NTNU
Innovation and Creativity

Ontology-Driven Query Reformulation in Semantic Search

Geir Solskinnsbakk

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Jon Atle Gulla, IDI
Co-supervisor: Terje Brasethvik, IDI
Jon Espen Ingvaldsen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Semantic search requires that user queries are understood semantically and can be related to the real contents of documents. Ontologies are semantic specifications of important domain properties and are independent of any particular application. For a search application, however, ontologies are more useful for interpreting queries and documents if they are tailored to the particular needs of the search process.

In this project, we will enrich the ontologies with weights that expose how central concepts and relationships are for a particular search context. Moreover, we will use text mining techniques to provide a mapping from concepts to words that are semantically linked to the concepts are frequently used in the documents available.

The candidate will extend the Lucene search core with a query reformulation component that makes use of this enriched ontology to modify the query semantically and map the concepts down to the appropriate words. The resulting search application is to be evaluated against traditional search engines for the petroleum domain.

Assignment given: 20. January 2007
Supervisor: Jon Atle Gulla, IDI

Abstract

Semantic search is a research area in which the goal is to understand the users intended meaning of the query. This requires disambiguation of the user query and interpreting the semantics of the query. Semantic search would thus improve the users search experience through more precise result sets. Moreover, ontologies are explicit conceptualizations of domains, defining concepts, their properties, and the relations among them. This makes ontologies semantic representations of specific domains, suitable to use as a basis for semantic search applications.

In this thesis we explore how such a semantic search system based on ontologies may be constructed. The system is built as a query reformulation module that uses an underlying search engine based on Lucene. We employ text mining techniques to semantically enrich an ontology by building feature vectors for the concepts of the ontology. The feature vectors are tailored to a specific document collection and domain, reflecting the vocabulary in the document collection and the domain. We propose four query reformulation strategies for evaluation. The interpretation and expansion of the user query is based on the ontology and the feature vectors. Finally the reformulated query is fired as a weighted query into the Lucene search engine.

The evaluation of the implemented prototype reveals that search is in general improved by our reformulation approaches. It is however difficult to give any definite conclusion to which query types benefit the most from our approach, and which reformulation strategy improves the search result the most. All four of the reformulation strategies seem to on average perform quite equally.

Preface

This report presents my Master thesis as part of the 5th year of the "Sivilingeniør i Datateknikk" course. The work has been carried out at the Department of Computer and Information Science, Faculty of Information Technology, Mathematics, and Elec- tical Engineering at NTNU. The work has been supervised by Professor Dr. Jon Atle Gulla, and co-supervised by researcher Dr. Terje Brasethvik, and PhD student Jon Espen Ingvaldsen.

I would like to thank my main supervisor Professor Jon Atle Gulla for many fruitful discussions and guidance in the research work. I would also like to thank Terje Brasetvik for feedback on my work, especially in the early stages. Jon Espen Ingvaldsen has given valuable comments on the report, both on structure and content, in addition to be one of the test subjects for the evaluation of the implementation.

Finally I would like to thank the four Master students Christian Bøhn, Øyvind Arne Evensen, Christian Laverton, and Morten Larsen Segelvik for taking the time in the finishing stage of their own Master projects to evaluate the implemented prototype.

Trondheim, June 17, 2007,

____

Geir Solskinnsbakk

Contents

I Introduction **1**

1 Introduction **3**
- 1.1 Background . . . 3
- 1.2 Project Goals . . . 3
- 1.3 Approach . . . 3
- 1.4 Expected Results . . . 4
- 1.5 Outline of this Document . . . 4

II State-of-the-Art **7**

2 Background **9**
- 2.1 Text Mining . . . 9
- 2.2 Information Retrieval . . . 9
- 2.3 Semantic Web . . . 10
- 2.4 Ontology . . . 10
  - 2.4.1 What Constitutes an Ontology? . . . 11
- 2.5 WordNet . . . 11
- 2.6 Concept Feature Vector . . . 11
- 2.7 Semantics & Semantic Search . . . 12

3 Technological overview **13**
- 3.1 Text Preprocessing . . . 13
  - 3.1.1 Tokenization & Transliteration . . . 13
  - 3.1.2 Stop Word Removal . . . 13
  - 3.1.3 Normalization . . . 14
  - 3.1.4 POS-tagging . . . 15
- 3.2 Latent Semantic Indexing . . . 15
- 3.3 Indexing . . . 16
  - 3.3.1 Inverted Files . . . 17
  - 3.3.2 Suffix Arrays . . . 17
- 3.4 Information Retrieval . . . 17
  - 3.4.1 Vector Space Model . . . 17
  - 3.4.2 Retrieval Performance Evaluation . . . 19
- 3.5 Web Ontology Language - OWL . . . 20
  - 3.5.1 OWL Lite . . . 21
  - 3.5.2 OWL DL . . . 21
  - 3.5.3 OWL Full . . . 21

4 Related Work **23**
- 4.1 Classification of Search Systems . . . 23

4.2 Classification of Semantic Search 24
4.3 Related Work 26

III Realization **33**

5 Approach **35**
5.1 Architecture 35
5.2 IIP Ontology 35
5.3 Indexing Phase 36
5.3.1 Preprosessing 37
5.3.2 Indexing 37
5.3.3 Feature vectors 39
5.4 Retrieval Phase 41
5.4.1 Query Reformulation 41
5.4.2 Document Retrieval 44

6 Implementation **45**
6.1 Frameworks 45
6.2 Implementation 46
6.2.1 Indexing 46
6.2.2 Feature Vector Construction 47
6.2.3 Query Reformulation 48
6.2.4 Document Retrieval 51
6.3 Class diagrams 52
6.3.1 Package index 52
6.3.2 Package featurevector 52
6.3.3 Package query 53
6.3.4 Package web 54

IV Evaluation **57**

7 Evaluation **59**
7.1 Evaluation Data 59
7.2 Evaluation Strategy 60
7.3 Evaluation Results 61
7.3.1 User Tests 61
7.3.2 Overlap 64
7.4 Evaluation Summary 67

8 Discussion **69**
8.1 Test Results 69
8.2 Improvements 70
8.2.1 Feature Vectors & Negative Feature Vectors 70
8.2.2 Performance 72
8.2.3 Ontology Reasoning 72
8.2.4 Challenges 72

9 Conclusion **75**

Bibliography **77**

V Appendix 81

A Implementation 83

List of Figures

1.1 The suggested approach 4

2.1 Ontology Spectrum [33] 11

3.1 Precision and Recall [6] 20

4.1 Classification of search systems 24
4.2 The combination of queries used in the search process [38] 29
4.3 Terms and query in DVS [42] 30

5.1 Overview of the approach 36
5.2 OWL definition of a *Christmas Tree* in the IIP ontology. 36
5.3 Part of the IIP ontology focused on the *Christmas Tree* [24]. 37

6.1 Screenshot of the web user interface 51
6.2 Package view of the implemented prototype 53
6.3 Class diagram of the `index` package 53
6.4 Class diagram of the `featurevector` package 54
6.5 Class diagram of the `query` package 54
6.6 Class diagram of the `web` package 55

7.1 Average score for the top 3 and top 10 hits over all the queries for each of the search strategies. 62
7.2 Average score for the top 3 and top 10 hits for each of the search strategies for queries 1 and 2. 62
7.3 Average score for the top 3 and top 10 hits for each of the search strategies for queries 3 and 4. 63
7.4 Average score for the top 3 and top 10 hits for each of the search strategies for queries 5 and 6. 63
7.5 Average score for the top 3 and top 10 hits for each of the search strategies for query 7. 64
7.6 Hits and overlap for queries 1 and 2 64
7.7 Hits and overlap for queries 3 and 4 65
7.8 Hits and overlap for queries 5 and 6 65
7.9 Hits and overlap for query 7 66

List of Tables

7.1 Proposed queries used in the evaluation. 61
7.2 Total overlap for the documents retrieved. 66
7.3 Total overlap for the paragraphs retrieved. 67
7.4 Total number of hits for each strategy. 67

A.1 Stop words 83

Listings

6.1 Pseudo code for the *simple* reformulation strategy 49
6.2 Pseudo code for the *best match* reformulation strategy 49
6.3 Pseudo code for the *ontology structure* reformulation strategy 50
6.4 Pseudo code for the *cosine similarity* reformulation strategy 50

Part I

Introduction

Chapter 1

Introduction

1.1 Background

As the amount of information grows, both in the Internet and in corporate document repositories, users are faced with the overwhelming task of finding information suited for their purpose. Search engines are needed to assist the user in this task. However, most of the search engines employed today utilise term matching to match the user query with the users information need. The user may not have a clear understanding of the domain, making it hard to find the correct information. To make matters even worse, users tend to use short queries, 3 or less terms, to describe the information need [13]. It is thus hard for the user to specify a few good keywords to accurately describe the information need.

Ontology driven Information Retrieval (IR) lets the user search in concept space rather than in keyword space[39]. This project will enrich the concepts of the ontology with contextual information in a relevant domain, trying to enhace the information retrieval task by letting the user search conceptually.

1.2 Project Goals

The main goal of this project is to explore how a semantic search system can be implemented, based on a domain specific ontology and a domain specific document collection. The ontology used in this project is the IIP [1] core ontology containing 18,675 concepts. The motivation for the project is to investigate the usefullness of such a system in a domain specific search setting.

A prototype of the search system is to be implemented using the suggested approach and the results of the prototype implementation will be evaluated.

1.3 Approach

As stated in the problem-description, this project will develop a semantic search prototype based on a semantically enriched ontology. The semantic enrichment is based on

[1] IIP, the Integrated Information Platform for reservoir and subsea production systems project. A research project funded by the Norwegian Research Council, project nr 163457/S30.

the work done in the project “Extending Ontologies with Search-Relevant Weights” [47]. The main part of the project is to explore ways to use the semantic enriched ontology to reformulate queries. Figure 1.1 shows an overview of the suggested approach. The first step is to build feature vectors for each concept in the ontology. The feature vectors are built using text mining techniques applied to a set of domain relevant documents. The query reformulation component is split in two phases, first query interpretation, and secondly query expansion. We have suggested four strategies to interpret the query. Through the query interpretation we find the most related concepts for the query. The query expansion uses the feature vectors as a basis to add semantically related words to the query. The semantically enriched query is lastly fired as a weighted query into the search index powered by Lucene.

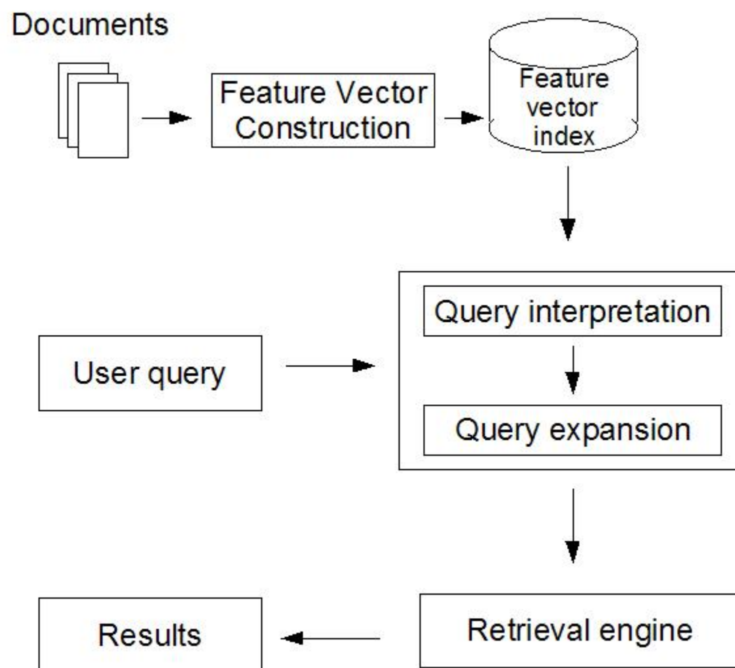

Figure 1.1: The suggested approach

1.4 Expected Results

The result of this thesis will be a semantic search prototype which uses an ontology and a collection of domain relevant documents to reformulate the user query. Through the evaluation of the implemented prototype we compare the query reformulation strategy with standard keyword search, shedding light on the possible improvement of search made by our approach with respect to keyword search.

1.5 Outline of this Document

Chapter 2 of this report gives an overview of the background related to this project. In Chapter 3 we present more detailed theory that is relevant for this project. We will in Chapter 4 give an overview of the current state-of-the-art on the field of semantic search. Chapter 5 gives a detailed description of the approach and techniques used. The

prototype and its implementation is described in Chapter 6, followed by the evaluation of the prototype and the obtained results in Chapter 7. Chapter 8 gives a discussion of our findings, challenges, and possible improvements to the approach. Finally, Chapter 9 gives the conclusion for the report.

Part II

State-of-the-Art

Chapter 2

Background

We will in this chapter give a short presentation of the relevant backgroun for this project.

2.1 Text Mining

Text mining has been defined as "*the discovery by computer of new, previously unknown information, by automatically extracting information from different written resources*" [26]. Text mining and information retrieval are related in the sence that information retrieval is concerned with retrieving information from a document repository (i.e. an index), while text mining is concerned with discovering new information in large amounts of text. This relationship is the same as the one found with data mining and data retrieval.

Text mining and data mining are highly related techniques, as many of the techniques applied to text mining problems have originated in the data mining field. The main difference between the two is that data mining tools are designed to operate on large bodies of structured data, such as databases or XML files, while text mining tools mainly handle unstructured or semi-structured data such as natural language documents, e-mails, or HTML files [14]. Examples of techniques applied to text mining are clustering, information extraction, summarization, concept linkage, and topic tracking [14].

2.2 Information Retrieval

In information retrieval the objective is to search for and retrieve documents fulfilling a users information needs. Classical information retrieval is based on keyword matching, requiring the user to state his information needs in the form of a keyword query. There exists several different information retrieval models, such as the vector space model, the boolean model, and the probabilistic model [6]. Documents are retrieved by matching them against the query supplied by the user. The documents retrieved may be ranked by the search engine, depending on the retrieval model used. As the amount of documents both on the web and in corporations continue to increase, information retrieval is an important area of research.

2.3 Semantic Web

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in co-operation"

Tim Berners-Lee, James Hendler, Ora Lassila
The Semantic Web, Scientific American, May, 2001 [9]

The World Wide Web (WWW) of today is not designed in a way that lets the computer *understand* its contents. Rather it is designed to let humans find and read the information they seek. The semantics of the web are, so to say, out of reach for machines; they can not understand and process the semantics of the information they handle. For instance, a machine that parses a web site only recognizes the web page as a bag of words and is not able to understand how the words are related to the semantic content of the web page. The Semantic Web is proposed as the next generation WWW, and aims at adding machine prosessable semantics to web resources, allowing machines to "understand" the semantics of documents and resources found on the Semantic Web [9].

To render the semantics in the Semantic Web accessible to machines, the semantics must be expressed in a machine processable manner, and the machines must have access to structured sets of information and inference rules to make automatic reasoning possible [9].

Ontologies are a central part of the Semantic Web vision. Ontologies give a conceptual description of a domain, modelling the domain with concepts, relations, and properties. In addition the ontology specifies inference rules on the concepts, allowing complex reasoning within the ontology. These ontologies can be utilized by the Semantic Web to semantically annotate the semantic contents of web pages. In contrast to todays situation, the semantically annotated web pages will let machines "understand" how the information in a web page is related to other pieces of information.

2.4 Ontology

The term ontology has several meanings, depending on who you ask. It has for a long time been used by philosophers to describe the theory of *being* and *existance*. In the information systems context however, an ontology is considered an agreement on a domain specification [48]. The ontology specification defines concepts, properties, and the relations among them within a specific domain. A commonly used definition of an ontology is: *an ontology is a formal, explicit specification of a shared conceptualization.* [20]. In other words, this means that the ontology provides a shared understanding of a domain, its concepts and how they are related to one another. The term formal refers to the ontology as being machine processable, and facilitates sharing and construction of intelligent agents that may *understand* the contents of the information they handle.

The use of ontologies has found its application in many areas, such as knowledge engineering and representation, database design, information retrieval and extraction, and knowledge management and organizing[21].

2.4.1 What Constitutes an Ontology?

In its simplest form a controlled vocabulary may be viewed as an ontology [33]. An example of this may be a catalog. The catalog contains a list of terms, providing an unambigous mapping from a term to its interpretation. Although this may be seen as a simple ontology, [33] provides a definition of what properties an ontology must hold. Figure 2.1 shows a spectrum of different types of ontology candidates, ranging from the simplest on the left side to the more complex on the right side. [33] has drawn a border in the spectrum between the *informal is-a* type and the *formal is-a* type, letting all the candidates to the right be considered real ontologies.

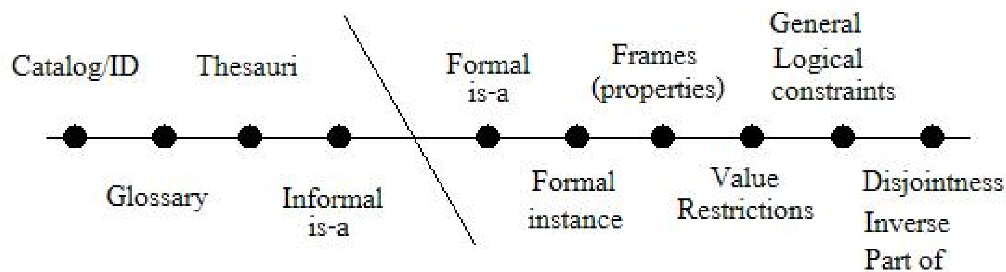

Figure 2.1: Ontology Spectrum [33]

2.5 WordNet

WordNet is a large lexical database for the english language containing nouns, verbs, adjectives, and adverbs[2]. Each wordgroup is grouped into cognitive synonym sets (synsets), where each synset defines a distinct concept. The groups are interconnected through semantic and lexical relations, leading to a network of related concepts and words [2].

WordNet is by some people considered a simple ontology. Taking another look at Figure 2.1 one can see that WordNet would fit in the left side of the scale, namely as a glossary or thesaurus, and would not be considered a real ontology using this definition.

2.6 Concept Feature Vector

[48] provides the following definition of a concept feature vector:

Let C^k be the feture vector of concept K, and let V be the collection of all index words in the document collection.

$C^k = (weight_1, weight_2, ..., weight_t)$

$V = (word_1, word_2, ..., word_t)$

C^k_i *denotes the representativeness of index word V_i to concept K.*

During the construction of the feature vectors, documents are assigned to each concept in the ontology. Two different approaches to document assignment are described by [48], and we will now repeat the simplest here. First all the documents in the document collection are indexed. In the next step, each concept is used as a query into the index.

The resulting documents above a minimum threshold are assigned to the concept feature vector and subsequently used in the construction of the feature vector.

2.7 Semantics & Semantic Search

Traditional Information Retrieval is based on keyword search, presenting the user with the challenge of specifying his information need in terms of keywords. This may sometimes be a hard task, since the user thinks of his information need in concept space (e.g. *the Iraq war*), and has to present a query in word space [39]. Concepts tend to be abstractions and need not be mentioned explicitely in a document covering the concept, making it difficult to match the concept with the keywords.

The objective of semantic search is to let the user search in concept space in contrast to keyword space as is the case of most full-text search engines. Semantic search may take on a wide range of approaches, from specialized indexing (e.g. indexing based on synonyms [39]), query reformulation approaches (e.g. [10, 13]), to semantic annotated text like in the Semantic Web. Additional knowledge, such as ontologies, is often used to find relations and conceptual representations of the content.

Chapter 3

Technological overview

This chapter will give an overview of the theory that is relevant for this project.

3.1 Text Preprocessing

Text preprocessing is usually the first step in a indexing procedure. This section will give an introduction to tokenization, transliteration, stop word removal, and normalization of terms. In addition we give a short introduction to part-of-speech tagging.

3.1.1 Tokenization & Transliteration

The first step of text preprocessing is usually tokenizaion, also called lexical analysis. A digital text can be viewed as a character stream. This stream must be converted to a representation that is appropriate for information retrieval purposes. The character stream is through the application of tokenization converted to a stream of tokens. Tokenization isolates word-like units from the character stream [19] by recognizing the word boundaries. Recognizing the word boundaries is a simple task for a human reading the text, but is somewhat more complicated for machines. Problems encountered during machine processing of the character stream include, among others, removal of tags in marked-up text, for instance HTML tags, handling of abbreviations, numbers, dates, and hyphenations.

