

Course Recommendation Using Aggregated Information Retrieval

Fredrik Karlsson
Daniel Perván

Department of Electrical and Information Technology
Lund University

Advisor: Anders Ardö

June 27, 2013

Printed in Sweden
E-huset, Lund, 2013

Abstract

Study plan scheduling is a natural part of each student's life cycle at LTH. In the fourth and fifth year, the students have to customize their education by selecting courses that will give them the possibility to graduate. The selection and scheduling is done ad hoc by the students and is most often a quite time consuming act; the information is spread across different web pages and a lot of values needs to be summarized manually by the student.

This spring, a web based tool for study plan scheduling, searching and browsing has been developed to help the students. Using aggregation of obtainable structured data about the courses given at LTH and crawled unstructured course material from the course web pages, the system helps the students both by letting them create and validate their study plan, as well as easily explore suitable courses at LTH.

Acknowledgements

We would like to thank:

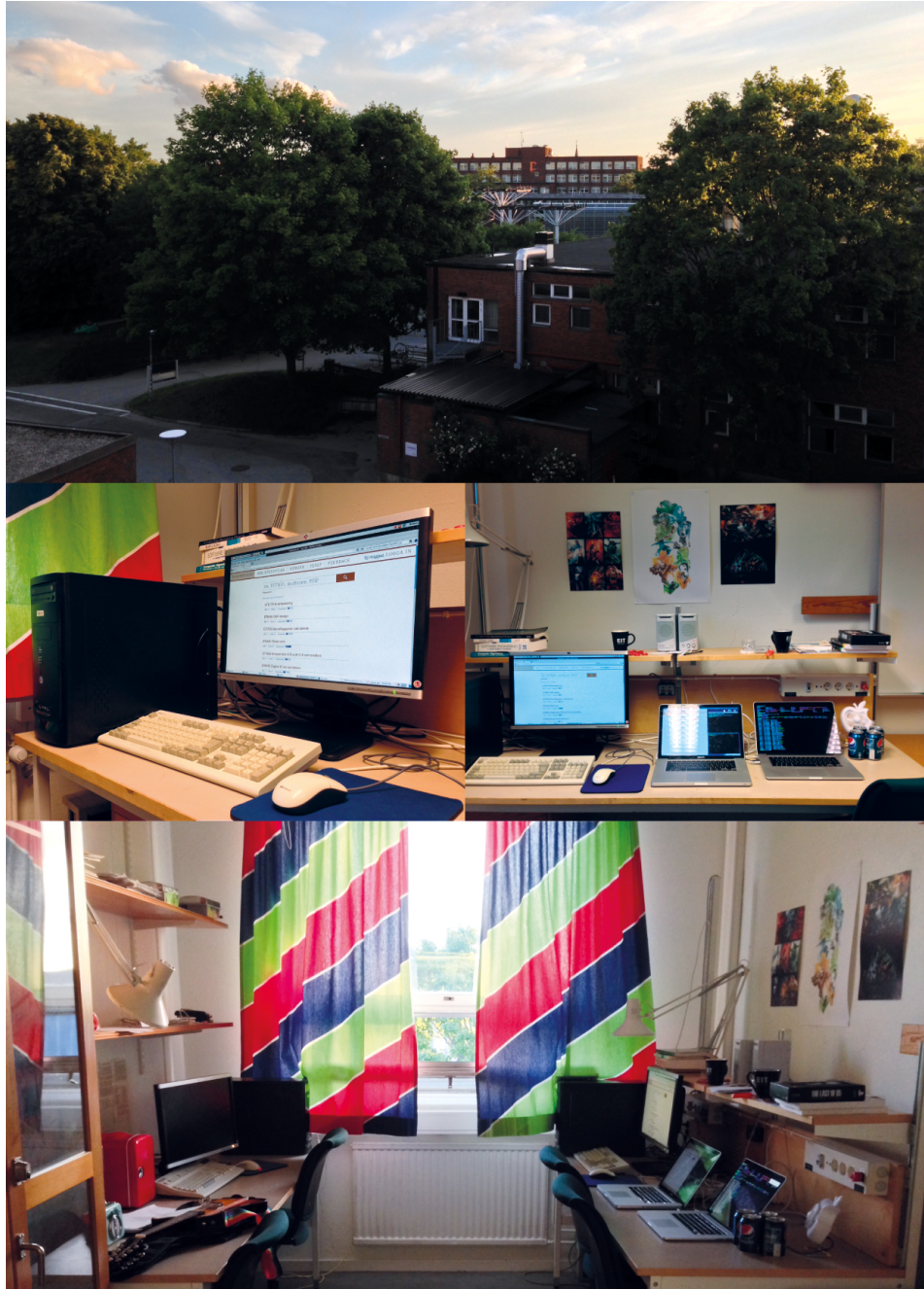
Our advisor Anders Ardö for the great insight and support that he has given us throughout the project.

Nora Ekdahl and Roger Berlin for their domain knowledge, suggestions and feedback on the prototype.

Karim Andersson for the data he has provided to us and the feedback we have received.

The technical staff at EIT for the computer support and hardware configuration.

Last but not least, we like to thank the students at the D and C programmes that sometimes almost voluntarily helped us to test and evaluate the system!



Exjobbsrummet

Table of Contents

1	Introduction	1
1.1	Problem statement	1
1.2	Prestudy	2
1.3	Background	3
1.4	Methods	6
1.5	Limitations	11
1.6	System structure	12
1.7	Responsibilities	12
2	Crawler	15
2.1	Introduction	15
2.2	Choice of implementation	15
2.3	Implementation	16
2.4	Results	19
3	Search Engine	21
3.1	Introduction	21
3.2	Choice of implementation	21
3.3	Overview of Solr	22
3.4	Configuration	23
3.5	Fields	24
3.6	Autocompletion	25
3.7	Search handler	26
4	Recommendation Engines	27
4.1	Similarity recommendations	27
4.2	Statistical recommendations	38
5	Course Dependencies and Validation	41
5.1	Parsing algorithm	42
5.2	Example	43
5.3	Graph structure	44
5.4	Graph visualization	45
5.5	Validation	45

6	Web Application Backend	47
6.1	Introduction	47
6.2	Implementation	47
7	User Interface	53
7.1	Course Information Page	54
8	Search Page	55
8.1	Layout	55
9	Study Plan Scheduler	59
9.1	Layout	59
9.2	Validation	62
9.3	Discarded interactivity	63
10	Course List	65
11	User Evaluation	67
11.1	Survey	67
11.2	Conclusions	68
12	Extensibility	71
12.1	Example	71
12.2	Suggested extensions	72
13	Discussion	73
14	Conclusions	75
15	Glossary	77
	References	79
A	Prestudy Questions	81
B	LoT data example	83
C	Code Examples	85
C.1	Home page	85
C.2	Gymnasist page	85
C.3	PageState Javascript object	88

Introduction

1.1 Problem statement

Choosing courses at LTH highly involves manual labour, where a significant amount of information about each course is typically not available in one place and some information not even explicitly stated anywhere. Instead it is required of the student to thoroughly read through the many different course web pages, the curriculum, timetable and many other sources. During this, the student is also required to consider eligibility of the course and the amount of hp (equivalent to ECTS points) the course will provide.

At the time of writing there exists no tool to easily accomplish this nor validate that the course composition can be used to graduate according to the Bologna model. Students are therefore forced to use ad hoc solutions such as complex spreadsheets or other rather creative methods.

Based on a prestudy using interviews of people studying their fourth or fifth year at LTH, a dedicated web service for study planning and course exploration might be a possible solution to the problems mentioned above.

In this master thesis we will examine the possibility of simplifying the process by implementing a prototype web based solution with aggregated searching of multiple sources and data structures. We will also try to enhance the result set by the use of statistical data from Ladok and data from CEQ at LTH. While some data may be mined directly from different databases or web pages, some may need to be produced through calculation.

Further, we will study the use of recommendation engines to suggest suitable courses in the system, likely using a hybrid type of engine based on relevant data that we may manage to get access to.

We will investigate if comparing course material by using VSM (Vector Space Model) or LSI (Latent Semantic Indexing) give useful results that can be used to find similar courses.

The system created as a part of the master thesis will be tested and evaluated by potential users of such system (e.g. students of LTH).

1.2 Prestudy

We have conducted a study to better understand the problems involved and how the students currently handle them. The study consisted of a number of interviews (some individual, others two at a time) using some written questions as a basis and we have then followed up interesting leads to get more detailed information. The paper with questions is available in Appendix A.

Some of the interview participants mentioned using the student's counselor for their programme. Therefore we also had a interview (no prepared questions) with him, to understand how he can help the students.

1.2.1 Results

While the test set is because of time constraints arguably too small and homogeneous to hold a more scientific relevance, the study could still discover some recurring trends among the test subjects as well as confirm some of our prior theories. Most students answered that a major challenge with choosing courses was that it was difficult to compare information between different courses. Typically, getting an overview of relevant information in general was a common nuisance when selecting courses.

Surprisingly, the absolute majority of the subjects claimed that they relied heavily on information provided by the CEQ system. Information that was considered as important varied from person to person, but "overall happiness with the course", "number of passed students" and "appropriate workload" occurred frequently. Many complained however that the CEQ data wouldn't always be accurate and that they mostly watched for numbers that swayed greatly out of the ordinary.

Documentation of students' study plans varies significantly. We encountered both complex solutions such as auto-calculating spreadsheets and manually updated text documents, as well as students simply attempting to keep all information in their heads. When we asked for the benefit of a united system concerning study planning and course exploration, practically everyone were greatly in favour of such a system and some even wondered why it didn't exist already.

However, when asked about a system that could automatically build an entire study plan, people were more sceptical, many claiming that it would be interesting in a technical sense and that someone might have some use for it, but that they would most likely not use it themselves.

Some students mentioned that they had meetings with the student counselor for their respective programme and thus received support and material from there. After an interview with the counselor we came to the conclusions that information received from him mainly consisted of the same information that is already publicly available on the web, though in a few different representations combined with formal guidelines and rules, as well as friendly advice of how to interpret the information.

1.3 Background

1.3.1 Web applications

The web has since the nineties been a great tool for distributing information[1]. However, with the introduction of modern web standards as well as the proliferation of standard compliant browsers, the web may today be used for much more. A web application may in short be described as a computer program built with web technologies and typically runs in a regular web browser.[22]

Even though there are some limitations to web technologies compared to native programming languages, they still feature benefits that make them valuable for some implementations:[23]

Cross platform Even though different browsers may render pages slightly different, it's still relatively easy to handle compared to the alternative of porting a native application to multiple platforms. With proper coding practices, the same application may work no matter processor, operating system or even I/O.

Instant distribution From a configuration management standpoint, releasing updates to an already released application is often complicated. Not all users update at the same time and some may choose to not update at all. With web applications as soon as the application is updated, every single user will instantly have access to that version as well.

Client system requirements In some ways, a web applications adheres to many of the principles of a thin client. While animations and similar still have to be rendered by the client, storage and many kinds of preprocessing may instead be handled by the server. This makes it especially useful for mobile units and other low powered machines.

Availability Web browsers are becoming more and more common and may today be found in everything from mobile phones and computers, to refrigerators. In contrast to native applications, no actual installation of a web application is required; if there is a web browser then the application will be available. Additionally, if all data is stored on the server then the user's settings and content may also be instantly available.

1.3.2 NoSQL

Historically, relational databases have very commonly been used as the conventional database structure of choice. A relational database is typically designed with a fixed predesignated schema for each and every table in the database. Every table require a primary index which is used to fetch the correct rows. Relational databases features the concept of foreign keys which enables easy merging of data between different tables. Lastly, this type of databases also features a hard consistency model called ACID, which defines how data is stored and handled.[18]

While relational databases are able to handle even the most complex data relationships and are typically considered a very reliable database structure, there are a few drawbacks. To handle this very strict consistency model, many relational

database systems have troubles with horizontal scaling[17] which means that as more rows are inserted into the system, the performance becomes worse. However, while the strict consistency model might be appealing or even an absolute requirement for some systems, for other this might be completely superfluous. Also, while it's technically possible to handle almost any data relationships with a relational database, the sheer complexity of the structure may eventually become exceptionally complicated to the degree where the programmers have great difficulties understanding the mere structure.

One solution to this problem is NoSQL systems. NoSQL refers to the use of database systems that aren't based on a relational database model. While the exact definition varies, many refer to it as *Not only SQL*, where SQL is one of the most common query languages for relational databases. Unlike relational databases, NoSQL systems don't necessarily have to comply with the strict ACID model. Instead they typically use looser models, sometimes categorised as BASE (Basically Available, Soft state and Eventually consistent)[18]. There are many different kinds of NoSQL systems available and each have their respective strength and weaknesses depending on what kind of problem they are supposed to solve. Because of the limitations and specialised nature found in many NoSQL systems, it's common to combine many different kinds of NoSQL databases as well as SQL databases to handle different parts of the system.

1.3.3 Searching

While searching data may be done in many different ways, they often share a few basic concepts. Generally, searching consists of two fundamental parts: a query and a collection of documents. Each document that is to be searched will first need to be described by a set of computationally readable keywords called index terms. What such term actually consists of varies from system to system, but is often composed of a single word from a document. More advanced search engines may instead use e.g. multiple words to somehow represent the general topic of the document. It is also common to apply additional preprocessing to the index terms in order to make it easier to process when searching. This might include storing additional spelling variations of a word, synonyms or even translations into multiple languages.

A very common type of preprocessing is stemming. The idea is that instead of having to store uncountable different inflected forms and variants of the same term (e.g. *walking*, *walked* and *walks*) that don't really add to the semantics, one may simply store only the most basic form of the term (e.g. *walk*) and then apply the same preprocessing on the query terms.

The search query is then also divided into index terms and later applied to the indexed document collection in order to find the relevant documents. The simplest concept of doing this is by merely finding the co-occurrences of each term between the query and a document, without any regards of neither grammar nor term position. This concept is often referred to as a so called *bag of words* representation[24, p.62]. A score for each document may then be calculated for example through simply counting the co-occurring term frequency which can be

represented as a *term-document matrix* seen below:

$$\begin{array}{cc} & \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{array}{c} t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} & \left[\begin{array}{cc} f_{1,1} & f_{1,2} \\ f_{2,1} & f_{2,2} \\ f_{3,1} & f_{3,2} \\ f_{4,1} & f_{4,2} \end{array} \right] \end{array}$$

Where d is an individual document, t is a term and f is the frequency of that term in a document. There are however many different ways to fill this matrix, depending on which model is used, but the core principles still remains. This core concept provides a very powerful basis for building a full fledged search engine.

There are three very common models that are based on the bag of words concept, namely the *boolean*, *vector space* and *probabilistic*. The boolean model is the simplest of the three and as its name might imply is based around boolean algebra. The model only regards the existence or absence of a term and not the number of occurrences in a document, why the term-document matrix is populated with solely boolean values representing the existence of a term within a specific document. The query consists of a boolean expression on terms to be searched for, and is then evaluated for every document to calculate their similarity. Because of its boolean structure, the boolean model cannot however handle partial matches and can only produce a boolean similarity score; either the query evaluates entirely as true or not.[24, p.64-65] Because of that, the boolean model can by itself only find a collection of documents without providing any internal ranking of the result set.

However, this is something that the vector space model directly tries to solve. Instead of only populating the term-document matrix with binary values, the vector space model uses calculated weights for every term. One very common way of calculating these weights is by using so called *TF-IDF* term weighting. TF stands for *Term Frequency* and refers to the number of times a specific term occurs in specific document. IDF stands for *Inverse Document Frequency* and refers to the inverse number of documents in the collection where the term occurs at least once. The TF-IDF score is calculated simply by:[24, p.68-77]

$$TF \cdot IDF = \frac{TF}{DF}$$

The theory behind the algorithm is that a term that occurs often in a document, is probably more relevant for describing that particular document and should be weighted higher when sorting. However, if it's a very common term that also appears in many other documents then it might not be important at all, even though it may appear very often in a single document. As always, there are a few different variations of the TF-IDF algorithm such as applying normalisation to the score based on document length, but the general principles stay the same.

To use these weights to calculate the similarity between two documents, the associated row in the matrix for the concerned documents is treated as a unit vector of N-dimensional space. The similarity score can then be calculated by the angle between these two vectors; the closer the two vectors are to each other, the more similar they are (e.g. by cosine similarity).[24, p.78]

1.4 Methods

1.4.1 Hardware

For this project to work, a server was needed. We got to borrow an old and rather slow machine with CentOS 6¹. During experiments and indexing with Solr we felt that the server did not have enough memory and so we had it increased from 2GB to 4GB. Later when using the almost completed prototype, the motherboard together with CPU and memory was upgraded. From a two-core AMD Athlon 64-bit (which sometimes had up to 100% load) we had it upgraded to a four-core Intel i5 and the memory was increased again to 8GB.

1.4.2 Software

CentOS uses yum² for software management and provides a simple way of installing and updating software components. To be able to get access to a wider array of software and versions, we configured yum to use extra repositories, namely the atomic repo³ and EPEL (Extra Packages for Enterprise Linux)⁴.

The following packages were installed through yum (not counting dependency packages):

zsh our favorite command-line shell

screen command-line window manager

apache httpd 2.2.15 web server

mysql-server 5.5 relational database

php 5.4 script language processor

php-tidy functions to sanitize and clean html in php

htop interactive process viewer

graphviz graph visualization software

java openjdk 1.5.0, 1.6.0, 1.7.0 java runtime

libreoffice office software suite with command-line conversation support

xvfb virtual display emulator

Some software is not available through any yum repositories and had to be manually installed. For this project we also installed:

Solr open-source search server using Lucene⁵

Neo4j transactional property graph database⁶

Tomcat open source servlet container⁷

¹<http://www.centos.org/>

²<http://yum.baseurl.org/>

³<https://www.atomicorp.com/channels/atomic/centos/6/>

⁴<http://fedoraproject.org/wiki/EPEL>

⁵<http://lucene.apache.org/solr/>

⁶<http://www.neo4j.org/>

⁷<http://tomcat.apache.org/>

Some of the software are servers and are running as daemons by themselves, but Solr was configured to run by Tomcat. Solr comes preconfigured with Jetty⁸ (another web server/servlet container), but in order for easier reboot and control over the Solr process, we used Tomcat instead as servlet container.

