University of Oxford

Extending and Optimizing an Ontology Matching System

Antón Morant, Wolfson College

supervised by Bernardo Cuenca Grau and Ernesto Jiménez-Ruiz

A dissertation submitted in partial fulfilment of the degree of Master of Science in Computer Science.

Oxford University Department of Computer Science

September 2011

This is my own work (except where otherwise indicated)

Candidate: Antón Morant

Signed:................................

Date:..................................

Contents

Chapter 1

Introduction and Background

1.1 Introduction

Ontologies, formal conceptualisations of a domain of interest, are extensively used in biology and medicine. An ontology specifies the meaning of the entities and relations relevant to an application domain by means of logical statements (expressed in Description Logics). Automated reasoning tools can then exploit their semantics to make explicit information that was implicit in the ontology.

Ontology developers often need to integrate several ontologies. This might be the case when they need to exchange or migrate data between different ontology-based applications, or as the first stage in the development of a new ontology, in which relevant parts of existing ontologies are reused. However, since different ontologies are typically developed by different groups of experts, even if they describe related domains it is reasonable to expect that their vocabularies will be quite different. For very small-sized ontologies, this problem can be simply addressed by manually identifying what are the names given in each ontology to common concepts. If the source ontologies are large, however, such task cannot be manually undertaken.

The problem of finding correspondences between entities in ontologies is called *ontology matching*. For large and complex ontologies, such task can only be efficiently accomplished by an automatic tool. Existing methods for mapping discovery are mainly based on lexical and structural techniques. These algorithms, however, do not take into account the semantic information included in the initial ontologies. Thus, found mappings could lead to unintended logical consequences when automated reasoning is performed over the integrated ontologies (i.e., the initial ones together with the mappings).

LogMap [1, 2] is a EPSRC-funded project whose aim is to develop an ontology matching system (the LogMap tool, or simply 'LogMap') that combines lexical, structural and semantic techniques. The latter are especially important to minimise the number of erroneous mappings introduced by lexical heuristics. Consider for example the scenario in which a user is developing an ontology about juvenile forms of arthritis, for which they want to start by integrating two independently developed ontologies: one describing types of arthritis and the other juvenile diseases. Both ontologies provide a description of different types of juvenile arthritis, but since they have been independently developed, their vocabularies are disjoint. The disease known as JRA (Juvenile Rheumatoid Arthritis), for instance, is represented by the concept Juvenile_Rheumathoid_Arthritis in the first ontology and by the concept Rheum_Arthritis_Juvenile in the second one. By using lexical techniques, LogMap is able to establish a correspondence between these concepts. Furthermore, if the addition of such correspondence would lead to a semantic error (i.e. a logical contradiction), LogMap can detect it, identify the mappings involved in the conflict and attempt to repair (by using semantic techniques). In realistic ontologies there may be thousands of such conflicts. Each such conflict may involve several mappings, and a given mapping can participate in many conflicts.

This chapter provides some context for the following ones introducing the background problem (sections 1.2 and 1.3), briefly analyzes the state of the art in ontology matching tools (section 1.4), describes in further detail the algorithm implemented by LogMap (section 1.5) and summarises the objectives and main contributions of this thesis (section 1.6).

Chapter 2 describes the methods used in the development of the thesis, addressing in particular experimental work. Chapters 3, 4 and 5 account for the extensions proposed for each addressed stage in the LogMap algorithm ('Lexical indexation', 'Structural indexation' and 'Mapping repair and discovery') and analyze the outcome of their evaluation. Finally, chapter 6 resumes the obtained conclusions and main contributions of this thesis.

1.2 Ontologies

This section introduces the concept of *ontology* and provides some related terminology[1].

[1]It is assumed that the reader is familiar with the basics of FOL.

Description Logics (DL) are family of decidable fragments of First Order Logic (FOL). Their vocabulary is composed of *atomic concepts* (unary predicates), *atomic roles* (binary predicates) and *individuals* (constants). DL semantics is given by syntactic translation into FOL, and a DL ontology corresponds to a FOL knowledge base. Ontologies are composed of two sets of statements: The terminological box *TBox*, typically containing FO sentences (i.e. with no free variables) without constants, and the assertion box *ABox*, comprised of ground atomic formulae.

Given an ontology $\mathcal{O}$, the following notions are defined:

- An ontology is **consistent** if it has a FO-model.
- $\mathcal{O}$ **entails** a statement α if every model of $\mathcal{O}$ satisfies α.
- An atomic concept (unary predicate) A is **satisfiable** with respect to $\mathcal{O}$ if there exists a model of $\mathcal{O}$ in which A is interpreted as a non-empty set.

1.3 Ontology matching

Ontology matching (or ontology *alignment*) can be defined on an abstract level as the task of finding correspondences between the vocabulary of different ontologies. [3]. This correspondences express relations between such entities, and the entities themselves can be either elements of the conceptual schema of the ontology (TBox) or individuals belonging to its dataset (ABox). The former is known as schema matching, and the latter as instance matching, although the general name of 'ontology matching' is often used to refer to schema matching[2].

Correspondences can be established between different types of entities, e.g. concepts (classes) or properties (relations). Although LogMap's techniques can be easily applied for property matching, the current implementation targets class matching. The term *mapping* is used in this thesis to refer to a correspondence established between classes. A formal definition of mapping (adapted from [3]) follows:

Given two ontologies, $\mathcal{O}_1$ and $\mathcal{O}_2$, a ***mapping*** *is a quintuple:*

$$\langle id, c_1, c_2, r, n \rangle$$

such that:

- *id* is an identifier of the given correspondence.

[2]Since LogMap focuses in schema matching exclusively, in this thesis the mentioned convention is followed using the term 'ontology matching' hereafter as an equivalent to 'schema matching' unless specified otherwise.

- c_1 and c_2 are classes of $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively.
- r is a relation holding between c_1 and c_2 such as equivalence ($\equiv$), subsumption ($\sqsubseteq$) or disjointness ($\perp$).
- n is a confidence measure (typically in the [0, 1] range) holding for the correspondence between c_1 and c_2.

1.4 State of the art

A wide range of approaches have been developed for ontology alignment. Furthermore, current matching tools often do not limit themselves to applying one single technique or method but combine several of them in a structured way. For instance, LogMap uses lexical techniques for establishing initial mappings and structural and semantic methods for refining or repairing them. Section 1.4.1 provides an overview of current state-of-the-art techniques in ontology matching while section 1.4.2 relates them to some of the most relevant matching tools at the moment.