Transliteration is an important part of translating terms from one language to another. The problem may not be as apparant in for example spanish/english as in japanese/english or arabic/english due to different alphabets and sound systems. Lets look at transliteration between english and japanese as an example. Japanese uses a syllabic alphabet called *katakana* to transliterate foreign loanwords and proper names. The english word *computer* would be transliterated into the japanese phonetic *konpiyuuta* [30, 43].

3.1.2 Stop Word Removal

Stop words are a group of words in the language which are used frequently and occur to often in documents to be good discriminators. In addition, stop words often carry little or no meaning, such as the words *the* and *or*. Since the words do not contribute significantly to the information retrieval process, these words are often removed during

the preprocessing of the document collection. Stop words typically belong to the word classes articles, prepositions, and conjunctions. A beneficial side effect of stop word removal is that the index size may be reduced by up to 40% [6].

Although generally stop word removal improves the performance of the application, an unfortunate side effect is that the recall (see Section 3.4.2) may be reduced. We will illustrate this with the phrase *to be or not to be*. Depending on the stop word list used, most of the words in the phrase are stop words, and may leave the phrase empty or just containing the word *be*. It is thus obvious that searching for the phrase in this situation might be futile, reducing recall. This unwanted reduction in recall has influenced some of the search engines to do a full text indexing, not removing the stop words [6].

3.1.3 Normalization

A word may appear in several syntactical variations, such as plural, gerund form and past tense with different suffixes. This introduces a problem when matching keywords with its inflectional variations. Normalizing the terms by stemming or lemmatization is a solution to this problem. Stemming reduces the term to its stem by suffix stripping, while lemmatization reduces the term to its morphological root. An example of a word and its syntactic variation is *running* and *runs*, which both are related to the base form *run*. In addition to increasing recall [16], stemming/ lemmatization provides index compression by mapping several syntactic variation of a term to its stem/lemma.

In addition to these normalization techniques will we briefly introduce entity normalization, which deals with normalizing for example a persons name, easing the task of retrieving documents referring to a specific person.

Stemming

A number of different algorithms that perform stemming exist, but the Porter algorithm is one of the most popular [6]. The porter algorithm is a suffix removal algorithm and uses a suffix list for removing and replacing suffixes. The list is specified as rules, and an example of such a rule is $s \rightarrow \oslash$, which converts the plural form into singular form by removing the plural s. The algorithm has five phases, and the rules are organized into five groups corresponding to the five phases of the algorithm. For a more detailed discussion of the Porter algorithm see [41].

Another example of a stemming algorithm is the ”S” stemming algorithm [25], which is a lot less complex than the porter stemming algorithm. The version of the algorithm described in [25] uses three rules to group singular and plural form of words. The ”S” stemming algorithm is a light stemming algorithm. [25] reports that the ”S” stemming algorithm reduced the number of terms from 8460 to 7489 unique stems in the Cranfield collection, while the Porter algorithm redused the same number of terms to 6028 unique stems.

The strength of a stemming algorithm is a measure for how much a word is altered in the stemming process. According to [16] a strong stemmer will, on average, increase recall, decrease precision, and decrease the size of the index.

Lemmatization

Lemmatization is highly related to Stemming, as the objective of the two techniques is the same. Stemming strips words of their suffix, while lemmetization reduces words to their morphological root. Lemmatization typically uses a dictionary in the normalization process, by identifying the inflection of the word through morphological analysis, and mapping the word to the normalized form found in the dictionary. However, lemmatization may also be applied using a rule-based approach [40].

Stemming tends to produce non-existing words, while lemmatization produces words that actually exist. An example of this is *computing*, *computed*, and *computes* which by stemming would be reduced to the stem *comput*, while lemmatization would correctly identify the normalized form as *compute*. Lemmatization has the drawback that it can not lemmatize unknown words. To solve this problem, [32] suggests to use lemmatization for all the known words, and to stem words not found in the dictionary.

Entity normalization

Entity normalization provides normalization of named entities, such as people, locations, companies etc. By adding markup of people etc. one may inhance the information retrieval task by recognizing that a query term actually refers to a person, and use this fact during search. The entities may be recognized by handcrafted finite state patterns, Hidden Markov Models, or a maximum entropy approach [7].

A presentation given by Aleksander Øhrn [29] shows how Fast [1] uses the feature. The example given in the presentation is an excerpt of a text reffering to *Leonid Kuchma* using only the term *Kuchma*. The term *Kuchma* is tagged as being the person *Leonid Kuchma*, and this information may be used to enhance the search, recognizing that the term *Kuchma* refers to the *person* Leonid Kuchma, even though *Leonid* is not mentioned together with *Kuchma* in the text.

3.1.4 POS-tagging

Words can be found in several different syntactic variations, and their meaning may be different depending on the context they appear in. One way of dealing with this is to use a Part-Of-Speech tagger [45]. Part-of-speech tagging labels all of the words in a sentence with their corresponding part-of-speech, such as adjective, noun, pronoun, verb or other catergories. In contrast to ordinary part of speech, part-of-speech tags have a much larger range. Verbs for instance, are not simply tagged as being a verb, but are tagged according to the tense of the verb. There exists several influencial tag sets, such as the Brown Corpus tag set and the Penn Treebank tag set.

Rule-based tagging and stochastic tagging are the two most common approaches for part-of-speech tagging. For an overview of rule based and stochastic taggers, see [4].

3.2 Latent Semantic Indexing

When using a search engine, the users enter a query and may expect to retrieve documents based on the concepts of the query terms, thus retrieving documents based

[1] Fast Search & Transfer, http://www.fast.com

on concept and not on keyword matching. Conventional retrieval techniques, such as the vector space model, using keyword based matching between documents and query reveals two problems [12]. These problems are *synonymy* and *polysemy*.

Synonymy deals with the fact that there exists several words to describe the same object or concept. People tend to use different words to refer to the same concept depending on variables such as context, needs, knowledge, and linguistic habits [12]. The main problem lies in matching the query terms against the document terms. A document may have content that discusses the same concept as the intended query, but as long as the document index does not contain the term given in the query, matching them up is not possible with classical information retrieval.

The use of a thesaurus to expand the query by adding synonyms of the query words to the query has been explored [6]. However, this introduces an unwanted effect by adding polysemy to the query, thus adding non-relevant documents to the retrieved documents and degrading precision.

Polysemy means that a word may have several meanings. An example of this would be the word *banke*, which in one context may refer to a *financial institution*, and in another context may refer to a *river bank*. Specifying a search word and matching it with the document index, may retrieve documents dealing with completely different concepts. Polysemy and synonymy causes respectively *precision* and *recall* to fall.

Latent Semantic Indexing (LSI) [12] is proposed as a solution to these two problems. LSI is based on the assumption that there exists an underlying semantic structure in the textual data. Instead of keyword matching, as in conventional information retrieval, LSI is used to perform concept matching. Statistical methods are used to estimate the semantic structure in the text, giving a conceptual representation of the relations between the terms and documents.

LSI uses Singular Value Decomposition (SVD) to estimate the underlying semantic structure in the text. First, the textual data is represented as a term by document matrix $t \times d$, where each row represents a term, and each column represents a document. Each cell value in the matrix, $f_{i,j}$, is a weighted frequency of the term in the document. The weight may be calculated by for example the $tf \times idf$ score. Next, SVD is applied to the $t \times d$ matrix, reducing the dimesionality of the matrix. The resulting matrix can be seen as dealing with concepts instead of terms.

The retrieval performance of LSI is in [31] reported to range from equal up to 30% better than the best prior methods.

3.3 Indexing

Given a document collection and an information need in the form of a query, a search application will attempt at retrieving the most relevant documents according to the information need. The simplest approach would the to search sequentially through the documents in the collection, retrieving the documents corresponding to the query. However, this does not provide efficiency in large collections [6]. Building an index to provide a more efficient and effective interface to the document collection is a better approach. There exist several approaches to construct such indexes, and among these are *inverted files*, *suffix trees*, *suffix arrays*, and *signature files*. We will in the next subsections give an introduction to *inverted files* and *suffix arrays*.

3.3.1 Inverted Files

Inverted files speed up the search process by allowing for easy acces to terms and frequencies [6]. The inverted file is built by constructing a list of all the terms found in the document collection, called the *vocabulary*. The terms included in the vocabulary are the terms that remain after preprocessing, i.e. if stopword removal has been employed, these are not included in the vocabulary.

For each term in the vocuabulary, there exists a list of documents in which the term appeares. The list may be extended to contain the position of the term within the document, easing positional queries such as phrase queries. The positional information of the term may be given in several forms, such as the character posistion of the first character in the term, the term posistion by counting terms, or by block number. Adding positional information based on blocks partitions the text into blocks of equal size, e.g. 256 characters, and assigns each term within the block the block number as posistion. Although block indexing reduces the size of the index, it adds complexity to the handling of phrase queries, as phrases may exist across block boundaries. The use of positioning information requires accounting for the removed stop words so that positional queries are handled correctly.

3.3.2 Suffix Arrays

Certain types of queries, such as phrase queries, may be expensive to solve using inverted files, and using other index structures than the inverted file may be preferable. Suffix arrays, which are space efficient implementations of suffix trees, are more effective at handling complex queires, such as phrase queries [6]. Using the suffix array for indexing allows for indexing both on term level and character level, which may provide benefits to other applications, such as genetic databases [6].

Although suffix arrays are good at handling complex queries, in most cases, where complex queries are not of priority, suffix arrays are outperformed by the inverted file [6]. For more information of suffix arrays and suffix trees, see [6].

3.4 Information Retrieval

Information retrieval (IR) is the field of searching for and retrieving information or documents that are relevant to a user specified information need. The users information need is specified as a query; a set of keywords that describes the information need. The query is matched against the index to retrieve the documents relevant to the user. IR comprises the representation, storage, organization of, and acces to information [6].

3.4.1 Vector Space Model

The vector space model handles partial matching of documents and queries in contrast to the boolean model, which does not allow partial matching. The documents and queries are represented as vectors in a n-dimensional vector space. Consequently, the document and the query has a vector representation containing the weight for each term. A document vector for document t, $d_t = \{w_{0,t}, w_{1,t}, ..., w_{i,t}, ..., w_{n,t}\}$, is constructed by assigning a weight $w_{i,t}$ to each term i in the document t.

Weighting Scemes

A number of different weighting schemes can be used together with the vector space model [3], and we will now give a short overview of these.

Term weighting is based on two important observations [3]:

- Terms occuring with a high frequency in a document are more relevant than terms occuring less frequent.
- Terms occuring in few documents discriminate better between documents than terms occuring in many documents.

Boolean weighting is one of the simplest approaches, and sets the term weight in the document to 1 if the term is present, and 0 if the term is not present. A slightly more sophisticated appraoch is to use word frequency. These schemes do however not take into account how well terms discriminate between documents, as listed in the second point above. We will now explain two weighting schemes that are a little more complex, namely the $tf \times idf$ and weirdness score.

The $tf \times idf$ weighting scheme takes both term frequency and term distribution among documents into account. The $tf \times idf$ score is based on two measures, the term frequency (tf) and the inverse document frequency (idf).

The term frequency is a measure of the importance of a term within a document, and is by [6] defined as:

$$tf_{i,j} = \frac{f_{i,j}}{max(f_{k,j})}, \text{where} \tag{3.1}$$

$tf_{i,j}$	=	The term frequency for term i in document j.
$f_{i,j}$	=	The raw term frequency of term i in document j.
$max(f_{k,j})$	=	The frequency of the most frequent occuring term k in document j.

Using the idf factor assures that term distribution in the document collection is taken into account. The idf factor is by [6] defined as:

$$idf_i = \log \frac{N}{n_i}, \text{where} \tag{3.2}$$

N	=	The total number of documents.
n_i	=	The number of documents containing term i.

The combination of the term frequency, $tf_{i,j}$, and the inverse document frequency, idf_i, assures that the term weight is balanced both with respect to term frequency within the document and how well the terms discriminate between documents [6].

The weirdness score discerns between prominent and non-prominent terms by using a reference collection. Terms with a high weirdness score are usually prominent in the document collection that is analyzed [23]. It is however not certain that this is the case. Terms that are not found in the reference collection will generate a weirdness score of *Infinity*. The infinity score may be interpreted as the term being domain specific for the analyzed document collection, and thus prominent, but it may also be the case that the infinity score is caused by misspelling of the term or other disturbances. The weirdness score for a term i is calculated according to Equation 3.3 [23].

$$W_i = \frac{\frac{F_{i,dc}}{N_{dc}}}{\frac{F_{i,rc}}{N_{rc}}}, \text{ where} \tag{3.3}$$

W_i = The weirdness score for term i
$F_{i,dc}$ = Number of occurrences of term i in the document collection
N_{dc} = Number of tokens in the document collection
$F_{i,rc}$ = Number of occurrences of term i in the reference collection
N_{rc} = Number of tokens in the reference collection

[46] gives a more comprehensive overview of weighting schemes.

Similarity Measures

When searching for documents by the application of a query, the goal is to find the documents that are most relevant for the query. As the documents in the vector space model are represented by a term vector, the similarity calculation will use the vectors for comparison.

A popular and simple measure for the similarity of two document vectors is the cosine similarity [6]. The documents are seen as vectors in a n-dimensional vector space, and the cosine similarity finds the angle between the vectors [6]. The cosine similarity is defined in Equation 3.4[6].

$$sim(d_j,q) = \frac{\vec{d_j} \bullet \vec{q}}{\mid \vec{d_j} \mid \times \mid \vec{q} \mid} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}}, \text{where} \tag{3.4}$$

$\vec{d_j}$ = The vector representing document j
$\vec{q}$ = The vector representing the query
$w_{i,j}$ = The weight of term i in document j
$w_{i,q}$ = The weight of term i in the query
t = The total number of terms
$sim(d_j,q)$ = The cosine similarity between d_j and q

By using the cosine similarity, the documents are each given a score between 0 and +1. This score gives the degree of similarity between the two vectors, and thus supports partial matching. In addition the cosine similarity between the query and the document is used to rank the retrieved documents. When retieving documents a threshold may be specified, leaving out documents below the threshold.

3.4.2 Retrieval Performance Evaluation

Two commonly used measures for retrieval performance are the *recall* and the *precision* measures [6]. These measures require that one knowes in advance all the relevant documents with respect to a certaian query. Many test collections exist in which such knowledge exists, but in real life, this is not always the case, making the retrieval evaluation harder. [6]. Figure 3.1 shows a document collection, the set of relevant documents, $\mid R \mid$, the set of retrieved documents, $\mid A \mid$, and the intersection of the two sets, $\mid Ra \mid$ for a given query.

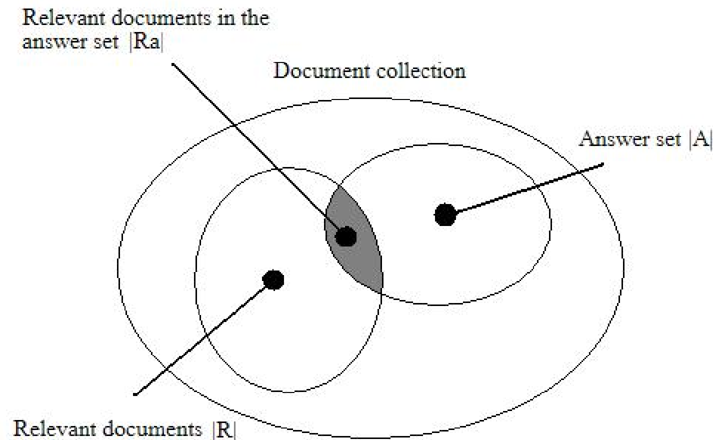

Figure 3.1: Precision and Recall [6]

Precision is defined as the quota of relevant documents with respect to the retrieved set of documents. Retrieval systems with high precision retrieve few documents that are not relevant. Equation 3.5 shows how this quota is calculated [6].

$$Precision = \frac{|Ra|}{|A|} \tag{3.5}$$

Recall is defined as the quota of relevant documents with respect to the total number of relevant documents. Retrieval systems with high recall retrieve a large portion of the relevant documents. Equation 3.6 shows how the recall is calculated [6].

$$Recall = \frac{|Ra|}{|R|} \tag{3.6}$$

In addition to the *recall* and *precision* measures other evaluation performance measures exist, such as the *harmonic mean* and the *E-measure* which are further explained in [6].

When comparing the performance of different retrieval algorithms, plotting the recall versus the precision of the algorithms in the same plot is a nice way of visualizing the difference in performance of the algorithms [6]. These plots are often reffered to as recall versus precision plots. For more information see [6].

3.5 Web Ontology Language - OWL

The Web Ontology Language (OWL) has been developed by the Web Ontology Working Group as a part of the W3C Semantic Activity. OWL is based on DAML+OIL and is placed on top of RDFS in the Semantic Web Tower [8, 33]. OWL extends RDF/RDFS by adding constructs to make the language more powerfull. Three different spieces of OWL have been defined, OWL Lite, OWL DL, and OWL Full. The main difference in the spieces is the expressiveness. OWL Lite is the least expressive, and OWL Full is the most expressive. The reason for having these three variants is that different users have different needs. Some users may favor the use of a language with limited expressiveness,

while keeping reasoning simple, yet others might need maximal expressiveness without any computational guarantees [5].

The layer which OWL is placed on, RDFS, is suitable for constructing classification hierarchies with type properties, and facilitates meta-modelling.

3.5.1 OWL Lite

OWL Lite adds expressiveness on top of RDFS with the OWL language. RDFS for instance, does not have the ability to claim equality between individuals, but OWL Lite has this possibility by the *sameAs* feature[5]. However, OWL Lite is subject to restrictions, using only some of the language features, and is more limited in the use of the features than OWL DL and OWL Full[5].

3.5.2 OWL DL

OWL DL adds even more expressiveness to the ontology than OWL Lite. An example of this is that OWL DL adds the possibility of stating that two classes are disjoint. This is much stronger than stating that the two classes are not equal in OWL Lite [5]. OWL DL uses the same vocabulatry as OWL Full, but is restricted in how the constructs are used, thereby guaranteeing complete reasoning.

3.5.3 OWL Full

OWL Full uses the full vocabulary of the OWL language for maximised expression. A strict segmentation of the vocabulary is applied in the case of OWL Lite and OWL DL, where no term can at the same time be stated as beeing an instance and a class [5]. OWL Full on the other hand does not have this restriction. It is much more similar to RDFS in this sence. An example of this is that in RDFS a class can have both a *type* and *subClassOf* relationship to a second class [5]. Due to the liberal nature of the language, theory shows that building a correct and complete reasoner for OWL Full is impossible [5].

The compatability between the different layers of OWL has been stated by [51] to be the following:

- Every legal OWL Lite ontology is a legal OWL DL ontology.
- Every legal OWL DL ontology is a legal OWL Full ontology.
- Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- Every valid OWL DL conclusion is a valid OWL Full conclusion.

Chapter 4

Related Work

This chapter will give an overview of the state-of-the-art for the field of semantic search. We first start by giving our own classification of search systems, followed by a classification of semantic search systems given by [35]. Finally we report the relevant work for this project.

4.1 Classification of Search Systems

This section will give a short overview of different search types, showing where semantic search fits in. Figure 4.1 shows what kind of matching different search systems do. The example query used to illustrate the search methods is *car*. Simple keyword mathcing systems rely solely on syntactic matching, retrieving instances that exactly match the query, i.e. the term *car* in the figure. Applying stemming and/or lemmatization to the sytstem allows for more relaxed matching allowing the system to retrieve inflectional variations of the query term. In the figure this is illustrated by the query *car* also matching the term *cars*, as *cars* is reduced to its stem/lemma *car*. Search systems in this category still rely on syntactic matching, and are called morpho-syntactic search systems. Full text search engines fall into this category. The main problems with search systems in these two first categories is that they require the user to know in advance the vocabulary of the domain they are searching [13]. This is not a large problem when searching for documents the user has already seen, or the user is familiar with the domian. However, it may be quite challenging to find the result if the domain is new to the user, or the user seeks new information [38]. Documents that actually are relevant on a conceptual level, may use different terms to refer to the concept of the users query, thus not enabling the search system to retrieve these documents.