1.4.3 Languages

For simplicity, we chose to use PHP⁹ as our main programming language. Both authors are fluent in PHP and we have been working with it in the past years.

Making a web application, it is required to write some CSS¹⁰ and Javascript¹¹. Javascript is executed by the browser and is implemented somewhat different in different browsers. Therefore it is quite handy to use frameworks that makes Javascript more platform/browser independent. There are a few solid frameworks that does this, such as jQuery¹², PrototypeJS¹³, MooTools¹⁴, YUI¹⁵ and we for the same reasons as for PHP we chose to use jQuery above the other alternatives.

The databases needs to be queried in some way; MySQL uses SQL¹⁶ and Neo4j uses Cypher¹⁷. Solr is queried by sending HTTP-requests using a special query syntax¹⁸.

Both Neo4j and Solr are open source Java¹⁹ applications, but no programming in Java had to be done in this project.

To be able to configure Solr, XML (eXtensive Markup Language)²⁰ was needed. Another markup language that was used is JSON (JavaScript Object Notation), that makes it easier to pass data structures between, in our case, backend PHP and frontend Javascript.

1.4.4 Software configuration management

Development of the system has been simplified by enabling collaboration on the coding side by the use of version control software. We selected Git²¹ for the task and set it up to have two developer repositories that are synchronized by a third production repository. The developer repositories are then pulling updates from the production repository when it is practical and pushing updates when they are tested. The repository directories are placed in the web server root directory and therefore all the repositories are accessible through the web server. Normally the

⁸<http://www.eclipse.org/jetty/>

⁹<http://www.php.net>

¹⁰<http://www.w3.org/Style/CSS/>

¹¹<http://www.w3.org/standards/webdesign/script>

¹²<http://jquery.com/>

¹³<http://prototypejs.org/>

¹⁴<http://mootools.net/>

¹⁵<http://yuilibary.com/>

¹⁶<http://dev.mysql.com/doc/refman/5.0/en/sql-syntax.html>

¹⁷<http://docs.neo4j.org/refcard/1.9/>

¹⁸<http://wiki.apache.org/solr/SolrQuerySyntax>

¹⁹<http://www.java.com>

²⁰<http://www.w3.org/XML/>

²¹<http://git-scm.com/>

web server would be configured to host the different repositories as sub domains, but this was not done due to us having limited control over the domain name.

Along with the collaboration functionality Git also provides the means to check differences between the edited and checked in versions of code which is very handy at times and makes sure that unwanted changes stays unchanged.

1.4.5 Data sources

We have received both publicly available data in an easily parseable format as well as anonymized private data and we have crawled the web for even more data.

Our primary data sources are:

LoT Läro- och timplaner, contains meta information about courses given at LTH

CEQ Course Evaluation Questionnaire, contains feedback from the students about the courses

Student results Ladok (the register for the storage and management of higher education study results) contains the students' results

Course homepages usually maintained by the departments and contains course material such as lecture slides and laboratory instructions

Master thesis courses The departments at LTH gives Master thesis courses and it is required for the students to have taken one of those to be able to graduate

Läro- och timplaner

The LoT-database hosted by LTH contains meta information about courses. We need the meta data to tell when a course is given, how many hp it gives and what presciences it requires (among many other things).

The partial database dump that we have received (in spreadsheet form) contains the fields:

Course code A six-character long codeword that represents the course (e.g. EITN01 or EIT070)

Course name The name of the course

hp The amount of hp that is given when the course is fully completed

Level The level of the course, either G1, G2 or A i.e. basic 1, basic 2 and advanced respectively. In order to graduate from the D and C programmes, a certain amount of hp needs to be studied in advanced courses.

Programme The programme that the course is given to

Specialization The specialization that the course is given to, or *ALLM* if it is general

Choosability When the course is in the general specialization, it might be mandatory or not for the programme.

Year In what year of the education it is most reasonable to study.

Start period Which period (läsperiod) the course is given in. lp1 starts in the autumn and lp4 is the last term and ends just before the summer break.

End period In which period the course ends.

Course responsible The email address of the course responsible

Prescience requirements A free-text containing the prescience requirements that needs to be fulfilled in order to study the course. These requirements are considered hard and are checked upon applying for the course using Ladok.

Prescience recommended A free-text containing the recommended presciences for the course. These usually say what kinds of general knowledge you need to know or what other courses you need to have taken in order to understand the course fully.

Teaching goals (1-3) Three fields with free-text saying what the students are about to learn in the course²².

Examination form How the student is examined during the course. Usually written exam or project assessment.

Contents A short text describing what the course is about and in which areas it teaches²².

Rationale A text describing the need for the course²².

Other Usually meta data saying that the exercises are preparatory for the labs or that the course has an examination in the middle of a course spanning over two periods, and such information.

Web page The URL to the course web page. Sometimes this is the web page of the department giving the course instead.

It is important to note that one course is represented by multiple rows in the database dump; there is a different row for each program and each specialization that can study the course. This allows for having different prescience requirements for different programmes taking the course and it allows for giving the course in different years for different programmes as well.

The LoT data was inserted into our relational MySQL database.

A real example of a row in this database dump is given in Appendix B on page 83.

²²Some of the fields are meant to be printed on the web and do therefore contain HTML elements.

CEQ

The CEQ data comes from students filling in questionnaires²³ after finishing courses they have studied. The questionnaires are voluntary and the results are published by LTH²⁴. The CEQ involves 26 questions where the students answer if the claim in the question fits the events in the course or not. The student can answer 1–5, where 5 means that the student agrees fully and 1 that it was not like that at all. These numbers are then converted into a scale between -100 and 100 (where a 5 is 100, a 4 is 50, 3 is 0, 2 is -50 and 1 is -100). We have received historical CEQ-data from the last five years. The fields we have received from the CEQ database are:

Course code A six-character long codeword that represents the course (e.g. EITN01 or EIT070)

Year and period The time index for the CEQ, e.g. 2008/09 lp3.

Overall, I am satisfied with this course Value in the range -100 to 100

The course seems important for my education See above.

Good Teaching See above.

Appropriate Assessment See above.

Appropriate Workload See above.

Clear Goals and Standards See above.

Registered students The number of students who were registered on the course

Passed students The number of students that passed the course.

We have received CEQ data only for the courses (i.e. course codes) that our system currently supports. When courses change (no matter how much or little) they change course code and therefore they lose their CEQ history (which is intended). In our case it makes some courses have a very long history while others have a very short history.

The CEQ data was inserted into our relational MySQL database.

Ladok

The data we have received from Ladok is anonymized and contains the grades for students studying non-mandatory courses that our system supports; the data is up to five years old. The anonymization is done in a way that replaces the personal identity number (Swedish: *personnummer*) with an increasing number. This allows us to follow individual students but not identify them directly.

The Ladok data contains these fields:

Student index The converted personal identity number

²³Example: http://www.ceq.lth.se/info/dokument/filer/CEQ-enkat_engwebb.pdf

²⁴<http://www.ceq.lth.se/>

Course code A six-character long codeword that represents the course (e.g. EITN01 or EIT070)

Programme code The 5-character programme code, e.g. TDATY for the 270hp (i.e. old) D programme and TADAT for the 300hp D programme

Year and term A concatenated five-number string containing year and which term (i.e. spring or autumn, where spring is equal to period 3 and 4, and autumn is equal to period 1 and 2), e.g. 20082

Grade For UG courses (i.e. courses that only gives grades pass or fail) it is a number 0 or 1 and for TH courses (i.e. courses that have grades 3, 4, 5 and fail) it is a number 0, 3, 4 or 5. In both cases 0 means that the student have failed the course.

The Ladok data was inserted into our relational MySQL database, but will never be shown directly in any way.

Course homepages

The list of course homepage URLs in the system have been mined from the LoT web pages²⁵ that show which courses are given to which programmes. We have mined the pages for the D and C programmes respectively and manually made sure that they are correct.

Course web pages are of varying quality and quantity. We are only interested in human readable media such as text and not images. Text can be found in many different file formats and we have focused to crawl HTML, TXT, PDF, PS and Microsoft Office formats (e.g. Powerpoint presentations, Word documents, etc.). Some courses have published all lecture slides as well as links to other sources of information that are relevant to the course, while others only present the meta data of the course (i.e. when it is given and how many hp it gives).

Our crawler will download and define the course material for the supported courses and the crawled data will later be indexed to enable for full-text searching and for similarity measures.

Master thesis courses

The master thesis courses have been mined from the LoT webpages²⁶. The master thesis courses are quite similar to each others (same level, hp and periods) and it was quicker to mine them directly than to ask for a database dump containing them.

1.5 Limitations

The goal of the project is to create a functioning prototype that help a student to completely plan his or her study plan. The project has limited time and resources

²⁵<http://kurser.lth.se/lot/?val=program>

²⁶<http://kurser.lth.se/lot>

why some limitations are to be expected. The prototype will receive fairly small amount of testing and code review, far less than what would be expected of a commercial product, but still sufficient for this prototype. It will also only support the current desktop versions of Safari, Chrome and Firefox. Other versions and browsers may still work, but there will be no testing done for those browsers. The prototype will neither officially support any browser for mobile devices.

The prototype will only fully support two different programmes, namely the D and C programmes at LTH. This also means solely supporting courses that students from these programmes are eligible for. We will also not take in consideration that the content of the two programmes may change over the years, and will instead merely assume that the programmes always stay the same. Similarly, other data such as data from CEQ, Ladok or LoT will not be updated in real time and will simply be preprocessed and stored in the early stages of the project.

With that said, one goal of the project is still to enable relatively simple maintenance, where if needed new and updated content can easily be added to the system.

Concerning optimisation, we will only consider performance for a handful of concurrent users. Scaling is not be an issue that will be handled for this particular project. However, the system should still feel fast and responsive when used by a single or a few different users.

1.6 System structure

This report subsequently describes every part and module of the system in detail in the respective following chapters. An overview of the structure of the system can be seen in Figure 1.1.

1.7 Responsibilities

While most of the work has been closely collaborated upon, some modules has been more influenced by one of the authors, whereas he has had main responsibility for its implementation.

Daniel had responsibility over Solr and its configuration, the PHP backend framework and Neo4j configuration. Fredrik had responsibility over the crawler, prescience parsing, validation and extensibility spikes.

The residual work has been fully collaborated between the two authors.

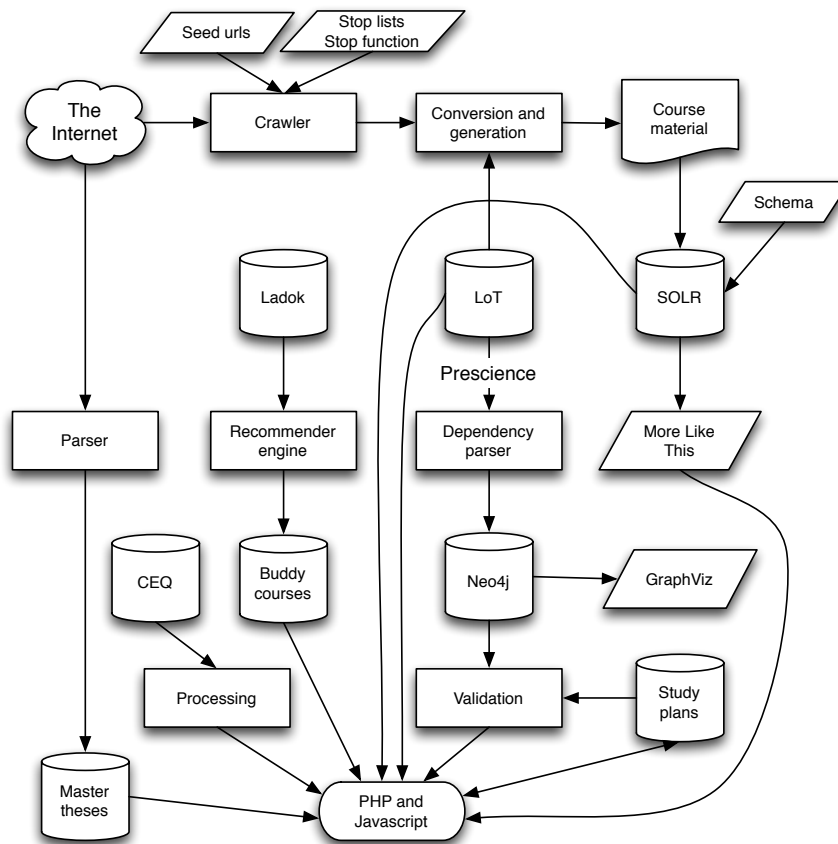


Figure 1.1: A diagram of the system structure

2.1 Introduction

Web crawling is the process used by search engines to collect pages from the Web[2]. The crawler is seeded with web pages (URLs) and then recursively finds new pages to crawl using hyperlinks found on the pages.

Many of the components in our system rely on data available on the course homepages to enable full-text search, course similarity comparison and course recommendation. For that purpose, we need a crawler to collect the course material for the supported courses. The seeded pages for our crawler are the course homepages for each course in our system. We only want the crawler to mine the documents relevant to the course and not to crawl outside of the course pages.

2.2 Choice of implementation

In our prestudy, we found that most course pages are created inside a web CMS (content management system) and therefore shares a lot of its page design with other course web pages given by the same department (i.e. using the same CMS). The page designs usually contains a lot of hyperlinks that are common to all courses such as department links and other university links (e.g. the LTH homepage).

There are several implementations of crawlers freely available on the internet today and therefore it is advisable to use an existing crawler. After sampling some crawler implementations (we installed and tried PHP-crawler¹ and Nutch²) and sampling some papers on crawlers[3, 4, 5, 6], it became evident that most crawlers are focused on containing the crawler within pages with similar semantics of the content while we are more interested in the page structure; specifically we want the crawler to only crawl pages that have a significant amount of links in common.

Since we did not find any crawler being able to carry out crawling the way we needed it to, we created our own. We based it on letting the page designs classify whether pages are relevant and if they should be followed further or not.

¹<http://en.wikipedia.org/wiki/PHP-Crawler>

²<http://en.wikipedia.org/wiki/Nutch>

2.3 Implementation

First the crawler parses the seeded course homepages for all hyperlinks (excluding Javascript and mailto-links) which are converted into absolute canonicalised URLs, taking in account any potential `<base>`-tags. A flow diagram of this step is seen in Figure 2.1.

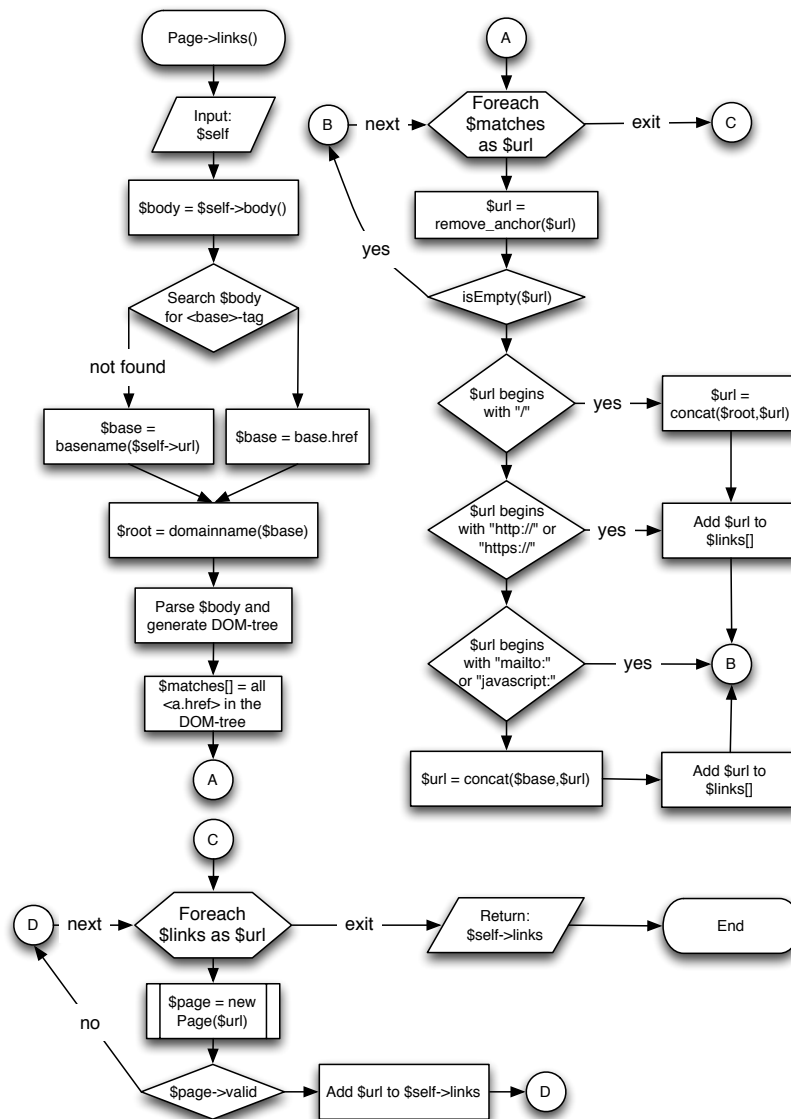


Figure 2.1: Flow diagram of initial hyperlink parsing

The links are then counted; if a link occurs in more than two courses, it is deemed as shared (i.e. not specific to any course) and all of its occurrences are thereby removed and put into a stoplist such that it never gets crawled. This technique effectively filters common pages such as the university homepage and department pages linked by all courses given by the department. Hyperlinks are further filtered through the use of custom stop functions which remove links containing certain substrings such as GET-parameters for individual laboratory enrollment pages.