1.4.1 Matching strategies

Ontologies contain knowledge of different types within them, and different approaches to ontology matching exploit different kinds of knowledge. Some of them perform lexical comparisons between terms, explore the context of the classes or extract logical consequences through reasoning over the ontologies' axioms. Instance data can also be accessed to analyze frequency distributions, or furthermore, external resources can be used during the process. Matching techniques can be classified in the following way [4]:

Similarity techniques
These techniques measure the degree of 'similarity' between concepts in order to decide whether they represent the same entity. Such exploited similarities can be of different kinds:

Linguistic techniques use label names of classes and properties to identify correspondences between them. They can be either purely *syntactic* comparisons, using just the structure of the words, or they can also use *semantic* information, accessing lexical resources such as WordNet [5].

Contextual techniques analyze the structural features of concepts comparing elements either from their external structure (e.g. hierarchically linked classes or related properties) or internal (e.g. property domain and range types). Contexts built this way are often represented by graphs, and their similarity can be determined with topological similarity measures.

Reasoning-based techniques
Reasoning-based techniques take as the input the ontologies together with an initial set of mappings (manually or automatically defined) and they reason over them to infer new mappings and to derive the implications of such correspondences. According to the type of reasoning performed, they can be:

Deductive techniques, which mainly use propositional (SAT) or description logics (DL) to logically express the candidate mappings as formulae and verify their satisfiability.

Probabilistic techniques, which make use of machine learning techniques or Bayesian networks to determine the probability whether two given concepts refer to the same entity.

1.4.2 Matching tools

An increasing number of ontology matching tools and frameworks are available at the moment which apply the previously described techniques (see section 1.4.1). Furthermore, it is a common tendency in these tools to provide an extensible system which integrates a number of *matchers* rather than focusing on only one algorithm, allowing customized parametrization.

Most matching tools are based entirely on lexical and structural (contextual) techniques. For instance, TaxoMap [6] lexically processes concept labels to obtain morphological and syntactic information, but also divides the hierarchical structure of the ontologies into partitions for better scalability. AgreementMaker [7] implements a stack of syntactical and structural matchers (allowing customization of the stack policies) and combines their results into the final set of mappings.

However, automatic lexical and structural techniques often lead to semantic errors which manifest themselves in the form of logical inconsistencies. For example, consider the scenario where the ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$ are being matched: two equivalence mappings connecting concepts that are related via an is-a relation in $\mathcal{O}_1$ to concepts that are disjoint in $\mathcal{O}_2$ would be conflicting.

Interest is therefore growing in designing tools that are based on or incorporate reasoning techniques. For example, ASMOV [8, 9] iteratively calculates similarity measures combining weighted lexical, structural (internal and external) and extensional (data instances) techniques, and then performs a series of semantic checks to prevent logical inconsistencies. KosiMAP [10, 11] extracts logical consequences about entities from ontology axioms through

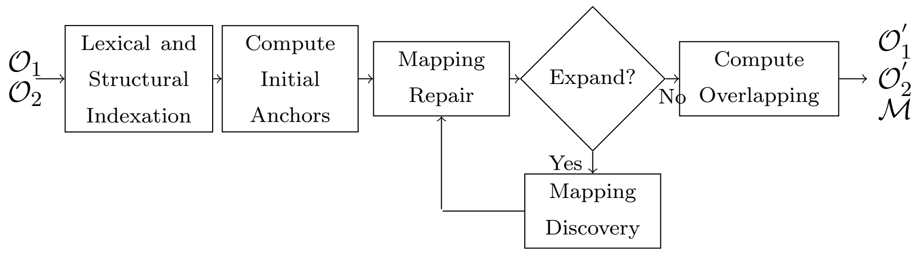


Figure 1.1: Overview of LogMap stages.

DL-reasoning, then combines lexical and structural similarity matchers to produce a 'pre-alignment' over which DL-reasoning is used to remove inappropriate mappings. CODI [12] provides a declarative framework for ontology matching (at schema and data levels) which uses lexico-structural matching techniques for mapping detection together with Markov logic for logical representation and evaluation of mappings.

1.5 LogMap

LogMap [2] is an ontology matching system which combines lexical, structural and semantic techniques to produce incoherence-free alignments. The motivation underlying LogMap's design is to provide a matching tool satisfying two needs which are commonly lacked by most available matchers: scalability and logical consistence.

Although currently available tools can deal efficiently with moderately-sized ontologies (thousands of classes), some bio-medical ontologies can have up to tens or hundreds of classes (e.g. SNOMED CT, see section 2) and are still beyond their reach. LogMap intends to make use of accurate but efficient methods in order to manage to process such large ontologies.

Since manually curating these errors would be extremely time-consuming, the scenario requires of a tool capable of automatically repairing such errors. LogMap uses reasoning techniques for efficient detection and repair of logical inconsistencies.

Figure 1.1 provides a schematic overview of the stages followed by the LogMap algorithm, briefly described next:

1. **Lexical indexation:** After parsing the input ontologies (received in OWL[13] format), lexical indices are built for each ontology to store the labels of its classes in a compact format which will allow efficient lookup. At this stage, linguistic techniques are used to enhance the index with spelling variations, synonyms, etc. Chapter 3 describes

tasks performed here in detail as well as the proposed extensions for this stage.

2. **Structural indexation:** The extended[3] hierarchy of each ontology is computed using structural heuristics or an off-the-shelf DL reasoner. An interval-labelling schema is then used to efficiently store structural information. See chapter 4 for a more detailed description of this stage as well as extensions implemented in this thesis.
3. **Computation of 'initial anchor' mappings:** Lexical indices are intersected producing a set of *exact* lexical matches. These are considered reliable mappings, and will be used as initial anchors for the iterative core of the algorithm.
4. **Mapping repair and discovery:** The expansion stage is an iterative algorithm composed of two steps: repair and discovery. The repair step takes as an input the so-far established mappings (anchors plus already repaired mappings) and the most recently discovered ones (the *active* mappings), and through reasoning techniques for inconsistency detection and repair 'cleans' the active mappings. The discovery step makes use of the already established mappings to explore the ontology hierarchies looking for new mapping candidates, and obtains new correspondences using an efficient lexical matcher [14]. These new mappings will be cleaned in the next iteration, and so on until a stop condition is reached (namely, no new mappings are discovered). Chapter 5 addresses this stage and describes proposed enhancements for the repair algorithm.
5. **Overlapping estimation:** In addition to the final set of mappings, LogMap produces fragments of the input ontologies corresponding to their estimated overlapping. These fragments are meant to be offered to the user if it is needed to look for missed mappings, since it is most likely that they will be found in the overlapping. For obtaining such fragments, LogMap includes classes involved in 'weak mappings', which establish correspondences between related entities but not necessarily equivalent (their label names are not exact matches, but share a number of terms) and therefore are not taken into account for mapping discovery.