The next class of search employs glossaries or thesaurus, e.g. WordNet, letting the system retrieve instances based on for example synonym relations. This class of search is somewhat semantic as it recognizes semantic relations among terms. In the figure this is illustrated by the query *car* being matched with a document referring to an *automobile*, which is a synonym of *car*. The actual mechanisms used to do this matching are mainly based on two approaches; indexing based on synonyms (e.g. [39]) and expanding the query with synonyms and related terms (e.g. [52, 36, 10]). Systems in this category somewhat solves the problem of the two first categories, by retrieving documents that contain similar words. However, this approach also introduces polysemy. Polysemy means that a word may have different meaning depending on context. Take for instance the word *bank*, which in one context may refer to *a financial institution*, while in another

context it may refer to *a river bank*. The effect of polysemy may thus clutter the result set by retrieving documents that are not relevant for the users intended meaning of the query.

Adding more power to the search system, by letting the system retrieve class instances, and searching conceptually, we now enter the semantic realm of search. Systems in this category are able to identify concept instances, and documents that are semantically referring to a specific concept. Many different techniques may be employed in this type of search, e.g. knowledge bases, ontologies, semantically annotated text, conceptual query expansion etc. Examples of such systems include [22, 44, 39, 42, 13, 50, 38]. The system that we are developing would also fall into this category. In the figure we have illustrated this category of search by the matching of the query *car* with a document reffering to a *Honda*, correctly identifying a Honda as an instance of the concept *car*. Systems in this class partially solves the problem of polysemy that is inherent in systems based on glossaries or thesaurus. Additional knowledge of the domain is taken into account when retrieving documents, for example by annotating text and using ontologies to find the correct interpretation of the query terms, and to disambiguate the query.

The final class of search is the most powerful; it employs reasoning to identify documents, or ressources that are related to the conceptual meaning of the query. A good example of a system in this category is the WineAgent 1.0 [1], which uses an ontology to infer the best suited wine for a specific meal. In the figure we have illustrated this point with the query *car* matching a document speaking of how the car is run, correctly identifying that the engine is a vital part cars functioning ability. Systems in this category are able to use e.g. ontologies to infer more complex relations between the query and the documents searched, making search systems in this class complex. The systems reffered to as examples in this section will be described shortly.

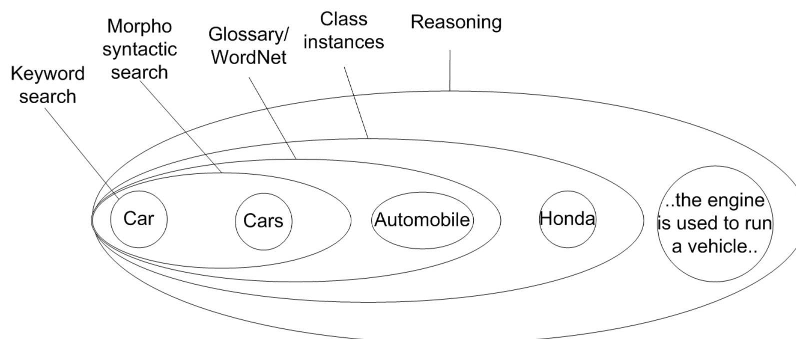

Figure 4.1: Classification of search systems

4.2 Classification of Semantic Search

According to Eetu Mäkelä [35], research on semantic search can be grouped into five directions. These are *augmenting traditional keyword search with semantic techniques*, *basic concept location*, *complex constraint queries*, *problem solving* and *connecting path discovery*. We will give a short introduction to each of the research directions, followed by a more thorough description of related work in the field of query expansion.

[1] http://onto.stanford.edu:8080/wino/index.jsp

Augmenting traditional keyword search with semantic techniques
This research direction aims at augmenting traditional keyword search with different semantic techniques. Unlike the four other reasearch directions presented, this direction operates mainly on data that is not semantically annotated. Techniques such as using ontologies as a basis for a wide range of approaches to augmenting the keyword search are used[35]. The next section will cover several examples of search systems in this category.

Basic concept location
This research direction is based on concepts, instances, and relations one can find in ontologies. Data in the semantic web is encoded using ontological classes and instances of these classes, the real data being the instance data[35]. In addition domain knowledge is based on the relations between classes in the ontology. Using this as a basis, one can locate data using the ontology and constricting properties of instance data by keyword filters as in the SHOE search system [27]. SHOE is a language specification based on SGML and XML that allows the user to define vocabularies and relations along with machine interpretable semantics of the vocabulary. These vacobularies are reffered to as ontologies. The SHOE search system is based on annotated documents using the SHOE ontologies. The user is first presented with a list of ontologies to choose from. Having chosen an ontology, the classes in the ontology are presented in a hierachy, displaying to the user the relations among the classes. Clicking one of the classes, e.g. *Article* (a subclass of *Publication*), the user is presented with properties applicable to the class. These may include *Author*, *Title*, etc. Next the user may enter keywords, used to filter the results. Partial matching of strings is used to increase matching ability. If the user is uncertain of what values are valid for a particular property, the user may be presented with a list of valid values, e.g. names of authors for the *Author* property present in the Knowledge Base (KB). When the user searches, a query based on the class and properites is generated, and entered into a KB containing the class instances (documents), relations, properties etc. The results of the search are presented the user in tabular form, including the URL of documents[27]. Many other systems based on this research direction exist, and [35] gives an overview of these systems.

Complex constraint queries
This research direction deals mainly with complex queries, in which the goal is to locate a "*group of objects of certain types connected by certain relationships*" [35]. In the setting of the semantic web this type of query is handled using graph patterns with constrictions on the type of the object nodes and property edges. These graph patterns are not as easy for the user to formulate as a standard keyword search, and therefore much research is done on user interfaces which aid the user in formulating such queries [35]. An example of a search system in this category is GRQL[37]. GRQL is a graphical user interface which aids the user in navigating a RDF(S) model, spcifically through the RDF(S) class definitions and properties. While browsing the GRQL system generates RQL (RDF Query Language) queries, used to locate information in the RDF(S) model. Thus releaving the user from learning and constructing complex RQL queries.

Problem solving
This research direction is based on formulating a problem and using ontological knowledge to infer the solution. The scenario given in [9], in which a doctors appointment is set up using agents to find a place, time, and doctor that satisfies the user falls into this research direction. However, implementations of systems of this type are rare and simple [35]. A query language intended for simple SQL-like queries in the semantic web based on a DL reasoner is described in [15]. An example of a system in this segment is

the Wine Agent 1.0 [2] which uses an ontology as a basis to reason about which wines are best suited to specific meals. The user specifies contraints, such as type of meal, and the Wine Agent uses the ontological knowledge (types of wine, wine instances, and wine properties) to infer the best suitable wine for the meal.

Connecting path discovery
In the above, the users seeks to find data objects, this research direction differs in that the links between the objects are seen as the interesting part. According to [35], one of the possible applicatoins of this research is in assessing security risk. For government agencies, such interesting links would be of the type "does there exist any connection between terror group *x* and potential recruits *y*?".

4.3 Related Work

We will in this section give an overview of the related work for this project. Refer to Section 4.1 for the classification of these systems. For more research projects on semantic search, see [35, 49]

Three research efforts that use WordNet[3] as a basis for query expansion are [52, 36, 10].

[52] is an early research effort, and uses the *noun* part of WordNet, version 1.3, to expand the query. The research was done using the TREC collection and the already compiled quries 101-150. For each query topic, an average of 2.7 (min 0, max 6) WordNet synsets were added manually, as the purpose of the research was to see if retrieval could be improved using WordNet. Terms from selected fields of the topic statement for each query is used as a basis for the query. The query expansion step uses the synsets that were manually added. Terms were added to the query based on the synsets, the descendants in the *is-a* hierarchy, the parents, and related synsets. Combinations of these approaches were used together with two different selections of original query terms. The first consists of the full topic description, while the second uses a less detailed topic description for the original query. The expanded query was built up of subvectors containing the original query, and one subquery for each of the expanded groups (synset, related synsets etc.). The calculation of the similarity between the query, Q, and a document, D, was done according to Equation 4.1, where each subquery type, $ctype$, was assigned a weight reflecting the importance of the added terms. The original query was usually weighted higher to reflect the importance of the user entered terms.

$$sim(D,Q) = \sum_{ctypei} \alpha_i D \cdot Q_i \tag{4.1}$$

[52] also proposed a method for automatic selection of synsets based on the relatedness to the original query. The experimental results showed that none of the approaches performed significantly better than the original query alone, due to the high level of detail in the original query. However, the author notes that in shorter, less detailed queries, the query expansion approach has potential to improve retrieval due to the small context in the short queries.

[36] describes a system that expands the query based on word sence disambiguation. This process is followed by post processing of the result set, extracting only the relevant parts of the documents, presenting the user with less text. The search system takes as input a keyword query or a sentence describing the information need, e.g. a question

[2] http://onto.stanford.edu:8080/wino/index.jsp
[3] WordNet, http://wordnet.princeton.edu

in natural language. The first step is to POS tag the input query, recognizing noun and verb phrases, and the head words. After the stop words are removed, the remaining words, x_i, are used for expansion of the query. The system employs word sense disambiguation of each query term using Internet search, WordNet and the context of the query by pairing words, mapping each word to its semantic form in WordNet. The internet search is used to rank the different sences of a word by counting hits for each of the sences provided by the word pairs. This is followed by a reordering of the rank using WordNet glosses. The ranked sences are used as a basis for the query expansion, and uses WordNet to add semantically similar words to the query based on the synsets of the words. [36] also employs post processing of the result set. The documents retrieved are searched on a paragraph basis, with the restriction that the query words appear within n consecutive pargraphs in the document. In addition to focusing the search, this reduses the amount of text the user has to sift through. The system was tested with 100 queries; 50 questions from real internet searches, and 50 questions from TREC-6. The queries were expanded using two approaches, one based on the NEAR operator, and one on the AND operator. For each of the two methods, one query was based on the input words, x_i, and one query was based on each input (query) word being replaced by its similarity list, W_i, found from WordNet. The queries have the form:

$$x_1 \text{ AND } x_2 \text{ AND } \ldots \text{ AND } x_n$$
$$x_1 \text{ NEAR } x_2 \text{ NEAR } \ldots \text{ NEAR } x_n$$

The queries constructed by the similarity lists, W_i, were constructed in the same manner. These four approaches retrieved whole documents. The last approach tested was the one applying the post processing using paragraph search. This approach used the similarity lists, W_is, and the AND operator. The results showed that the paragraph based approach performed best, with a precision of 43% , and 90% of the questions being answered correctly for the TREC based questions, the researchers find the results encouraging, and higher than for the other approaches. Especially they note that answering 90% of the questions correctly is significantly higher than current systems. Although the system is based on WordNet to expand the query, disambiguation is employed as a counterweight to the polysemy effect introduced.

[10] is a search system that uses WordNet to expand the query within the geographical domain. The expanded query is based on the synonymy and meronymy relationships for the query terms. The system uses POS tagging as a first step in the query expansion process. Each term is tagged, and is expanded based on the POS tag. All proper nouns are checked against WordNet synsets, and the synset is added if it is of type {*country,state,land*}, with exeption to stopwords and the term itself. The meronyms of the term are retrieved, and all the terms in the synset or gloss containing *capital* are added to the query with exeption to the term *capital*. An example of such a query and its expansion provided by [10] is *shark attacks off Australia and California*. The terms *shark* and *attack* are not expanded, as they do not have the {*country,state, land*} synset among their hypernyms. Next, *Austalia* is considered for expansion. The term is expanded as it has the synset {*Australia*, *Commonwealth of Australia*}, adding the phrase *Commonwealth of Australia* to the query. Looking at the meronyms for *Australia*, we find that the meronym *Canberra, Australian capital, capital of Australia - (the capital of Australia; located in the southeastern Australia)* contains the term *capital*, subsequently adding the term *Canberra* to the query. Looking at the next query term, *California*, the corresponding synset used for expansion is {*California, Golden State, CA, Calif.*}, and the terms *Golden State*, *CA*, and *Calif.* are added to the query. For the term *California* two meronyms were retrieved containing the term *capital*, and the terms

{*Los Angeles*, *City of Angles*} (*...motion picture capital of the world...*) and *Sacramento* (*capital of California*) are added to the query. The final query, after processing, is *"shark attacks" Australia California "Commonwealth of Australia" Canberra "Golden State" CA Calif. "Los Angeles" "City of Angels" Sacramento.* The query is subsequently fired against a search engine based on Lucene. [10] reports a small increase in recall, but a deteriation of the average precision. The authors remark that the results are based on topics from the TREC-8 collection and that they believe the geographical entities mentioned in the TREC-8 queries refer to political entities; "U.S.A." refers to the American government[10].

[39] describes two approaches to conceptual information retrieval using WordNet and Latent Semantic Analysis (LSA). The first approach uses Word Sence Disambiguation (WSD), based on several WSD measures (including WordNet) and the combination of them to identify concepts. WSD is applied to both documents and queries, and the documents are indexed both traditionally, i.e. in a term based index, and concept (synset) based. The terms not found during the application of WSD were traditionally indexed, while words that were disambiguated were indexed in the concept based index by adding conceptually similar terms and related terms from WordNet synsets. This approach was evaluated against two different test collections; the Cranfield collection consisting of 1400 documents and 225 queries, and one test set containg 2714 image captions, and 47 queries. The evaluation pointed out that the concept based index approach applied to the Cranfield collection actually deteriorated the performance with restpect to standard information retrieval. The researchers note that this may be due to errors in the WSD process, or the fact that the queries are quite detailed, containing 8 or 9 terms, leading to unnecessary disambiguation of the query, as it already holds much context. In the second test set however, the researchers found a significant improvement when using the concept based indexing. They point out that the short documents in this collection and the short queries benefitted from the conceptual indexing, by adding related concepts, and thus improved the retrieval performance over standard indexing techniques and keyword search.

The second approach used by [39] is based on a domain specific document collection in combination with a variant of LSA to add conceptually related terms to the query. LSA is used to derive conceptually similar terms, and since some of the terms are not related to the concept, postprocessing using WordNet was employed removing words not related. The evaluation was done with 3 different domain specific test collections to derive conceptual representations of the terms. The top 5 concepts (terms) for a query terms were added to the query. An example provided by [39] is the query term *reservation* in the travel domain, which was expanded with the concepts *agent*, *confirmation*, *hotel*, *customer*, and *rates*. The expanded queries were submitted to google, and were evaluated against the original query. The results showed from moderate to dramatic increase in presicion for the top 50 results of the 10 queries used, due to the added context in the query.

[38] describes a semantic search system based on ontologies. The domain of the system is history, and both time interval concerns as well as conceptual representation of the documents are adressed. The information model used to represent documents is based on one conceptual part, and one temporal part in which an extention of the *temporal vector space model* is used to allow for fuzzy matching of time intervals. In addition the documents are conventionally indexed using a traditional bag of words approach. The conceptual part of the documents are modelled as vectors, containing weighted ontology instances instead of terms, allowing to use the standard vector space model. The temporal aspect is modelled as a weighted set of time intervals. Thus having both

a conceptual representation of the document and a bag of words representation, the retrieval system may use the conceptual information to retrive documents if such information is present, or use the bag of words representation if the conceptual information is not present. During query reformulation the ontology is used to disambiguate the query by presenting the user with a set of ontology instances found in the ontology. This approach is based on full text search on the ontology class labels. For instance, a user entering the query *Churchill* will be presented with the instances of *Churchill* present in the ontology, allowing the user to select the correct interpretation. Further, the query process is based on multiple queries. The query is reformulated using several ontology based heuristics, executed seperately into a full text search engine. The results of the various queries are finally combined using Bayesian inference, see Figure 4.2. The formal evaluation of [38] has not yet been done, and is part of the further work.

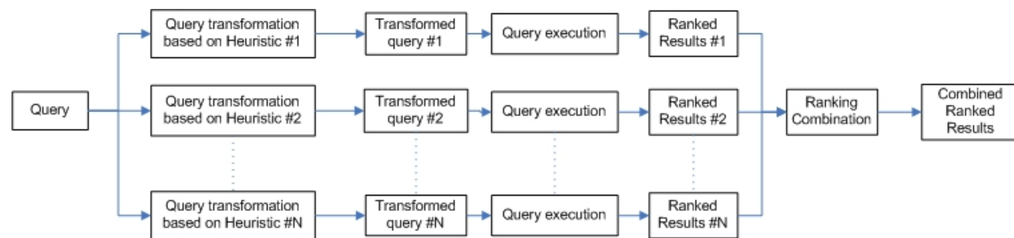

Figure 4.2: The combination of queries used in the search process [38]

Semantic Search[22] takes a different approach, by augmenting traditional IR search results with semantic data through the TAP infrastucture, which is used to publish/-consume data on the Semantic Web. The general idea is that the traditional keyword search is augmented with semantic data by mapping the query terms to concept labels in an RDF repository. The matching concepts are returned together with the standard keyword search results and added to the results page of the user interface. For example when searching for a musician, the augmented information may be concert schedules, albums, picture etc. The authors also present a refinement to the keyword search, where the user may specify the intended meaning if multiple concepts match. The refinement is not based on query expansion, but rather on mechanisms to decide if the document matches the intention. An example of this would be if a user searches for a person, documents containing the persons email address would likely be a document that pertains to the users intended meaning.

[44] combines keyword search with a spread activation algorithm to locate concept instances. The goal is to return instances that are related to a term, even if the term is not mentioned in the instance itself. An example would be to return research areas, students tutored, publications etc. that relate to a person, even if the terms are not mentioned explicitly. Standard text search is applied to a document collection, followed by application of the spread activation algorithm to the returned results. A graph (ontology properties and conepts) is searched in which the links between concepts are initially weighted using the initial text search score. [44] reports that the approach proved successfull in two applications tested.

[42] describes an approach to query expansion using an automatically constructed similarity thesaurus. The main idea behind the query expansion is to add terms to the query based on the similarity to the concept of the query, and not that the expanded query terms should be similar to the query terms. The query expansion model used is based on a probabilistic approach. Figure 4.3 shows terms in Document Vector Space (DVS). The terms are found in DVS, since the similarity thesaurus is based on DVS in

which the role of terms and documents are interchanged. The figure shows the query terms t_1 and t_2, along with the other terms in DVS. q_c is a representation of the query concept, which is calculated as the centroid of the query, q. The fine lines represent the similarity between the terms, and the thick lines represent the similarity between q_c and the terms in DVS. We can see from the figure that the term which is closest to t_1 is t_3, and that the closest term to t_2 is t_6. Using the most basic expansion approach, these terms would be added to the query, however, recall that the idea was to expand the query based on the concept of the query, q_c. This would lead to the choice of expanding the query by t_4 and t_5 since they are the closest terms to the query concept, q_c.

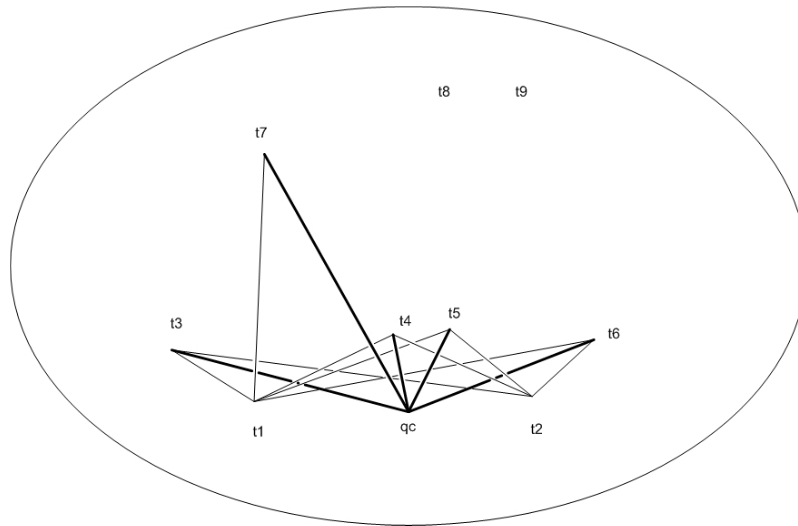

Figure 4.3: Terms and query in DVS [42]

[42] tested the approach on three test collections of different size, and found that the improvement of the approach relative to the original query seemed to increase as the size of the collection increased. The researchers note that this mainly is due to the quality of the similarity thesaurus, since larger collections contain more domain knowledge. They also found that adding more terms to the query seemed to improve the results; the performance of the approach applied to the smaller collections started to decrease when the number of additional terms were above 200, while for the largest collection, the results were still improved by adding more than 200 terms to the query. The improvement of the approach is reported to be between 20-30 % [42].