With the new set of hyperlinks, the crawler then downloads the linked documents and decides whether their links will be followed further or only indexed. Documents that are not HTML or are located on a different domain than the course homepage are only indexed and never followed. We also apply a similarity check where we compare the hyperlinks of a document with the links on the respective course homepage. If a document is deemed too dissimilar, it is regarded as outside the scope of the web pages of the course and is therefore not followed. Without this last filter the crawler might have found a large amount of irrelevant documents such as other department pages not relevant to the course. Since we are mostly interested in human readable course documents (such as slides, lab preparatory papers and exercises), the crawler also removes documents based on their file type without indexing. This includes files like images, source code and binary data.

This whole second step is then done depth first recursively until there are no more valid links to follow. After each recursion we update the hyperlink count and filter the new links using the stoplists described in the previous step. A flow diagram of this step is seen in Figure 2.2.

When the crawler has downloaded all documents for all courses it removes duplicate HTML documents and reinserts originally seeded homepages if they have been removed by any of the previous filters. Documents are compared by hashing the content of the body (using MD5³ for speed and simplicity) with all HTML-tags removed.

Finally we generate one additional document for every course using data from the LoT database.

Since the crawled material will be inserted into Solr, the result from the crawler is post-processed by:

- Deep data inspection that removes irrelevant file types such as source code and images that could not be removed earlier due to their file extension
- Removing documents from certain domains deemed irrelevant
- Converting .doc, .docx, .ppt, .pptx and .ps to .pdf

The file type conversion of Microsoft Office-type files is done using the command-line tool version of LibreOffice⁴ (a freeware software suite with programs similar to Microsoft Office). Since we failed to run LibreOffice headless (i.e. without a

³<http://www.ietf.org/rfc/rfc1321.txt>

⁴<http://www.libreoffice.org/>

display), we had to let LibreOffice use an emulated display through Xvfb⁵. The .ps-files are converted using ps2pdf⁶ on the command-line.

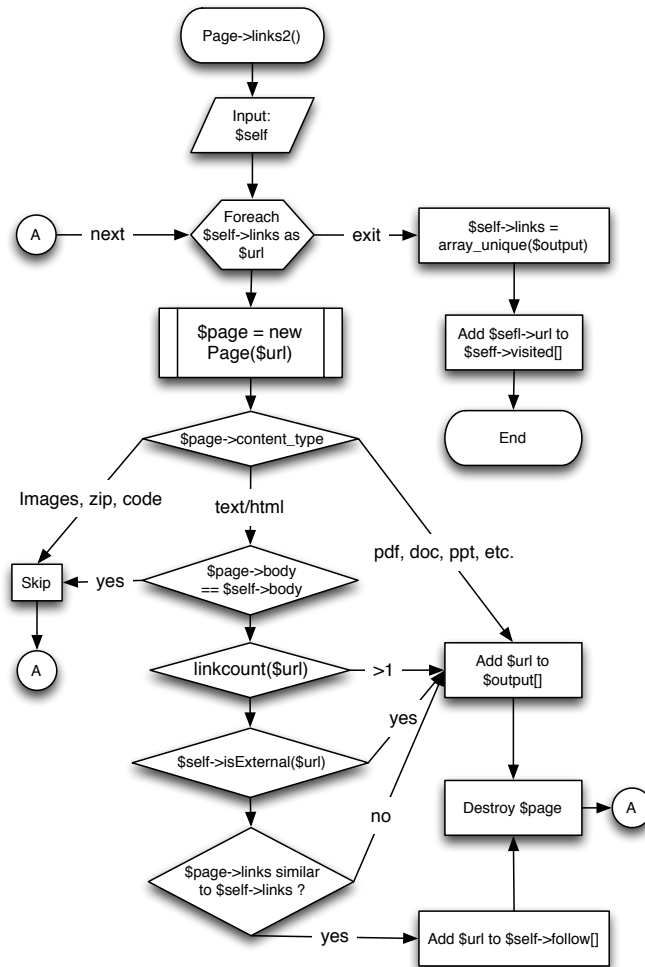


Figure 2.2: Flow diagram of the inner function of the recursive crawling

⁵<http://en.wikipedia.org/wiki/Xvfb>

⁶<http://www.ps2pdf.com/>

2.4 Results

The crawler handled 191 courses and downloaded (cached) 12178 documents. The number of residual documents after filtering and removal was 6416 (which is 53% of the considered documents) and of the residual documents, 89 needed to be converted into PDF (about 1.4%).

The post process removed:

- 228 documents because of duplication
- 233 URLs because of unusable filetypes
- 411 URLs because of blocked domains

The execution time for the crawler from a clean state is about 2-3 hours on our server. Most of the time is spent requesting and downloading web pages.

Since the crawler only crawls a subset of all courses given at LTH, some departments are not crawled for more than a few courses, resulting in that their page design will not be exploited. In order to make the crawler benefit from these as well, some extra courses were crawled, but not indexed later on. This gave the crawler more hyperlink counts that could be used to discard more pages.

Some departments (such as the Math department) does not use a CMS for their course web pages and in some cases they only use a PDF-document for course homepage. In those cases it is hard for our crawler to find sub pages to follow if they do not share enough links. Hopefully these course homepages have all important links on the homepage itself and not too many unique links outside the course pages.

Number of documents	Number of courses	Percentage of the courses
1	29	15,18%
≤ 10	94	49,21%
≤ 50	158	82,72%
≥ 100	10	5,24%
≥ 200	4	2,09%

Table 2.1: Breakdown of number of documents per course

The final distribution of number of documents per course is shown in Table 2.1. and is a quite fluctuating number in reality. This is partly due to our crawler but especially to the course homepages themselves. Some courses have very few documents on the web (like a single PDF), while others have an extreme amount of external linkage or contain links to old course pages with shared links.

The crawler can be tuned to better define the boundaries of the course material for a course, meaning which documents are considered to be part of the course material as well as which pages shouldn't be crawled at all. This is done by manually tuning the stoplists and stop functions. Both the stoplists and stop functions can potentially be iteratively enhanced by running the algorithm multiple times and evaluating the result. While we did apply some additional tuning by this method, it could still be explored considerably in order to improve the results.

The data created in the crawling stage is later qualitatively evaluated during the similarity study and also when used in the search engine.

3.1 Introduction

In order to easily find a course in the vast collection of course material, a search engine is required. A regular search engine is generally divided into two distinct parts: indexing and searching. Indexing is plainly the task of creating an index that may later be easily searched through. An index is conceptually not very different from the index in a regular handbook in the sense that the engine creates an easily searchable key-value list of search terms and where their corresponding document is located. The engine may then use this index to find documents without having to expensively iterate through the whole collection.

The goal was to integrate a search engine that would let the user find courses by searching for any words or sentences that may have a connection to any of the course data available, may it be course code, names, programming languages, topics or anything else that has been indexed.

3.2 Choice of implementation

Our choice of search engine was constrained by a few different criteria. Firstly, the indexer is required to successfully handle multiple different document formats, ranging from pure ASCII text and HTML pages to complex (and sometimes broken) PDF and Powerpoint documents. Secondly, while the prototype only has to handle a few simultaneous users at a time, the server hardware would at least initially be relatively old and limited with low amount of both RAM and processing power. The prototype still needs to be fast and responsive even with this limited hardware.

Further the engine would need to be easily integratable with the rest of the system, preferably providing some kind of API to query the engine directly with PHP.

Finally, the engine would need to be able to provide the corresponding course code for multiple documents. From our experience, search engines are generally specialised at finding individual documents based on a search query and the contents of that particular document. However, in our case the documents by themselves serve little purpose; instead we are only interested in finding a course based on the contents of the many indexed documents associated with it.

While it would be technically possible to implement a custom search engine from scratch using PHP and MySQL, because of time constraints and the impractically vast scope of such project, we rather wanted to configure (and extend where needed) an existing publicly available search engine to fit our needs.

We had already in an early state been looking at Apache Lucene¹ for our choice of search engine. Lucene is a mature open source project maintained by the Apache Software Foundation and is used by many large companies and organisations around the world, including Twitter[19], LinkedIn and IBM[20]. After further study we did however turn to another Apache project called Solr², which is an enterprise search engine solution based on Lucene with several additional features, including being preconfigured out of the box to be queried through HTTP and output both serialised PHP objects and JSON. Overall Solr seemed to fit our needs on all points, why we chose it for our implementation.

3.3 Overview of Solr

The basic Solr configuration consists of several configuration files, modules and plugins. Most of them may be left as is, while a few require extensive configuration to properly fit our needs.

There are three primary configuration files that needed to be changed: the core schema, the core configuration file and finally the primary configuration file.

The basic structure of Solr is seen in Figure 3.1. The primary configuration file handles global settings for Solr, where its main purpose is listing all available cores and their locations. A Solr Core is conceptually an isolated instance of a Solr configuration, complete with its own index and configuration files. With the use of cores, a single Solr installation may host multiple search services that may be accessed by simply pointing the request URL to the appropriate core.[21]

For each Solr Core there is then a dedicated configuration file as well as a schema. The schema contains all fields, field types as well as how these fields will be populated when documents get indexed. The schema also handles how each field will be handled when the index is queried.

The configuration file for a Solr Core handles most of the settings for Solr itself. This includes file paths for the different modules as well as configuration of the request handlers. A request handler (as its name implies) handles all incoming HTTP requests to Solr. The handler register itself for a specific URI (e.g. `/select` or `/autocomplete`) which results in all requests to

```
/coreName/handleName?parameter1=value&parameter2=value
```

will be managed by the respective request handler. The handler itself is composed of one or more so called *components* which to the request parameters (including any default or overriding parameters that the handler itself may specify) are

¹<http://lucene.apache.org/core/>

²<http://lucene.apache.org/solr/>

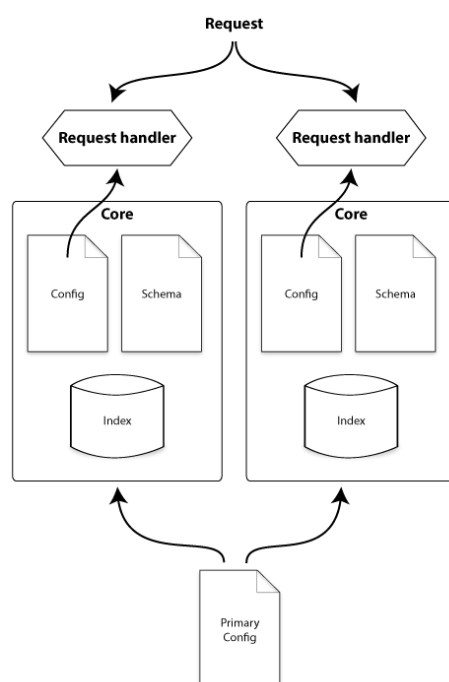


Figure 3.1: Basic structure of Solr cores

then forwarded to. The component then finally performs the actions it's specified for (e.g. spell checking or highlighting).

The result is then returned back to the user in one of a few different formats, depending on how the request handler is configured.

3.4 Configuration

Basic installation of Solr turned out to be a rather straightforward process where the downloadable package is preconfigured to work with rudimentarily functionality out of the box. Since Solr is queried through HTTP requests it is dependent on a web server, whereas the default configuration is bundled with a simple HTTP server called Jetty.

Naturally, further configuration would still be needed in order to fully cover our needs. We had to configure the schema to hold all the required fields and upon analysis of the crawled data, we concluded that the schema would only need to support a document title, the document text and the course code. At a later stage we also added a field for the course name.

First of all, we would need a way to properly index the many different document types that had been crawled. By default, Solr only handles properly structured XML-files and can't index HTML or PDFs without aid. Thankfully the `ExtractingRequestHandler` module, based on Apache Tika, was made exactly for

this purpose. The module automatically identifies the document type and extracts the relevant data. However, while it was able to properly index even rather broken HTML, ASCII and PDF files, it would systematically crash when trying to index some Microsoft Word files and e-fax files. Our solution was to build a PHP based middle layer between the crawler and the indexer where all files that are not HTML, ASCII or PDF will be converted to PDF when possible before they are handed to the indexer.

When configuring the fields, we soon realised that the number of distinct human languages used in different courses would turn out to be a problem in need of a solution. Every language needs slightly different preprocessing to function optimally; for example stemming for one language will not work at all for another language and the same with stopwords. Again, the solution was the `ExtractingRequestHandler` module which additionally has support for identifying languages and labeling the fields accordingly. We then created two versions for both the document title and text fields; one for Swedish and one for English which were by far the two most common languages. The reason for not supporting additional languages was mostly because of time constraints as every language would later need to be handled separately when integrating the search module into the prototype.

3.5 Fields

Solr is based around a defined schema, which contains what kinds of data for a document is stored in the index. These are called fields and use definitions from field types, which configure what kinds of preprocessing is applied to the field. Many fields can share a single field type.

3.5.1 Title field and text field

Both the text field and the title field share the field types `text_en` and `text_sv` for respective language. Several layers of preprocessing is done by Solr on both the fields themselves and by the query when searching in the fields.

The two field types are of type `text` and uses a standard tokeniser to create the initial bag of words. It's a simple tokeniser that splits the text on whitespace and removes punctuation as well as some special characters. Later, all tokens that are either less than 3 characters or greater than 35 characters in length are removed. The rationale behind this is foremost to remove documents containing irrelevant byte data that slipped through (e.g. text documents with images) and that it's very unlikely that documents contain actual human readable words of more than 35 characters in length. The remaining tokens are then lowercased in order to achieve case insensitivity.

Additionally, we apply a localised list of stopwords as well as a stemmer to further increase search accuracy. Stopwords are merely a list of common and semantically unimportant words that will be removed from the bag of words. For English this list would include words like *I*, *the* and *an*. [24, p.226]

Stemming is plainly the task of reducing tokens to its stem in order to remove semantically unnecessary variants of the same token. The tokens *walking*, *walked* and *walks* can for example be reduced into the single token *walk*. [24, p.226]

The english variant of the field type also applies a possessive filter to remove trailing -s from the tokens.

Lastly, we apply two N-gram filters; one from left to right and one right to left. N-grams are basically subtokens of every token. Instead of merely saving the token *walk* a left-to-right N-gram filter would also create the tokens *w*, *wa* and *wal*. With both filters applied the complete list of tokens becomes *w*, *wa*, *wal*, *walk*, *k*, *lk* and *alk*. N-grams are an effective way of enabling partial matches where users may search for incomplete words and still find what they are looking for. Without N-grams the query *car* would not match the word *racecar*, while with N-grams it will.

N-grams cannot however be applied to the query, since this will result in an excessive amount of false positive matches. For example the query *walking* will now incorrectly match both *ingredient* and *walrus*.

The original text is also stored in Solr in order to be used for result highlighting later.

3.5.2 Course code

The course code field contains the course code and therefore needs a rather small amount of preprocessing. The field is of type `text` and is tokenised with a simple whitespace tokeniser that only splits on whitespace. The field is then lowercased without any further filtering.

3.5.3 Course name

The contents of this field is self-explanatory. Again the field is of type `text` and uses a standard tokeniser which works well for this purpose. Since the course name consists of rather small amounts of text, we didn't manage get satisfying results from the automatic language identifier, why we instead chose to use a single field for both Swedish and English. Because of this, we don't use a stoplist nor stemmer and only applies a lowercase filter and N-grams. However, we also keep a second copy of the course name, but without N-grams such that it won't interfere with the autocomplete function explained below.

3.6 Autocompletion

In order to provide a more streamlined user experience we wanted the system to support autocompletion on course codes and course names. To achieve this we used a feature called *faceting*. Faceting is a technique for generating counts for certain properties. In our case we wanted to simply get a list of the most common words that begins with the typed letters. Configuration was rather straightforward where we merely created a specific auto-complete request handler and set it to return maximum five facets with a user specified prefix, ordered by number of occurrences. Due to the mixed quality of the contents of the text field, we configured the autocomplete handler to solely search within the course code and course name fields. However, we had to specifically create a dedicated variant for the course name field that didn't use N-grams. This is because we only wanted the handler to

suggest complete words and not partial ones that would otherwise be found with N-grams activated.

3.7 Search handler

Our basic search handler is set to search all of the indexed fields, namely: title, text, course code and course name.

To make sure that the appropriate course is always returned when a user searches for a valid course code we wanted the course code field to be ranked much higher than other fields. Solr implements the concept of *boosting*, where a field (or term) may be configured to be ranked relatively higher or lower than default. Using this system, we boosted the course code field greatly, giving it a boost value of 10. We also boosted the title field slightly with a boost value of 2. After some basic internal testing we found ourself often wanting to merely search for a specific course name and not necessarily the contents of a course. Because of that, we finally also boosted the course name field with a value of 10.

We also have three additional components activated for the search handler, namely highlighting, grouping and spell checking. The former is set to highlight words contained in the text field that somehow match the search query. It then returns a snippet of formatted HTML containing a total of 200 characters of text surrounding the matching words. With standard highlighting being remarkably slow, we use the fast vector highlighter which is noticeably faster, but require the fields being vectorised in advance (and thus increasing index file size).

The spell checker uses the contents of the text field to suggest spell corrections. It does however not take language in account when checking and instead suggests any word that may be close to the misspelled word.

Lastly, we use grouping. A big problem with the standard search handler is that it is specialised at finding *documents* that are relevant to the provided query, which as explained earlier, is not what we want. Instead we want it to return *courses* that have relevant documents connected to them. There are a few more or less creative ways of achieving this with Solr, but our way of handling it is with the *grouping component*. This component is configured with a common field which is to be grouped on. All documents that share the same value on this field are concatenated in the result as one single result item. In our case we group on the course code field and returns the five highest scoring documents for each course.

Recommendation Engines

To make it easier for students to select and explore viable courses, we have implemented two recommendation engines.

The first recommendation engine uses similarities in course material to recommend courses, and the second uses statistical data to recommend courses that students have studied at the same time before. In some sense, the search engine can also be seen as a recommendation engine, but one that requires a keyword or a phrase to do its magic.