1.6 Summary of contributions

The main goal of this MSc thesis is to propose and implement extensions to improve the matching techniques currently implemented in LogMap. Such

[3]The class hierarchy of each ontology is extended with additional information such as more complex subsumption axioms or explicit disjoints

enhancements must be integrated into LogMap for their evaluation on realistic ontologies.

Extensions are proposed addressing some of the stages of LogMap's algorithm as follows:

- **Lexical indexation** is enriched with WordNet synonyms and stemming tools for improving the system's recall, and the results for determined ontologies are shown to benefit from the stemming extension. Furthermore, a significant efficiency boost is obtained as a side-effect of stemming.
- **Structural indexation** makes use of an interval-labelling schema provided by an external library. A new library has been designed in order to carry out this task. It is shown that the new component, optimized for storing ontology hierarchies, is capable of greatly reducing execution times for structural indexation (up to 97%).
- The repair algorithm in the **mapping repair and discovery** stage is enhanced with two different optimized methods. These new versions manage to improve efficiency, significantly reducing execution times for ontology matching (up to 17% excluding the indexation stages).

Besides the direct benefits obtained from proposed extensions, the library provided for structural indexation presents an additional advantage. Since it integrates into LogMap rather than being an external process (as the library used by previous LogMap releases is), it allows the system to be delivered as a self-contained tool, therefore qualifying to enter the OAEI 2011 Campaign for ontology alignment evaluation. Publication of LogMap's participation in the matching contest is pending [15] (details are given in section 6.2.1).

Chapter 2

Methodology

This chapter aims to provide a frame of reference for methods referred to throughout this thesis. In particular, details are given on the assessment methods and test case ontologies used for experiments.

2.1 Assessing the proposed extensions

Since one of the objectives set for this thesis is to evaluate the impact of proposed extensions to the LogMap system, a number of experiments have been carried out for that purpose. Since the objectives fixed for the different extensions vary, two types of experiments were designed: quality experiments (section 2.2 and efficiency experiments (section 2.3). Note that, however, both types were needed for some of the extensions (for instance, to know the extra cost of a newly introduced technique). Furthermore, due to the number of tests that had to be run, and the amount of results to be processed, a flexible evaluation framework has been implemented ad-hoc to run series of tests and extract diverse information from the output logs.

Scalability is one of the stated objectives for the LogMap project. Therefore, knowing that the enhancements proposed would allow the system to retain such property was crucial. Real-world large ontologies were therefore chosen as test cases for experimentation (the same that are used for evaluation of LogMap), together with one pair of ontologies from a renowned ontology matching benchmark:

- SNOMED CT Jan. 2009 version ($306,591$ classes)
- NCI version 08.05d ($66,724$ classes)
- FMA ($78,989$ classes)

- NCI Anatomy (3,304 classes) and Mouse Anatomy (2,744 classes), both from the OAEI 2010 benchmark [3, 16]

The conducted experiments were run on two standard laptops: a Windows 7 4GB RAM system and a Linux Fedora 4.6GB RAM system. To ensure coherence, all time measures were run on the same machine (Linux).

It is worth noting that LogMap is parametrized by a number of configurable values such as confidence threshold for expansion or maximum number of iterations. For evaluation purposes, parameter values have been fixed to LogMap's 'defaults' for all conducted experiments.

2.2 Measuring the quality of the results

As it is traditional in the field of information retrieval and common practice in the context of ontology matching evaluation, a matching system is evaluated by comparing the set of obtained mappings with a previously designated *gold standard*, i.e. a manually curated collection of mappings which is agreed to be thet *most correct* solution to the matching problem. Precision, recall and F-measure scores are obtained from this comparison and used as the main quality measures to assess matching tools. Let M be the set of found mappings and GS the set of correct mappings. The mentioned measures are computed as follows:

Precision: Ratio of correct found mappings with respect to total number of found mappings, i.e. 'How many of the found mappings are correct?':

$$P = \frac{|M \cap GS|}{|M|}$$

Recall: Ratio of correct found mappings with respect to total number of correct mappings, i.e. 'How many of the correct mappings were found?':

$$R = \frac{|M \cap GS|}{|GS|}$$

F-measure: Dice's coefficient of the found and correct mappings sets. This measure combines precision and recall scores and applies a hard penalty to very unbalanced systems[1], i.e. 'How close are the found mappings to the correct mappings?'::

$$F = \frac{2 \times |M \cap GS|}{|M| + |GS|} = \frac{2 \times P \times R}{P + R}$$

[1]Otherwise, if for instance the arithmetic mean were used, extreme cases could be aimed for in order to attain higher scores, e.g. reporting all possible mappings as found would yield perfect recall, and reporting none at all would achieve perfect precision.

For assessing each of the extensions proposed in this thesis, they were implemented and integrated into LogMap. Then, experiments were run under identical conditions with and without the assessed extension. Precision, recall and F-measure scores were extracted and compared for analysis and evaluation.

As for the gold standards, the following resources were employed:

- The mappings FMA-NCI, FMA-SNOMED and SNOMED-NCI included in UMLS Metathesaurus [17] version 2009AA (the mappings are extracted from the UMLS distribution files [18]).
- The OAEI 2010 Anatomy track gold standard [3, 16].

2.3 Measuring efficiency

For some of the extensions designed in this thesis, the objective was to improve the system's efficiency and therefore it was needed to determine such improvement. Execution times were run for the test cases with and without the extensions, and results were compared to conclude whether the proposal had succeeded in achieving its goal.

Besides this, it was found necessary to run efficency tests on other extensions as well to measure whether they had a negative impact on the system's peformance (e.g., enriching lexical indices with WordNet synonyms for labels greatly increases the cost of the mapping extraction process).

Chapter 3

Lexical Indexation Stage

The first step after parsing the input ontologies is building the lexical indices that will be used for establishing the initial anchor mappings. This indices take the form of inverted files, a data structure widely used in information retrieval applications.

For each class, LogMap takes the English name of each class, as well as any other alternative label (usually stored in the ontology as OWL annotations), and splits them into single words. Thus, each label produces a set of words which is then introduced as the key in the inverted index to store the class identifier (an integer).

For instance, the ontology Mouse Anatomy contains the class labelled {thorax_bone} which is annotated with the synonym {upper_body_bone}. Hence, the inverted file index contains two entries for this class: one of them has {thorax,bone} as its key and the other has {upper,body,bone}. Both entries map to the same numeric identifier (2008).

Once the lexical index has been built, the initial set of anchors can be efficiently computed by intersecting the sets of keys in each inverted index, thus obtaining exact matches: classes whose labels contain exactly the same words (disregarding the order).