[13] shows an approach for expanding queries on a conceptual level. The concepts are calculated based on a single document in two different test cases; namely the single best ranked document or the single top ranked relevant document using a simple two word query into the TREC Assosiated Press collection. The two word queries were manually generated by the researchers based on the TREC full text queries (24 queries in total were used). The initial thought was to calculate concept lattices based on the whole collection, but the researchers found that with the ammount of text it was not feasible. They instead calculated a sublattice based on each of the two test cases. The sublattice was used to find concepts (combinations of terms) that in the setting of the test collection had a conceptual meaning. A single document in the collection would typically yield a few hundred concepts. Experimental results showed that even using non-relevant documents could result in good expansion. An example of this is query 58, in which the top document, which was not relevant, and the top relevant document both expanded the original query *rail* and *strike* with *commuter*. Navigation of the concept lattice was used to expand the query by concepts; selecting a subconcept, navigating

down the lattice, would add terms to the query and specialize the query, while selecting a superconcept would navigate up the lattice and remove terms making the query more general. A small simulation program was used to query and navigate the lattice, and the results showed an improvement in the retrieval performance.

An ongoing research project is [50]. The project is very much relevant to our project, since the basic idea is similar. Ontologies are used as a basis for query enrichment. The ontology is semantically enriched by building a feature vector for each of the concepts in the ontology. Text mining techniques are used to build the feature vectors, specifically [50] has used the k-nearest neighbor algorithm to extract terms from a document collection. These terms give a description of concepts that reflect the terminology of the concept in the document collection. The terms of the feature vector are weighted to reflect the importance of the terms with respect to the concept. When refining the query, each concept in the user query is replaced by the corresponding feature vector, resulting in a more contextual description of the concept query term. The expanded query is entered as a weighted query into a standard vector space model retrieval engine, presinting the user with the results. The project is as mentioned still in progress, and the approach has not been evaluated formally. Preliminary results show that the quality of the search result is very reliant on good quality fature vectors. [50] also points out further research work. Among the most important are refinement of the term weight calculations by reasearching alternative approaches to assign terms to the feature vectors, and researching approaches to use the ontology and its semantic relations for post-processing of the result set.

Part III

Realization

Chapter 5

Approach

This chapter will present the suggested approach. The first section gives an overview of the system, while the second section gives an introduction to the ontology used. This is followed by a detailed description of the approach.

5.1 Architecture

We have chosen to split the approach into two separate phases, the indexing phase and the retrieval phase. Figure 5.1 gives an overview of the approach, and shows what actions are taken in the two phases. The indexing phase, as the name indicates, deals with the indexing of documents, both when it comes to the search index and the feature vectors. This phase is not intended to be run often; only when new documents are added either to the search index or the index used to create feature vectors. The retrieval phase deals with reformulating the user query and supplies the user with a ranked list of documents. We will in the next sections give a more thorough description of the separate steps within each of the phases.

5.2 IIP Ontology

The ontology used in this project is the IIP ontology, and the domain of the ontology is the subsea petroleum industry. The IIP ontology is a part of the ongoining research project Integrated Information Platform [1].

The construction of the IIP ontology is an effort towards achieving semantic interoperability in the petroluem domain. Using a terminology standard for the petroleum industry, ISO 15926, the objective is to construct an ontology which provides the industry with an unambiguous and consistent terminology in the subsea petroleum domain[24].

The ontology is built using the web ontology language (OWL), and mostly contains hierarchic relations. The main reason for concentrating on hierarchical relations is that the ontology hierarchy is not deemed sufficiently stable, thus relationships and constraints will be added with time [24].

The *Christmas tree* component is a set of connected parts that sit on top of a wellhead controlling the flow out of the well. Figure 5.2 shows the OWL definition of the concept

[1] http://research.idi.ntnu.no/IIP/

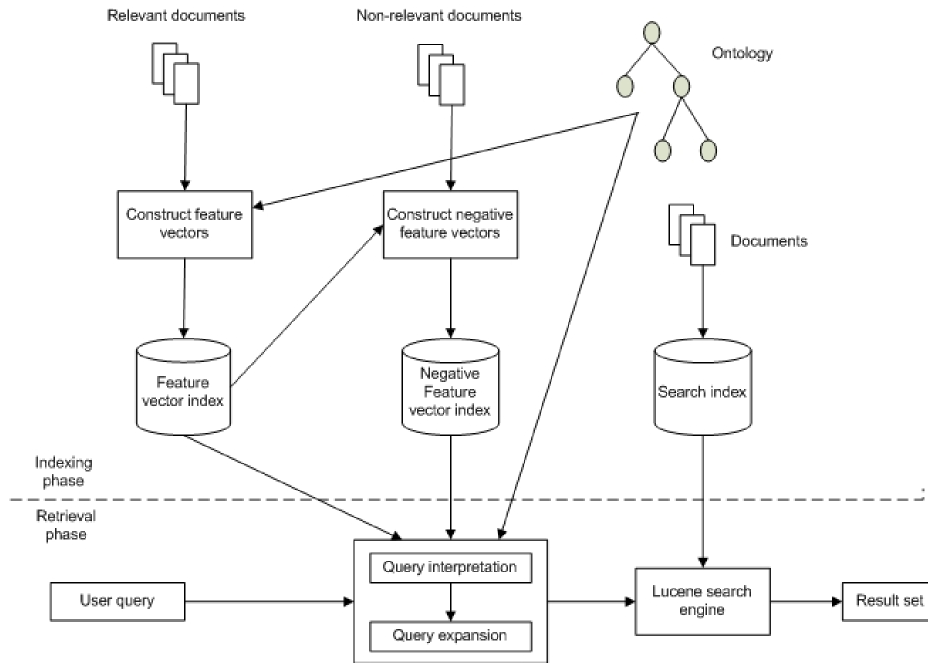

Figure 5.1: Overview of the approach

Christmas tree in the IIP ontology, not including relations and constraints. We see from the figure that the *Christmas tree* is a subclass of the *Artefact* class, and that the ontology includes a defintion of a *Christmas tree*.

```
<owl:Class rdf:about="#CHRISTMAS_TREE">
  <dc:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    CHRISTMAS TREE
  </dc:title>
  ...
  <dc:description rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    An artefact that is an assembly of pipes and piping parts, with valves
    and associated control equipment that is connected to the top of a
    wellhead and is intended for control of fluid from a well.
  </dc:description>
  <rdfs:subClassOf rdf:resource="#ARTEFACT"/>
</owl:Class>
```

Figure 5.2: OWL definition of a *Christmas Tree* in the IIP ontology.

Now let us turn to Figure 5.3 which contains a subset of the class hierarchy focused around the *Christmas tree* class. From the figure we see that the *Christmas tree* is a subclass of *Artefact*, which again is a subclass of *Inanimate physical object*. We can also note that *Pipe* is a subclass of *Artefact*, *Piping network connection* and *Pipeline component*. Finally we see that there exists three subclasses of *Christmas tree*, namely horizontal, vertical, and subsea.

The complete ontology contains about 50,000 concepts arranged in a hierarchy[24]. This project uses only a subset of the full ontology, containing 18,675 concepts.

5.3 Indexing Phase

The goal of the indexing phase is to produce a set of feature vectors for each of the concepts in the ontology and a separate search index which includes all the documents one wishes to search. We will now describe the document preprocessing, the indexing, and the construction of the feature vectors.

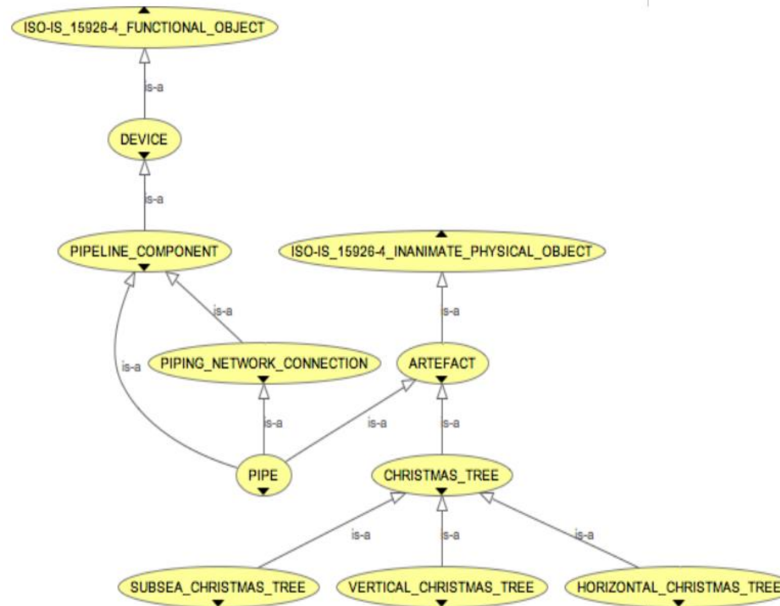

Figure 5.3: Part of the IIP ontology focused on the *Christmas Tree* [24].

5.3.1 Preprosessing

The preprocessing step prepares the documents for the indexing step. The documents are in this step converted from a character stream to a token stream. The first step in doing this is removing all unwanted information from the text, i.e. HTML and other script like tags. The character stream is next tokenized, or split into word-like units called tokens.

Stopwords, words found in a large portion of the documents, are not good descriminators [6], therefore we remove them from the token stream. The stop word list used in the implementation can be found in Table A.1 in AppendixA.

We have chosen to use a light stemming, only removing plural s, following the rule: $s \rightarrow \emptyset$, for all terms not ending with *ss*. The light stemming is chosen so that the side effect of a stronger stemmer (e.g. the Porter Stemmer), namely decreased precision [16] is reduced, while still increasing recall somewhat.

5.3.2 Indexing

The indexing phase produces several indexes which are to be used by different parts of the system. All the indexes are built as inverted indexes [6], to facilitate easy acces to terms, frequencies, documents, and feature vectors.

The indexes we generate are:

- **A main search index**
  The index contains the documents the user wishes to search. The documents are indexed by paragraphs.
- **A document search index**
  The index contains the same documents as the main search index, but the documents are indexed as whole documents.

- **An index used to construct the feature vectors**
  The index contains only relevant documents used as a basis for constructing the feature vectors.
- **A feature vector index**
  The index contains the feature vectors, supporting searching and retrieving the vectors.
- **An index used to construct the negative feature vectors**
  The index contains non-relevant documents with respect to the ontology domain used to construct the negative feature vectors.
- **A temporary negative feature vector index**
  The index is at temporary storage of the negative feature vectors.
- **A negative feature vector index**
  The index contains the negative feature vectors, and supports search and retrieval.

This section will mainly be concerned with describing how we construct the main search index, and the indexes used for the construction of the feature vectors and the negative feature vectors. The three remaining indexes will be discussed in the following section.

The main search index is the index which the user queries with the reformulated query. Traditionally one has used documents as the building blocks of indexes, but we have chosen to build the search index on the paragraph. We believe that indexing on the paragraph level will add presicion to the search by keeping the search context focused, especially with large documents covering a wide range of topics. We have chosen to use natural boundaries in the text, such as two or more consecutive line breaks, to denote a paragraph boundary. This however introduces a challenge; what should be viewed as a paragraph? Strictly using the consecutive line break approach can potentially leave us with a lot of paragraphs containing a few words or a short sentence, due to headings in the text, problems with parsing HTML etc. Therefore we have defined a paragraph as a string of a minimum number of characters which is bounded by two or more consecutive line breaks in both ends. We have proposed to set the minimum charater bound to 1200 characters. This has not been researched, and the size of the paragraphs will not be evaluated as part of the implementation.

In addition we have chosen to let each paragraph be boosted by the paragraph above and below, to make sure that we have sufficient context when searching. We have set the boost factor for the paragraphs above and below the main paragraph to 0.25 (the main paragraph has boost factor 1.0). As the first and last paragraph of a document do not have paragraphs above and below, respectively, we have set the related paragraph for these to the boost factor times 2, in our case 0.5.

This index lets the user retrieve each indexed paragraph for viewing.

The document search index is based on the same documents as the main search index, and the documents are indexed conventionally, i.e. as whole documents. This index is not used for search with our reformulation strategy, but is used for evaluation purposes.

The index used to construct the feature vectors. This index is, as the name implies, an index used for the construction of the feature vectors. The index is built on documents deemed to be within the correct domain. Actually, the index consists of three separate indexes, one for whole documents, one for paragraphs, and one for sentences. The documents are indexed on these three different levels to reflect three views of the

documents. The first view considers the document a document in the conventional sence, i.e. the whole document as a collectin of terms, and indexed as one unit in the *document index*. The next view considers each of the paragraphs in the document as a single document, logically linked to the parent document. Each paragraph is indexed separately in the *paragraph index*. The third and last view considers each sentence in the parent document as a single document, logically linked to the parent document. Each sentence is indexed separately in the *sentence index*.

The reason for this splitting is that we found in [47], that splitting the documents into three different semantic views improved the construction of the feature vectors by altering the boost for terms found in the different parts of the index.

The index used to construct the negative feature vectors. The construction and purpose of this index is very much the same as the index presented last. The main difference is that this index is built of documents that are not relevant at all to the domain. This creates a basis for the construction of the negative feature vectors, which will be described shortly.

5.3.3 Feature vectors

This section will give a description of the approach taken to generate the feature vectors. The approach to construct the feature vectors is largely same as the one described in [47] and is based on a method described by [48], repeated here for convenience.

The indexes based on the relevant documents (document index, paragraph index, and sentence index) are used as a basis for the construction of the feature vectors. Each concept in the ontology is used as a query into the three indexes. Since many of the concepts are named by multiple terms, the query is presented the index as a phrase query. This ensures that only the documents containing the phrase are returned as relevant for the concept. These documents (documents, paragraph-documents, and sentence-documents) are assigned to the feature vector for the concept. As explained in [47], the reason for having three indexes is that we concider the terms as being relevant at three different levels. Terms found in a relevant document, are probably relevant for the concept. At the next level, terms found in the same paragraph as the concept, ergo closer to the concept phrase, should in theory be more relevant to the concept. Following the same logic we view the terms found in the same sentence as the concept phrase to be most relevant to the concept.

Having all the relevant documents (documents, paragraph-documents, and sentence-documents) assigned to the feature vector, the vector is set up by adding all the terms to the vector. The three levels of relevance are reflected in the feature vector by weighting the terms found in the three levels differently. The frequency of each term is multiplied by a constant for each of the three cases. We have chosen to set these constants to 0.1 for terms found in the same document, 1.0 for terms found in the same paragraph, and 10.0 for terms found in the same sentence as the concept. These settings have not been researched extensively, but [47] found that these settings performed better than the more conservative 0.25, 0.50, and 1.0. The freqeuncy for each term is summed up for the concept vector, across all documents containing it and assigned to the concept.

The term frequency in the concept vector is found using Equation 5.1.

$$vf_{i,j} = \alpha \cdot \sum_{d \in D} f_{i,d} + \beta \cdot \sum_{p \in P} f_{i,p} + \gamma \cdot \sum_{s \in S} f_{i,s}, \text{ where} \tag{5.1}$$

$vf_{i,j}$	=	The term frequency for term i in concept vector j.
$f_{i,k}$	=	The term frequency for term i in document vector k.
D	=	The possibly empty set of relevant documents assigned to j.
P	=	The possibly empty set of relevant paragraph-documents assigned to j.
S	=	The possibly empty set of relevant sentence-documents assigned to j.
α	=	The constant for terms found in D, we use 0.1.
β	=	The constant for terms found in P, we use 1.0.
γ	=	The constant for terms found in S, we use 10.0.

The vectors we now have are the basic vectors, but the final score for the terms in the vectors are based on the $tf \cdot icf$ score. Recall from Section 3.4.1 that the $tf \cdot idf$ score is based on the term frequency and the inverse document frequency. The icf factor is in fact just the same, the inverse *concept* frequency which is calculated in the same way as the idf factor, only assuming that the feature vectors now are our documents. The final frequency for each term in the concept feature vector is found using Equation 5.2.

$$tficf_{i,j} = \frac{vf_{i,j}}{max(vf_{l,j})} \cdot \log \frac{N}{n_i}, \text{ where} \tag{5.2}$$

$tficf_{i,j}$	=	The tficf score for term i in concept vector j.
$vf_{i,j}$	=	The term frequency for term i in concept vector j.
$max(vf_{l,j})$	=	The frequency of the most frequent occuring term l in concept vector j.
N	=	The number of concept vectors.
n_i	=	The number of concept vectors containing term i.

The feature vector is normalized to the unit vector length, 1.0, so that searching the vector will reflect the prominence of the term within the vector. At last the feature vectors are indexed in a separate index called the feature vector index. This allows for searching the feature vectors, both with respect to terms and concept name.

The negative feature vectors are based on the index containing the non-relevant documents. These vectors are created to balance the new query, i.e filter out documents in the search process that are not relevant for the domain.

For each concept that has a feature vector in the feature vector index, we build a negative feature vector consisting of terms that are atypical of the ontology domain. Using the concept name as a query into the index containing the non-relevant documents, in the same way as for the feature vectors, might seem as the correct approach. However, since the index contains non-relevant documents, this will probably result in few hits. We have therefore chosen to use the top 5 terms for each of the feature vectors as a reformulated semantic query into the non-relevant document index. From this stage on the construction of the negative feature vectors is the same as for the feature vectors.

Having the negative feature vector complete with $tficf$ score for each term, there almost certainly is some overlap between the feature vector and its corresponding negative feature vector. The approach taken to remove some of the overlap is to remove all terms from the negative feature vector that also are found in the corresponding feature vector. In addition we remove all terms found in more than x% of the feature vectors. The constant x has not been researched, so we have chosen to set it to 5%. This hopefully ensures that the negative feature vector mostly contains terms that are atypical for the domain.

The top y terms in each negative feature vector are indexed without any weights, so that one can retrieve the top y terms when the query is to be reformulated. This index

is called the negative feature vector index. We have chosen to set y to 15, the same number of terms as used in the expansion of the query with the feature vectors.

5.4 Retrieval Phase

The retrieval phase starts with the user entering a query. The system reformulates the query using the feature vectors and the corresponding negative feature vectors and presents the weighted reformulated query into the main search index, retrieves the results and presents them to the user. In addition the user is presented with the results of two standard keyword search approaches using the original query, based on pargaraph search and document search, for comparison.

5.4.1 Query Reformulation

The query reformulation approach has been divided into four separate strategies. These have been named *simple reformulation*, *best match reformulation*, *ontology structure based reformulation*, and *cosine similarity based reformulation.* The query reformulation process has two steps; query interpretation and query expansion. The first step is to map the query terms to one or more concept(s), and the second step expands the query based on the concept feature vectors of the chosen concepts. The basic difference in our four strategies is in how the query terms are mapped on to one or more concept(s).Once the query terms are mapped on to concepts, the query expansion process is similar for each of the four strategies.

The concept(s) are used as a basis for the new query; the top x terms in the concept feature vectors are used to form the new query, where the $tficf$ score is used to weight each term in the query. Recall from Section 5.3.3 that the feature vector is normalized to unit length so that search reflects the prominence of a term within the vector. When the query is reformulated, the top x terms constitute the feature vector, and the new feature vector is normalized. This normalization is not based on length, but on total weight. This is done since we find it more logical that the total weight should be reflected for each query term. For instance, a query of two terms, expanded by two different feature vectors should maintain the weight ratio between the original query terms.

We have chosen to use the top $x = 15$ terms of the query vector for expansion, and give each feature vector a total weight of 10.

In addition the negative feature vector for each of the concept vectors is added to the query in an attempt to remove non-relevant documents from appearing in the result set. The reformulated query has the form:

$term_{0,i}^{w0} \dots term_{x,i}^{wx}\ NOT\,(\,negterm_{0,i} \dots negterm_{x,i}\,)$, where

$term_{n,i}$	=	Term n in feature vector i.
wn	=	The normalized weight for the nth term in vector i based on the $tficf$ score.
$negterm_{n,i}$	=	Term n in negative feature vector i, corresponding to feature vector i.

We have not researched how the negative vectors should be handled, so we propose to use a simple NOT, which leads to documents containing any of the terms in the negative feature vector to be filtered out of the result set. This approach is not so

sophisticated, so more research is needed on how the negative terms can be handled in a best possible way.

A situation that may arise is that one or more of the query terms are not found in the feature vector index, making it impossible to map the term on to a concept. This is handled by letting the term not found be viewed as a single term in the concept vector for the term, and the term is weighted as much as the other feature vectors, namely with a weight of 10.