4.1 Similarity recommendations

4.1.1 Introduction

When exploring possible courses to study, it might be interesting to know if there are more courses like a given course that was fun or interesting. We want our prototype to be able to list similar courses, but the choice of algorithm is not obvious by any means. Therefore we have done an evaluation of some different algorithms on our dataset. By using the contents of the found course material for each course, we have created several different similarity experts, which consist of different algorithms that rank courses similar to a given course.

In addition to the similarity experts, we are also investigating if combining their results through different voting algorithms and merging creates better similarities.

Most algorithms are using term vectors (i.e. vectors representing term occurrences in a document) for their similarity measurement. The term vectors for all courses have been created by extracting the terms in each document in Solr and then summing up the term frequencies for each course.

4.1.2 Algorithm families

We are evaluating similarity experts of three different types:

- Set-based
- Term frequency-based (TF-based)
- Concept-based

All of the results created by the experts are precalculated and their results are stored in a database for quick retrieval without recalculation.

Set-based

Set based algorithms only use the vocabulary overlap of courses and by using set theory creates a similarity index.

The two set-based similarity algorithms to be evaluated are: Jaccard similarity coefficient[9] and Dice's coefficient[8]. They are quite similar in their definitions (where A and B are sets to be compared):

$$JaccardSim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

$$DiceSim(A, B) = \frac{2|A \cap B|}{|A| + |B|}$$

Term frequency-based

The TF-based algorithms uses the term frequency (i.e. tf) in each course as well as the term document frequency (i.e. df). The term document frequency is often used as a denominator and is therefore usually called inverse term document frequency (i.e. idf).

The tf and idf are used to create weights for every term in the documents and the weights are then used to represent the courses as an array of terms (Vector Space Model, VSM) with weights.

The similarity between two vectors can be measured using cosine similarity that calculates the angle between the vectors. The similarity between two courses are calculated by cosine similarity using the respective term vectors.[9]

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

The measure is a real number between 0 and 1, where 1 means that the vectors are equal and 0 that they have nothing in common.

The different TF-based algorithms differs in how their weights are calculated, whereas some use logarithm functions to normalize numbers. There are four somewhat different weighting schemes that we have evaluated:

normalized tf_idf:

$$weight(t) = \frac{tf \cdot idf}{T}$$

where T is the number of different terms in the course

glasgow (log-normalized tf_idf):

$$weight(t) = \frac{\log(1 + tf) \cdot \log(N \cdot idf)}{\log T}$$

where N is the number of courses and T is the number of different terms in the course

aug:

$$weight(t) = 0.5 + \frac{0.5 \cdot tf}{max(tf_t)} \cdot max\{0, \log \frac{N - df}{df}\} \cdot \frac{1}{T}$$

where T is the number of different terms in the course

aug_log:

$$weight(t) = 0.5 + \frac{0.5 \cdot df}{max(tf_t)} \cdot max\{0, \log \frac{N - df}{df}\} \cdot \log \frac{1}{T}$$

where T is the number of different terms in the course.

aug_log was meant to be the same expression as aug, but with the only difference that the normalization was the logarithm of the original normalization. Due to a small programming error (sometimes referred to as a bug), the term frequency was substituted for the term document frequency in the first term. This was not found out until much later, when the code was reviewed for the master thesis paper.

Concept-based

Information retrieval has been enhanced over the years by the use of merging terms into concepts. The idea is that terms that adhere to a concept are used in a similar way. Two well known ways of creating concepts from terms are LSI (Latent Semantic Indexing)[10] and PLSI (Probabilistic Latent Semantic Indexing)[11]. These will effectively lower the number of different words and combine them into concepts, making it possible to match documents by their semantic meaning and not just their vocabulary use.

LSI uses Singular Value Decomposition (SVD) to lower the high dimensionality of terms into much fewer concepts[10].

PLSI is an improved form of LSI and uses a well built statistical ground to create concepts containing terms. PLSI can be calculated by iterating the expectation-maximization (EM) algorithm to find the maximum likelihood of a term being in a concept. This iterative process takes much time when the amount of concepts increases.[11]

The results of LSI and PLSI gives the courses new vectors in lower dimensional concept space, and these vectors can also be checked for similarity by the use of the cosine similarity measure.

LSI

We calculated LSI using a standalone command-line executable called SVDLIBC¹, that runs SVD on a dataset formatted in a special way (sparse text). SVDLIBC has support for a number of parameters, but we only used the `-o` (output directory) and `-d` (number of dimensions i.e. concepts) parameters. We created three different sized LSI datasets with 50, 100 and 150 concepts each.

PLSI

¹<http://tedlab.mit.edu/~dr/SVDLIBC/>

Since there isn't a freely available PHP implementation of PLSI, we implemented our own, using the algorithm descriptions given in [12]. The algorithm ends when it converges and we chose to check if the mean absolute difference between the last and current iteration in a matrix element was below 0.01 as end condition.

The PLSI script took quite a while to run for a very small number of concepts (i.e. 2) on our server and therefore we used the easiest means of parallelism that PHP can handle: C-style process forking². The most time consuming part of the PLSI algorithm can easily be split into two independent parts that calculates the different probability matrices. The fork did however not double our performance since the different matrices are of different size and takes an uneven amount of time to generate, but we still got a significant performance boost.

In retrospect, we should have implemented PLSI in C or some other language with high computational performance, if we needed it to be faster. With that in mind, speed was not really an issue for us since we precalculate all of our similarity results.

In the end we used a Macbook Pro (with an Intel i7 processor) to run the PHP script and produce a result with 20 concepts. Compared to LSI, PLSI is extremely slow to create the same number of concepts.

4.1.3 More Like This

We have also evaluated a Solr module called MLT (More Like This) which uses the term vector of a given document, calculates term weights and searches Solr using the terms and weights.

The actual MLT algorithm is not well documented, but a kind soul has analyzed the source code and created a write-up which makes it quite clear in [13].

MLT algorithm overview:

- Creating term vectors for each field (separately, only counting occurrences in the same field)
- Removing stopwords
- Filters out term frequencies less than 2
- idf is calculated as: $\log(\frac{N}{df+1}) + 1$
- The term scores are then calculated as: $tf \cdot idf = tf \cdot (\log(\frac{N}{df+1}) + 1)$
- After sorting the different term scores, picks the 25 terms with highest frequency for querying
- Boosting the terms (downwards) using: $boosted = score / max_score$, where max_score is the score of the term with highest score
- Search using the created query vector with boosting values³⁴

²<http://php.net/manual/en/function.pcntl-fork.php>

³Note that the search is carried out using a vast complement of different pre- and post-processing

⁴Also note that the terms are only searched for in the fields where they occurred the most and not in all fields where they exist

Configuration

Configuring the Solr MLT module for basic usage is done by creating a new request handler in the Solr core configuration file. The MLT request handler is then implemented very similar to a regular search request handler, sharing many of the common search parameters. The result set for MLT is divided into two separate parts.

First, MLT does a regular search to find the document to base the recommendation on. In our case we only wanted to precisely fetch a course with a specific course code, why we only search in the course field.

The second part of the MLT result set contains the actual recommended documents based on the previously found query document. The latter part needs very little configuration and merely requires a list of fields to return (in our case title and course code), the minimum TF and DF score (1 and 2 in our case) and whether boosting should be applied or not (which it will in our case).

To our dismay the MLT handler does not support the grouping module that we use for concatenating documents on the respective course. Because of this, simply running the MLT handler would only return the most similar documents to the highest ranking document for a specific course. Most of the time the MLT handler would thereby pointlessly return a list of documents belonging to the same course that the recommendation was initially based on.

The solution to this problem was to take advantage of a Solr feature called multivalued fields. A multivalued field can effectively contain multiple independent rows of data, making it possible to let a single indexed document contain the data of all documents belonging to a single course. However, the current version of the `ExtractingRequestHandler` does not as far as we could see support iteratively insert data into fields as the documents are indexed. With that said, it is still possible to subsequently export the original data from one index to another without having to rely on the `ExtractingRequestHandler`. In the end, we settled with creating an additional Solr core specifically for the MLT module. The data was then exported from the original Solr core and successfully concatenated so that every course only consists of a single document each. The original core is however still used for regular searching.

4.1.4 Combining expert results

The similarity experts work in different ways and they seem to show (different) valid results among their top ranked results. Therefore it would be viable to combine their efforts into creating result sets that merges the rankings from the experts in a way that keeps the most relevant results in the top rankings.

We have decided on four merging algorithms that should be evaluated together with the experts. Two of the algorithms are voting algorithms, meaning that they let multiple similarity experts hand out weighted scores that are then combined in different ways. Another algorithm is the Run-off algorithm[15], that iterates breadth-first through the similarity experts rankings and picks courses that have been found enough times. The last algorithm is a simple round robin merger[15] which just merges the results breadth-first and skips items that have already been collected.

The two voting algorithms are Borda count[14] and Inverted nauru[15]. They are both based on letting the experts vote and give their top ranked courses the most points. If the voting algorithms only use one expert, the sorted result is the same as for the expert. The points are summarized and then the result is returned as a course list, where the courses are sorted by their respective point totals.

Borda count

Borda count is similar to the Eurovision Song Contest scoring method and it works by letting the experts give $R+1-position$ points to course at place *position* where R is the total number of rankings (i.e. 10 in our case).[14]

Inverted Nauru

Inverted Nauru only differs from Borda count by the points given. The amount of points is given by: $\frac{1}{position}$. For example, the top ranked course by an expert gains 1 point, the second $\frac{1}{2}$, the third $\frac{1}{3}$ and so on. The idea is that the point difference between position 1 and 2 is greater than between 9 and 10.[15]

Run-off

The run-off algorithm works by selecting courses in a sorted fashion by letting the course found in enough (by a threshold) experts first be the top ranked, and so on. The default threshold is: $\text{floor}(\frac{\text{numberOfExperts}}{2})$. [15]

We have enhanced the standard algorithm so that if not enough courses are found to fill the list, the threshold is decreased by one until the list is filled. The benefit of using this algorithm is that it totally ignores experts that are in minority to have top ranked a course until a majority have found it.

Round Robin

Round Robin is the simplest of the result merging algorithms and contrary to the others, it does not need a memory to keep track of points, it simply scans the experts and adds courses to the list if they are not already present in the list. The immediate disadvantage with this algorithm is that it does not care about how many experts have found a course, just that someone have. This makes round robin perform poorly when inadequate experts are used as input.[15]

4.1.5 Evaluation

TREC-style average precision (TSAP) can be used to score search engines. It mends the common problem of not knowing enough information to perform a standard precision/recall evaluation by skipping the recall part and using weighted precision taking into account result positioning.[16]

We have used a slightly modified version of TSAP with cutoff N (denoted TSAP@N), where N is 5 or 10. The cutoff is used to only analyse the top N

results. For an expert E and cutoff N , the TSAP@N is calculated by

$$TSAP(E, N) = \frac{1}{N} \sum_{i=1}^N \frac{E_i}{i}$$

where $E_i = 1$ if the element in position i in expert E is deemed relevant in the comparison, otherwise 0.

Data gathering method

The evaluation data has been created by using a web tool that lets users select a course to evaluate and then mark its suggested courses to be similar or not to the selected course. The set of courses to evaluate against is the merged expert results and to get rid of some biasing, the merged pool of courses have been shuffled before being displayed.

The evaluation requires that the test subject have a working knowledge of all the courses in the set, as well as the course that is being evaluated.



Figure 4.1: Snapshots from the TSAP evaluation tool: Index page (left) and voting page (right)

We have carried out three rounds of TSAP:

- Round 1 was executed by us (the two authors), testing 22 courses.
- Round 2 was executed by us also, but with less experts feeding the merging algorithms.
- Round 3 was executed by 16 other students (and former students), testing 25 courses.

The results from Round 1 was used to modify the evaluation by tuning the voting algorithms for Round 2. Round 3 was carried out by more people to hopefully get a more balanced and less biased result.

4.1.6 TSAP round 1

22 evaluations done by the two authors using courses in different departments for tests. The results are shown in Table 4.1 where bold rows represent combined results and a higher score is better. In this round, the combined results are based on all of the similarity-ranking-algorithms.

Algorithm	Cutoff	Score
mlt	5	0.280
runOff	5	0.258
bordaCount	5	0.257
invertedNauru	5	0.256
glasgow	5	0.254
aug_log	5	0.252
aug	5	0.249
tf_idf	5	0.236
roundRobin	5	0.223
jaccard	5	0.215
dice	5	0.215
lsi100	5	0.181
lsi50	5	0.163
mlt	10	0.160
bordaCount	10	0.146
runOff	10	0.145
invertedNauru	10	0.145
aug_log	10	0.143
glasgow	10	0.142
aug	10	0.140
tf_idf	10	0.137
roundRobin	10	0.130
jaccard	10	0.123
dice	10	0.123
lsi150	5	0.111
lsi100	10	0.102
lsi50	10	0.095
plsi	5	0.065
lsi150	10	0.061
plsi	10	0.037

Table 4.1: The results from TSAP round 1

Comments

- MLT (Solr More Like This) wins both TSAP@10 and TSAP@5 and it is even superior to the voting algorithms using all similarity experts.
- 3 out of 4 voting algorithms are placed top 5 for TSAP@10, the only voting algorithm missing is Round Robin, which is placed together with the set-based algorithms. RoundRobin is considerably worse than the other voting algorithms
- LSI and PLSI are both placed at the bottom end of the list. Both PLSI and LSI are supposed to be superior for TF-based document querying using cosine similarity, but in this context they both perform badly, probably because they are not tuned in such a way that the concepts can be used to distinguish similarities in the given material.
- Dice and Jaccard are both set based similarity measures and does not take into account the term frequencies, only their presence. Their mathematical definitions are quite close, and in our test they perform exactly the same.
- The simplest term frequency based similarity ranker, `tf_idf`, performs worst in both TSAP@10 and TSAP@5 and is superseded by the logged versions.
- Glasgow and `aug_log` performs about the same and has swapped positions in the different cutoffs. Even though `aug_log` is faulty implemented it still performs fairly well.
- Run-off and borda count also performs about the same and has also swapped places in the different cutoffs. Both their scores are very similar to inverted nauru, and their results could be considered being of the same quality, although the way they create them differs much in algorithmical terms.

4.1.7 TSAP round 2

Using the results from the first round, input of the voting algorithms has been slightly changed. Dice has been removed (having the same results as Jaccard), LSI and PLSI has been removed (showing that they do not contribute). The hypothesis is that this change should improve the score of the voting algorithms and especially the Round Robin.

Comments

The results from TSAP round 2 is shown in Table 4.2. and as expected, the Round Robin algorithm benefited greatly from the changes, in fact more so than initially anticipated, ending up scoring very close to the previously dominating MLT. However, the test is still limited by the small homogenous test group and relatively few tests why small errors may impact greatly on the scores. The `tf_idf` algorithm is noticeably higher on the list even though no changes have been made to the algorithm. One explanation could be that because of the generally unreliable nature of the algorithm where it sometimes provides very good results and other times utterly flawed, the score may fluctuate greatly if the data set is too small.

A proposed solution can be to increase the size of the test group.

Algorithm	Cutoff	Score
mlt	5	0.356
roundRobin	5	0.354
tf_idf	5	0.353
bordaCount	5	0.334
glasgow	5	0.328
runOff	5	0.325
invertedNauru	5	0.324
aug_log	5	0.309
aug	5	0.305
jaccard	5	0.275
mlt	10	0.206
roundRobin	10	0.202
tf_idf	10	0.199
bordaCount	10	0.194
invertedNauru	10	0.188
runOff	10	0.187
glasgow	10	0.187
aug_log	10	0.175
aug	10	0.174
jaccard	10	0.160

Table 4.2: The results from TSAP round 2

Algorithm	Cutoff	Score	Diff. to TSAP 1	Diff. to TSAP 2
mlt	5	0.306	0	0
roundRobin	5	0.290	8	0
glasgow	5	0.284	2	2
runOff	5	0.283	-2	2
bordaCount	5	0.283	-2	-1
tf_idf	5	0.280	2	-3
invertedNauru	5	0.279	-3	0
aug	5	0.275	-1	1
aug_log	5	0.256	-3	-1
jaccard	5	0.209	0	0
mlt	10	0.178	0	0
roundRobin	10	0.169	8	0
tf_idf	10	0.165	5	0
runOff	10	0.165	-1	2
bordaCount	10	0.164	-3	-1
invertedNauru	10	0.163	-2	-1
glasgow	10	0.161	-1	0
aug	10	0.157	-1	1
aug_log	10	0.149	-4	-1
jaccard	10	0.123	0	0

Table 4.3: The results from TSAP round 3

4.1.8 TSAP round 3

One potential flaw with the results from the previous tests is the somewhat limited test group solely consisting of the two authors. The third round of the test was carried out through an online public form distributed to a larger group of students of roughly a bit more than a hundred people. While we didn't expect a response rate of more than 10-20 people, the greatly increased test group should still provide more reliable data.

Comments

The results of TSAP round 3 is shown in Table 4.3 and as expected, the tf_idf algorithm has gone down significantly and now has a much more feasible score. MLT still outperforms all other algorithms, even more so than the voting algorithms, both at a cutoff of 5 as well as 10. While it was anticipated that the round robin algorithm would improve because of the changes from the second round, the degree of it is surprising. Also interesting is the relatively low score of the two aug algorithms, placing second to last at both cutoffs. Judging from the score, one can also draw the conclusion that the glasgow algorithm is relatively effective at producing reliable list of five courses, but becomes less effective when increasing

the list, dropping four places between the two cutoffs.

In the end, it's obvious that MLT is the most favourable algorithm to use for this task, even outperforming the different voting algorithms.

4.2 Statistical recommendations

4.2.1 Introduction

To provide additional options for course recommendation, we wanted to be able to suggest courses based on previous students choice of courses. To do this we use data from the national student documentation service called Ladok which accurately log all course selections as well as grades for every examination element that a student has participated in. Because of privacy concerns we only gained access to the last five years of data with the personal identity numbers thoroughly replaced with anonymous ID numbers.