The objective of this thesis with respect to the lexical indexation stage is to increase the number of intended mappings detected by the system (its recall). This is achieved by enriching the set of alternative labels for each class. During computation of the initial anchors and during mapping discovery, mappings are established between classes depending on a lexical similarity measure that takes into account such alternative labels. Hence, an enriched lexical index increases the probability of an intended mapping being ‘detected’.

LogMap already makes use of the annotations contained in the ontologies to look for alternative labels and of the UMLS Lexicon to search for spelling variants. This thesis introduces two additional enhancements for the lexical indexation: using the WordNet lexicon (section 3.1) and stemming (section 3.2).

3.1 WordNet

WordNet [5] is a lexical English database containing information on semantic relatedness for words, including relations such as synonymy (same meaning), antonymy (opposite meanings), hyperonymy-hiponymy (hierarchycal subordination), meronymy (part-of relationship) and others.

An external tool (*WordNetFetcher*) was implemented to extract synonymy information from the WordNet database in the following way: An ontology is precomputed in order to extract a 'bag of words', i.e. the set of words contained in all class labels across the ontology's signature. Then, the tool accesses the database and obtains a list of synonyms for each word, storing them in the form of a map into an output file.

This map can be loaded during the lexical indexation stage of the algorithm and efficiently queried to obtain synonyms for the words contained in a label. Then, all possible combinations of each label substituting each word for one of its synonyms are produced (under a controlled limit to avoid huge numbers of cases) and stored as alternative labels for the class in the index.

The tool *WordNetFetcher* is written in C to make use of the WordNet search API, which allows direct access to the WordNet database. This method is fairly efficient (see section 3.3.1), and furthermore, it only requires to be executed once. Then, LogMap only needs to load the map file, which is also efficiently accessed to fetch synonyms.

Table 3.1 shows fragments of the lexical indices generated for the Mouse and NCI Anatomy ontologies, in particular the entries related to the classes Mouse:Right_Oviduct and NCI_Anat:Right_Fallopian_Tube. The WordNet extended index for Mouse shows how it has been enriched with new entries for Mouse:Right_Oviduct by producing alternative labels. This alternative labels result from substituting the the word *oviduct* with the synonyms found in WordNet for it: {*oviduct, uterine tube, fallopian tube*}.

The result of including these additional entries is that the intersection of the inverted file keys, which would be empty without WordNet expansion, contains now the key {*tube, right, fallopian*}, which leads to successfully establishing a mapping between the classes Mouse:Right_Oviduct and NCI_Anat:Right_Fallopian_Tube. This is an intended mapping (it is contained

Lexical inverted indices		
	Key	***Ids***
Mouse	oviduct, right	2651
NCI Anatomy	tube, right, fallopian	2804
Mouse (wnet)	uterine, tube, right	2651
	tube, right, fallopian	2651
	oviduct, right	2651
NCI Anatomy (wnet)	tube, right, fallopian	2804

Class Ids to URIs indices		
	Id	***URIs***
Mouse	2651	Mouse:Right_Oviduct
NCI Anatomy	2804	NCI_Anat:Right_Fallopian_Tube

Table 3.1: Fragment of the lexical indexes for Mouse and NCI Anatomy ontologies, showing the changes when extending with WordNet.

in the Gold Standard used for evaluation) that is not detected by LogMap without the WordNet extension.

3.2 Stemming

Stemming is a lexical procedure, widely used in the field of information retrieval, consisting on removing suffixes from words in order to transform them into a common root shared by all words from the same lexical family. This root may or may not be a English[1] word; instead, it is a token which represents the whole set of words included in that lexical family.

In the context of ontology matching, stemming allows to recognize variations of the same word and deal with them as if they were the same, hence detecting mappings between concepts whose names include different forms of the same word. The number of detected mappings (the system's recall) can this way be increased.

The LogMap system has been extended to allow the use of stemming. The first approach followed the lines of the WordNet extension previously mentioned (section 3.1) and consisted of preprocessing the ontology's 'bag of words' to produce an index mapping words to their stemmed version. This map was to be loaded during lexical indexation by the LogMap algorithm. However, the experiments showed that stemming can be implemented very efficiently, therefore this approach was discarded and the preprocessing is no

[1]It is presumed here that the original language is English. Note that, for most stemming algorithms, there are adapted versions available for a variety of languages.

Lexical inverted indices		
	Key	***Ids***
Mouse	intestine, epithelium	1021
NCI Anatomy	intestinal, epithelium	2965
Mouse (stemming aggro)	intestin, epithel	1021
NCI Anatomy (stemming aggro)	intestin, epithel	2965
Mouse (stemming non-aggro)	intestin, epithel	1021
	intestine, epithelium	1021
NCI Anatomy (stemming non-aggro)	intestin, epithel	2965
	intestinal, epithelium	2965

Class Ids to URIs indices		
	Id	***URIs***
Mouse	1021	Mouse:Intestine_Epithelium
NCI Anatomy	2965	NCI_Anat:Intestinal_Epithelium

Table 3.2: Fragment of the lexical indexes for Mouse and NCI Anatomy ontologies, showing the changes when extending with stemming.

longer necessary. The current approach obtains stemmed versions running the algorithm on the fly during lexical indexation, avoiding thus having to load extra files or keeping extra maps in memory.

Systems applying stemming usually substitute each word for its stemmed version. Although this increases the chances of finding matches between ontology concepts, the index might however lose accuracy since the original form of the words composing the concept's label is discarded. This is due to the fact that labels are not only used by LogMap for finding the initial anchors but also for obtaining lexical similarity scores at later stages (see mapping repair and discovery in section 1.5). It might be therefore beneficial not to discard the original labels in the lexical index when using stemming. For this purpose, the LogMap extension handling stemming allows for two different modes to be selected: *aggressive* substitutes each word with its stemmed version when building the index while *non-aggressive* regards the stemmed version as if it were a synonym of the original, and keeps both version in the index.

Table 3.2 shows fragments of the lexical indices generated for the Mouse and NCI Anatomy ontologies, in particular the entries related to the classes Mouse:Intestine_Epithelium and NCI_Anat:Intestinal_Epithelium. Index versions are shown for the cases of no stemming, aggressive stemming and non-aggressive stemming (as described above). When the aggressive mode is selected, the existing entries are modified by substituting each word in the key by its root. For instance, for the Mouse ontology, the key {*intestin, epithel*} is used instead of {*intestine, epithelium*}. On the other hand, in the non-aggressive case, entries are created for both the original label and the stemmed version.