The original query terms are boosted in the final query to reflect the actual terms the user entered. The boost factor is one that should be researched further, but we have chosen to set it to a modest 3.0, which is added to the query term in the final query. However, we might have the situation in which the best chosen concept to expand does not contain the query term used among the top 15 terms. This may especially be the case in the best match strategy, in which one of the terms may be top 15, and the other(s) may be below the limit. We handle this by adding the query term with a weight of 3.0 to the query.

The reformulated query is used as a weighted keyword query into the search index, returning the best matches for the query.

Simple Reformulation

The simple reformulation strategy is the naive approach, using each term in the user entered query as a separate query into the feature vector index. Each term generates a ranked list of concepts, ranked by the $tficf$ score for the term within the concept vector. The top concept vector for each term is chosen for the query expansion. I.e. the concept vector with the best $tficf$ score for each term is picked as a semantic representation of that term.

Thus a user query of n terms will result in a query expanded by n concept vectors.

Best Match Reformulation

The best match reformulation tries to map the terms over to a single concept. The basic assumption is that the query terms entered by the user are related, and that this relation should be possible to find in the feature vectors. This is done by using the query terms as a query into the feature vector index, requiring all terms to be present in the feature vector. The concept vectors containing all terms are returned, and a ranked list of concepts is generated by calculating a total score for each concept according to Equation 5.3. This approach assumes that all terms are equally important, as they have equal wheight in the score calculation.

$$Score_c = t_{0,c} + t_{1,c} + \ldots + t_{n-1,c}, \text{ where} \tag{5.3}$$

$Score_c$	=	The score for concept c.
$t_{n,c}$	=	The $tficf$ score for query term n in the feature vector for concept c.

The concept chosen for expansion is the one with the highest score according to Equation 5.3.

Ontology Structure Based Reformulation

The ontology structure based reformulation strategy is a strategy in which one attempts to disamiguate the query. The simple reformulation strategy simply maps each query term to the best matching concept vector. However, this may not be the most correct mapping, it may be the case that the first terms best concept is strongly related to the second best concept of the second term. This relation will not be recognized by the simple reformulation strategy.

The goal of this strategy is to use the ontology as a basis for finding the best match between a pair of query terms. We generate a graph, represented by the concepts in the ontology as nodes and the child/parent relations as edges in the graph. For each query term the top 15 ranked concepts according to $tficf$ score is used in the graph search. The graph search is executed as a breadth first search [11], finding the length of the path between every pair of concepts. Since the ontology is large, and taking the fact that relatedness in this setting is based on concepts being close to one another in the ontology, we have chosen to search to a maximum depth of 5.

The score is calculated using Equation 5.4.

$$Score_{cin,cjm} = tficf_{i,n} \cdot tficf_{j,m} \cdot \frac{1}{path(cin,cjm)}, \text{where} \tag{5.4}$$

$Score_{cin,cjm}$	=	The score for the concept with rank n related to query term i and the concept with rank m related to query term j.
$tficf_{k,l}$	=	The t$ficf$ score for query term k in the ralated concept ranked as l.
$path(cin,cjm)$	=	The path length between the concept with rank n related to query term i, and the concept with rank m related to query term j in the ontology. Note that the path length between cin and cjm is set to 0.5 if $cin \equiv cjm$.

The pair of concept vectors with the best score is chosen for the query expansion.

Consine Similarity Based Reformulation

Just as with the ontology structure based strategy, the cosine similarity based strategy is an attempt to disambiguate the query. However, the approach is slightly different. The approach is based on the consine similarity measure between pairs of concept feature vectors. We showed in [47] that using the cosine similarity between feature vectors did reflect some semantic relations between the concepts. We have therefore proposed to incorporate the cosine similarity to disambiguate the query.

As with the ontology structure based strategy, we first generate the top 15 concepts for each term in the user query. The next phase is to calculate the score for each pair of concepts. This is done according to Equation 5.5.

$$Score_{cin,cjm} = tficf_{i,n} \cdot tficf_{j,m} \cdot sim(cin,cjm), \text{ where} \tag{5.5}$$

$Score_{cin,cjm}$	=	The score for the concept with rank n related to query term i and the concept with rank m ralated to query term j.
$tficf_{k,l}$	=	The $tficf$ score for query term k in the ralated concept ranked as l.
$sim(cin,cjm)$	=	The cosine similarity between the concept with rank n related to query term i and the concept with rank m related to query term j in the ontology.

The pair of concepts with the best score is chosen for the query expansion.

Because of computational complexity the ontology structure based strategy and cosine similarity based strategy will only be implemented for 2 query terms.

5.4.2 Document Retrieval

The reformulated query (top terms in the feature vectors + top terms in the corresponding negative feature vectors) is presented to the main search index as a weighted query. The results are ranked and presented to the user. Recall that the index is built on paragraphs, letting the user browse the paragraph that scored best and the document as a whole.

Chapter 6

Implementation

This chapter will give a description of the implemented prototype. First we will describe the APIs and additional software that has been used for the implementation, next we will give a textual description of the prototype. The last section in this chapter gives a structural description of the prototype, illustrated with class diagrams.

6.1 Frameworks

This section will give a short overview of the APIs other than Java (version 1.6.0), and additional software used during the implementation of the prototype. For more information about each of the elements please consult the web page for the API/software.

Lucene The open source project Lucene [1], is used as a basis for the implemented prototype. Lucene is part of the Apache Software Foundation [1]. The book *Lucene in action*[17] gives a more thorough walkthrough of Lucene and its posibilities.

Lucene is designed to be an easy to use Application Programmers Interface (API), which features both text indexing and text search. The API is written in Java, but is also available in other languages [1].

All indexing and searching of the created indexes are done with the help of the Lucene API version 2.1.0. The Lucene `IndexWriter` class has been used for indexing text, both documents and feature vectors. A custom `Analyzer` has been used to tokenize and preprocess the tokens (stemming, stopword removal, etc.). Lastly the Lucene `IndexSearcher` class has been used to search the indexes created.

OWLApi The implemented prototype uses the open source API OWLApi [2] to handle ontology reading and representation.

GoogleAPI We have used the GoogleAPI [3] to access the Google[4] search engine for the purpose of gathering URLs for a document collection.

[1]Apache SoftWare Foundation http://www.apache.org
[2]http://sourceforge.net/projects/owlapi/
[3]http://code.google.com/apis.html
[4]http://www.google.com

PDFBox PDFBox [5] has been used to extract text from pdf documents. PDFBox has been used as a commond line tool to extract the text from pdf documents downloaded, and is not used as a part of the implementation.

JericoHTML the JericoHTML API [6] has been used as an aid to parse HTML.

Note that the prototype has been developed on a Windows XP system and has not been tested on other platforms.

6.2 Implementation

This section will describe the details of the implemented prototype. We have chosen to split this section in four subsections, indexing, feature vector construction, query reformulation, and document retrieval.

6.2.1 Indexing

As mentioned in Chapter 5 we use several indexes in the prototype. This section describes only the indexes built upon the document collections used, not the feature vector index and negative feature vector index.

The indexes used to construct the feature vectors and negative feature vectors are based on a set of relevant and non-relevant documents, respectively. We have built one document index, one paragraph index, and one sentence index for both cases. In the document index, the full document is indexed, in the paragraph index each document is split into paragraphs and indexed, and in the sentence index each document is split into sentences and indexed. The text is split using simple regular expressions, and we have chosen to use the same boundaries as in [47]. The paragraph boundary is considered to be two or more consecutive newline sequences, possibly interupted by whitespace. A puctuation mark, ".", "!", or "?" followed by a white space is considered to be a sentence boundary.

The main search index is based on paragraphs and includes both relevant and non-relevant documents. We have in this case used a slightly different approach to defining a paragraph. We still use the same boundary as above, but to get a reasonable amount of context, we have set the minimum paragraph length to 1200 characters. This implies that the text first is split according to the paragraph boundaries, and then starting at the first paragraph, paragraphs smaller that 1200 characters are merged, leaving us with fewer, but larger paragraphs. In the index, each paragraph (main paragraph) is indexed along with its neighbor pargraph above and below (if they exist). The neighboring pargraphs are given boosts as stated in Section 5.3.2. The main paragraph is the paragraph that is returned to the user during search, the neighboring paragraphs are used only to boost the search.

The document search index is based on the same documents as the main search index, but here we use traditional indexing, i.e. the documents are indexed as units.

The preprocessing step of the index creation is done by a custum class, `StemAnalyzer`, which is a implemented as a subclass of the Lucene `Analyzer` class. The tokenization

[5] http://www.pdfbox.org/
[6] http://jerichohtml.sourceforge.net/doc/index.html

is done by the Lucene class `StandardTokenizer`. Next the tokens are converted to lowercase, and accents are removed, followed by the applicatoin of a custom built `NumberFilter`, which removes tokens consisting of purely numbers. The stopwords among the tokens are next removed. The stopwords we have used have been downloaded from the "TDT4215" course homepage [28], and may be found in Table A.1 in Appendix A. The last part of the preprocessing is the stemming of the tokens with the `LightStemFilter` class. We have chosen to only lightly stem the terms, this has been done using the conversion $s \rightarrow \emptyset$, for tokens not ending with *ss*. Both the `NumberFilter` class and the `LightStemFilter` class have been implemented as subclasses of the Lucene `Filter` class. The rest of the techniques explained here use already implemented Lucene `Filter` classes.

Class `Indexer` is responsible for creating the indexes for the construction of the feature vectors and the negative feature vectors (document, paragraph, and sentence indexes).The `IndexSearchDocs` class is responsible for creating the main search index and the document search index. Both these classes use the `StemAnalyzer` class for preprocessing, and the Lucene `IndexWriter` class to build the actual indexes and write them to disk. The `IndexWriter` class uses a single Lucene `Document` instance to represent a document (document, sentence-document, and paragraph-document) in the index. The content of the `Document` instances is represented by Lucene `Field` classes, which may hold a variety of information. We have used one `Field` instance to hold the filename, and one to hold the textual content which is indexed. However, when it comes to the main search index, this is slightly different. We have here used up to 7 `Field` instances. These `Fields` hold the following information:

- The filename
- The main paragraph number
- The main paragraph content
- The number of the paragraph above (if one exists)
- The content of the paragraph above (if one exists)
- The number of the paragraph below (if one exists)
- The content of the paragraph below (if one exists)

In addition we set the boost factor for the paragraphs above and below using the `Field.setBoost(float)` method.

6.2.2 Feature Vector Construction

Feature vectors. The class `TFVector` is the class responsible for setting up the feature vectors and indexing them. The feature vectors are built by retrieving all the relevant documents for each concept, using the concept name as a phrase query into the three indexes by the `Searcher` class. The `Searcher` class uses the Lucene `IndexSearcher` class to retrieve these documents from the Lucene index. We also use the Lucene `IndexReader` class to access the terms and frequencies of the documents. The feature vectors are built in sequence, and the vectors are indexed based on frequency alone. This leaves the actual calculation of the *tficf* score to the query reformulation part of the system. The vector is built as a `HashMap` which contains all the terms and their altered frequencies based on document (1.0), paragraph (10.0) and sentence (100.0) weight. Each vector is indexed as a single Lucene `Document` with one `Field` containing

the concept name, and one `Field` containing the terms. The terms and frequencies are added to the content `Field` using the `MapReader` class, which converts the `HashMap` into a stream. The class `BasicAnalyzer` tokenizes the stream, using only the Lucene `StandardTokenizer` class, as the terms used for the construction of the feature vectors are retrieved from an existing index, and have already been subject to preprocessing. The Lucene `IndexWriter` class is used to write the feature vector index to disk.

Negative feature vectors. The class `AntiVectorCreator` is responsible for creating the negative feature vectors. This is done in two runs. The first run creates what we refer to as "raw" negative vectors, and the second run cleans the vectors and writes the final vectors to disk. The first run builds the "raw" negative vectors in much the same way as the ordinary feature vectors, with the exception that we do not use the whole documents to build these vectors; only the paragraph-documents, and sentence documents. Also, we use the indexes based on the non-relevant documents. One other significant difference is that we use the top 5 terms for each concept feature vector as a weighted query into the indexes. The Lucene `IndexSearcher` class is used to retrieve the relevant documents for each vector, and the vectors are constructed in the same way as the feature vectors, with the exception to the weights used. We have used a term weight of 1.0 for terms found in the same paragraph, and a term weight of 10.0 for terms found in the same sentence, still reflecting the same relation between paragraph terms and sentence terms. These raw vectors are indexed in a temporary index using the Lucene `IndexWriter` class, `Document` class, and `Field` class. The content is also here converted to a stream by `MapReader` and tokenized by `BasicAnalyzer`. We add two `Field`s to each `Document`; one containing the concept name, and one containing the "raw" vector. The second run cleans the negative feature vectors. This is done by first reading the negative feature vector using the Lucene `IndexReader` class, and retrieving the corresponding feature vector. The terms found in the corresponding feature vector are removed, so are the terms found in more than 5% of the feature vectors. The cleaned negative feature vector is finally written to disk using the same formalism as above. Recall from Chapter 5 that only the top 15 terms (if there are that many) for each vector are indexed without frequencies. The frequencies are not indexed , as they do not play any role in the query reformulation strategy.

6.2.3 Query Reformulation

The four query reformulation strategies have been implemented in four classes;

- *Simple reformulation* in `SimpleWeightReformulator`
- *Best match reformulation* in `MultiTermReformulator`
- *Ontology structure reformulation* in `OntologyStructureWeightReformulator`
- *Cosine similarity reformulation* in `CosineReformulator`

All these classes are subclasses of the abstract class `QueryReformulator` which contains some basic shared code and defines the interface. All of the classes reform the query based on the approach described in Chapter 5. Each term is used as a query into the feature vector index by the Lucene `IndexSearcher` class, which retrieves the related feature vectors. The terms and their frequencies are retrieved using the Lucene `IndexReader` class. The *tficf* score is calculated for each of the terms, and the query is reformulated according to strategy. In addition the negative feature vector (if one exists) is retrieved for each of the concepts chosen for expansion in the final query. Finally the query is collapsed; terms appearing more than once in the expanded query

are collapsed in the final query vector, by adding the weight of each terms occurence. The boost for the user entered query terms are finally added to the query.

We will now use pseudo code to show how the implementation handles the actual query interpretation and expansion. Listing 6.1 shows pseudo code for the *simple* reformulation strategy. Q represents the user query and q_i represents term i in the user query. The variable fv holds the best feature vector, and the variable nfv holds the corresponding negative feature vector. The for loop loops over all the query terms, retrieving the best feature vector based on the $tficf$ score, and its corresponding negative feature vector (if it exists). The 15 highest weighted terms in the feature vector are used to expand the query. The top 15 negative terms are added with a negation, i.e. *-term*, to the query.

Listing 6.1: Pseudo code for the *simple* reformulation strategy

```
for (each qi in Q){
    retrieve all feature vectors containing qi;
    find the fv with highest tficf score for qi;
    expand the query with the top 15 terms in the best fv;
    retrieve the negative fv corresponding to fv, nfv;
    expand the query with the negated top 15 terms in nfv;
}
```

The reformulation of the user query by the *best match* reformulation strategy is shown in Listing 6.2. Q represents the user query, and q_i is used to refer to term i in the query. fv is used to represent a feature vector, and $tficf_i$ is used to refer to the tficf score for term i in the feature vector currently operated on. Finally nfv is used to refer to the negative feature vector corresponding the the concept of the fv chosen for expansion.

Listing 6.2: Pseudo code for the *best match* reformulation strategy

```
retrieve all fvs containing all terms in Q;
for (each fv retrieved){
    score = 0;
    for (each qi in Q){
        score += tficfi in fv;
    }
}
choose the best fv based on score;
expand the query with the top 15 terms in fv;
retrieve the negative fv corresponding to fv, nfv;
expand the query with the negated top 15 terms in nfv;
```

Listing 6.3 shows the pseudo code for the *ontology structure* based reformulation strategy. As before, q_i represents query term i in the user query Q. fv[i] is an array holding the top 15 feature vectors for term i. The variable fv_l is used to designate the feature vector with rank l in the corresponding feature vector array. $tficf_{n,k}$ is used to refer to the $tficf$ score for query term n in feature vector k. The function $pathLength(i, j)$ finds the length of the path in the ontology between the concepts of feature vector i and feature vector j,with a maximum depth of 5. We use nfv_k to refer to the negative feature vector corresponding to feature vector k.

Listing 6.3: Pseudo code for the *ontology structure* reformulation strategy

```
for (each q_i in Q){
    fv[i] = retrieve the top 15 fvs based on tficf score for q_i;
}
for (each fv_i in fv[0]){
    for (each fv_j in fv[1]){
        score_{i,j} = tficf_{0,i} · tficf_{1,j} · 1/pathLength(fv_i,fv_j);
    }
}
choose the pair of concepts (fv_i,fv_j) with the highest score;
expand the query with the top 15 terms of fv_i;
expand the query with the top 15 terms of fv_j;
retrieve the negative fv corresponding to fv_i, nfv_i;
retrieve the negative fv corresponding to fv_j, nfv_j;
expand the query with the top 15 negated terms of nfv_i;
expand the query with the top 15 negated terms of nfv_j;
```

Next, Listing 6.4 shows the pseudo code for the *cosine similarity* reformulation strategy. Q represents the user query, while q_i represents term i in the user query. We use fv[i] as a variable to hold the top 15 feature vectors for term i. The variable fv_l holds the feature vector with rank l in the corresponding feature vector array. $tficf_{n,k}$ is the $tficf$ score for query term n in feature vector k. Finally, the function $cosineSimilarity(i, j)$ computes the cosine similarity between feature vectors i and j. The variable nfv_k is used to refer to the negative feature vector corresponding to feature vector k.

Listing 6.4: Pseudo code for the *cosine similarity* reformulation strategy

```
for (each q_i in Q){
    fv[i] = retrieve the top 15 fvs based on tficf score for q_i;
}
for (each fv_i in fv[0]){
    for (each fv_j in fv[1]){
        score_{i,j} = tficf_{0,i} · tficf_{1,j} · cosineSimilarity(fv_i,fv_j);
    }
}
choose the pair of concepts (fv_i,fv_j) with the highest score;
expand the query with the top 15 terms of fv_i;
expand the query with the top 15 terms of fv_j;
retrieve the negative fv corresponding to fv_i, nfv_i;
retrieve the negative fv corresponding to fv_j, nfv_j;
expand the query with the top 15 negated terms of nfv_i;
expand the query with the top 15 negated terms of nfv_j;
```

Note that in Listings 6.3 and 6.4 the indexes in the arrays are set to 0 and 1. This is due to the fact that these two reformulation strategies, ontology structure and cosine similarity, have only been implemented for the case of two query terms.

After the query has been expanded by the chosen approach, the query is collapsed according to the approach given in the top of this section.

6.2.4 Document Retrieval

The class `ParIndexSearcher` is responsible for retrieving the result for the user query. For each search request three searches are performed. The first search is based on our reformulation strategies, and uses the relevant reformulator class to reform the query. The reformed query is then fired as a weighted keyword query into the main search index using Lucenes `IndexSearcher` class. The ranked hits are returned as Lucene `Hits` objects. The second search uses the user entered query as a standard keyword search into the main search index. The third search uses the user entered query as a stadard keyword search into the document search index. Both of the last queries use the `IndexSearcher` class for searching and the `Hits` class to represent a ranked list of hits.

We have chosen to implement a very simple web server to present the search interface and search results to the user. The web server is, as said, very simple, and handles simple HTTP GET requests following a certain syntax for the requested object. The server is implemented based on the Java classes `ServerSocket` and `Socket`. The `ServerSocket` binds to a port and waits for incoming connections. When connections arrive, a new thread (`ConnectionHandler`) is started to handle the request. The `ConnectionHandler` is used to present the user with the search interface, the search results (parsing the `Hits` into HTML), or a chosen file/paragraph from a link. The results are presented to the user in tabular form; the first column shows the results for the reformulated query, the second column shows the results for the paragraph based keyword search, and finally the third column shows the results of the document based keyword search. For each result, the first 200 characters of the paragraph, or document in case of the document search, is displayed, along with a link to the paragraph content and the document content, and the score of the hit. Figure 6.1 shows a screenshot of the implemented web user interface.