4.2.2 Implementation

Some preprocessing was required before it was possible to query the recommender for suggestions. For simplicity, we do this in two separate parts. In step one, we group all courses given in a certain period pairwise and averages the result for all students for respective course such that we get the combined average grade for both of those two courses at every specific period. However, since the grades at LTH can either be in the form failed/3/4/5 (TH) or merely the binary failed/passed (UG), we assume that all passable grades are equal to the score of 1 and failed grades the score of 0. If we instead would try to adjust the average value depending on the TH grades, the end result would then be unfairly biased towards either TH or UG courses. While only counting passed and failed grades results in lower precision, it at least makes the result more unbiased.

The data structure at the end of the first part can be seen in Table 4.4.

Year	Term	Course 1	Course 2	Average grade
------	------	----------	----------	---------------

Table 4.4: Data structure (database row) containing the calculated average grade for two courses given at a certain time.

With the new data prepared we proceed to the second part of the preprocessing. For the data to be useful we have to cover three additional factors. Firstly we only want a score which is based on all occurrences of two specific courses and not only on one particular period. We also want to account for how the grades for the two concerned courses differ from the grades overall. Finally we want to count the number of students who study the two courses together compared to only one of the courses together with any other course. The resulting equation for calculation of the new score is as follows:

- $G_i = \{\text{failed/passed}, \dots\}$ is a set containing the individual grades from course i

- $\overline{G_i}$ is the average grade across course i
- $\overline{G_{ij}} = \frac{\overline{G_i} + \overline{G_j}}{2}$ is the average grade across course i and course j (which are given in the same period)
- $|G_i|$ is the number of grades (i.e. students) in course i
- $|G_{ij}| = |G_i \cup G_j|$ is the number of students taking BOTH course i and course j at the same time

the score is then given by:

$$score(i, j) = \frac{\overline{G_{ij}} |G_{ij}|}{\overline{G_i} |G_i|}$$

This means that if many students choose to study two specific courses at the same time instead of any other courses, then those two courses will be ranked higher by the recommendation engine. If however only a very few of the students actually pass relative to students studying other courses, then they will be ranked lower.

4.2.3 Result

After preprocessing we ended up with 1,238 rows of data (down from 7,039) each containing two courses and a score between 0 and 1. The usefulness of the generated data is evaluated in the prototype user evaluation.

Course Dependencies and Validation

Courses at LTH have different requirements that need to be fulfilled in order to be allowed to study the course. The most common sort of the requirements are of the form that the student need to have passed some courses prior, but can also be that the student needs to have studied a certain amount of hp. Requirements can also be both hard and soft, meaning that in a case of a soft requirement, the student only need to have passed some compulsory parts of the course, such as laborations and not have passed the course exam. The requirements can therefore be thought of as dependencies between courses together with some conditional checks.

The course requirements are originally stored in free text, but they are generally written in a uniform way, probably from old texts being copied to create the new requirements, which make them somewhat parseable. The majority of the requirements are written in CNF (conjunctive normal form). To be able to study the course you need to have completed at least one course in each group of requirements. The boolean expression that is derived from this assumption has the form:

$$Req = (A \vee B) \wedge C$$

This means that you can study the course if you have completed at least A and C, or B and C. An easy method to get all the different course states that fulfils the expression is to create a graph and then programmatically create all paths through it.

We wanted our system to be able to validate the study plans filled in by users and answer the question if a course can be studied when the student wants to study it. Quite early in this project, we wanted to work with the graph database Neo4j¹, and the most promising area to apply a graph upon was the courses themselves. Neo4j provides a schema-less database with nodes and relationships where both can have individual properties containing data. While nodes may be fetched individually, the strength of Neo4j is that it can quickly perform pattern matching of nodes and relationships and thereby traverse large graphs with ease. Neo4j is queried by the custom querying language Cypher².

¹<http://www.neo4j.org/>

²<http://docs.neo4j.org/chunked/milestone/cypher-introduction.html>

5.1 Parsing algorithm

The first and perhaps most important part of formalizing the course prescience dependencies is the parsing of the free-text representation.

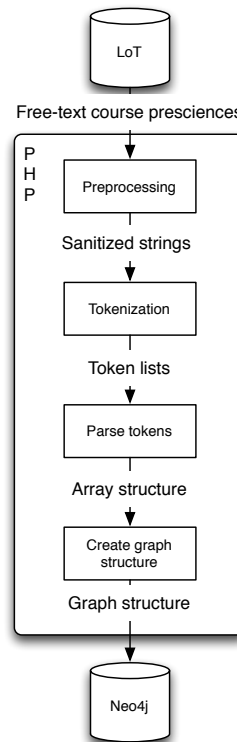


Figure 5.1: Flow diagram of the dependency parsing algorithm

The free text requirements are first preprocessed by removing course names so that they will not interfere with the tokenization process. The residual text is then tokenized by a regex containing different words and lexical constructs that can be used to create the boolean expressions of the requirements as well as describe the different special conditions that also needs to be passed in order to study the course. The token lists are then parsed using a state machine to create a graph-like structure using arrays, which is then converted into graph nodes and assigned relationships and properties before inserted into the graph database.

The graph nodes in PHP are provided by a package called Neo4jPHP³, which is available as a PHAR (PHP Archive)⁴. It also provides an API to communicate with Neo4j.

³<https://github.com/jadell/Neo4jPHP>

⁴<http://www.php.net/manual/en/intro.phar.php>

5.2 Example

In this example of the course EDA221 (Computer graphics), the prescience requirement text is the following:

EDAA01 Programmeringsteknik - fördjupningskurs eller EDA027 Algoritmer och datastrukturer samt FMA420 Linjär algebra.

This is tokenized into a list containing:

["EDAA01", "eller", "EDA027", "samt", "FMA420"]

The list is then parsed from the left into an array structure:

1. "EDAA01" is a course code, add it to current group
2. "eller" is converted into the boolean OR and is skipped
3. "EDA027" is also a course code and is added to the current group
4. "samt" is converted into the boolean AND and therefore the current group is closed and a new one is created
5. "FMA420" is a course code and is added to the new group.

This leaves us with the multidimensional array:

[["EDAA01", "EDA027"], ["FMA420"]]

The multidimensional array is then converted into a graph structure:

- The Start and End nodes are created
- The Start node relates to both elements in the first group
- Each element in the first group is then related to all elements in the second group
- All the elements in the last group is related to the End node
- The graph structure is then inserted into the Neo4j using the PHP interface.

no, he has a thermal detonator! The boolean expression for this example is then:

$$Req_{EDA221} = (EDAA01 \vee EDA027) \wedge FMA420$$

The subgraph for the example is shown in Figure 5.2.

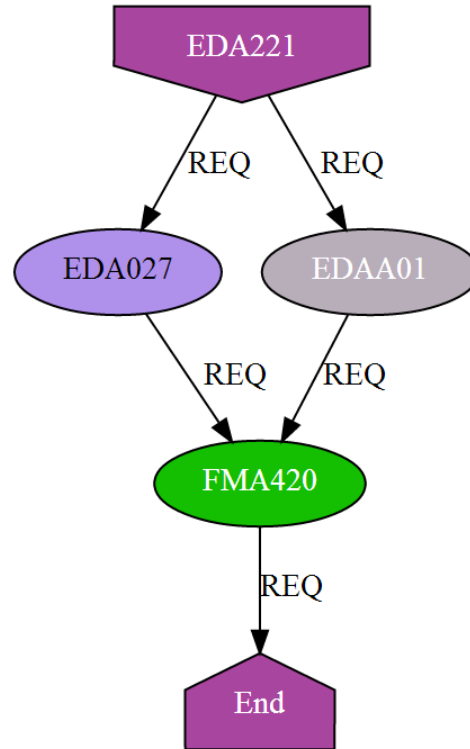


Figure 5.2: Subgraph of the prescience dependencies of EDAA21

5.3 Graph structure

The graph structure (shown in Table 5.1.) we created uses the fact that we want to represent the course prescience dependencies as CNF and that we want to easily find the Start and End nodes for a given course.

Node type	Properties	Outgoing relationships
Start node	tag = course code	REQ, REQ_D, REQ_C
in-between-nodes	stag = course code	REQ, REQ_D, REQ_C, ORIG
End node	end = course code, special = special conditions	-
Root node	-	COURSE

Table 5.1: Graph database structure

By using indexes in Neo4j, it is quite easy to select a Start node as well as the End node. They have different properties, but share one value, the course code, that are stored in the properties **tag** and **end** respectively. The in-between-nodes are then returned as parts of the paths found between the Start and End nodes

by only following `REQ`, `REQ_D` or `REQ_C` (i.e. requiring) relationships. The need for more than one kind of requiring relationship is due to that the course presciences are sometimes individual for the different programmes. In reality it is not used very often, especially when looking at courses which is given for the quite similar programmes D and C.

The `ORIG` relationship type is used to let the in-between-nodes point to the Start nodes that represents their dependency graphs. In addition to the other node types, there is also a singular Root node (with `ID=0`) that has a `COURSE` relationship to all of the Start nodes. This allows for writing cypher queries easier that uses all of the Start nodes as part of the query.

The `ORIG` and `COURSE` relationships were not needed for the validation algorithm to work, but they were quite interesting and handy to have when exploring the possibilities of Neo4j and its querying language Cypher.

Other functions than the course validation was implemented together with Neo4j as well, but they were not included in the final prototype system. These include a traversal algorithm that given a list of courses returns a list of courses which presciences are fulfilled for. The traversal is another way of querying Neo4j (in contrast to Cypher) and is supported by Neo4jPHP. Traversal requires rules written in Javascript that is run in the Neo4j Java application and they are run during the traversal of the graph. The reason for this function never to be implemented into the prototype was due to it being difficult to explain and visualize to the user.

5.4 Graph visualization

To easier visualize the graph and get ideas for possible queries to the graph database, we implemented a converter to be able to show our Neo4j database with GraphViz⁵, a graph visualization tool. By combining Neo4jPHP and the PHP GraphViz library `Image_GraphViz`⁶, it was just a matter of converting the output from Neo4jPHP into the GraphViz dot format and styling the different nodes and relationships. To make the graph drawing look a bit more fancy and appealing, colors for the nodes are randomized every time the graph is drawn.

5.5 Validation

Neo4j has support for a lot of different graph algorithms including some which returns all paths between two nodes.⁷

The algorithm which validates if a student can study a course needs a few parameters:

- The course to be validated
- The time coordinates for when it is to be studied

⁵<http://www.graphviz.org/>

⁶http://pear.php.net/package/Image_GraphViz

⁷<http://components.neo4j.org/neo4j-graph-algo/snapshot/>

- All courses which are planned (both earlier and later than the course to be validated)
- The programme that the student is studying

The algorithm starts by asking Neo4j to find all paths between the Start and End node for the course to be validated as well as fetching any special conditions (which are included as a property in the End node) which might apply. Then the algorithm checks all paths and whether any of the paths can be followed by only passing nodes which represents courses that the student has studied prior (i.e. are planned earlier in the study plan) to the course being validated.

If there were any special conditions that needed to be fulfilled the algorithm will carry out some special checks such as if the student has enough hp at the time or enough hp within a certain set of courses. If any special checks was carried out, their boolean result is returned, otherwise the earlier results from the path check is returned.

The validation algorithm actually have two output results; the first being a boolean that represents if the student is eligible for the course in question, and the second a string containing the special conditions. This algorithm is called by the prototype system using the data the user have inputed in its study plan. The prototype also checks whether the course is studied in a study period where the course is given in addition to the course prescience dependency validation. Since we know that we can't fully validate all special conditions (such as corresponding courses and the students language capabilities) the different combinations of eligibility and special conditions creates two different warnings. If the special conditions only contains validatable data, the course is classified fully by the eligibility output, but if there are special conditions that cannot be validated, the course is classified as maybe possible to study.

Web Application Backend

6.1 Introduction

When we started building the prototype we had right from the beginning a few different principles in mind that we wanted the system to comply with. First of all, the overall goal of the prototype would ultimately be to facilitate the process of designing one's personal study plan. This means that features would generally need to have a practical purpose where some would regrettably be cut out if they don't sufficiently help the user. Secondly, the prototype would need, to as far as possible, be module based with a relatively easily modifiable core. While this is in general a good coding practice, we mainly wanted this type of structure to make it easier for us to work independently on different parts of the system while minimising the integration process. Albeit not our primary goal, we also wanted to code with extensibility in mind, such that the prototype could be extended into a stable product in the future if needed.

The PHP backend interacts with the underlying layers of the system that is described in earlier chapters and a diagram depicting the whole system is shown in Figure 6.1.

6.2 Implementation

6.2.1 PageBuilder class

To achieve a stable module based structure we would need a common solution to build pages with and to handle the modules themselves. For this purpose we built a PHP class named PageBuilder together with a closely associated PHP interface named PageResource. To create a new page, the programmer allocates a new PageBuilder object and initiates it. The PageBuilder will then automatically generate the basic page structure, including scripts, stylesheets and the navigation. The programmer can then continue adding content as usual until finally closing the page structure by invoking the associated method. The Page object usage is exemplified in Appendix C.1 on page 85.

Most modules that would be created for the prototype relies heavily on both Javascript and CSS, sometimes even requiring entire third party libraries to function. To simplify the process of integrating such module on a page, we created the

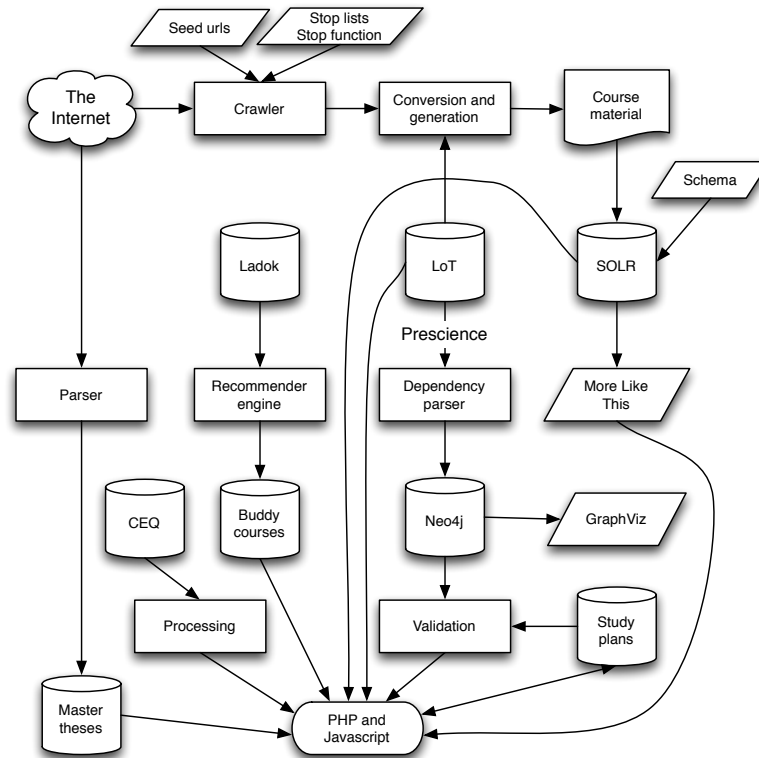


Figure 6.1: A diagram of the system image

PageResource interface which all modules are required to implement. The interface specifies a few additional methods that when called will return a list of file paths to any external Javascript or CSS files that is required by the module. It is also possible to return pure Javascript or CSS code directly if needed. Additionally, the interface specifies a method for returning HTML intended to be placed at the top of the page.

The module methods are however not expected to be called by the programmer directly. Instead the PageBuilder class has methods for automatically loading single or multiple page resources. When initiating the PageBuilder, the programmer merely has to specify what modules to load, and the PageBuilder will automatically prepare the necessary scripts, styling and HTML for the modules to behave properly.

6.2.2 Solr class

While we had already configured Solr to successfully search for courses based on course codes, course names or just free text, it would still need a PHP layer for

easier integration, as well as additional filtering that could not easily be done through Solr itself. In our case, Solr is not expected to be queried directly by the programmer whereas all requests to Solr are entirely managed by the class.

A diagram of basic usage of the Solr class can be seen in Figure 6.2.

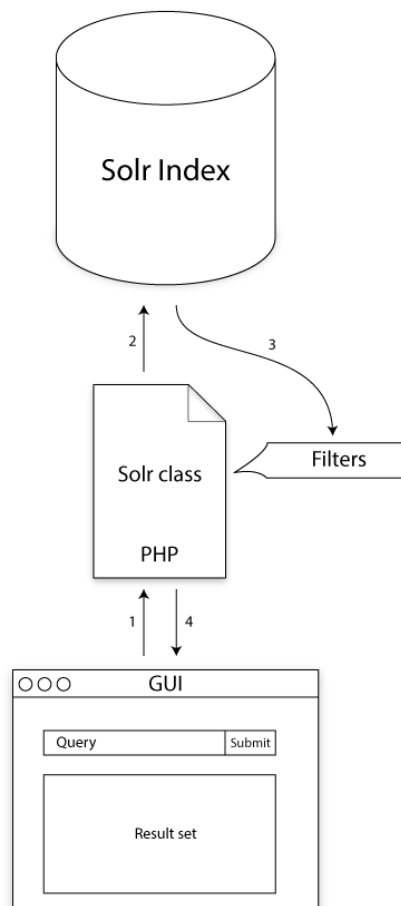


Figure 6.2: Flow chart of course searching

GUI

First of all a GUI would need to be made. We wanted as far as possible the prototype to feel simple and familiar to the user, why our design choices have been heavily influenced by other web search services such as Google Search¹ and Microsoft Bing². The Solr class implements the PageResource interface and is

¹<http://www.google.com>

²<http://www.bing.com>

set to handle the entire GUI, including both search controls, filtering tools and the result set. Searching is mainly done through AJAX requests, but will also seamlessly make use of regular GET-requests if needed (such as at initial page load). Additionally, the search bar autocompletes on course names and codes through the use of the Solr autocomplete handler that we had configured prior.

The result set consists of a simple list of courses, a snippet of highlighting from the most relevant course document, as well as some basic information about the course. A typical item in the result set may look like Figure 6.3.

