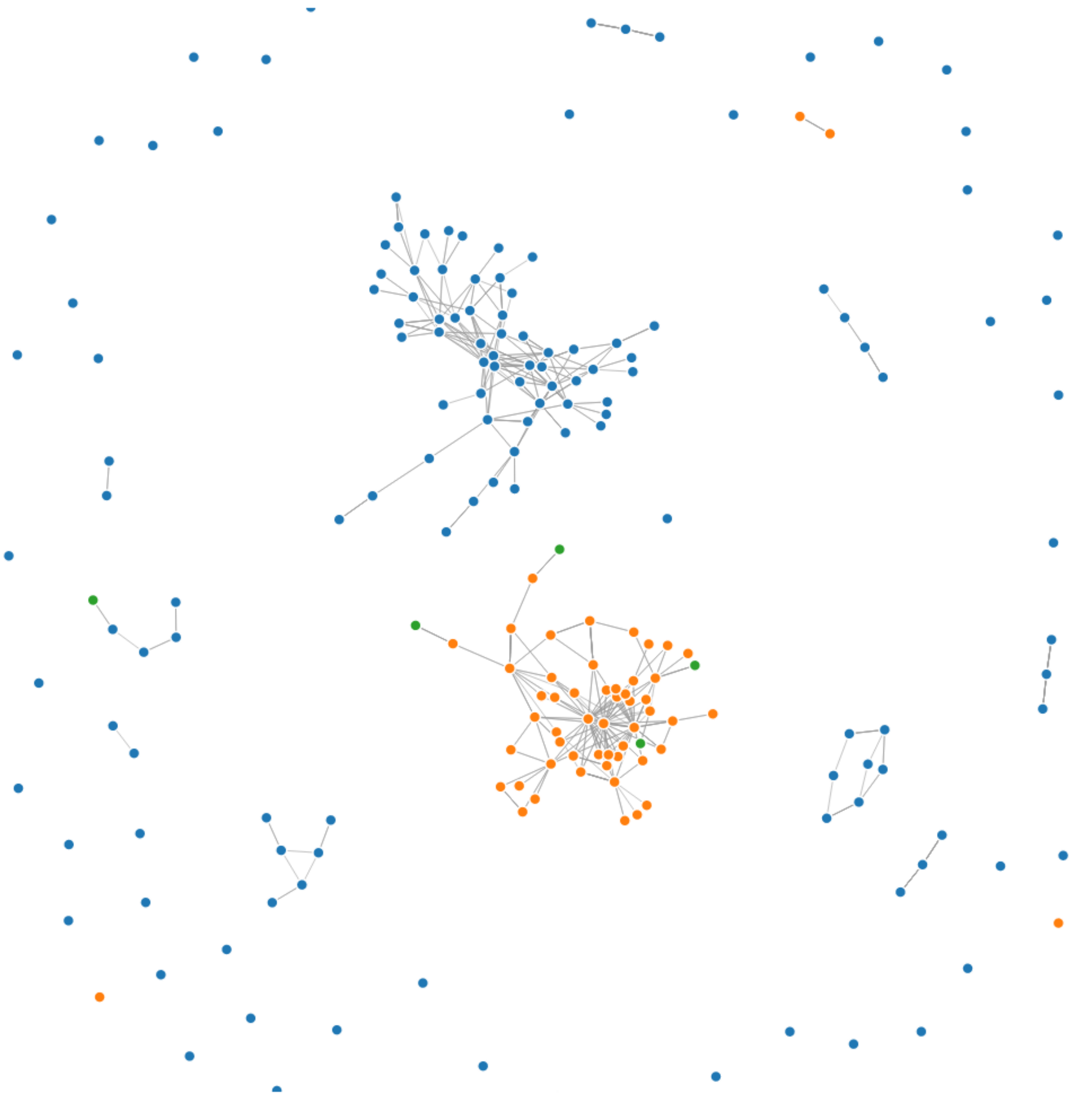

Figure 3.1: Initial Force-Directed draft

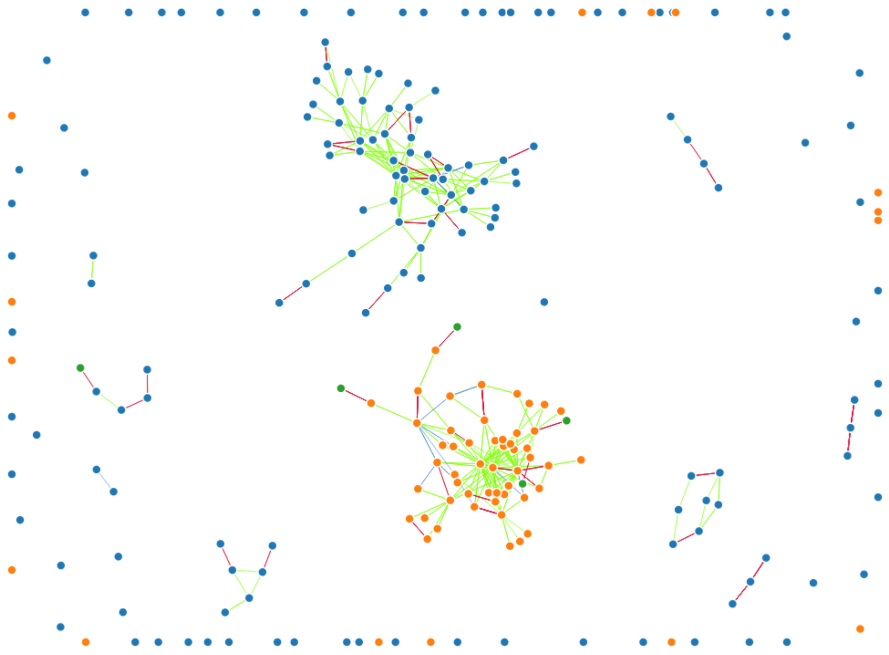
Figure 3.2: Force-Directed draft with colour-coded links and bounded area

some gateway modules.

I then modified this example to take the type of requisites into account, colour-coding them. I used green for pre-requisites, blue for co-requisites, and red for anti-requisites.

Having familiarised myself with D3 and data-handling, I created a column-oriented draft. Once it was ready, I took it and used it to create the final visualisation.

3.3 Column-oriented Layout

3.3.1 Idea

The idea of the column-oriented layout was that each column would represent a level, 1000-5000. I considered having columns per semester. However, this would require some nodes to have double-width for whole-year modules and would further complicate how links between levels would be shown. Therefore, I decided against it and stuck with a column per level. The requisites would be smooth, colour-coded lines between the modules. The idea was that on hover/mouse-over, the links would show what modules the module selected en- or disabled, and if the person chose to take the module, what they would need to take before that.