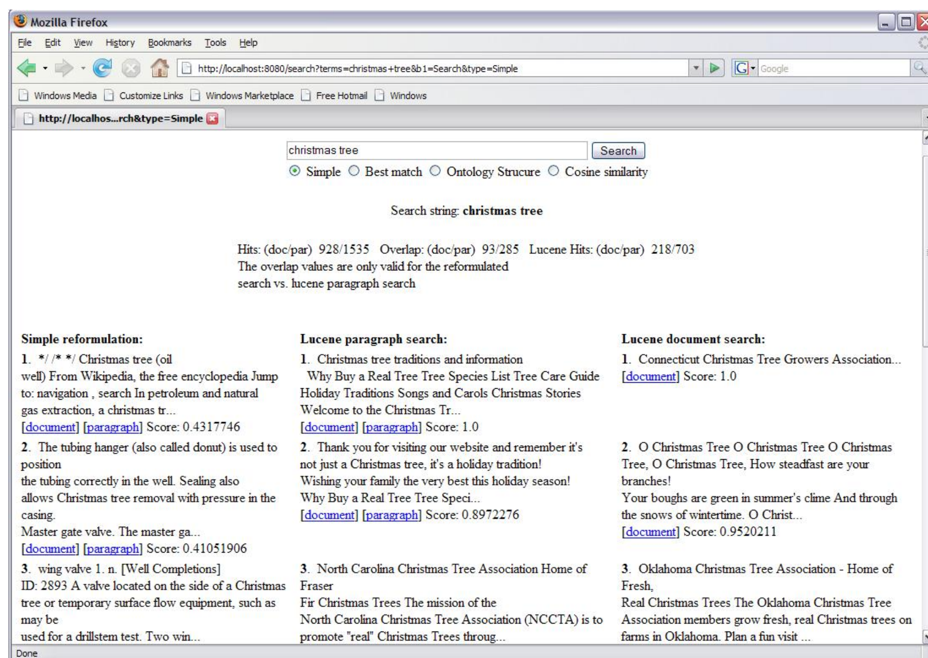

Figure 6.1: Screenshot of the web user interface

6.3 Class diagrams

We will now give a short description of the structure of the implemented prototype. To illustrate this we have used one overall package diagram and class diagrams for the most important packages.

Figure 6.2 shows the overall structure of the packages of the implemented prototype. The packages `index`, `featurevector`, `query`, and `web` will be discussed in closer detail in the next subsections.

The package `googleSearch` includes two classes; `DocCollection` which is responsible for retrieving search results from the Google search engine through the GoogleAPI, and `Downloader` which is reponsible for downloading the hits from URLs.

The package `utils` contains 5 classes. The class `Config` is a class containing static variables that hold paths, and other constants used by the prototype. The class `DocRepresentation` contains static methods to read text files, and convert them to arrays of sentences or paragraphs. The class `MFileNameFilter` implements the java interface `FileNameFilter` and is used to filter files based on extension. `OntologyReader` is used to read OWL ontologies. Finally the class `StripHTML` is used to strip HTML from the downloaded documents and uses the JericoHTML API.

The package `ontology` contains a single class, `OntologyRepresentation`, which builds a graph representation of the ontology used to search for the shortes path between two concepts. The class is used by the `OntologyStructureWeightReformulator` class in package `query`.

Package `run` contains five classes which are responsible for executing the different parts of the application. `RunBuildFVBasisIndexes` is responsible for building the indexes used as basis for the feature vector and negative feature vector construction. `RunBuildSearchIndexes` builds the main search index, based on paragraphs, and the document search index. The class `RunFVBuilder` is responsible for the construction of the actual feature vectors and negative feature vectors. The class `RunGoogleInterface` collects document URLs from the Google search engine and subsequently downloads them, while the class `RunServer` starts the server. All of these five classes simply consist of one main method.

6.3.1 Package index

This package mainly contains code that is associated with indexing and analyzing (preprocessing) of text. The class `FileLister` is reponsible for listing files in a specific directory. `ParIndexSearcher` is the class that is mainly responsible for handling reformulation, using the query reformulation classes in package `query`, and user search into the indexes. The classes contained in the `index` package are shown in Figure 6.3. The purpose of the remaining classes not discussed here have been discussed in Section 6.2.

6.3.2 Package featurevector

The `featurevector` package contains code that is used to construct and index the feature vectors and negative feature vectors. The class `TFVector` is a subclass of `FVCreator` and is the class responsible for creating and indexing the feature vectors. `Searcher`

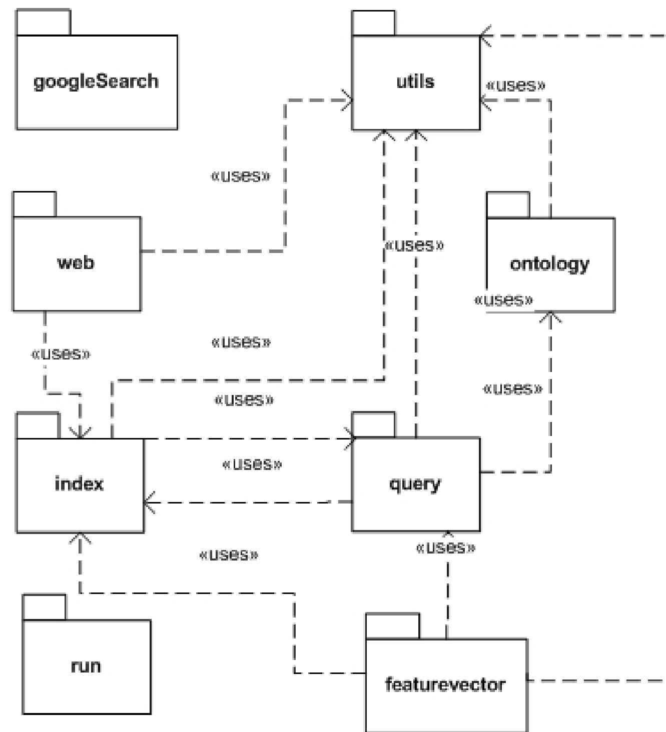

Figure 6.2: Package view of the implemented prototype

is used by `TFVector` to retrieve the relevant documents, paragraph documents, and sentence documents for each concept. The negative feature vectors are constructed and indexed by the class `AntiVectorCreator`. Both `TFVector` and `AntiVectorCreator` use the class `MapReader` to convert the internal `HashMap` representation of the vetors to a stream, making them suitable for indexing by Lucene. Figure 6.4 shows the internals of package `featurevector`.

6.3.3 Package query

The `query` package contains the main reformulation code. Each of the four strategies have been implemented in a separate class, which is a subclass of the abstract `QueryReformulator` class. The `QueryReformulator` class contains code that retrieves

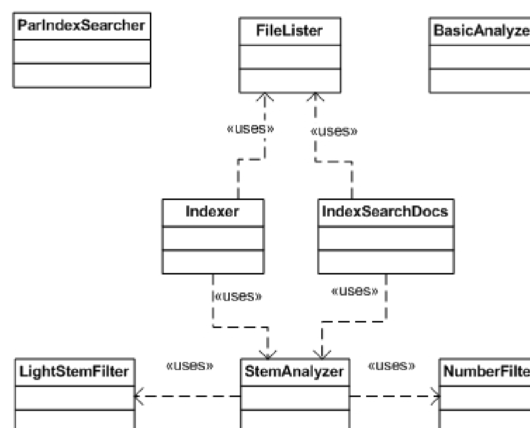

Figure 6.3: Class diagram of the `index` package

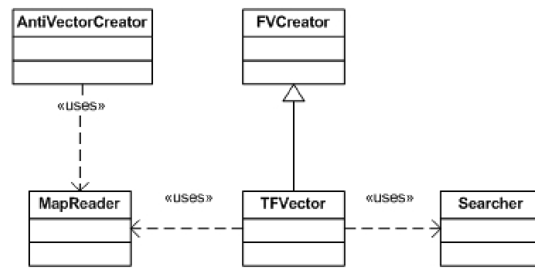

Figure 6.4: Class diagram of the `featurevector` package

a single feature vector, which is represented by the `FeatureVector` class. The class `FeatureVectorResult` is used to hold intermediate results used by the query reformulation classes. The class `FVSearcher` is used to obtain ranked lists of feature vectors based on the query terms entered by the user. The *simple* reformulation strategy is implemented in `SimpleWeightReformulator`, the *best match* reformulation strategy is implemented in `MultiTermReformulator`, while the *ontology structure based* reformulation strategy is implemented in `OntologyStructureWeightReformulator`. Finally, the *cosine similarity based* reformulation strategy is implemented in the class `CosineReformulator`. Figure 6.5 shows the classes of the `query` package.

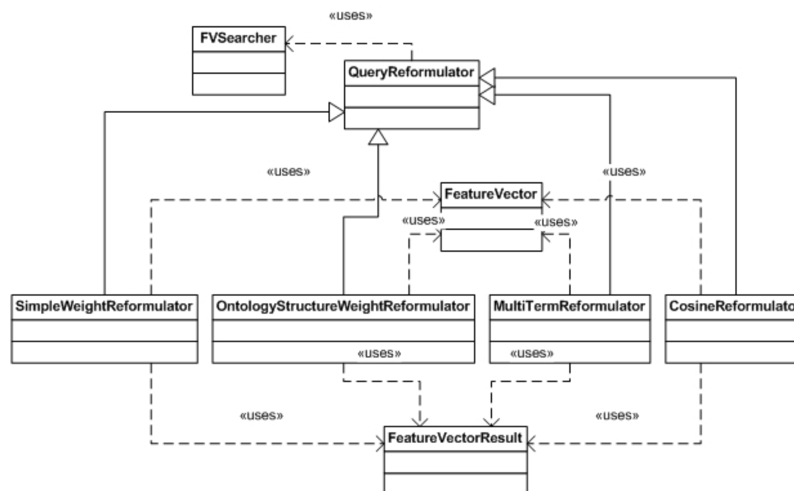

Figure 6.5: Class diagram of the `query` package

6.3.4 Package web

The package `web` contains the code that sets up a very simple web server and handles the arriving requests. The main class is the `Server` class, which sets up a `ServerSocket` that waits for incoming connections. For each new connection a new `ConnectionHandler` thread is started, which handles the request from the user browser. The `ConnectionHandler` class uses the class `index.ParIndexSearcher` to retrieve the search results. Finally the class `Request` parses the user request, retrieving search method, query terms, files/paragraphs requested etc. Figure 6.6 shows the classes of the `web` package.

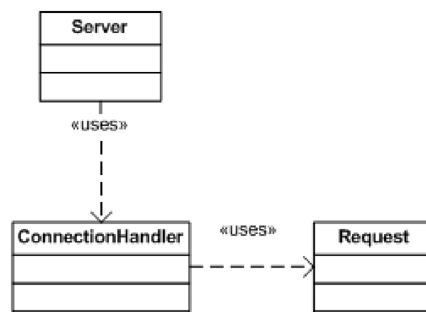

Figure 6.6: Class diagram of the `web` package

Part IV

Evaluation

Chapter 7

Evaluation

This chapter presents the evaluation of the prototype. The first section will give an overview of the evaluation data used, the next section will describe the evaluation strategy. This is followed by the evaluation results, and finally we give a summary of the observations made.

7.1 Evaluation Data

The domain of the evaluation data is subsea petroleum equipment and installations. The ontology we are using in the prototype evaluation is the IIP [1] core ontology containing 18,675 concepts. Many of the concepts were not recognized during the construction of the feature vectors, leaving us with 2195 concept feature vectors. However, this was expected, and as we pointed out in [47], resons for missing concepts when building feature vectors may be caused by the concept name not being mentioned in the text because it may be a phrase not reffered to in "daily" speak, or that the concept phrase is broken up in the text. The 2195 concept feature vectors were used as a basis to build the negative feature vectors, and the prototype built 2006 negative feature vectors.

The document collection we have used to build the feature vectors is the Schlumberger Oilfield Glossary [2]. The glossary is provided by S.L. Tomassen, and is in the form of small text files containing definitions of terms in the domain. The files contain no tags, so the files are used as they are. These files vary in size from a few bytes to approximaltely 4 KB of text, totalling approximately 2,2 MB of text spread over 4132 files.

For the search index we downloaded some documents designated as both relevant and non-relevant. These documents were found using the Google search engine, more specifically using the Google Search API [3]. The URLs of the top 250 results were stored to file and subsequently downloaded. The relevant documents were retrieved using the query *christmas tree wellhead petroleum subsea* and the non-relevant documents were downloaded using the phrase query "*christmas tree*". Due to sites not responding, .xls files etc. the number of relevant documents used was 130, totalling approximately 4 MB of text after stripping HTML and extracting text from pdf files. The number of

[1] IIP, the Integrated Information Platform for reservoir and subsea production systems project. A research project funded by the Norwegian Research Council, project nr 163457/S30.

[2] http://www.glossary.oilfield.slb.com/

[3] http://code.google.com/apis.html

non-relevant documents used is 99, totaling approximately 0.5 MB of text. We also used the Schlumberger Oilfield Glossary in the search index.

The negative feature vectors were built using the feature vectors and 82 files from the non-relevant document collection. The documents used to build the negative feature vectors are not among the documents included in the search index.

7.2 Evaluation Strategy

The scope of this evaluation is concerned with two aspects of the implemented prototype. The first is how relevant the hits returned to the user are, and the second is how well the hits are ranked. The base-line used as a comparison for our prototype is straight forward keyword search implemented with Lucene. We have chosen to use two implementations of Lucene, one in which the documents are indexed in a standard document index, and one in which the documents are indexed by paragraph with a boost factor for the neighboring paragraphs. The paragraph based index is the same as the one described in Section 5.3.2 used by our reformulated queries. Note that system performance in terms of speed is not within the scope of this project, and so will not be subject to evaluation.

Evaluating the implementation using standard precision and recall measures is not viable for this project. This is mainly due to the fact that we do not have an overview of which documents in the document collection searched should be deemed as relevant for each of the queries. The approach chosen for the evaluation of the ranking of the hits is to specify several queries, and have the users evaluate the top 10 hits for each query and search strategy. We have defined four groups of queries, to distinguish between different types of search. The four groups and their charactaristics are listed below.

- **Single concept**. The query terms together identify a single concept in the ontology.
- **Two conepts**. Each single term in the query identifies a singel concept in the ontology.
- **Implisit concept**. The query terms are closely related to one of the existing concepts in the ontology.
- **Concept + keyword search**. The query generates a concept and uses the remaining terms, which are not associated with any concept, for keyword search.

For each of the first three query groups, we have defined two queries and for the last group we have defined one query which you can find in Table 7.1. The queries will be executed with all four of our reformulation strategies and the two lucene approaches, with exception to the last reformulation strategy. The last query (query 7 in Table 7.1) belongs to the fourth query group, and only one of the terms will have a feature vector associated with it, resulting in all four of our reformulation strategies constructing the same expanded query. Thus for the last query the test subjects are asked to evaluate the simple reformulation approach together with the two Lucene keyword approaches. The total number of test cases is 39, requiring the users to evaluate 390 documents (although some of them may be duplicated from one search strategy to another).

The test subjects were presented with a short description of the search domain, and a short description of the scale used to score the documents. The scale used is a simple 0-2 point scale. A score of 0 designates the document as being irrelevant, a score of 1

Table 7.1: Proposed queries used in the evaluation.

Query nr.	Group	Query
1.	1	christmas tree
2.	1	valve control
3.	2	tree valve
4.	2	tree pipe
5.	3	wellbore casing
6.	3	shelf seismic
7.	4	tree ekofisk

designates the document as being related to the search domain, and lastly a score of 2 designates the document as being relevant with respect to the query.

For each query we will evaluate the top 3 hits and the top 10 hits, as [18] found that users tend to pay most attention to the top ranked results. The choice of evaluating the top 3 and top 10 ranked hits is thus a way of reflecting this observation in the evaluation.

Ideally we would have subsea petroleum domain experts evaluate the system, but we do not have the resources available to do such an evaluation. We will therefore use 5 persons (one PhD candidate and 4 Master students) with little domain expertise to evaluate the system. This is not an optimal evaluation approach as deciding if a hit should have score 1 or 2 may be hard. We therefore argue that using 5 test subjects will somewhat even out the uncertainty in this evaluation.

We have also chosen to evaluate the overlap between our four search strategies and the Lucene paragraph based approach. All of these five approaches search the same index, and lets us directly compare the overlap. The overlap is calculated for both paragraph and document overlap. The overlap is used as a measure of how many of the same documents/paragraphs are found by our approaches and the keyword based approach, and will indicate how much of the result set is filtered out in our approaches. This will by mainly due to the negative vectors, as all documents in the result set containing any negative feature vector terms are restricted from appearing in our result sets.

7.3 Evaluation Results

We will in this section present the results from the evaluation of the implementation, both the user tests and the overlap between our approach and the Lucene paragraph based keyword search.

7.3.1 User Tests

Figure 7.1 shows the average score of each of the search strategies calculated over all of the seven queries and all test subjects. We may from the figure note that the performance of our four reformulation strategies perform quite equally over all the query types. We also note that the keyword based search performs, on average, less well than the reformulated queries. However, we see that the keyword search based on the whole documents performs slightly better than the keyword search based on the paragraph indexing. This may suggest that the size of the paragraphs is to small and that to little

context is available for search with the regular keyword based search. This result is worth noting, as the results from the reformulated query are based on the same index as the paragraph based keyword search.

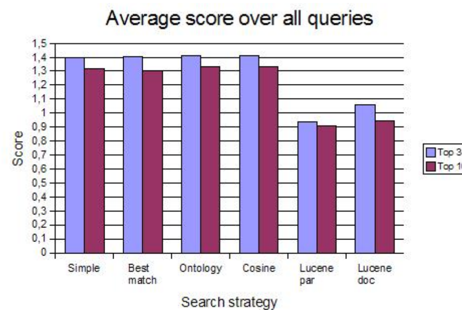

Figure 7.1: Average score for the top 3 and top 10 hits over all the queries for each of the search strategies.

Figures 7.2 through 7.5 show the results for each of the queries found in the four different query groups. The results presented are an average value calculated over all the test subjects.

The results of the two queries from the first query group (single concept) are shown in Figure 7.2. From the figure we see that query 1 shows significantly better results than the two keyword based approaches. The result indicates the value of using domain specific terms and the filtering the negative feature vector provides for the search. The results for query 2 shows that the reformulation approaches in general actually perform slightly worse than the best keyword based approache. We also note that the ontology structure based reformulation performs best, being slightly outperformed by the keyword search based on the full documents.

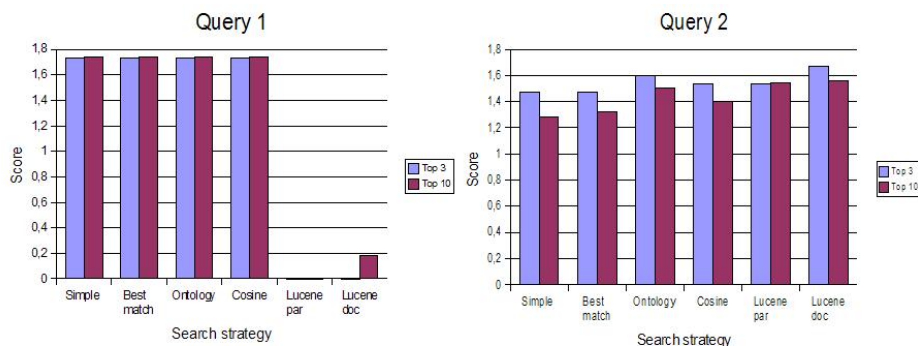

Figure 7.2: Average score for the top 3 and top 10 hits for each of the search strategies for queries 1 and 2.

Figure 7.3 presents the results from query group two (two concepts), specifically queries 3, and 4. We see that our four approaches perform quite equally for query 3, and that they score on an average less points than the keyword based search. The difference between the keyword based approaches and our reformulated query approaches is however not so large. Turning to the results for query 4 we see some small differences in performance between our four approaches. For the top 3 ranked hits, the simple reformulation strategy scores highest, closely followed by the best match, cosine, and lucene document search approaches. We can also note from the figure that the top 3 ranked hits for four of the six approaches score lower than the top 10 ranked hit average. This

may suggest that the ranking of the documents is not optimal. Although the top 3 score for the Lucene document search is close to the top score of the simple reformulation strategy, we see that compared to the top 10 score, and the Lucene paragraph based search our reformulated search strategies perform significantly better.