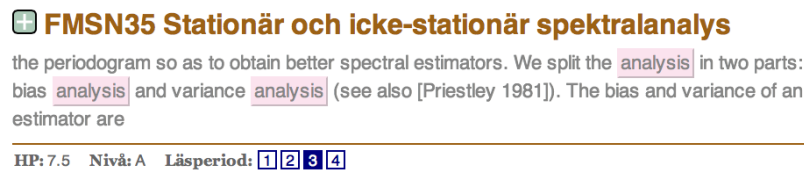


Figure 6.3: Example of a search result item for the query *analysis*

Clicking on the result set item brings up a lightbox (i.e. an asynchronous modal overlay) with extensive information about the course. If the user is logged in, the course may also directly be added to the course basket by clicking on the green button with the plus sign.

Filtering

We additionally support a few different filtering options. In order to minimise the amount of duplicated data, we store a significant amount of course data in a MySQL database separate to the Solr installation. Because of this, the filtering is done entirely by the PHP class and not by Solr. To do this we always fetch a considerably sized result set from Solr even though only a limited set will actually be presented to the user. By doing this we assure that the presented result will not unnecessarily become shortened because of applied filters. Since requests to Solr are done internally to localhost and because of how Solr scales with the size of the result set, the performance actually merely takes an insignificant hit by this oversized result set.

6.2.3 Course class

The course class is an abstract container class for managing all kinds of different courses and their respective data. The class provides basic getters and setters for the most common course types, including course code, course name, number of hp, when it is given and whether the course is advanced or not.

The course class is in turn extended by three different subclasses: LTHCourse, CustomCourse and SplitCourse.

6.2.4 LTHCourse class

This class handles all courses that are given by LTH. It is initiated solely with a course code and then lazily populates itself from different data sources when needed. The basic course information that is inherited from the superclass is fetched from the MySQL database. Part from the basic course information, the class also contains MLT and CEQ data.

6.2.5 CustomCourse class

The primary purpose of this class is to enable the users to create their own courses on the study plan page. This may be the case when a they have taken a course outside of LTH or a course that for some reason hasn't been registered in the system. The class is initiated with a course code which it then uses to fetch the custom course from the respective table in the MySQL database that has been created earlier.

6.2.6 SplitCourse class

A split course generally represents a mandatory course that spans over multiple periods and consists of different parts with separate examination. A prime example of a split course in three parts is the course "calculus in one variable" (FMAA01, endimensionell analys), which spans over three periods with separate exams in each period. Split courses are created when the mandatory courses are inserted in a user's study plan and they are stored in database together with the rest of the user's courses.

6.2.7 Lightbox class

To minimise the need of constantly reloading entire pages we built our own specialised lightbox module. The class implements the PageResource interface and relies heavily on Javascript. The content may be populated either by loading entire pages through AJAX or by simply explicitly assigning the content as a string when calling the Javascript function. We also created a simple navigation bar for the lightbox which is automatically handled by Javascript object. When multiple calls to the lightbox are made, the Javascript object keeps a stack of the requests and displays a convenient "back"-button such that the user may pop the stack and go back to the previous lightbox. If the lightbox is closed the stack is reset, however.

The prototype is a web application that attempts to mimic a web shop where a user picks out courses and puts them into a *course basket*. Instead of checking out the basket, the user rather places the courses from the course basket into a *study plan*. The latter may conceptually be imagined as a manual planning procedure involving paper cards representing courses.

The planning procedure lets the student put course cards into different stacks on a desk organised in a grid pattern. A row represents a year and a column represents a study period. The course basket works in this case like a stack of cards where unplaced course cards are kept. The student may move cards to and from the course basket, as well as between stacks at anytime. Cards may also be discarded completely, except for a few mandatory cards (courses) that can't be discarded.

The web shop part of our system lets the student *shop* for courses. The student can find courses both by searching as well as browsing. The student may also shop by the use of recommendations based on similarities between courses or courses that other students have already studied together.

The final prototype consists of an extensive web based GUI that is based on the building blocks supplied by the backend. The most important aspect when creating the front-end of our prototype is to make it easy to use. We expect that students will use this tool relatively seldom and that they will not likely read the supplied help page. We also believe that they will likely not use the tool at all if they don't understand it immediately.

The prototype is divided into four primary pages¹:

Search page GUI for the search engine. Lets the user search for courses as well as add them to the user's course basket. This is also the start page of the system.

Study plan scheduler Tool for building and viewing the user's study plan. Courses added to the course basket will appear here.

Course list Extensive list of all courses available to respective program. Also lets the user add courses to the course basket.

Help page Simple support page that describes the interface and how to use the system.

¹Not counting the login and select programme pages

The subsequent chapters will describe these pages in detail.

7.1 Course Information Page

In addition to the primary pages, several pages need to show complete information about a specific course. For this we built a new page intended to be opened in a lightbox that displays all available information about a particular course. This includes general course information, CEQ data, similar courses and complementary courses. The information is divided into several collapsible categories to minimise space and clutter. The CEQ data is presented by color coded percentages of the weighted mean, an arrow to indicate the trend and a line chart showing entries from previous years. For drawing the graphs we use a third party library called jQuery Sparklines². The Course Information Page is shown in Figure 7.1.

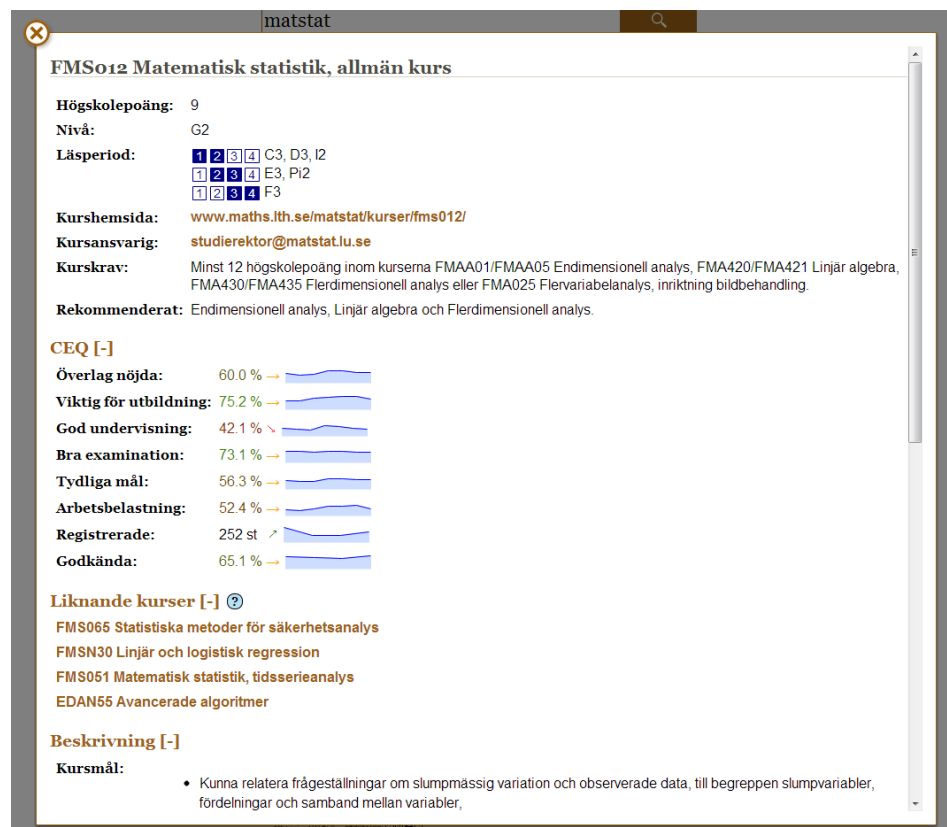


Figure 7.1: The Course Information Page showing FMS012

²<http://omnipotent.net/jquery.sparkline>

This is the entry page of the prototype and is thus the first thing the user sees. The page handles course searching as well as being one of the primary methods of adding courses to the user's course basket.

8.1 Layout

Figure 8.2 shows the top of the search page after the user has entered the query *analys*. The page includes three major components:

- Search field with a submit button
- Toolbox for advanced filters
- Result set

8.1.1 Search field

The search field is a regular text field where the user may write his or her desired search query. The query may consist of course code, course name or any free text that might occur somewhere in the course material. As seen in Figure 8.1 the search field also features word completion for both course code and course name. To accept a suggested word, the user may either click on the word or navigate to it with the keyboard and press enter. The result set will also automatically update continuously while the user is writing.

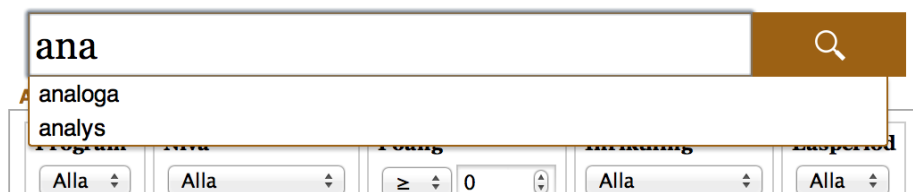


Figure 8.1: A screenshot showing the search field as well as the word completion

Avancerat

Program <input type="button" value="D"/>	Nivå <input type="button" value="Alla"/>	Poäng <input type="button" value="≥"/> <input type="text" value="0"/>	Inriktning <input type="button" value="Alla"/>	Läsperiod <input type="button" value="Alla"/>
Kurslängd <input type="button" value="Alla"/>		Valfrihet <input type="button" value="Alla"/>	Visa valda <input type="button" value="Alla"/>	

Sökningen tog 0.333 sekunder

☒ **FMSN35 Stationär och icke-stationär spektralanalys**
 the periodogram so as to obtain better spectral estimators. We split the **analysis** in two parts: bias **analysis** and variance **analysis** (see also [Priestley 1981]). The bias and variance of an estimator are
 HP: 7.5 Nivå: A Läsperiod:

☐ **FMA430 Flerdimensionell analys**
 i parameterform, skalär- produkt, vektorprodukt. Från **analysen** behövs deriva- tans definition och i samband med båg- längd integraler och **analysens** huvudsats. För avsnittet om ytor bör man också vara bekant
 HP: 6 Nivå: G1 Läsperiod:

☐ **FMAA01 Endimensionell analys**
 Viktiga länkar Nuvarande kurs: Endimensionell **analys** A3 2013 (M, MD) Endimensionell **analys** A2 2012 (M, MD) Endimensionell **analys** A1 2012 (M, MD) Kurshemsidor till gamla kurser Vita
 HP: 15 Nivå: G1 Läsperiod:

☒ **FMI040 Energisystemanalys: Förnybara energikällor**
 förnybara energikällor, särskilt bioenergi, vindkraft och solenergi. Kunna beskriva och **analysera** system för förnybara energikällor ur miljösynpunkt. Beskriva och kritiskt diskutera samhällseliga
 HP: 7.5 Nivå: A Läsperiod:

☐ **FMI050 Energisystemanalys: energi, miljö och**

Figure 8.2: A screenshot of the final version of the web application showing the Search page

8.1.2 Advanced filters

The advanced filters are hidden as default, but can seamlessly be toggled by clicking on the *avancerat* link. For convenience, the state of the filter box is saved so the user doesn't have to toggle it every time he or she visits the site. The system supports the following filters which can also be seen in Figure 8.3:

Program All, D, C

Program that should be eligible to take the course.

Level All, Basic, Advanced

Type of course.

Points \geq , \leq , = followed by [number]

The desired number of hp a course should be worth.

Specialisation All, [list of specialisations for each program]

Which specialisation a course should be part of.

Study period All, autumn 1, autumn 2, spring 1, spring 2

When the course should be given.

Course length All, 1, 2, 3, 4 periods

How many periods a course should span.

Mandatory All, only mandatory, only optional

Whether the course should be mandatory or not

Show picked All, not in course plan

Hide courses already in the user's study plan. Is only available when logged in.

Avancerat

Program D ▾	Nivå Alla ▾	Poäng \geq ▾ 0 ▾	Inriktning Alla ▾	Läsperiod Alla ▾
Kurslängd Alla ▾		Valfrihet Alla ▾	Visa valda Alla ▾	

Figure 8.3: A screenshot showing the advanced filters

8.1.3 Result set

The result set consists of a list of courses that match the submitted query and filters. The respective course name is presented at the top of each item. Below the title there is a snippet of highlighted text, showing what part of the course material (if any) that was matched. In the footer of each item, there are a few fields of important information about the course, including number of hp, level and which study periods the course is given in. If the user is logged in there is also a button in the top left of every item to add or remove the course from the user's course basket.

If the user clicks on a course, the course information lightbox will be presented with extensive information about the course.

A typical result set item can be seen in figure 8.4.

FMSN35 Stationär och icke-stationär spektralanalys

the periodogram so as to obtain better spectral estimators. We split the **analysis** in two parts: bias **analysis** and variance **analysis** (see also [Priestley 1981]). The bias and variance of an estimator are

HP: 7.5 Nivå: A Läsperiod: [1](#) [2](#) [3](#) [4](#)

Figure 8.4: Example of a search result item for the query *analys*

Study Plan Scheduler

This part of the web application handles the creation and modification of the user's study plan as well as validating if it will be possible to graduate when it is studied.

9.1 Layout

2008/2009 (39 hp) [-]

Läsperiod 1 - Ht1 (8 hp)

- Programmeringsteknik* (G1) EDA016 D1 7.5 hp
- Datorer och datoranvändning* (G1) EDA070 D1 3 hp
- Endim A1* (G1) FMAA01 D1 5 hp

Läsperiod 2 - Ht2 (12.5 hp)

- Programmeringsteknik Endim A2* (G1) FMAA01 D1 5 hp

Läsperiod 3 - Vt1 (12.5 hp)

- Endim A3* (G1) FMAA01 D1 5 hp
- Programmeringsteknik - fördjupningskurs* (G1) EDA01 D1 7.5 hp

Läsperiod 4 - Vt2 (6 hp)

- Linjär algebra* (G1) FMA420 D1 6 hp

Kurskorg

- Datorer i system* (G1) EDAA05 D1 8 hp
- Ingenjörprocessen för programvaruutveckling - metodik* (G1) ETSA01 D1 5 hp
- Elektronik* (G1) ETIA01 D1 8 hp

Studieplanen i siffror

År	lp1	lp2	lp3	lp4	hp
2008	8	12.5	12.5	6	39
2009	4.5	22.5	17	16.5	60.5
2010	7.5	22.5	15.5	14.5	60
År	lp1	lp2	lp3	lp4	hp
Totalt:	-	-	-	-	159.5 (0)

Med nuvarande kursplan kan du inte få ut någon examen inom några specialiseringar för det valda programmet. (Dock:

- Du måste ha minst 45hp (varav 30hp avancerat) inom en specialisering
- Obligatoriet måste slutföras
- Det saknas avancerade poäng (minst 75hp inkl. examensarbete)
- Det saknas högskolepoäng (minst 300hp)

Figure 9.1: A screenshot of the final version of the web application showing the Study Plan Scheduler page

The study plan scheduler shown in Figure 9.1. contains five integral parts:

- Toolbar (top-left)
- Course basket (top-right)

- Study plan (below the toolbar)
- Statistics (below the study plan)
- Study plan validation (bottom)

The study plan and the course basket both contain course objects that represent courses.

9.1.1 Course objects

The course objects can be moved between the different periods and the course basket by drag and drop¹. While dragging, the different drop zones are colored differently based on if the course is given at the time the zone represents; valid zones are colored green and the others are colored grey.

The course objects shows some of the most important information directly:

- The name of the course
- The level of the course
- A red asterisk if the course is mandatory
- The course code
- Specialization inherency
- Amount of hp



Figure 9.2: Example of a course object being hovered

The course objects also have their own tools that are shown when they are hovered by the mouse pointer. The tools, shown in Figure 9.2., are from the left:

Course information Opens a lightbox that shows information about the course

Buddy courses and similar courses Opens an overlay that shows courses that students have studied at the same time as the course as well as courses that are similar to the course

¹The drag and drop is supported by the Sortable-class in the Javascript package jQueryUI (<http://jqueryui.com/>)

Disable/enable course Toggles the course being disabled, i.e. not providing any hp and virtually like the course is in the course basket. Useful when trying different course options out.

Mark as reexam Makes the course one period long and skips validation for correct period. This tool is used when a course is meant to be reexamined and not studied at its full length.

Move to basket/remove If the course is in the study plan the tool moves the course to the course basket and if the course is already in the course basket it will be removed.

9.1.2 Toolbar

The study plan toolbar has gone through many changes during the iterations and only contains two tools in the final version of the prototype. The residual tools are a button to expand/contract all years in the study plan and a link (which is hidden by default) to reset the study plan and allow for the user to change programme.

9.1.3 Course basket

The course basket is a staging area much like a shopping basket on an E-commerce web site. It contains courses that the user possibly wants to (or have to) study but that have not yet been placed in the study plan. The user can interact with courses in the course basket either by dragging them or to use the course tools that are displayed on hovering over a course object.

The course basket also contains links to add courses from different sources. There are two special kinds of courses that can only be added from here: Master thesis courses and custom courses.

The master thesis courses are shown in a list and simply added, but the custom courses requires more information to be provided by the user. The custom courses are meant to make the user able to add courses not given by LTH into the study plan. The custom courses requires the user to input a six-character course code, a course name, the amount of hp, the level and the length of the course.

9.1.4 Study plan

The study plan is the main area of the study plan scheduler and contains courses planned to be studied as well as courses already completed. Courses are placed in a period and each year have four periods that can hold courses. Courses that are longer than one period are represented by placing special course objects in the subsequent periods to indicate that the course continues. The study plan contains five years by standard, but can be extended indefinitely by adding more years.

The study plan also presents the amount of hp being earned during a period and summed up over the year. In the prototype, all hp from a course is given in the end of the course and therefore the amount of hp can be pretty low in some periods with many long courses.

9.1.5 Statistics

The statistics table contains information about how the gained hp is divided both in time and in programme specializations. It is meant as an overview to know where more courses can be fitted and to know how many points are within the different programme specializations. The table shows both normal hp and advanced hp given from courses that are of the advanced level. These numbers are also the basis for the study plan validation.