As a result of the described changes in the indices, the intersection their sets of keys is now non-empty and contains the key {*intestin, epithel*} (this holds for both the aggressive and non-aggressive options). Thus, a mapping is established between the classes Mouse:Intestine_Epithelium and NCI_Anat:Intestinal_Epithelium. This is an intended mapping (it is contained in the Gold Standard used for evaluation), and it is not detected by LogMap when the stemming extension is not activated.

3.2.1 Stemmer selection

Since there are multiple stemming algorithms available, it was necessary to carry a minimal survey on existing algorithms. An extensible framework was designed to allow different stemming tools to be plugged into LogMap; so that the user can choose among them when running the system if desired.

The following tools were chosen to be included in the stemming extension of LogMap because of their well-known good performance, relevance and availability of implementations:

- **Porter:** Porter Stemmer [19].
- **Porter2:** Porter 2, new version of the previous one and included as part of the SnowBall project [20].
- **Paice:** The Lancaster (Paice-Husk) Stemming Algorithm [21].
- **Lovins:** Lovins stemmer [22].
- **LovinsIter:** Actually same tool as *Lovins*, but enabling its 'iterated' options which recursively stems the term until no more changes are performed on the input.

It was verified that some stemming tools behave in a slightly more aggressive way than others, producing shorter roots for the same words. Also, some stemmers may act more accurately than others, for instance when dealing with irregular forms of words. This differences have an impact on the system's precision (too aggressive stemming matches unrelated words) and recall (too moderate or unsophisticated stemming fails to match related words), and therefore the mentioned tools were assessed under equal conditions to compare their performance (see detailed results in section 3.3.2).

Since Porter 2 turned out to be the most balanced and achieved, in general, best results during testing, it was chosen as the 'default stemmer' for

	No WordNet			WordNet		
	P	R	F	P	R	F
Anatomy	95.84	72.75	82.71	91.05	73.67	81.44
FMA-NCI	76.78	83.06	79.79	60.74	84.47	70.67
FMA-SNOMED	78.23	19.49	31.21	36.70	19.99	25.88
NCI-SNOMED	76.04	58.39	66.06	48.33	59.73	53.43

Table 3.3: Evaluation of WordNet application for lexical indexation: Precision, Recall and F-measure.

LogMap. All mentioned facts and collected data in this thesis related to the stemming extension comes therefore from the Porter 2 version unless stated otherwise.

3.3 Evaluation

The proposed extensions to the lexical indexation stage were tested in order to compare their impact both on the obtained results and on the execution times. Experiments were run for the ontologies in the OAEI Challenge Anatomy track (Mouse and NCI Anatomy), FMA, NCI and SNOMED.

Section 3.3.1 contains results for the WordNet extension, assessment of the application of stemming are shown in section 3.3.2, and a comparison of execution times for both extensions is included in section 3.3.3.

3.3.1 WordNet evaluation

Table 3.3 shows the obtained results for the specified ontologies when enabling the WordNet extension for the lexical indexation stage. Overall, the WordNet extension achieves the objective of improving the recall score of the system. However, this modest increase (up to 1.41% for FMA-NCI) is counteracted by the massive drop of precision, leading to significantly decreased F-measures for all ontology pairs.

In order to explain for the poor performance of the extension, it is worth noting that there are two factors that mitigate its positive impact:

- High specialization of the input ontologies: Since LogMap is mostly designed to target biomedical ontologies, the test datasets use highly specialized vocabulary. Such terms are difficult to find in the WordNet database and, when found, the obtained synonyms are often too general and lead to errors.
- Annotated synonyms for classes: In the tested ontologies, classes often include annotations containing alternative labels. These already

	Anatomy			**FMA-NCI**			**FMA-SNOMED**			**NCI-SNOMED**		
	P	R	F	P	R	F	P	R	F	P	R	F
None	95.84	72.75	82.71	76.78	83.06	79.79	78.23	19.49	31.21	76.04	58.39	66.06
Porter	95.32	73.67	83.10	74.82	85.09	79.63	66.72	19.63	30.33	73.09	55.37	63.01
Porter2	95.33	73.86	83.23	74.38	85.44	79.52	66.51	19.66	30.35	73.18	58.91	65.28
Paice	92.96	75.58	83.37	53.55	87.20	66.35	55.53	20.17	29.59	59.33	59.34	59.34
Lovins	94.83	72.42	82.12	68.57	82.06	74.71	63.36	19.79	30.16	69.55	55.37	61.65
LovinsIter	93.70	72.42	81.69	66.41	82.40	73.54	59.78	19.81	29.76	65.84	55.38	60.15

Table 3.4: Comparison of stemming tools for the lexical indexation stage: Precision, Recall and F-measure.

	No Stemming			**Stemming aggro**			**Stemming non-aggro**		
	P	R	F	P	R	F	P	R	F
Ana	95.84	72.75	82.71	95.33	73.86	83.23	95.33	73.93	83.28
F-N	76.78	83.06	79.79	74.38	85.44	79.52	73.94	85.85	79.45
F-S	78.23	19.49	31.21	66.51	19.66	30.35	65.61	19.66	30.26
N-S	76.04	58.39	66.06	73.18	58.91	65.28	70.78	59.05	64.39

Table 3.5: Evaluation of the stemming extension for lexical indexation (Porter2 stemmer). Precision, Recall and F-measure shown. Abbreviations for input ontologies: *Ana* = Anatomy, *F-N* = FMA-NCI, *F-S* = FMA-SNOMED, *N-S* = NCI-SNOMED.

provide LogMap with a not extensive but accurate set of synonyms, therefore WordNet's contribution is perceived mostly as ‘noisy’.

Hence, evidence suggests that the developed extension for inclusion of WordNet synonyms is not adequate for the LogMap system, unless it is applied for a different (less specialized) domain or the user's primary need is recall regardless of the precision obtained.

3.3.2 Stemming evaluation

As described in section 3.2, four stemming tools were included for this extension (they are five if both versions of Lovins are counted). A comparison of how each of them performs for the specified ontologies is shown in table 3.4. Aggressive stemming was used for this purpose.

The Porter2 tool outstands as the most balanced, achieving in general the best F-measure scores. Porter performs quite closely to it, only significantly below for NCI-SNOMED (where recall is quite lower than Porter2's). Paice's algorithm is much more aggressive than them, producing remarkable higher recall results for every test, but with the drawback of much lower precision with a subsequent drop in the F-measure. Finally, Lovins and its recursively-iterated version present a very aggressive behaviour noticeable by their poor precision scores, which not being compensated by a very high recall make these last tools unsuitable for this application.

As mentioned, Porter2 shows the best performance and is therefore the default tool for the stemming extension. It is worth noticing however that, should the user be interested in attaining high recall over precision, changing from Porter2 to Paice could be benefitial in such specific scenario.