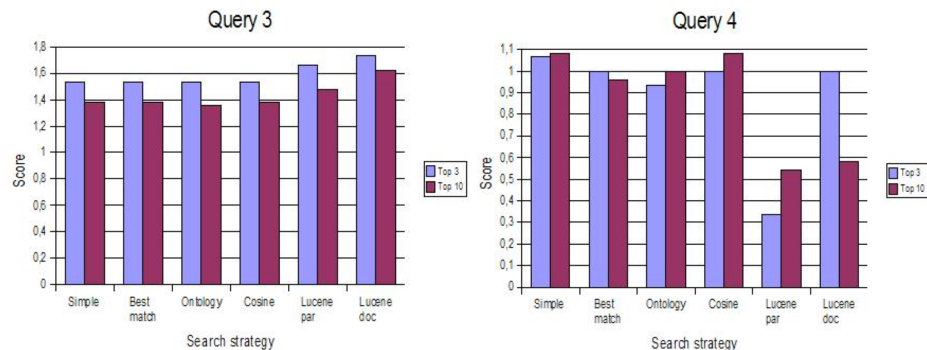

Figure 7.3: Average score for the top 3 and top 10 hits for each of the search strategies for queries 3 and 4.

The results for the queries in query group 3 (implisit concept) are shown in Figure 7.4. From the figure we note that for both queries 5 and 6, our reformulation strategies perform better than the purely keyword based approaches. However, the difference is not significant. We also note that the average score is less than 0,9 even for the best search strategy in query 6, indicating few relevant documents in the result set.

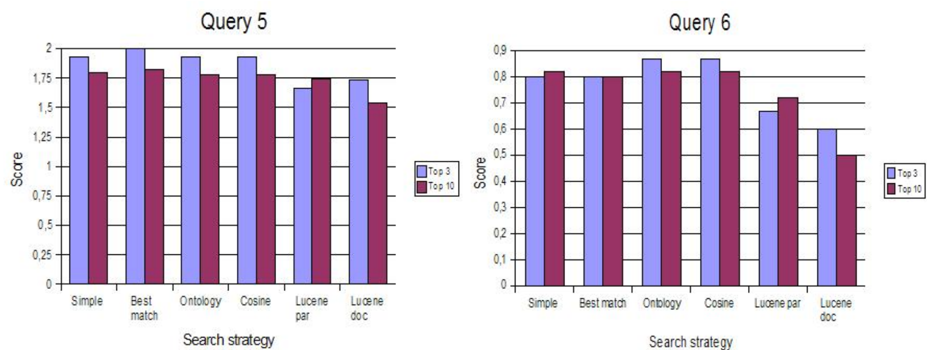

Figure 7.4: Average score for the top 3 and top 10 hits for each of the search strategies for queries 5 and 6.

The results of the one query in query group four (concept + keyword search) is shown as query 7 in Figure 7.5. Recall that the reformulated approaches all return a single (the same) feature vector for expansion for all four reformulation approaches, and they are all collapsed into one search strategy. In the case of query 7 we see that it is clearly the case that the reformulated query performs better than the pure keyword based approaches.

We see that out of the seven queries, one of our reformulation strategies produced the best overall result in 5. In the last two (queries 2, and 3) the Lucene keyword based document search was the best. However, if we compare only our approaches and the Lucene paragraph based keyword search, we see that in only query 3 does the keyword search perform better than our reformulated search strategies. This would probably be the most correct observation as these searches are directly comparable, searching the same index. These observations seem to point in the direction that the added context in

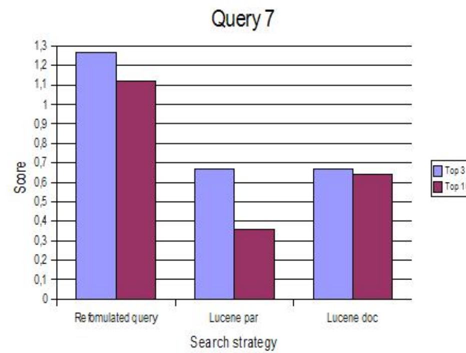

Figure 7.5: Average score for the top 3 and top 10 hits for each of the search strategies for query 7.

the conceptual query expansion does have a positive impact on the information retrieval performance. However, the statistical basis is to small to give any conclusive evidence.

7.3.2 Overlap

This section will present the results concerning the overlap between our four approaches and the paragraph based keyword search. Figures 7.6 through 7.9 show the results, and also include the total number of hits for each search strategy. We can already note by a quick glance at the figures that the reformulated queries tend to produce a significantly larger result set than the keyword based approach. This is not unexpected as adding more terms to a query increases the number of documents which may match parts of the query.

Figure 7.6 shows the results for queries 1 and 2 in query group one. We note that for query 1 the overlap is quite low, and the ratio between overlapping paragraphs and documents is for each method significantly higher than the ratio between total paragraph and document hits. This tells us that our approaches may be able to find more short documents, containing less context. For the second query we may especially point out that the ontology structure based method has a very large overlap, both with respect to document and paragraph. In addition we observe that the number of hits for the ontology structure based reformulation generates significantly fewer hits than any other reformulation strategy.

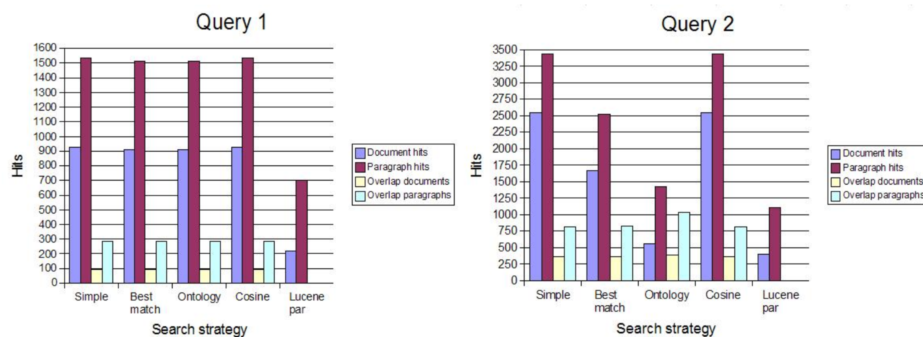

Figure 7.6: Hits and overlap for queries 1 and 2

In Figure 7.7 we find the results for the overlap for queries 3 and 4 in query group two. For query 3 we see that the total number of hits is quite consistent over all four

of our search strategies, and that the overlap also is quite stable over the four search strategies. In query 4 we see that the overlap is significantly higher for the *best match* strategy with respect to the other reformulation strategies.

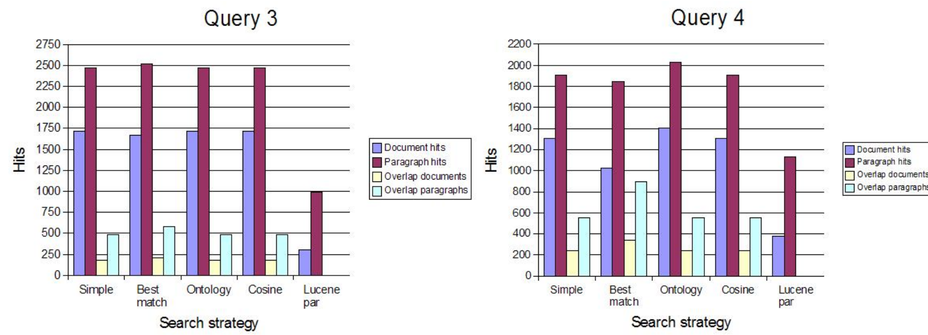

Figure 7.7: Hits and overlap for queries 3 and 4

Figure 7.8 shows the overlap results for queries 5 and 6 in query group three. We see that all the four reformulation strategies have a large overlap for query 5, this means that most of the documents and paragraphs found by our reformulated search strategies are also found by the keyword based approach. This indicates that few results have been filtered. The same is the case for the for query 6 where almost all results, both documents and paragraphs found by our strategies are also found by the keyword approach.

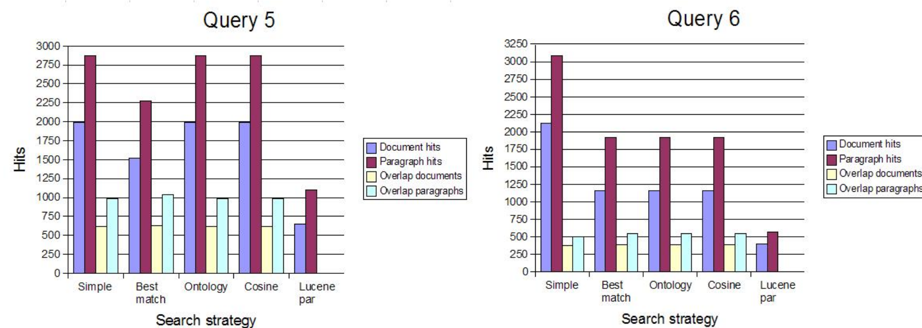

Figure 7.8: Hits and overlap for queries 5 and 6

The results for query 7 in query group four is given in Figure 7.9. Once again, recall that queries in this group containing only two terms will return the same result, independant of the reformulation strategy used. We have therefore collapsed the overlap for our reformulation strategies into one result. We see from the figure that the overlap is not so large, and that many of the documents found by the keyword approach have been filtered out in our reformulated search approach.

Tables 7.2 and 7.3 show the overlap of our approaches and the Lucene paragraph based search calculated over all 7 queries. Looking at a specific row, we read out the percentage of overlap between the approach with the row label and the column lable. For instance, looking at the row labeled Simple in Table 7.2 we see that 72,74% of the documents found by the simple reformulation strategy were also found by the Best match strategy (second column). Viewing the results from the Best match point of view (second row), we see that 95,03% of the documents found by the best match strategy were also found by the Simple reformulation strategy (first column).

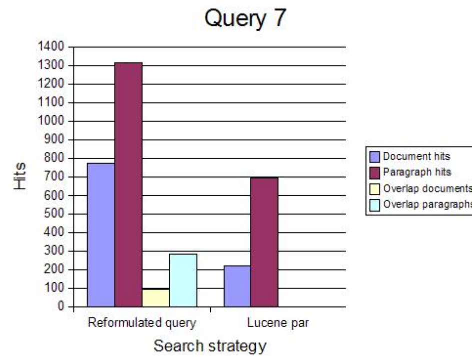

Figure 7.9: Hits and overlap for query 7

Table 7.2: Total overlap for the documents retrieved.

	Simple	Best match	Ontology	Cosine	Lucene par
Simple	100,00%	72,74%	70,33%	89,08%	17,25%
Best match	95,03%	100,00%	84,29%	98,21%	24,35%
Ontology	94,10%	86,32%	100,00%	97,36%	23,56%
Cosine	97,34%	82,14%	79,51%	100,00%	18,97%
Lucene par	76,30%	82,44%	77,89%	76,81%	100,00%

From Tables 7.2 and 7.3 we see that the overlap is quite large between our four reformulation strategies. The first row in the tables shows the percentage of documents found by the Simple reformulation strategy that also were found by other strategies. We note that the overlap, especially for the Simple versus Best match and Ontology reformulation strategies is lower that for the other approaches. Looking at Table 7.4 which shows the total number of hits produced, we see that the Simple reformulation strategy has generated the largest result set. This is not unexpected, since it will in every query expand every term in the original query by one feaure vector. Best match on the other hand, reduces the query to a single feature vector, and produces a smaller result set due to the smaller query used. The Ontology structure and Cosine similarity approaches both may in one extreme expand the query by only one feature vector, if the best pair of feature vectors to expand by actually are the same feature vectors. They may also expand the query by one feature vector for each term in the original query, if the best pair chosen for expansion are not equal. The high degree of overlap we have seen from Tables 7.2 and 7.3 indicates that the reformulation approaches seem to generate relatively equal result sets, which may explain why our approaches perform so equally when looking at Figure 7.1.

We may also note that Table 7.4 shows that our reformulation strategies tend to find fewer paragraphs per document found. As the index contains many short documents from the Schlumberger Oilfield glossary, this may indicate that the reformulation strategies find more short documents, containing less context. This is also not unexpected, since the queries of the reformulated searches firstly contain a lot more terms than the pure keyword search, and secondly that the added context allows the search engine to retrieve more short documents that contain less context.

Table 7.3: Total overlap for the paragraphs retrieved.

	Simple	Best match	Ontology	Cosine	Lucene par
Simple	100,00%	76,69%	75,25%	90,15%	23,53%
Best match	91,64%	100,00%	84,32%	95,07%	32,10%
Ontology	92,31%	86,56%	100,00%	95,84%	30,80%
Cosine	96,91%	85,53%	83,98%	100,00%	25,55%
Lucene par	56,48%	64,48%	60,26%	57,06%	100,00%

Table 7.4: Total number of hits for each strategy.

	Simple	Best match	Ontology	Cosine	Lucene par
Total paragraph hits:	16616	13905	13545	15457	6923
Total document hits:	11385	8714	8509	10419	2574
Avgerage paragraphs per document:	1.46	1.60	1.59	1.48	2.69

7.4 Evaluation Summary

This evaluation does not claim to be exhaustive by any means, but we have made some observations regarding the performance of our reformulation strategies versus standard keyword search. This section will give a short summary of our obervations.

- The evaluation has shown us that on average our approach is somewhat better than the purely keyword based search, both with respect to full document search, and paragraph search. We note that for the first query, all of our reformulation strategies perform superior to the keyword based search. Many of the non-relevant documents contain many instances of *christmas tree* in the traditional sence, making these documents appear in the keyword based search. These documents have been filtered out by our approach, both thanks to the added context used in the expanded query, and the negative feature vectors filtering out documents containing typical *christmas holiday* type terms.

- There is no clear answer to which reformulation approach performs best, both overall and with respect to specific query types. For two of the query groups (group one and group two) the results show that our approaches perform best in one query (queries 1 and 4) and that the keyword based approach is better for the other query (queries 2 and 3). For the two other query groups our approach seem to perform better than the keyword based approach (queries 5, 6, and 7).

- It is interesting to note that the queries in which our approaches are "beat" by standard keyword search, the difference in performance is not that significant. In several of the queries (queries 1,4,and 7) in which our approaches perform better than the keyword based approach, we find that the difference is a lot larger, at least for the difference between our best approach and the keyword based paragraph search. Note that these are directly comparable as they search the same index structure.

- When comparing our approaches with the pure keyword search based on paragraphs, we find that out of the seven queries, one of our approaches seem to perform best in six of them. This is an interesting result, that points out that the added context of the expanded queries are able to correctly recognize more paragraphs than using the pure keyword based search.

- We have also shown that the overlap between our queries and the keyword based paragraph search, varies quite a bit. We may interpret this result as queries having a large degree of overlap are not as ambigous as the queries having a small degree of overlap. Again the filtering of documents is responsible for removing hits which are to some degree ambigous (i.e. *christmas tree* in the petroleum domain versus the holiday sence).
- We have observed a large overlap internally between our four reformulation strategies, indicating that they produce very similar result sets. This may substantiate why our reformulation strategies have a quite equal performance overall.
- Lastly we note that the reformulated search strategies tend to produce fewer paragraphs per document retrieved. Seen in context with the document collection used, in which a large portion of the documents are short, this indicates that the added context in the query is able to recognize a larger portion of short documents as being relevant or semi-relevant than the pure keyword based approach.

Chapter 8

Discussion

This chapter will give a discussion of our findings from the evaluation, followed by a section discussing improvements and further work.

8.1 Test Results

As pointed out in the Evaluation chapter, the evaluation has not been exhaustive. We also pointed out that ideally we would have domain experts evaluate the implementation, but we do not have the time and resources to employ such an evaluation. Also the statistical basis for the evaluation (five test subjects, and seven queries) is not deemed sufficient to give any conclusive evidence. We have however made some observations which will now be discussed.

In the evaluation we noted that the search context for the paragraph based search possibly was to small. This may be the case for the plain keyword based search. We argue that the reformulated queries contain a richer context, enabling these queries to correctly identify more paragraphs. However, it is clear from the differences between Lucene paragraph search and Lucene document search that the aspect of the paragraph size is one that should be researched further. It was not possible for this project to evaluate the reformulated queries based on both paragraph search and full document search as it would be to demanding for the test subjects. The test subjects reported using between two and three hours, and adding document search to the evaluation would amount to almost a full days work.

The tests show that on average our reformulation strategies seem to perform quite eqaully, and better than the purely keyword based approaches. Especially when comparing our reformulated searches to the keyword search based on paragraphs this trend is obvious. As we argued above the added context in the reformulated queries is an important part of this result. In addition we showed that the overlap internally among our reformulation strategies was high. This may explain why the reformulated queries performed surprisingly equally.

The reformulation strategies are all based heavily on the $tficf$ score for the query terms. It may be the case that we have not given optimal parameter settings for the various strategies, leading to equal concept expansions. Thus it may be the case that the $tficf$ score is dominating in the concept mapping phase for all reformulation strategies, leading to quite similar concept expansions. This is a matter that should be further explored.

When it comes to which reformulation strategy works best overall, and in specific query types, we have no conclusive evidence. We may note that our approach seems to perform better for queries handling queries in group three (implisit concept), and group four (concept + keyword search). However, by viewing Figure 7.4 we see that the best performing approach for query 6 in query group tree has a score of approximately 0.86 for the top 3 hits, indicating few relevant document in the result set.

The implemented prototype uses OR matching, returning documents even if they match only a single term. This may be viewed as a week spot in the system as it generates a very large result set. One could imagine using AND queries, forcing all of the terms to be present in the document. This could possibly increase presicion of the search, on the other hand requiring all the terms to be present in the documents returned reduces recall, by filtering out documents that do not contain one or more of the terms. For the number of terms used in expanded queries in the implemented prototype this would probably filter out too many of the relevant documents, at least with the size of the paragraphs used. Balancing this effect out by carefully researching the number of terms to use with an AND query is a matter to research further. One other solution to this problem could be to group terms in the feature vector based on significance, for example requiring the top 3-5 terms to be present in a relevant document using the AND operator, and viewing the rest of the terms used for expansion as support terms using the OR operator.

8.2 Improvements

This section will give a discussion of different improvements to the implementation.

8.2.1 Feature Vectors & Negative Feature Vectors

We have chosen to use the top 15 terms in the feature vectors for the expanded query. We have not done any evaluation on the optimal number of terms to expand. However, the weight for terms further down the feature vector seem to have small numeric values, so the impact of these terms may be so small that we can ignore them. On the other hand, as stated we have not done any formal evaluation of this, and more research is needed to find the optimal number of terms used in the expanded query. This number may depend on the type of query, the size of the document collection used to build the feature vectors, and the quality of the feature vectors.

We have not researched how to handle the negative feature vectors in a query setting. The approach used in the prototype was to add the terms in the negative feature vectors as a NOT query, simply filtering out the documents containing the terms in the negative feature vectors. As stated in Chapter 6 we have removed all terms in the negative feature vectors which also occur in the corresponding feature vector, and all terms that occur in more than 5% of the feature vectors. This approach may leave some significant domain terms in the negative feature vectors, actually working against the objective of the negative feature vectors by filtering out documents that actually may be relevant. We propose a new way of handling the negative feature vectors, an approach which is slightly more sophisticated than the simple NOT expansion. By adding negative weight to the feature vectors, one could imagine documents containing negative feature vector terms would be penalized, reducing the final score in the similarity calculation between document and query. This would remove the problem of strict filtering as in the NOT

approach, and it would be possible for documents containing only very few negative feature vector terms to retain a high score if the overall relevance is high. More research is needed on the area of negative feature vector construction and handling to find a less naive approach to constructing them and using them in the expanded query.

The construction of the negative feature vectors used only the paragraph- and sentence-documents, in contrast to the approach given in Chapter 5. We argue that this has not had a significant impact on the negative feature vectors, as the weighting differenses are so large, leading to the contribution of the document vectors being quite small. We have done small informal tests which conclude that only minor changes in ranking are achievable when using document-, paragraph-, and sentence-documents versus only using paragraph-, and sentence-documents. The important point is that we still retain the largest part of semantic context through the paragraph-, and sentence-documents. However, this assumption is based on informal tests, and is a matter that should be futher explored.

We found a small bug in the implementation during the evaluation; if the ontology structure method finds a single best concept to expand, the feature vector is added to the final query only once. If the cosine similarity method finds a single best concept to expand, the feature vector is added to the final query twice. This leads to a slight change in the ranking of the documents with respect to the two approaches. Using a simple feature vector, $fv = (t_1^x, t_2^y)$ where t_i^k represents term i with weight k, we will explain the difference. The boost, b, for the user entered query term, t_1, is added only once to the query. The final query, q, for the ontology structure based method will be $q = (t_1^{x+b}, t_2^y)$, while the cosine similarity based final query would be $q = (t_1^x, t_2^y, t_1^x, t_2^y) => (t_1^{2x+b}, t_2^{2y})$, making the boost factor less significant in the latter case.