9.1.6 Study plan validation

The study plan validation box shows if the study plan is possible to graduate or what needs to be fulfilled in order to be able to graduate using it. When the study plan is valid, it will also show which specialization is fulfilled for graduation.

9.2 Validation

The study plan scheduler performs two kinds of continuous validation, both of the course prescience requirements and the study plan graduability.

9.2.1 Course required prescience validation



Figure 9.3: Example of course prescience validation error message

All courses in the study plan are validated whenever a course is moved or is being enabled/disabled. If a course fails validation its border is colored orange or red depending on if the course presciences can be validated perfectly or if they require information that only the student knows to be validated correctly. If a course is placed in a wrong period (one where the course is not normally given) the course is also colored orange. A course that in some way fails validation displays a small question mark when hovered with the mouse pointer that can be clicked in order to get more information about what failed. Figure 9.3. shows a error message for

a course where the course presciences can be perfectly validated, but where the required courses are either missing or in planned too late. Some courses, such as the master thesis courses require the student to have earned enough hp and that is also validated.

9.2.2 Study plan validation

Whenever the study plan is changed, it is validated. The validation is done to conform to the current requirements to graduate from the D and C programmes². It checks whether the study plan fulfils:

- A total of at least 300 hp
- At least 75 hp from advanced courses
- That all the mandatory courses are planned
- That there are at least 45 hp (whereof at least 30 hp in advanced courses) that are included in a specialization
- That there is a planned master thesis course

9.3 Discarded interactivity

The first iteration of the study plan scheduler allowed for the user to color courses red, white and green depending on their status using photoshop-like brushes. When the brushes were active, the mouse pointer was changed to reflect which color was being used. Courses colored red represented failed courses, green courses were completed and white courses were being uncertain or planned in the future. The coloring brushes were discarded since the system did not require to know whether a course was completed or not. Except for the coloring brushes, a remove brush was added to easily move courses to the course basket or out of existence. In this iteration the courses could also only be dragged using a small red handle shown in the right corner of the course object.

Iteration two removed the color brushes, but kept the brush concept and created a few new brushes. One of them allowed for courses to be disabled (virtually being in the course basket but still placed in the study plan). Another brush made the buddy course lightbox to displayed and final brush made it possible to drag courses to any period (not just to the valid ones, that was the standard constraint at the time).

Iteration three added a context menu (i.e. right-click menu) on the course objects to more easily apply the tools on a course.

Iteration four removed the drag handle and iteration five removed the tool that allowed for freely moving the courses and made it into the new default behaviour (i.e. removed the moving constraint).

Iteration six removed the context menu and all of the brushes and moved all of the tools into their final position, below the course when hovering it with the mouse pointer. This rather large discarding of tools and their interactions was

²e.g. <http://www.student.lth.se/infocom/c300/>

based on feedback from potential users who had problems understanding how to use the tools or even notice their existence. The conclusion was that the web was ready for much more interactions than the users of that time.

Chapter 10

Course List

The course list page contains the courses given to a specific programme. It shows the mandatory courses and all the possible specializations with their included courses in separate tables.

Our course list page is an enhanced version of an already existing web page in use at LTH. The enhancements includes the ability to client-side sort the columns and the extension of CEQ figures directly in the table.

The course list page is meant as a browsing tool when exploring possible courses to study. Since it is required to have a certain amount of hp that consists of courses in a specialization, it is very important to be able to get an overview of those courses.

Programvara



Kurskod	Kursnamn	Poäng	Nivå	Läsperiod	Nöjd	Angelägen	Undervisning	Belastning	Godkända
EDA230	 Optimerande kompilatorer	7.5 hp	A	1 2 3 4	82.2 %	85.6 %	62.9 %	43.1 %	42.5 %
ETSN05	 Programvaruutveckling för stora system	7.5 hp	A	1 2 3 4	56.2 %	80.8 %	54.9 %	44.5 %	96.9 %
FDAN20	 ...	7.5 hp	A	1 2 3 4	68.6 %	73.2 %	68.6 %	67.5 %	94.4 %

Figure 10.1: Example of a sortable table with courses

Figure 10.1. partly shows one of the tables on the course list page. The table headers are (from the left):

- Course code
- Course name (with a "Add course to basket"-button to the left)
- Total amount of hp given
- Course level
- Period
- CEQ - Overall satisfaction
- CEQ - Important for my education
- CEQ - Good teaching
- CEQ - Appropriate workload

- CEQ - Passed students

The table columns are sortable (both ascending and descending) and it is even possible to sort by multiple columns at the same time by shift-clicking the column headers. The sorting is made by a very useful and easily used jQuery plug-in simply called `tablesorter`¹, which is enhanced by custom comparators to be able to sort the custom data cells in the table.

One of the goals with the course list page was to present as much useable information to the user as possible, but a great constraint with working with tables are their width; when there are too many columns in a table it becomes difficult to read. The last five columns, which contain CEQ figures, are selected based on what the students (in our limited prestudy) used the most when selecting courses to study. All information about a course is available through the course information lightbox opened by clicking the course name.

The table rows are integrated with the study plan scheduler in such a way that courses can easily be added and removed from the course basket. It is also possible to see if the course is part of the mandatory courses or if it is already planned in the study plan.

¹<http://tablesorter.com/docs/>

When the prototype was finished, we wanted to see how the intended users (i.e. students at the D and C programmes) envisage the system. We performed a test session in a computer room at LTH where a reasonably random selection of test users could try the system. We tried to lead the test users as little as possible, but answered direct questions and accepted vocal feedback. The task the test users were assigned was to try to create their study plan for the rest of their education. Throughout the test, we sporadically asked the students about their initial thoughts and feelings about the different parts of the system, without trying to lead the answers significantly.

The designated task was easier for older students who often had previously planned at least parts of their education before trying the system. The younger students were in the beginning concerned that they had to fully create their definite study plan on such short notice. They were however eventually able to be assured enough to however at least create hypothetical study plans, even though the reality might turn out differently.

After the test users were done with testing the prototype, they filled out a web questionnaire that the user evaluation is based upon. The number of test users (15 students) is relatively low, but it is sufficient for this user evaluation.

11.1 Survey

We structured the survey such that it wouldn't take too much time and effort of the user to complete it while still providing necessary information to draw proper conclusions from the results.

Question 1 was asked to confirm that our prototype fills its primary purpose to actually help students to finish their study plan. If the student wasn't able to finish his or her study plan we were however also interested in knowing whether the system failed fundamentally (e.g. the students couldn't finish their study plans because of technical limitations of the system) or if there were merely some minor issues that made the student stop. For that purpose we additionally included an optional associated question (Question 2) where the students could freely explain why they didn't complete their study plan. We wanted to know what advanced features of the system that the testers used. For this, we added questions 3, 4, 6, 7, 9 and 10 to cover all parts of the advanced features. Additionally we also wanted to

know how the students used the fundamental parts of the system, more specifically how the students added courses to the study plan. To cover this, we added the multiple choice Question 8 to let the students specify how they added courses, as well as Question 11 to see whether the system is sufficiently comprehensive or not.

To gather the students' thoughts about the ease of use of the system, we added two more questions, Question 5 and Question 12, asking about how easy it was to use the system, as well as whether they had to use the help page or not.

Lastly, we end the survey with a question asking if the students would consider using this system for real if it was publicly available, as well as an option to add any free text of final thoughts and opinions. We can add that we specifically put Question 13 last in hope that this would let the tester more thoroughly think through his or her opinions about the system by answering all the previous questions first.

The results from the user evaluation survey is shown in Table 11.1.

11.2 Conclusions

The results from the user evaluation turned out to be relatively promising to the favour of the project. All students who answered the survey stated that they would use the system if it was publicly available. 93% of the students answered that they thought the system was easy or very easy to use. From observing the students at the testing phase, we could also gather that while many students had some initial confusion of how to use the system, they would rather quickly learn from simply trying out the different functionality. The students would then finally manage to use the basic features with relative ease. Interestingly enough, very few answered that they ever used the help page. When directly asked about the topic, many answered that they hadn't even reflected on that such page actually existed, even though it's clearly visible in the navigation.

While 60% of the testers managed to successfully complete their study plans, the remainder answered that they were *almost* able to complete their plans. Question 2 however revealed that the absolute majority only failed to complete their plans due to very minor reasons, mostly related to not being able to decide on the last few courses in their plans and not because of limitations in the system itself.

To our annoyance very few students answered that they used any of the more advanced features such as creating custom courses. Those who did use them also had a rather varying (albeit generally inclined towards the positive) experience of the perceived accuracy and efficiency of the tools. While these features may arguably belong to the more academically interesting parts of the project, the end users are still apparently mainly interested in the most basic functionality of the system, whereas the advanced features are merely seen as optional nice-to-have assets.

Together with the excellent result from Question 13, the rather positive free text answers as well as the generally positive results from the survey overall, we happily conclude that from a user standpoint the system is a success!

No.	Question	Alternative breakdown
1.	Did you finish your study plan?	Yes (60%) No (0%) Almost (40%)
2.	If you answered <i>Almost</i> to the above question, please write why so here	FREE-TEXT
3.	Did you use <i>similar courses</i> and/or <i>buddy courses</i> ?	Yes (40%) No, did not need it (47%) No, was there such functions? (13%) No, didn't understand it (0%)
4.	Did you use <i>Mark as reexam</i> ?	Yes (60%) No, did not need it (40%) No, was there such functions? (0%) No, didn't understand it (0%)
5.	Did you use the Help page?	Yes (13%) Yes, but it didn't help me (0%) No (87%)
6.	Did you create custom courses?	Yes (20%) No, did not need it (47%) No, was there such functions? (7%) No, didn't understand it (27%)
7.	Did you ever disable a course?	Yes (13%) No, did not need it (53%) No, was there such functions? (13%) No, didn't understand it (20%)
8.	How did you add courses? (Multi-select)	Search (73%) Similar courses (40%) Buddy courses (13%) Course list (93%) Custom course (13%) Master thesis (53%)
9.	If you used <i>similar courses</i> , how well did it perform? (Range: 0 (bad)...5(very well))	1 (0%) 2 (14%) 3 (14%) 4 (29%) 5 (43%)
10.	If you used <i>buddy courses</i> , how well did it perform? (Range: 0 (bad)...5(very well))	1 (0%) 2 (17%) 3 (33%) 4 (33%) 5 (17%)
11.	Did you ever need to look for any information outside the system?	Yes (35%) No (65%)
12.	How <i>cognitive</i> did you find the system? (range: 0 (not at all)..5(very))	1 (0%) 2 (7%) 3 (0%) 4 (73%) 5 (20%)
13.	Would you use a system like this if it was available?	Yes (100%) No (0%)
14.	Comment the system (bugs, desired features, etc.)	FREE-TEXT

Table 11.1: The results from the user evaluation

We have tried to build our prototype system in such a way that it is easy to expand. Most functionality is created in a generalised way that combines lower level functions to make up for the final results in the high level functions. This both reduces the amount of duplicated code in the code base and it allows for easily creating similar functions.

12.1 Example

There are quite a few different things one can do using the results from the free-text search done by Solr. To test the extensibility of our prototype we implemented a tool meant for senior high school students (gymnasielever) that helps them to select which LTH programme and which specialization they should study by using their already existing interests. The tool works by letting the students enter a few keywords that they would like to have in the courses of the programme that they are about to study. Then a search is done for each of the keywords and the final result is then created from combining the subresults of the searches.

The subresults are ordered lists of courses and to make that into programmes and specializations each course gets a voting power that is:

$voting_power = \frac{1}{pos}$, where pos is the result position starting with 1 (similar to Inverted Nauru on page 32).

Then each course in the results votes with its voting power for the programmes and specializations where it is included. For example, if EDAN20 (Language Technology) is found in a result in third place, it will vote 1/3 point to D and the pv (software development) specialization. The last step is to sum up all the points given to different programmes and specializations and order the results and present them to the student.

Since the scope of our prototype limits its courses to be mostly courses given to the D and C programmes, its use is quite limited, but still presents valid results for keywords where there are searches with satisfying results. The time needed to implement this example (shown in Appendix C.2 on page 85) was under half an hour, due to all the functions already existing on the lower level.

Another similar extension is a tool that returns course teachers instead of programmes and specializations, which would be helpful for journalists and other people who wants to get in contact with people in academia.

12.2 Suggested extensions

Due to the scope of the prototype, some extensions are simply to enlarge the datasets that makes up for the system and to extend its use to the rest of LTH. Other extensions are enhancements that makes the current prototype system help the students even more. During the final user evaluations we gathered a few suggestions for enhancements that would make this system help students even more:

CSN check Count the number of hp per term (i.e. spring and autumn).

Export options Export to pdf, txt, csv, spreadsheet, etc.

Import options Import from Ladok results and/or course registrations.

Add prescience required courses when not in study plan When adding a course with unfulfilled prescience requirements, add also the courses needed to study the course.

Support for prior years and the old 270hp programmes The requirements and mandatory courses changes over the years and support for this would be extremely important.

Support for partly completed courses Some course gives the student hp for completing parts of a course. This is also important for knowing if CSN will still hand out financial support.

Planning summer courses Have a separate period where one can store summer courses.

Alternative mandatory courses Some mandatory courses can be swapped for other courses.

Calendar synchronization Automagically generate a synchronized calendar with the courses in the study plan.

Reexamination and examination dates Show exam and reexam dates, so that one won't get double booked and have enough time between exams.

Support for equivalent courses In some prescience requirements the word *equivalent* is used and it would be nice to know which courses are deemed to be equivalent.

While the overall results of the project were satisfactory, there are still a number of both planned and unplanned limitations that would need to be taken care of before a potential product could be released. Perhaps the most obvious limitation is the overall lack of data, especially for other programmes than the D and C programmes. The system is however built in such a way that adding additional programmes would hopefully require small amount of effort. With that said, new programmes would bring an assortment of new odd special cases of both course requirements as well as structures of course web sites. While it's impossible to predict what these kinds of special cases would be, the system is originally designed with special cases in mind.

There are however a few limitations that applies even to the already implemented programmes. For example the current data collection is gathered solely from one particular point in time and doesn't cover neither future nor historic changes to the courses or requirements. Some courses change their names over the years and some are replaced with entirely new courses. In reality the system should handle overlapping courses and name changes, but in its current state it doesn't. Another issue is the concept of *student batches* dependent on when they started their programme. When students begin their programmes they apply to a specific programme plan (which in fact might not even be completed at the time) that they have to pass in order to graduate. Even students who linger on beyond the standard programme schedule (because of e.g. failed courses or lower study speed) will still need to complete the programme plan of their original student batch. The system however doesn't take this in consideration at all; it merely handles each term disjointly as they were scheduled at the point in time of collection of the data. In our specific test cases this is usually not a problem, but in reality for a very small set of students this limitation could lead to drastically inaccurate validation of their study plan.

Overall, our specific implementation of the validator is only a proof of concept that it's *possible* to validate a study plan, but to actually guarantee validity the system needs to be configured with many additional rules and special cases to fully handle all of the requirements for graduation.

The quality of the course prescience validation system is dependent on the parsed data structure created from the LoT data. The assumption that the prescience requirements can be parsed is generally true, but if the data existed in a formal and fully parsable form it could potentially be perfectly validated. Based

on discussions with personnel at LTH, we apprehend that the work of creating such formal data is already underway.

We found the need for tagging prescience requirements with special conditions that might apply to the validation. While these special conditions may include multiple complex requirements, they are in reality often linked to single courses in the prescience requirements (such as when a course can be partly completed). It would therefore also be helpful if the special conditions were formally stored together with the course that it applies to in the requirements. Because of these special conditions, we realised that our validator would apart from fully validating a prescience requirement as passed or failed, also needs to be able to validate it as uncertain whether the prescience requirement was fulfilled or not. It would arguably be more helpful for the end users if the validator was more certain of its validation than it is in this prototype.

The prototype validates and warns the user of any errors found in the study plan. The users reported however during the user evaluation that they would like even more feedback from the system. One of the problems that they pointed out was that the system did not display a message when a period had too many courses such that the workload would become impractical for the student. Since this is not a hard limitation, we decided to allow the users to decide themselves how much they want to study, but it seems that the more checks and validations we implement, the more users require of the system.

Our prototype has been user evaluated and system tested by potential end users. Since our prototype was adapted for students of the D and C programmes, we used students from those programmes for our testing. These students are generally accustomed to web applications as well as different computer applications that they build and test during their education. Students from other programmes might however not have the same knowledge and experience of computer systems and may very well have other usage difficulties with our prototype.

We tried a few different more or less unconventional ways of interacting with the study plan. It quickly became evident that it was not obvious how to implement an interface that the users easily understood and could handle; many users had trouble with the initial implementations. Our guess is that the test users are used to simple websites and have preconceived ideas of how to use them and therefore have a hard time adjusting to other ways of interacting with web applications. We believe that if the prototype would have been a native desktop application with a customized graphical style, it would not have been an issue in the same way since it is more common to interact with such applications in non-standard ways.

The prototype supports a few features that enables the students to get course recommendations. The recommendation engines are interesting and works fairly well from a technical standpoint. Even though many testers appreciated the advanced features, several test users did not seem to think that the features were absolutely necessary in order to create their study plan. We imagine that a much smaller system that would only contain the course list, the statistics and the study plan without validation would probably be sufficient to satisfy most students and let them plan their education. This simpler system would be much easier to implement and maintain, and it would be clearer to the students that the tool is not omnipotent and that it require additional work of the user.

After the largely positive user evaluation, we conclude that the prototype indeed fulfils its goal of helping students to plan their study plan. The system enables users to with relative ease, search for desired courses and add them to the users' study plan. The system is also able to successfully validate their study plans to inform the students whether it is possible to follow their plan and if it would eventually provide them with a final degree. The absolute majority of the students who answered the provided feedback form, claimed that they would use a system like the prototype if it would actually be released and maintained.