Table 3.5 shows a comparison of results obtained for the specified ontologies disabling stemming, enabling aggressive stemming and enabling non-aggresssive stemming.

Ontologies	**None(s)**	**WordNet(s)**	**Stemming aggro(s)**	**Stemming non-aggro(s)**
Anatomy	7	24	9	9
FMA-NCI	187	1383	175	230
FMA-SNOMED	606	2893	470	649
NCI-SNOMED	756	7770	605	1173

Table 3.6: Evaluation of LogMap execution times if enabling the extensions: none, WordNet, aggressive stemming and non-aggressive stemming.

Results show that the developed extension succeeds in increasing the system's recall, from 0.17% for FMA-SNOMED to 2.79% for FMA-NCI with the non-aggressive option. However, it is counteracted by a loss of precision which causes the F-measure to decrease in all cases except for the Anatomy track (rises in 0.57% for the non-aggressive mode).

It is also verified that, in general, the aggressive and non-aggressive options perform similarly in terms of F-measures, although the latter attains slightly higher recall in exchange for worse precision values. For these reasons, it can be concluded that the aggressive method is overall preferable to the non-aggressive alternative.

In a similar way as it was commented for WordNet in section 3.3.1, there are two factors that counteract the positive impact of stemming in the system:

- High specialization of the input ontologies: Stemming algorithms are language-specific, and tools used for this extension are English stemmers. Highly specialized vocabulary as biomedical often contains non-English terms, acronyms, latinisms and other types of words for which it is difficult to find a useful root. These cases often lead to noise when stemming is enabled.
- Use of UMLS Lexicon: As it was previously mentioned in this chapter, LogMap uses the UMLS Lexicon, a lexical resource for specialized terms which contains spelling variants, plurals and normalizations. This lexical enrichment contributes to diminishing the visibility of the stemming extension effects.

In conclusion, experiments suggest that the stemming extension is beneficial if recall is prioritized over precision, and that it is recommendable to use it for the specific scenario of the Anatomy track of the OAEI Ontology Matching Challenge.

3.3.3 Execution times evaluation

Lexical index enhancing techniques can have an important impact (positive or negative) on the execution time of the system, for three reasons:

- Building the lexical indices takes additional time since there are additional tasks to do(trivial).
- If the employed technique enriches the index with supplementary label names (as it is the case of WordNet and non-aggressive stemming), the time required at later stages to handle each of the alternative labels each time two classes are compared increases significantly.
- If the employed technique, on the other hand, reduces the total number of labels in the index (as aggressive stemming does in certain cases), then there is a reduction in the time spent dealing with class labels at later stages.

Since this time differences, positive or negative, can be significant, experimentation was conducted to measure them. Table 3.6 resumes the evaluation results of the proposed lexical indexation extensions for the specified ontology matches. Execution times of the LogMap system are shown for the cases of no extensions enabled, WordNet enabled and both aggressive and non-aggressive stemming (Porter2 tool).

According to the results, execution times increase heavily when using the WordNet extension (up to 928% extra time for NCI-SNOMED). Regarding stemming, the non-aggressive mode also increases execution times, although in a much more moderate way (up to 55% extra time for NCI-SNOMED). Aggressive stemming, on the other hand, presents the added benefit of reducing execution times (up to 20% of time saved for NCI-SNOMED).

From the analyzed results and previous conclusions in sections 3.3.1 and 3.3.2, it follows that WordNet has too high an efficiency penalty to be suitable for inclusion in LogMap. Regarding stemming, the non-aggressive version increases execution times while aggressive stemming reduces them as an added advantage and behaves generally better (in terms of the F-measure). Hence, aggressive stemming stands as a useful extension for LogMap under certain conditions.

Chapter 4

Structural Indexation Stage

The second stage of indexation is structural indexation. The (extended) class hierarchy of each input ontology is used to obtain useful information for later stages in the algorithm. LogMap's scalability heavily relies therefore on accessing efficiently the stored information.

LogMap's extended structural index consists of the following elements:

- The *inferred hierarchy* of the ontology, describing all the parent-child links between its classes.
- Disjointments explicitly declared in the ontology.
- Additional complex class axioms (e.g., those stating subsumption between an intersection of named classes and a named class).

This extended hierarchy is stored using interval-labelling schemas, optimized data structures for DAGs and trees which present efficient use of memory and access speed. For this purpose, the ontology classiffication is treated as a DAG, where each class is a node with links pointing to its children (and from its parents).

The first release of LogMap made use of an external Python library [23] for handling the interval-labelling schemas. The external library was running on a local server and communicated with the system through sockets, receiving the classified ontology and sending back the computed schemas. For each ontology, two DAGs were to be processed: one containing the descendants hierarchy (links from parents to children) and another containing the ancestors hierarchy (links from children to parents).

This approach presents two main disadvantages: First, communication with the library and the additional call for the reverse hierarchy are very time-consuming. Second, needing to run the external server prevents the system from being one self-contained component, which is desirable for entering the OAEI ontology matching challenge.

The objective of this thesis with respect to the structural indexation stage is to provide a new implementation of the interval-labelling schema library in Java, in order to improve LogMap's efficiency and to allow direct integration in the system (without needing to run an external process).

4.1 The interval-labelling schema library

A new library has been designed and built implementing the algorithm for compression of transitive relations into interval-labelling schemas described in [24], a technique that has been applied to ontology hierarchy indexing and shown to significantly reduce the cost of typical queries over large class hierarchies [25, 23]. The package offers a data structure that takes as input a classified ontology (in the form of a map of classes to their children) and then builds the corresponding interval-labelled graph following the steps described in figure 4.1.

The algorithm in [24] uses postorder indices for nodes, while the library formerly used by LogMap implements a preorder walk instead. For the designed library, an extensible set of classes was built so that the user can select which of the modes to use. It is woth noting that they differ in the output intervals' format, but their behaviour is exactly equivalent in terms of performance and correctness.