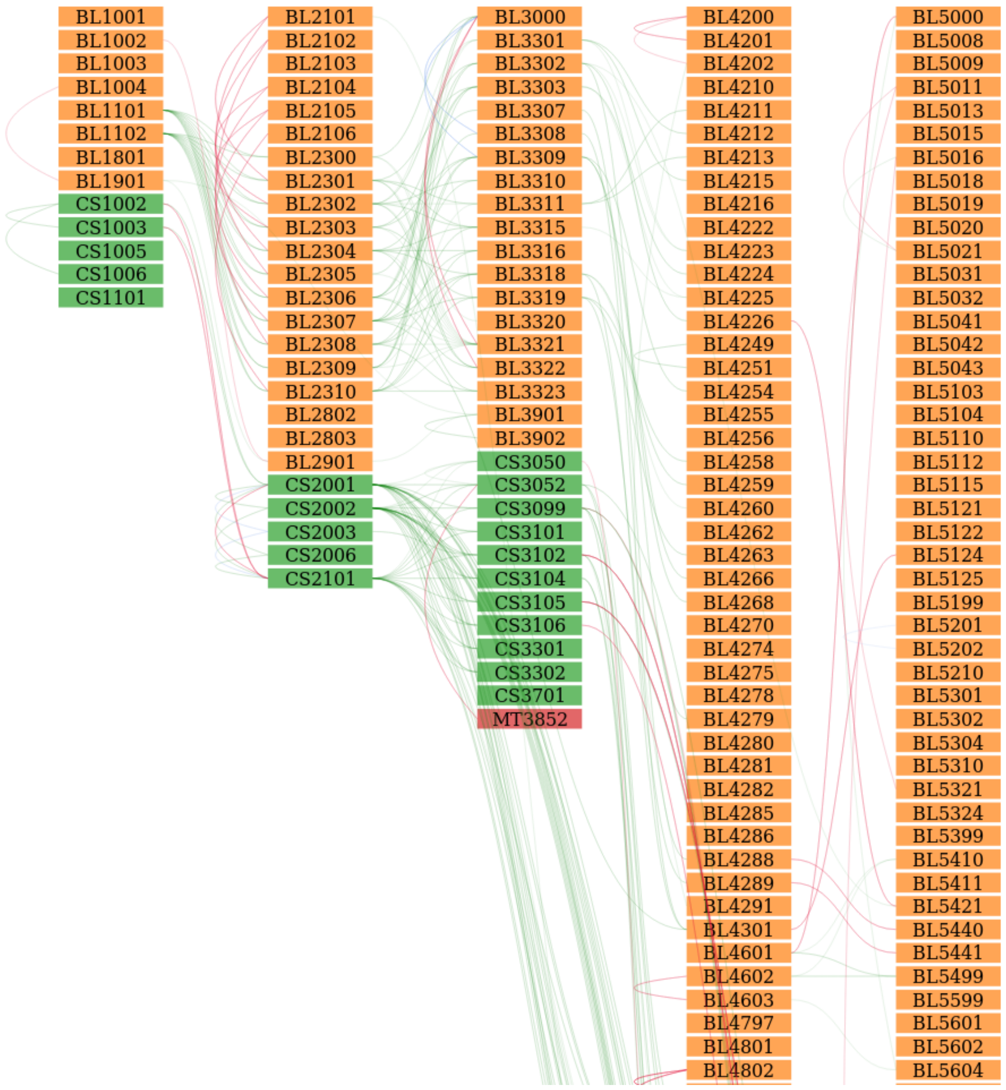

Figure 3.3: Initial column-oriented draft

3.3.2 Data pre-processing

To take full advantage of D3 and the DOM, I used D3-nest to nest both the modules and the requisites. The modules were nested by level, using a function which returned the 3rd character of their code, i.e. the level. The requisites were nested by their type.

To be able to highlight the ancestors of a node, I needed a function which found these going back an arbitrary number of links with potential loops. This was a more problematic part of the project, and I eventually ended up with two solutions. The first was using a graph-library on the server-side and using it to find the ancestors of a node. The second was an idea by Iain Carson, which enabled me to do it client-side. By constructing a `Node` object in JavaScript and using it to represent each module, I could add a method to the prototype of the object, thereby causing all instances to have that method. The method itself, called `returnNodes`, checks whether the node that called it is already in the list of relevant nodes and if it is, returns. If the node is not in the list, it adds it, finds all the links and ancestors of that node, adds the links to the list of links traversed, and recursively calls itself on all the ancestor nodes. This results in the list of nodes eventually containing all the ancestors, and the list of links containing all links traversed to construct the node-list. These lists are then used to associate CSS-classes denoting ancestry with the relevant DOM-elements, which in turn allows the browser to immediately select all ancestors by finding which DOM-elements have the `ancestor-of-<Node>` class.

3.3.3 Visualisation using D3

The nodes are spaced out using D3's `scale_band` method. This automatically also figures out the size of the bands, and allows specifying what percentage of the band should be padding. For the width, I set this to 50%, i.e. the space between each column should be 1 column-width, and I set the space between each row to be 15%, based on experimentation.