While the search module does provide advanced filtering options, the prototype does in fact not actually enhance the result set by the use of neither Ladok nor CEQ data. However, the data is used for providing additional features and information; the Ladok data is used by the recommender engine as historical data, and the CEQ data is processed and presented to the users so that they may use it to assess a course. Both the Ladok data as well as the CEQ data goes through various layers of preprocessing in order to be used by its respective module in the system.

Two different variants of recommendation engines have been implemented. The first uses similarities in the course material and suggests courses that are deemed similar to a given course; the second uses historical data and suggests courses that other students have taken at the same time as a given course. Both implementations perform fairly well and produce useful results, but it is hard to measure how well they perform since the perfect result is unknown.

VSM and its variants performed rather well and could potentially be used in the prototype to supply similar courses. LSI and PLSI, which reduces words into concepts, did however not perform according to our initial estimation. Both algorithms removed a significant amount of granularity from the course material which reduced the quality of the result set instead of enhancing it, like it does when they are used for searching. Other models for measuring similarity between courses was evaluated whereas the Solr's MoreLikeThis module performed best in the tests.

AJAX Asynchronous JavaScript and XML, a way of letting web applications update parts of the view without having to update the whole page

Apache Lucine Open source search engine framework

Apache Solr Open source search suite based on Lucine

ASCII Character encoding based on the English alphabet

Browser A computer program that displays web contents

C programme A programme at LTH that is specialised in information and communication technologies

Classifier A piece of software that can classify contents, either by parsing or machine learning

Content management system An application that provides easy creation, editing and publishing of contents

Course code Identification code of six characters for a specific course

Crawler An internet bot program that follows links structures and downloads contents from the web, most often for the purpose of indexing

CSN Centrala studiestödsnämnden, gives students financial support if certain requirements are met

D programme A programme at LTH that is specialised in computing and the creation of computer systems

DF Document frequency, how many documents that a term occurs in at least once

Full-text search Search containing any text in the database

GET-parameter Parameter sent to the web server through the URL

GET-request A request sent to a web server requesting a web page or some contents

GUI Graphical User Interface, the front end of a computer system

hashing Converting variable length data into fixed length data

- hp** One hp (högskolepoäng) is equivalent to one ECTS (European Credit Transfer System) point.
- HTML** Markup language for building websites
- Hyperlink** A link from one web page to another
- IDF** Inverse document frequency, $1/DF$
- IR** Information retrieval
- JSON** Javascript-like text representation of data. Suitable for transferring data between different programming languages.
- Ladok** Swedish national system for handling study results of higher education
- LoT** (Läro- och timplaner) Database with programme curricula, time tables and course information
- lp** A time period in which courses can be given, there are four lp in a year
- LTH** The Faculty of Engineering at Lund University
- Neo4j** Open source graph database
- PHP** Server side scripting language for web development
- Query** A request for a special set of data
- Recursive** See Recursive
- Regex** A language that provides the descriptions needed for automated text processing
- SQL** (Structured Query Language) A common query language for relational databases
- Term** A word
- TF** Term frequency, how many times a term occurs in a specific document
- URL** Web address

References

- [1] CERN, *The birth of the web*. 13 June 2013
<http://home.web.cern.ch/about/birth-web>
- [2] C. Castillo, *Effective Web Crawling*. University of Chile, November 2004
http://www.chato.cl/papers/crawling_thesis/effective_web_crawling.pdf
- [3] J. Akbari Torkestani, *An adaptive focused Web crawling algorithm based on learning automata*. Appl. Intell., 37:586–601. 2012
- [4] T. Furche et al, *OXPATH: A language for scalable data extraction, automation, and crawling on the deep web*. The VLDB Journal 22:47–72. 2012
- [5] M. Kumar, R. Vig, *Learnable Focused Meta Crawling Through Web*. Procedia Technology 6:606–611. 2012
- [6] S. Chakrabarti, M. van den Berg, B. Dom, *Focused crawling: a new approach to topic-specific Web resource discovery*. Computer Networks 31:1623-1640. 1999
- [7] DW. Shattuck et al, *Online resource for validation of brain segmentation methods*. Laboratory of Neuro Imaging, University of California. 2008
<http://sve.loni.ucla.edu/instructions/metrics/jaccard/>
- [8] DW. Shattuck et al, *Online resource for validation of brain segmentation methods*. Laboratory of Neuro Imaging, University of California. 2008
<http://sve.loni.ucla.edu/instructions/metrics/dice/>
- [9] C. Manning, P. Raghavan, *Information Retrieval and Web Search, Lecture 6*. Department of Computer Science, University of Vermont.
<http://www.cs.uvm.edu/~xwu/wie/CourseSlides/TFIDF.pdf>
- [10] Susan T. Dumais, *Latent semantic analysis*. Annual Review of Information Science and Technology 38:188–230. 2004
- [11] T. Hofmann, *Probabilistic latent semantic analysis*. Department of Computer Science, Berkeley, University of California. 1999
- [12] A. Kaban, *Machine learning, Lecture 5: Probabilistic Latent Semantic Analysis*. School of Computer Science, University of Birmingham. 2012

- [13] A. Johnson, *How MoreLikeThis works in Lucene*. [web log], 2008.
<http://cephas.net/blog/2008/03/30/how-morelikethis-works-in-lucene/>
- [14] L. Bowen, *Introduction to contemporary mathematics: The borda count method*. University of Alabama, 2001.
<http://www.ctl.ua.edu/math103/Voting/borda.htm>
- [15] A. Alba et al, *Applications of Voting Theory to Information Mashups*. In Proceedings of the 2008 IEEE international Conference on Semantic Computing, pages 10-17. 2008
- [16] Y. Lu, W. Meng, C. Yu, K. Liu, *Evaluation of Result Merging Strategies for Metasearch Engines*. WISE conference, pages 53-66. 2005
- [17] R. Cattell, *Scalable SQL and NoSQL Data Stores*. 2010
<http://cattell.net/datastores/Datastores.pdf>
- [18] Oracle, *Oracle NoSQL Database*. 2011
<http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf>
- [19] Twitter, *Open Source Thanks*. 30 May 2013
<https://dev.twitter.com/opensource/thanks>
- [20] Apache, *Powered by Lucene*. 30 May 2013
<http://wiki.apache.org/lucene-java/PoweredBy>
- [21] Apache, *Solr Terminology*. 31 May 2013
<http://wiki.apache.org/solr/SolrTerminology>
- [22] Dominique Hazaël-Massieux, W3C Staff, *What is a Web application?*. 5 June 2013
<http://people.w3.org/~dom/archives/2010/08/what-is-a-web-application/>
- [23] Alverno College, Basic technology learning module *Benefits and Drawbacks of web applications*. 5 June 2013
<http://depts.alverno.edu/cil/basic/webapps/benefits-drawbacks.html>
- [24] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, *Modern Information Retrieval, Second edition*, Pearson Education Limited. 2011

Prestudy Questions

Our prestudy consisted partly of interviews with students on the D and C programmes. Before the interviews, they were handed a paper (shown in Figure A.) with questions that they could think about for a while. During the interview, all the questions were asked and at the end they could add their thoughts that had come up during the interview.

Kursplanering på LTH

Vi ska skriva ett exjobb om verktygsbaserad kursplanering och skulle vilja få reda på hur studenter vid LTH tacklar problemet att välja kurser varje period i fyran och femman, dokumenterar detta och ser till att sina kurser kan användas för att ta ut en examen inom en specialisering.

Frågor

1. Planerar Du kurser till dina läsperioder i god tid eller bara en läsperiod framåt?
 - a. Hela utbildningen?
 - b. Ett år?
2. Hur gör Du när Du väljer vilka kurser du skall ta i en läsperiod?
3. Dokumenterar Du dina valda kurser på något sätt?
4. Vet Du vilka kurser som företag uppskattar att du studerat?
5. Vilka problem har Du när du planerar din utbildning?
6. Skulle du beskriva hela processen med val av kurser som lätt eller svår?
7. Skulle ett verktyg där Du kan mata in kurser, söka på kurser och hitta kurser som liknar andra kurser hjälpa dig?

Figure A.1: Prestudy question paper

LoT data example

This appendix contains a real life example of a row in the LoT database. EDA260 is a mandatory course given to the D programme in the second year and spans over two periods. Some database fields are empty while some contains HTML tags.

Course code EDA260

Course name Programvaruutveckling i grupp – projekt

hp 6

Level G2

Programme D

Specialization ALLM

Choosability 1 (*i.e. mandatory*)

Year 2

Start period 2

End period 3

Course responsible Boris.Magnusson@cs.lth.se,Gorel.Hedin@cs.lth.se

Prescience requirements Samtliga obligatoriska moment i kursen EDA061 eller de obligatoriska momenten under första läsperioden av EDAF10. Dessutom godkänd tentamen i någon av kurserna EDAA01, EDA027, EDA061, EDAF10.

Prescience recommended *empty*

Teaching goals 1 kunna redogöra för och motivera de olika delteknikerna inom extremprogrammering kunna redogöra för principer för versionshantering

Teaching goals 2 kunna utveckla och leverera en hållbar programvaruprodukt i samarbete med andra kunna tillämpa tekniker och verktyg för automatiserad testning, refaktorisering, och versionshantering kunna tillämpa iterativ planering och uppföljning kunna tillämpa parprogrammering

Teaching goals 3 *empty*

Examination form För godkänt krävs fullgjorda laborationer, godkänt på kontrollskrivningen samt fullgjorda planeringsövningar, långlaborationer och godkänd projektredovisning under kursens andra läsperiod. Detaljer kommer att finnas i kursprogrammet.

Contents En konkret iterativ, så kallad agil, programutvecklingsmetodik används där studenterna tränas i att arbeta i grupp. Den använda metoden tar sin utgångspunkt i idéer från extremprogrammering (XP) med deltekniker som iterativ planering, automatiserad testning, test-first, parprogrammering, refaktorisering och täta leveranser. <p>Kundens/användarens krav formuleras och prioriteras i samarbete med studenterna. Därigenom får studenterna inblick i de olika rollerna i processen exempelvis som kund/användare, projektledare och utvecklare samt förståelse för användarens behov och hur de kan hanteras. Kursen ger praktisk erfarenhet av hur ett småskaligt projekt kan drivas och ger därmed en referensram för påbyggnadskurser, som behandlar metodik för programutveckling för större projekt och organisationer.</p><p>Kursen går över två läsperioder. Under den första perioden varvas föreläsningar med laborationer på enskilda moment som planering, testning, versionshantering och refaktorisering. Under andra perioden delas studenterna in i grupper om cirka 10 personer. Varje grupp driver ett programutvecklingsprojekt som en serie av planeringsmöten varvade med långlaborationer och med en avslutande projektredovisning.</p>

Rationale Många civilingenjörer kommer under sin karriär att samarbeta med andra i utveckling av programvara. Syftet med kursen är att ge kunskaper om och praktisk erfarenhet av hur man samverkar i ett team för att ta fram en programvaruprodukt. Fokus ligger på metoden extremprogrammering, en högiterativ, så kallad agil, utvecklingsprocess som syftar till hållbar utveckling av mjukvara. Kursen tar upp principer för samarbete med beställaren, planering, hållbar design/implementation, testning och leverans. Kursen fungerar samtidigt som fördjupning inom objektorienterad programmering.

Other Kurstyp projekt.

Web page cs.lth.se/eda260

Code Examples

The prototype consists of about 8000 lines of PHP code (and other nested languages), not counting third party code such as GraphViz and Neo4JPHP, but including experiments and processing code. There is also about 1000 lines of CSS stylesheet code and 2000 lines of javascript (not counting third party such as jQuery).

The measurements have been carried out by running

```
find . -name '*.php' | xargs wc -l
```

in a server terminal and is therefore an approximation.

C.1 Home page

The home page of the prototype is a simple example showing the usage of our Page object (i.e. `$PAGE`) and the Solr module. `index.php`:

```
<?php
require_once('layout.php');
$search = new SolrSearch();
print $PAGE->setTitle('Rekommendr - Kurssök')->setCurrentPage('
    rekommendr')->addResource($search)->getHeader();
print $search;
print $PAGE->getFooter();
?>
```

C.2 Gymnasist page

The example given in the Extension chapter (the gymnasist page) is made from a quite short PHP script. It uses the Page object and then initiates searches using the Solr module and then uses the referenced Course objects to calculate the score. The same script contains both the form for the input as well as the processing and the result presentation.

gymnasist.php:

```
<?php
require('layout.php');

print $PAGE->getHeader();
print "<h1>Gymnasist</h1>";
if(!isset($_POST['action'])) {
    printForm();
} else {
    showResults();
}
print $PAGE->getFooter();
exit;

function printForm($j=5) {
    $inputs = '';
    for($i=0; $i<$j; $i++) {
        $inputs .= "<input type=\"text\" name=\"query[$i]\"/><br>";
    }
    print<<<HTML
<form method="post">
<h2>Nyckelord<h2>
<input type="hidden" name="action" value="getStudies"/>
$inputs
<input type="submit" value="Hitta utbildning">
</form>
HTML;
}

function showResults() {
    static $programTranslation = array(
        'F' => 'Teknisk fysik',
        'E' => 'Elektroteknik',
        'M' => 'Maskinteknik',
        'V' => 'Väg- och vattenbyggnad',
        'A' => 'Arkitekt',
        'K' => 'Kemiteknik',
        'D' => 'Datateknik',
        'I' => 'Industriell ekonomi',
        'W' => 'Ekosystemteknik',
        'B' => 'Bioteknik',
        'BI' => 'Brandingenjör',
        'C' => 'Informations- och kommunikationsteknik',
        'N' => 'Nanoteknik',
        'Pi' => 'Teknisk matematik',
        'L' => 'Lantmäteri',
        'MD' => 'Maskinteknik - teknisk design',
        'BME' => 'Medicin och teknik',
    );
    $courses = array();
    $times = array();
    foreach($_POST['query'] as $query) {
        if(!$query)
            continue;
        $solr = new SolrSearch($query);
        foreach($solr->getResults()->results as $i => $item) {
            $score = 1/($i+1);
            $courses[$item->course->tag] = $item->course;
            $times[$item->course->tag] = isset($times[$item->course->
```

```

tag]) ? $times[$item->course->tag]+$score : $score;
}

$specs = array();
$specCourses = array();
foreach($courses as $course) {
    foreach($course->getSpecsArray() as $spec) {
        if($spec[2]!='ALLM')
            continue;
        $key = $spec[0].':'. $spec[2];
        $score = $times[$course->tag];
        $specs[$key] = isset($specs[$key]) ? $specs[$key]+$score
            : $score;
        $specCourses[$key][] = $course;
    }
}
arsort($specs);
echo "Nyckelord: ", trim(implode(' ', $_POST['query']), " ");
echo "<h2>Du bör plugga:</h2>";
$i=0;
foreach($specs as $key=>$score) {
    $i++;
    if(preg_match("^(\\w+)\\:(\\w+)", $key, $matches)) {
        $program = $programTranslation[$matches[1]];
        $spec = $matches[2];
    } else {
        debug($key);
    }
    $cs = array();
    $scores = array();
    foreach($specCourses[$key] as $c) {
        $scores[$c->tag] = $times[$c->tag];
    }
    arsort($scores);
    foreach($scores as $tag=>$score) {
        $c = $courses[$tag];
        $cs[] = "<span title=\"{$c->name}\">{$c->tag}</span>";
    }
    $cs = implode(' ', $cs);
    echo $i, ". ", $program, " - ", $spec, ": ", $cs, "<br>";
    if($i==5)
        break;
}
echo '<br><a href=?>Gör om sökningen</a>';
}
?>

```

C.3 PageState Javascript object

This is a Javascript object that handles browser history primarily when changing the layout with AJAX.

```
var pageState = new function() {
    var that = this;
    var data = {};
    var namespaces = Array();
    var title = false;
    var getParams = {};
    this.setTitle = function(t) {
        title = t;
        document.title = t;
    }

    this.getTitle = function() {
        return title;
    }

    var getStateObj = function() {
        return {data:data, namespaces:namespaces, title:title,
            getParams:getParams};
    }

    this.readStateObj = function(obj) {
        data = obj.data;
        namespaces = obj.namespaces;
        title = obj.title;
        getParams = obj.getParams;
    }

    this.addParam = function(par,val) {
        getParams[par] = val;
    }

    this.addParams = function(obj) {
        for (o in obj) {
            if (o != "")
                that.addParam(o,obj[o]);
        }
    }

    this.addCurrentParams = function() {
        var url = $.url();
        var params = url.param();
        that.addParams(params);
    }

    this.unsetParam = function(par) {
        delete getParams[par];
    }

    this.getHref = function() {
        var r = '';
        var first = true;
        for (par in getParams) {
            if ($.isArray(getParams[par])) {
                for (var i=0; i<getParams[par].length; i++) {
```



```

        r += !first ? '&' : '?';
        r += par+'[]='+getParams[par][i];
        first = false;
    }
} else {
    r += !first ? '&' : '?';
    r += par;
    r += getParams[par] !== undefined ? '='+getParams[par] : '';
    first = false;
}
}
if (r == '') {
    r = window.location.pathname;
}
return r;
}

this.handleState = function() {
    for (var i=0; i<namespaces.length; i++) {
        var name = namespaces[i][0];
        var func = window[namespaces[i][1]];
        func(data[name]);
    }
}

this.pushState = function(namespace, d) {
    if (JSON.stringify(d) == JSON.stringify(data[namespace])) {
        return false;
    }
    if (namespace) {
        data[namespace] = d;
    }
    var t = title ? title : document.title;
    history.pushState(getStateObj(), t, that.getHref());
    return true;
};

this.replaceState = function(namespace, d) {
    if (namespace) {
        data[namespace] = d;
    }
    var t = title ? title : document.title;
    history.replaceState(getStateObj(), t, that.getHref());
};

this.registerNamespace = function(namespace, func) {
    return namespaces.push(Array(namespace, func));
};
return this;
})();

$(window).bind('popstate', function(e) {
    var state = e.originalEvent.state;
    if (state != undefined) {
        pageState.readStateObj(state);
        pageState.handleState();
    }
});

```