4.2 Optimizing data structures: Encoding descendants and ancestors in a single structure

In order to optimize memory usage, the library stores both the descendants and ancestors information in a single structure. For this purpose, for each class in the input ontology a *node* is created storing all information related to it:

- *Class identifier assigned by LogMap to each class.*
- *List of child nodes in the classified ontology.*
- *List of parent nodes in the classified ontology.*
- *Parent node in the descendants optimal spanning-tree.*
- *List of child nodes in the descendants optimal spanning-tree.*
- *Parent node in the ancestors optimal spanning-tree.*

Algorithm: Creation of interval-labelled schema

1. Create nodes for each class, and build map relating class identifiers to class nodes.
2. Build DAG representing the classified ontology.
3. Both for descendants and ancestors, do:
   (a) Find DAG's root (create if necessary).
   (b) Obtain optimal spanning-tree. This step is optional (any spanning-tree can be used), but it guarantees that the data structure will only store the least possible number of intervals.
   (c) Perform preorder (or postorder) walk of the spanning-tree, assigning to each node a preorder (or postorder) index. Additionally, each node is assigned a *tree interval* of the form [i, j], where i is the node's preorder index and j is the maximum preorder index among its descendants (or j is the node's postorder and i is the minimum postorder among its descendants).
   (d) Compute and compress inherited intervals. Each node's list of intervals, which only contained its *tree interval*, is now extended with its descendants' intervals. This is accomplished performing a depth-first traversal of the whole hierarchy (not only the spanning-tree this time), so that each node 'inherits' all of its descendants' intervals by inheriting those belonging to its direct children. At the same time, intervals are compressed, so that if for the same node one interval is subsumed by another interval, the former is discarded, and if they are contiguous both are merged into a new one.

Figure 4.1: Algorithm for building the interval-labelled schema for ancestors and descendants.

- *List of child nodes in the ancestors optimal spanning-tree.*
- indexDesc : *Preorder (postorder) index in the descendants tree.*
- indexAnc : *Preorder (postorder) index in the ascendants tree.*
- intervalsDesc : *List of descendants intervals.*
- intervalsAnc : *List of ancestors intervals.*

After the compressed index has been created, data structures in a node relating to other nodes can be cleared to save memory, so that the effective memory usage is limited to (disregarding class and index identifiers, which are integers) one list of intervals for descendants and another for ancestors. Furthermore, having that the algorithm compresses the list intervals and that an optimal spanning-tree has been used to assign them, these lists are of minimal size.

Additionally, the artificial roots mentioned in 4.1 which may need to be created to perform the preorder or postorder walks are automatically unattached and reattached from their related nodes when not needed. This is, if the hierarchy root has been artificially added, then it is attached before the descendants traversal but unattached after it, and likewise for the inverted root and the ancestors traversal. If they were kept attached to the graph during each of these walks, every node[1] would be labelled with one extra interval for the artificially added inverse root. This way, attachment and reattachment of artificial roots improves memory usage.

4.3 Querying the hierarchy

Figure 4.2 shows a fragment of the NCI ontology together with two interval-labelled DAGs representing the corresponding descendants and ascendants hierarchies. These hierarchies are indexed by the designed library while disjointness and complex class axioms are stored in separate structures not addressed by this thesis. Once the interval-labelled schema has been obtained by the library, LogMap can access it for efficiently answering queries about parent-child relationships. The following are examples of typical queries over ontology hierarchies which only require simple integer operations (class label abbreviations as stated in Figure 4.2):

- *'Is* Smegma *a subclass of* Anatomy*?'*:
  - Check if indexDesc(S)=7 is contained in any of intervalsDesc(A)={[5,9]}.
  - $7 \in [5, 9]$
  - Answer is *Yes*
- *'Is* BiologicalProcess *an ancestor of* CellularSecretion*?'*:
  - Check if indexAnc(BP)=4 is contained in any of intervalsAnc(CS)={[2,4]}.
  - $4 \in [2, 4]$
  - Answer is *Yes*
- *'Do* ExocrineGlandFluid *and* TransmembraneTransport *have descendants in common?'*:
  - Check if the intersection of any interval from intervalsDesc(EFG)={[6,7]} and any interval from intervalsDesc(TT)={[3,4]} is non-empty.
  - $[6, 7] \cap [3, 4] = \emptyset$
  - Answer is *No*

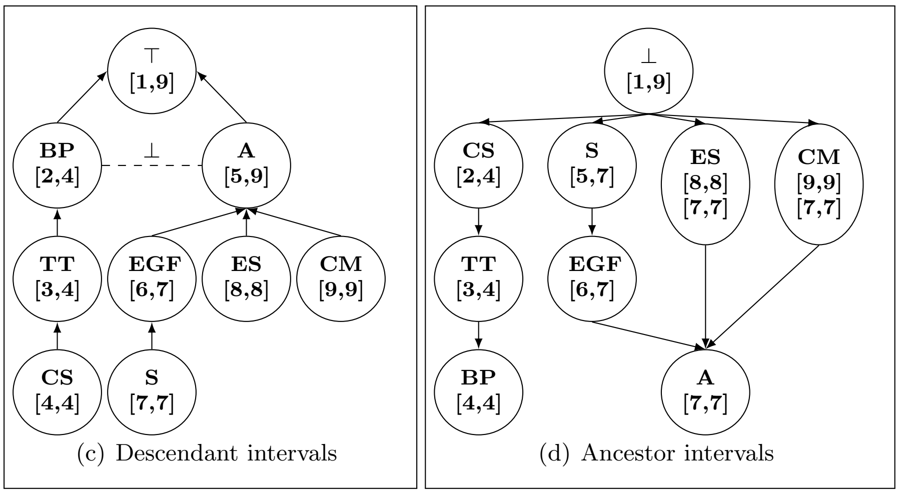
[1]To be rigorous, nodes within the same branch as the inverse root in the minimal spanning-tree would not store an extra interval. Their tree-interval would already contain the index of the inverse root.

α_1 : Anatomy $\sqsubseteq \neg$BiologicalProcess
α_2 : TransmembraneTransport $\sqsubseteq \exists$BP_hasLocation.CellularMembrane
α_3 : $\exists$BP_hasLocation.$\top \sqsubseteq$ BiologicalProcess
α_4 : $\top \sqsubseteq \forall$BP_hasLocation.Anatomy
α_5 : CellularSecretion $\sqsubseteq$ TransmembraneTransport
α_6 : ExocrineGlandFluid $\sqsubseteq \exists$AS_hasLocation.ExocrineSystem
α_7 : $\top \sqsubseteq \forall$AS_hasLocation.Anatomy
α_8 : $\exists$AS_hasLocation.$\top \sqsubseteq$ Anatomy
α_9 : Smegma $\sqsubseteq$ ExocrineGlandFluid
α_{10} : ExocrineGlandFluid $\sqcap$ ExfoliatedCells $\sqsubseteq$ Smegma