Each level in the nodes and edges have an SVG-group associated with them. This allows for easy translation of the columns and also allows the developer to easily inspect the relevant elements in the browser. A rectangle is drawn for each node, using the band-scales to determine width and height. Each `rect`'s fill colour is determined by the school, using the default D3 10-hue categorical colour scale, '`d3.schemeCategory10`'. The module code is then put as `text` in the center of the `rect`, towards the bottom. This is not a perfect solution, however, centering the text in the box is difficult due to the `rect`'s coordinates referring to the top-left corner of the rect, whereas the `text`'s coordinates, when centering, refer to the bottom-center of the text (and bottom-left when not centering). Taking the font-height into account is near-impossible, due to what browsers use which default font and whether users have changed that, and therefore, placing the `text` at the center of the `rect`, towards the bottom seemed like the best approach.

The links are drawn using cubic bezier curves. To make the endpoints match with the edges of a `rect` instead of always using the top-left corner, the width and height is taken into account. The path can be calculated using the D3 horizontal cubic bezier curve generator, ‘`d3.linkHorizontal()`’, with the ‘horizontal’ bit referring to the direction of the tangents of the cubic bezier links. This generator returns a series of SVG-path commands used to draw the line. If the link is between two modules in the same column, the path is instead calculated manually to achieve an arc (using a quadratic bezier line) rather than a straight line, which is what the generator returns in this case.

3.3.4 Styling and Interactivity

SVG-paths default to constructing a closed loop with a fill between the lines. Therefore, the fill is set to `none`. To make the highligthting “pop” more, the nodes and links have a lower opacity by default. Due to the large number of links and their overlapping, links have a minimally visible opacity of 0.1. Any `highlighted`-classed nodes and links have increased opacity and the nodes also have a black outline.

When a node is moused over, the `highlighted` class is added to all ancestor nodes and all links involved in this, resulting in them being highlighted. When the mouse then leaves the node, all `highlighted` classes are removed from the DOM-elements containing them.

Chapter 4

Conclusion and Future Work

A database model and implementation capturing module requisites and programme requirements was achieved. The model has proven to be adequate, as it was able to store the data scraped from the official university website. Accessing the database through a server, several visualisations were designed. These have provided several new insights into the state and structure of requisites, e.g. various old modules which are still needed as they could have been taken by students in their later years, and the fragmentation of some sub-categories of modules. The column-oriented layout has been incorporated into a website which allows the user to interact with it and filter the data, bringing the system closer to a deployable state. The column-oriented layout greatly facilitates establishing an overview and exploring consequences since each level is immediately visible compared to having to scroll through multiple PDFs. Whilst the project is not complete, it is now in a stage where someone could further develop it.

Several additions could be made to the project. The most essential would be scraping and inserting the degree programmes into the database, as well as scraping more modules, e.g. from the school of Mathematics and Statistics. Using the programme data, a system which displayed what programmes were available given the current module selection could be implemented. Or vice-versa: a system which highlighted the core modules for selected programme(s) could also be implemented. Additional filters could be added to filter by programmes, an academic year range, or perhaps a maximum number of years from the current to display modules in. Finally, a third network overview called "hive plots" should be implemented to provide another overview of the network. My initial idea was to have the schools be axes and to have the 1000-level modules nearest the centre, increasing the level going outwards. A potential problem with this visualisation is the number of requisites between modules in the same school. However, this visualisation should still be attempted.

Adding a feature to save the selection for retrieval after the current browser session is beyond the scope of this project, as it would then start to resemble a competitor to the existing advising system, which is not the aim of the project.

Chapter 5

Acknowledgements

I would like to thank Dr. Dharini Balasubramaniam for funding and supervising this project; Dr. Ruth Letham for her help with constructing the database model; Ms. Alice Lynch for enabling me to easily scrape the data from the web; Mr. Iain Carson for the help with D3, especially the `returnNodes` method, and advice with website design; and Dr. Uta Hinrichs for her help and feedback concerning the network- and column-visualisations.

Appendix A

Relational Model

Note: In general, all integer IDs are auto-incremented.