We have used paragraphs strictly bounded by two or more consecutive line breaks, both in the construction of the feature vectors and negative feature vectors, and in the case of the paragraph based search index. Although we have in the paragraph based index added som more context by taking into account the neighboring paragraphs, it would be interesting to see if the search would improve by using semantic paragraphs instead of grammatical paragraphs. A semantic paragraph would be a part of the text that keeps within a speceific topic or context, possibly spanning multiple grammatical paragraphs. Using text mining techniques to recognize topic boundaries in the text could be applied so that the text would be semantically partitioned. [34] has proposed such a partitioning based on, among other techniques, clustering. This could potentially be a large improvement, especially with large texts covering a wide range of topics.

Term Filtering

The feature vectors and negative feature vectors we have built are based upon a document collection which has been preprocessed in a standard way; tokenization, stopword removal, light stemming, etc. Exploiting POS tagged text to only use the most interesting word groups would be a possible way of improving the feature vectors and negative feature vectors, and should be researched further.

We also propose to further improve the system by generating a stop word list of domain specific terms that do not discriminate well among documents. The stop word list would be used during construction of the feature vectors, and would enable us to achieve higher quality feature vectors used in the process of expanding the query. On the other hand, users may want to search for such terms, leading to a problem in how such searches are handled. We propose to add user terms that exist in such a stop word list to the query

as a keyword search, as we did for terms not found in the feature vector index (query group 7 in the evaluation).

The system as it is implemented, builds the feature vectors based on the concept names. However, many of the concept names contain typical stop words, like *on off control* and *on off valve*. We have employed stop word removal during the indexing of the documents used as a basis for the construction of the feature vectors. Stop words have also been removed from the concepts used as a query into the index, removing parts of the concepts name. This may be a drawback, as there almost certainly is a difference between a *valve*, in a general sence, and a *on off valve*, that is a more specific type of valve. We propose to not use stop word removal during the construction of the feature vectors, but rather filter out the stop words after the feature vector has been created. This may lead to better semantic representations of the concepts, reflecting the actual difference between them.

One other question that arises is *do we actually need whole documents, paragraph-documents, and sentence-documents in the feature vector construction?*. The weight given to terms found in whole documents are diminishing with respect to the terms found in paragraphs and sentences. This could possibly lead to adding whole document vectors for a document in which only a small paragraph or section actually is relevant for the concept of the feature vector, thus possibly deteriorating the quality of the feature vector. Further research is needed to find if the whole documents are needed in this construction process.

8.2.2 Performance

The performance of the implementation may be improved dramatically by altering the feature vector index. In the implementated prototype the $tficf$ score is calculated on the fly during the query expansion process, something that is unnecessary as these weights only need to be calculated once. Building a special index to allow the $tficf$ weights to be calculated during the indexing phase would mean that many calculations would be avoided in the query reformulation stage, improving the responce time of the system.

8.2.3 Ontology Reasoning

In the approach we have used an ontology as the basis for construction of feature vectors used to expand the query. The ontology structure based method has also used the hierarchical structure of the ontology to find the best concepts to expand by. Using the ontology to allow more complex resoning during the query expansion process, taking into account other relations than hierarchical, such as properties, would be a natural extension of the approach. The additional reasoning may provide the query expansion process with more relevant connections than the hierarchical structure does.

8.2.4 Challenges

How do we handle that the vocabulary in the domain altering over time? This is an important question since the vocabularity of a given domain may alter by development of new terms, and terms slightly altering meaning in a new business setting. We propose to use document collections from different points in time to represent the vocabulary

at a given point in time, and constructing feature vectors for each point in time. In addition documents may be associated with some time period. Thus during search, the search concepts may map to different vectors in time, more accurately describing the vocabulary in a given time space. The different mappings could then be used to search different parts of the index, releaving the user from remembering or learing the vocabulary of the past.

The functioning of the system today assumes that all the terms in the query are equally important. However, this may not be the case. We propose to let the user weight the terms in the entered query to reflect the related importance among the terms. When expanding the query, these weights would be incorporated into the final weighting of the query, letting the user have some influence on what parts of the query he is most interested in (e.g. a main term and supporting terms). In addition we propose that the user may have an impact on which terms to expand conceptually, by letting the user escape terms he wishes to use as simple keywords in the query.

Today we only include simple terms, and do not add any phrase terms to the feature vectors. This is possibly a weakness of our system, as certain terms may only be relevant when seen in connection with one or more other terms, e.g. a phrase. Adding these terms could possibly add more power to the system. We therefore propose to further research methods for adding such phrases by the means of co-occurence analysis, assosiation rules, or other techniques.

The implementation relies on quite many parameters, which have not been formally evaluated. Among these are:

- The weight of single terms not expanded by feature vectors (query group 7)
- The number of terms in the feature vector to use for expansion
- The weight calculation for the terms in the expanded query
- The boost for query terms in the original query
- The paragraph boost used in the main search index
- How do we handle the negative vectors in a best possible way, and what terms should/should not be included in the negative vectors?
- The assigning of terms to feature vectors
- How to handle the calculation of weights of terms assigned to feature vectors

All of these parameters and others should be researched further to find if there exists any optimal settings that would perform well over a large range of domains and ontologies, or if these parameters must be tailored to each domain and ontology.

Chapter 9

Conclusion

We have in this project explored how a semantic search system based on ontologies may be constructed. The ontology has been semantically enriched by exploiting text mining techniques to construct feature vectors for the concepts in the ontology. A domain relevant document collection has been used in the process of constructing the feature vectors, letting the feature vectors reflect the vocabulary of the domain. We have used the concept feature vectors as a basis, together with the ontology, for the development of a query reformulation module that semantically reformulates the query entered by the user. The expanded query has been used as a weighted keyword query, reflecting the relative importance of each term, into a vector space search engine based on Lucene and paragraph indexing.

We have suggested four different query reformulation strategies, which differ mainly in how the query is interpreted. The four strategies have been evaluated against standard keyword search based on both paragraph and document indexing of the document collection. We found that on average our reformulation strategies seemed to perform better than the keyword based search, especially when our strategies were compared to the keyword search based on paragraph indexing. In six of the seven queries given to the test subjects, one of our reformulation strategies gave the best result. It is worth noting that the average score for all of our reformulation strategies were quite equal, and that the overlap between our four methods was high, possibly explaining the likeness in resuls. In fact, the reformulation strategies performed so similarly, that it is not possible to point out one of the strategies as being better than the others in general. We also found that there was no obvious pattern showing that one of the reformulation strategies distinguished itself with respect to a certain type of query.

Although we did not find any conclusive evidence supporting that one strategy is better than the others, we think that the general idea of expanding the query based on a semantically enriched ontology is promising. This point of view is also supported by our findings. However, a lot of work remains, as there are many variables in play, and researching these will take much effort.

We would finally like to stress the fact that the evaluation has relied on few test subjects, and that a larger evaluation, both including a larger set of queries, and test subjects with domain knowledge, has to be done for more conclusive evidence of the approach.

Bibliography

[1] Webpage. Apache Lucene. http://lucene.apache.org/java/docs/index.html , Accessed 27.05.2007.

[2] Webpage. WordNet. http://wordnet.princeton.edu/, Accessed 27.05.2007.

[3] K. Aas and L. Eikvil. Text categorisation: A survey. Technical report, Norwegian Computing Center, 1999.

[4] S. Abney. Part-of-speech tagging and partial parsing. Corpus-Based Methods in Language and Speech., 1996.

[5] Antoniou, Franconi, and van Harmelen. Introduction to semantic web ontology languages. In Reasoning Web, Proceedings of the Summer School, 2005.

[6] Beaza-Yates and Ribeiro-Neto. Modern Information Retrieval. ACM Press / Addison-Wesley, 1999.

[7] P. Bellot, E. Crestan, M. El-Bèze, L. Gillard, and C. de Loupy. Coupling named entity recognition, vector-space model and knowledge bases for TREC 11 question answering track. In TREC, 2002.

[8] T. Berners-Lee. Semantic web on xml. Keynote presentation for XML 2000. Slides available at: http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide1-0.html , Accessed 27.05.2007
Reporting avialable at: http://www.xml.com/pub/a/2000/12/xml2000/timbl.html , Accessed 27.05.2007.

[9] T. Berners-Lee, O. Lassila, and J. Hendler. The semantic web. Scientific American, 2001.

[10] D. Buscaldi, P. Rosso, and E.S. Arnal. A wordnet-based query expansion method for geographical information retrieval. Working notes for CLEF Workshop.

[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to algorithms. McGraw Hill/MIT Press, 2nd edition, 2001.

[12] S. Deerwester, S.T. Dumais, T.K Landauer, G.W. Furnas, and R.A Harshman. Indexing by latent semantic analysis. Journal of the Society for Information Science, 41(6):391–407, 1990.

[13] T.P. Weide F.A. Grootjen. Conceptual query expansion. Data & Knowledge Engineering, (56):174–193, 2006.

[14] W. Fan, L. Wallace, S. Rich, and Z. Zhang. Tapping into the power of text mining. Communications of the ACM, 49(9):76–82, 2006.

[15] R. Fikes, P. Hayes, and I. Horrocks. Owl-ql: A language for deductive query answering on the semantic web. Technical report, Knowledge Systems Laboratory, Stanford University, Stanford, CA, 2003.

[16] W.B. Frakes and C.J. Fox. Strength and similarity of affix removal stemming algorithms. SIGIR Forum, 37(1):26–30, 2003.

[17] O. Gospodnetic and E. Hatcher. Lucene in Action. Manning Publications, 2005.

[18] L.A. Granka, T. Joachims, and G. Gay. Eye-tracking analysis of user behavior in www search. In SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, pages 478–479, New York, NY, USA, 2004. ACM Press.

[19] Grefenstette and Tapanainen. What is a word? what is a sentence? problems of tokenization. Proceedings of the 3rd International Conference on Computational Lexiography, pages 79–87, 1994.

[20] T. R. Gruber. A translation approach to portable ontologies. Knowledge Acquisition, 5(2):199–220, 1993.

[21] N. Guarino. Formal ontology and information systems. In the Proceedings of Formal Ontology in Information Systems, 1998.

[22] R. Guha, R. McCool, and E. Miller. Semantic search. WWW '03: Proceedings of the 12th international conference on World Wide Web, pages 700–709, 2003.

[23] J.A. Gulla, T. Brasethvik, and H. Kaada. A flexible workbench for document analysis and text mining. NLDB 2004, pages 336–347, 2004.

[24] J.A. Gulla, S.L. Tomassen, and D. Strasunskas. Semantic interoperability in the norwegian petroleum industry. In Dimitris Karagiannis and Heinrich C. Mayer, editors, 5th International Conference on Information Systems Technology and its Applications (ISTA 2006), volume P-84 of Lecture Notes in Informatics (LNI), pages 81–94. Köllen Druck Verlag GmbH, Bonn, Klagenfurt, Austria, 2006.

[25] D. Harman. Journal of the american society for information science, 42(1):7–15, 1991.

[26] M. Hearst. What Is Text Mining? http://www.ischool.berkeley.edu/~hearst/text-mining.html , Accessed 27.05.2007.

[27] J. Heflin and J. Hendler. Searching the web with shoe. Artificial Intelligence for Web Search. Papers from the AAAI Workshop. WS-00-01., pages 35–40, 2000.

[28] Webpage. The homepage of the course "TDT 4215 Knowledge in document collections". English stopword list. http://www.idi.ntnu.no/emner/tdt4215/resources/englishST.txt, Accessed 27.05.2007.

[29] A. Øhrn. Context and semantics in search, presentation at norwegian semantic days, 2007. Presentation available at: http://www.abelia.no/getfile.php/Semantiske%20dager/P1.3%20Alexander%20%D8hrn%20-Context%20and%20semantics.pdf, Accessd 7.06.2007.

[30] K. Knight and J. Graehl. Machine transliteration. Computational Linguistics, 24(4), 1998.

[31] T.K. Landauer, P.W. Foltz, and D. Laham. An introduction to latent semantic analysis. Discourse Processes, 25:259–284, 1998.

[32] S.O. Løkse. Konseptekstraksjon fra store dokumentsamlinger. Master's thesis, NTNU, 2005.

[33] D.L. McGuinness. Ontologies come of age. Spinning the Semantic Web, pages 171–194, 2003. D. Fensel, J. Hendler, H. Lieberm, W. Wahlster (Eds.).

[34] W. Min, L. Zhensheng, and G. Yuqing. Study on semantic paragraph partition in automatic abstracting system. In Systems, Man, and Cybernetics, 2001 IEEE International Conference on, volume 2, pages 892–897. 2001.

[35] E. Mäkelä. Survey of semantic search research. http://www.sange.fi/~humis/sw/semantic_search.pdf, Accessed 27.05.2007.

[36] D.I. Moldovan and R. Mihalcea. Using wordnet and lexical operators to improve internet searches. IEEE Internet Computing, (4):34–43, 2000.

[37] D. Kotzinos N. Athanasis, V. Christophides. Generating on the fly queryes for the semantic web: The ics-forth graphical rql interface (grql). In Proceedings of the Third International Semantic Web Conference, pages 486–501. 2004.

[38] G. Nagypal. Improving information retrieval effectiveness by using domain knowledge stored in ontologies. In OTM Workshops 2005, LNCS 3762, pages 780–789. Springer-Verlag, 2005.

[39] R. Ozcan and Y.A. Aslangdogan. Concept based information access using ontologies and latent semantic analysis. Technical report cse-2004-8, University of Texas at Arlington.

[40] J. Plisson, N. Lavrac, and D. Mladenic. A rule based approach to word lemmatization. Proceedings of the 7th International Multi-Conference Information Society IS 2004, 2004.

[41] M.F. Porter. An algorithm for suffix stripping. Program, 14(3):130–137, 1980.

[42] Y. Qiu and H.P. Frei. Concept based query expansion. In SIGIR '93: Proceedings of the 16th annual ACM SIGIR Conference on Research and Development in Information Retrieval, pages 160–169. ACM Press, Pittsburgh, Pennsylvania, USA, 1993.

[43] Y. Qu, G. Grefenstette, and D.A. Evans. Automatic transliteration for japanese-to-english text retrieval. In SIGIR '03: Proceedings of the 26th annual international ACM SIGIR Conference on Research and Development in Informaion Retrieval, pages 353–360, New York, NY, USA, 2003. ACM Press.

[44] C. Rocha, D. Schwabe, and M. P. de Aragão. A hybrid approach for searching in the semantic web. Proceedings of the 13th international conference on World Wide Web, 2004.

[45] P. Rosso, E. Ferretti, D. Jimenez, and V. Vidal. Text categorization and information retrieval using wordnet senses. GWC, Proceedings, pages 299–304, 2003.

[46] G. Salton and C. Buckley. Term-weighting approaches in automatic retieval. Information Processing & Management, 24(5):513–523, 1988.

[47] G. Solskinnsbakk. Extending Ontologies with Search-Relevant Weights, 2006. Technical report, Norwegian University of Science and Technology, Trondheim, Norway.

[48] X. Su. Semantic Enrichment for Ontology Mapping. PhD thesis, Norwegian University of Science and Technology, 2004.

[49] S.L. Tomassen. Research on ontology-driven information retrieval. In Robert Meersman, Zahir Tari, Pilar Herrero, and et al., editors, OTM Workshops 2006, volume 4278 of LNCS, pages 1786–1795. Springer-Verlag, Montpellier, France, 2006.

[50] S.L. Tomassen, J.A. Gulla, and D. Strasunskas. Document space adapted ontology: Application in query enrichment. In Christian Kop, Günther Fliedl, Heinrich C. Mayer, and Elisabeth Matais, editors, 11th International Conference on Applications of Natural Language to Information Systems (NLDB 2006), volume 3999 of LNCS, pages 46–57. Springer-Verlag, Klagenfurt, Austria, 2006.

[51] F. van Harmelen and D.L. McGuinness (Editors). Webpage. owl web ontology language - overview. http://www.w3.org/TR/owl-features/, Accessed 27.05.2007.

[52] E.M. Voorhees. Query expansion using lexical-semantic relations. In SIGIR '94: Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 61–69. ACM Press, 1994.

Part V

Appendix

Appendix A

Implementation

Table A.1: Stop words used in the implementation[28]

a	a’s	able	about	above
according	accordingly	across	actually	after
afterwards	again	against	ain’t	all
allow	allows	almost	alone	along
already	also	although	always	am
among	amongst	an	and	another
any	anybody	anyhow	anyone	anything
anyway	anyways	anywhere	apart	appear
appreciate	appropriate	are	aren’t	around
as	aside	ask	asking	associated
at	available	away	awfully	b
be	became	because	become	becomes
becoming	been	before	beforehand	behind
being	believe	below	beside	besides
best	better	between	beyond	both
brief	but	by	c	c’mon
c’s	came	can	can’t	cannot
cant	cause	causes	certain	certainly
changes	clearly	co	com	come
comes	concerning	consequently	consider	considering
contain	containing	contains	corresponding	could
couldn’t	course	currently	d	definitely
described	despite	did	didn’t	different
do	does	doesn’t	doing	don’t
done	down	downwards	during	e
each	edu	eg	eight	either
else	elsewhere	enough	entirely	especially
et	etc	even	ever	every
everybody	everyone	everything	everywhere	ex
exactly	example	except	f	far
few	fifth	first	five	followed
following	follows	for	former	formerly

Continued on next page

Table A.1 – continued from previous page

forth	four	from	further	furthermore
g	get	gets	getting	given
gives	go	goes	going	gone
got	gotten	greetings	h	had
hadn't	happens	hardly	has	hasn't
have	haven't	having	he	he's
hello	help	hence	her	here
here's	hereafter	hereby	herein	hereupon
hers	herself	hi	him	himself
his	hither	hopefully	how	howbeit
however	i	i'd	i'll	i'm
i've	ie	if	ignored	immediate
in	inasmuch	inc	indeed	indicate
indicated	indicates	inner	insofar	instead
into	inward	is	isn't	it
it'd	it'll	it's	its	itself
j	just	k	keep	keeps
kept	know	knows	known	l
last	lately	later	latter	latterly
least	less	lest	let	let's
like	liked	likely	little	look
looking	looks	ltd	m	mainly
many	may	maybe	me	mean
meanwhile	merely	might	more	moreover
most	mostly	much	must	my
myself	n	name	namely	nd
near	nearly	necessary	need	needs
neither	never	nevertheless	new	next
nine	no	nobody	non	none
noone	nor	normally	not	nothing
novel	now	nowhere	o	obviously
of	off	often	oh	ok
okay	old	on	once	one
ones	only	onto	or	other
others	otherwise	ought	our	ours
ourselves	out	outside	over	overall
own	p	particular	particularly	per
perhaps	placed	please	plus	possible
presumably	probably	provides	q	que
quite	qv	r	rather	rd
re	really	reasonably	regarding	regardless
regards	relatively	respectively	right	s
said	same	saw	say	saying
says	second	secondly	see	seeing
seem	seemed	seeming	seems	seen
self	selves	sensible	sent	serious
seriously	seven	several	shall	she
should	shouldn't	since	six	so

Continued on next page

Table A.1 – continued from previous page

some	somebody	somehow	someone	something
sometime	sometimes	somewhat	somewhere	soon
sorry	specified	specify	specifying	still
sub	such	sup	sure	t
t's	take	taken	tell	tends
th	than	thank	thanks	thanx
that	that's	thats	the	their
theirs	them	themselves	then	thence
there	there's	thereafter	thereby	therefore
therein	theres	thereupon	these	they
they'd	they'll	they're	they've	think
third	this	thorough	thoroughly	those
though	three	through	throughout	thru
thus	to	together	too	took
toward	towards	tried	tries	truly
try	trying	twice	two	u
un	under	unfortunately	unless	unlikely
until	unto	up	upon	us
use	used	useful	uses	using
usually	uucp	v	value	various
very	via	viz	vs	w
want	wants	was	wasn't	way
we	we'd	we'll	we're	we've
welcome	well	went	were	weren't
what	what's	whatever	when	whence
whenever	where	where's	whereafter	whereas
whereby	wherein	whereupon	wherever	whether
which	while	whither	who	who's
whoever	whole	whom	whose	why
will	willing	wish	with	within
without	won't	wonder	would	would
wouldn't	x	y	yes	yet
you	you'd	you'll	you're	you've
your	yours	yourself	yourselves	z
zero				