A.1 School-relevant

school(id: INT UNSIGNED, name: VARCHAR(127))

prefix(code: VARCHAR(2), school_id*: INT UNSIGNED)

programme_school(school_id*: INT UNSIGNED, school_id*: INT UNSIGNED)

Note: Modules keep track of their offering school.

A.2 Programme-relevant

programme(id: INT UNSIGNED, name: VARCHAR(127), duration: VARCHAR(63), `type`: VARCHAR(15))

requirement(id: INT UNSIGNED, min_year_of_study: TINYINT(1) UNSIGNED, max_year_of_study: TINYINT(1) UNSIGNED, programme_id*: INT UNSIGNED)

`group`(id: INT UNSIGNED, min_credits: INT(3) UNSIGNED, max_credits: INT(3) UNSIGNED, min_grade: TINYINT(2) UNSIGNED, max_grade: TINYINT(2) UNSIGNED, requirement_id*: INT UNSIGNED)

module_group(group_id*: INT UNSIGNED, module_code*: VARCHAR(6))

Note: There are no tables for the relationships, as requirement and `group` keep track of what programme or requirement they belong to respectively; A module group is a disjunction of modules as part of a requirement (and should probably be renamed, along with `group`).

A.3 Module-relevant

academic_year(<u>title</u>: `VARCHAR(9)`)

module(<u>code</u>: `VARCHAR(6)`, name: `VARCHAR(255)`, description: `VARCHAR(2047)`, credit_worth: `INT(3) UNSIGNED`, re_assessable: `BOOL`, external_requirement: `TEXT`, school_id*: `INT UNSIGNED`)

graded_through(<u>module_code</u>*: `VARCHAR(6)`, <u>assessment_type</u>*: `VARCHAR(63)`, <u>academic_year</u>*: `VARCHAR(9)`, percentage: `TINYINT(3) UNSIGNED`)

assessment(<u>`type`</u>: `VARCHAR(63)`, description: `TINYTEXT`)

timetable(<u>module_code</u>*: `VARCHAR(6)`, <u>academic_year</u>*: `VARCHAR(9)`,
<u>url</u>: `VARCHAR(255)`)

time_frame(<u>module_code</u>*: `VARCHAR(6)`, <u>`start`</u>: `DATE`, <u>`end`</u>: `DATE`)

A.3.1 Requisite-relevant

requisite(<u>id</u>: `INT UNSIGNED`, academic_year*: `VARCHAR(9)`,
semester_number*: `TINYINT(1) UNSIGNED`, academic_level: `VARCHAR(3)`,
`type`: `VARCHAR(4)`, source_module*: `VARCHAR(6)`)

requisite_group(<u>group_id</u>: `INT UNSIGNED`, requisite_id*: `INT UNSIGNED`)

disjunctive_group(<u>group_id</u>*: `INT UNSIGNED`, <u>module_code</u>*: `VARCHAR(6)`)

Note: Requisites keep track of the module they belong to; Requisite groups keep track of the requisite they belong to and exist purely to provide a requisite group ID.

A.3.2 Teaching-relevant

taught_by(module_code*: VARCHAR(6), lecturer_uname*: VARCHAR(127), academic_year*: VARCHAR(9), semester_number*: TINYINT(1) UNSIGNED)

lecturer(user_name: VARCHAR(127), title: VARCHAR(15), first_name: VARCHAR(127), surname: VARCHAR(127))

taught_in(module_code*: VARCHAR(6), semester_number*: TINYINT(1) UNSIGNED, academic_year*: VARCHAR(9))

semester(`number`: TINYINT(1) UNSIGNED, name: VARCHAR(10), start_month: int(2) UNSIGNED, end_month: INT(2) UNSIGNED)

taught_through(module_code*: VARCHAR(6), teaching_type*: VARCHAR(63), academic_year*: VARCHAR(9), hours: SMALLINT(3) UNSIGNED)

teaching(`type`: VARCHAR(63), description: VARCHAR(511))