(a) NCI ontology fragment

β_1 : $\forall x.$Anatomy$(x) \rightarrow \neg$BiologicalProcess(x)
β_2 : $\forall x.$TransmembrTransport$(x) \rightarrow \exists y.$(CellularMembr$(y) \wedge$ BP_hasLocation$(x, y))$
β_3 : $\forall x.(\exists y.$BP_hasLocation$(x, y)) \rightarrow$ BiologicalProcess(x)
β_4 : $\forall x.\forall y.$BP_hasLocation$(x, y) \rightarrow$ Anatomy(y)
β_5 : $\forall x.$CellularSecretion$(x) \rightarrow$ TransmembrTransport(x)
β_6 : $\forall x.$ExocrineGlandFluid$(x) \rightarrow \exists y.$ExocrineSystem$(y) \wedge$ AS_hasLocation$(x, y))$
β_7 : $\forall x.\forall y.$AS_hasLocation$(x, y) \rightarrow$ Anatomy(y)
β_8 : $\forall x.(\exists y.$AS_hasLocation$(x, y)) \rightarrow$ Anatomy(x)
β_9 : $\forall x.$Smegma$(x) \rightarrow$ ExocrineGlandFluid(x)
β_{10} : $\forall x.$ExocrineGlandFluid$(x) \wedge$ ExfoliatedCells$(x) \rightarrow$ Smegma(x)

(b) FOL translation



(c) Descendant intervals (d) Ancestor intervals

Figure 4.2: Fragment of NCI extended hierarchy and labelled schema (FOL translation shown for reference).
Abbreviations: *BP*=BiologicalProcess, *A*=Anatomy, *TT*=TransmembraneTransport, *CM*=CellularMembrane, *EGF*=ExocrineGlandFluid, *CS*=CellularSecretion, *ES*=ExocrineSystem, *S*=Smegma.

Ontology	**Original time (s)**	**Improved time (s)**	**Time reduction (%)**
Mouse Anatomy	0.546	0.016	97.07%
NCI Anatomy	1.279	0.046	96.40%
FMA	35.038	2.044	94.17%
NCI	32.666	1.435	95.61%
SNOMED	961.071	33.197	96.55%

Table 4.1: Evaluation of structural indexation times before and after the integration of the new interval-labelling schema library.

- *'Do* Smegma *and* CellularSecretion *have ancestors in common?'*:
  - Check if the intersection of any interval from intervalsAnc(S)={[5,7]} and any interval from intervalsAnc(CS)={[2,4]} is non-empty.
  - $[5,7] \cap [2,4] = \emptyset$
  - Answer is *No*

4.4 Evaluation

The designed interval-labelling schema library was tested for correctness in two stages: First, small-sized unit tests were run during and after development. The library was then integrated into LogMap and put in exploitation, and it was checked that the obtained output for realistic ontologies (FMA, NCI, SNOMED) matched the one obtained with the previous library.

The next stage in the evaluation was measuring the impact of integrating the designed library instead of running the formerly used external one. Table 4.1 shows the assessment results for the library. The experiments consisted on measuring the execution times of the structural indexation stage for the ontologies Mouse, NCI Anatomy, FMA, NCI and SNOMED using both libraries. Table shows obtained times for each library for , as well as the efficiency improvement expressed as the percentage of reduced time.

Results show that execution times are reduced by a large extent with the new library both for small 'laboratory' ontologies such as those from the OAEI Anatomy track, as well as for reallistic ontologies such as FMA, NCI or SNOMED. The case of the latter is of special relevance, where the indexing time has been improved from 961 seconds to 33 seconds.

This huge efficiency boost can be explained by optimizations described in section 4.2 together with the fact that the library has been integrated directly into the LogMap system and is no longer needed to run and to communicate with an external process.

Chapter 5

Mapping repair and discovery

Once the lexical and structural indices have been built and the initial anchors have been established, the next stage is mapping repair and discovery, in which anchors are used to start the search for new mappings (see section 1.5). This task is carried out by the core of the LogMap system, an iterative process consisting of two alternate steps: repair and discovery; which continue until no more mappings are found. Each step can be briefly described as follows:

- During the **repair step**, a (possibly incomplete) reasoning algorithm is used to identify unsatisfiable classes with respect to the merge of the input ontologies and the set of mappings established so far. A 'greedy' diagnosis algorithm is then used to automatically repair (if possible) the detected unsatisfiabilities.
- During the **discovery step**, previously found and 'cleant' mappings are used as anchors to explore the input ontologies in an efficient way searching for new mappings.

This thesis focuses in enhancing the repair step. Algorithmic details of the actual design (section 5.1) as well as proposed modifications and extensions (sections 5.2, 5.3) are described next.

5.1 The repair step

Automatically found mappings are prone to leading to undesired logical consequences, which manifest themselves as unsatisfiable classes. The objective

Procedure Repair$_1$
Input: *List*: Ordered classes; $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_M$ Horn-propositional theories.
Output: $\mathcal{P}_M$: set of repaired mappings

```
1: for each $C \in List$ do
2:     $\mathcal{P}_C := \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_M \cup \{\mathsf{true} \rightarrow C\}$
3:     $\langle sat, \mathcal{P}_{act} \rangle := \mathsf{DowlingGallier}(\mathcal{P}_C)$
4:     if $sat = \mathsf{false}$ then
5:         $Repairs := \emptyset$
6:         $repair\_size := 1$
7:         repeat
8:             for each subset $\mathcal{R}$ of $\mathcal{P}_{act}$ of size $repair\_size$ do
9:                 $sat := \mathsf{DowlingGallier}(\mathcal{P}_C \setminus \mathcal{R})$
10:                if $sat = \mathsf{true}$ then $Repairs := Repairs \cup \{\mathcal{R}\}$
11:            end for
12:            $repair\_size := repair\_size + 1$
13:        until $|Repairs| > 0$
14:        $\mathcal{R}$ := element of $Repairs$ with minimum confidence.
15:        $\mathcal{P}_M := \mathcal{P}_M \setminus \mathcal{R}$
16:    end if
17: end for
18: return $\mathcal{P}_M$
```

Table 5.1: Repair in LogMap. A call to DowlingGallier returns a satisfiability value *sat* and, if *sat* = false, it optionally returns the relevant *active mappings* ($\mathcal{P}_{act}$).

of the repair step is to remove some of the found mappings in order to eliminate such unsatisfiabilities, hence yielding a set of 'clean' mappings.

In order to detect unsatisfiable classes, LogMap uses the Dowling and Gallier algorithm for propositional Horn satisfiability [26]. The structural indices of the input ontologies are used together with the established mappings to produce a propositional Horn theory. Then, a call to Downing and Gallier finds whether a given class is satisfiable with respect to the propositional theory. This is a sound and highly-scalable technique, but possibly incomplete. That is, detected unsatisfiabilities are guaranteed to be present in the non-propositional merge of the input ontologies and the set of mappings, while not every unsatisfiability entailed by the merge of the ontologies and mappings discovered so far is guaranteed to be detected. This is due to the loss of information when the ontologies' axioms are projected to Horn propositional form. However, experiments show that for instance, from the more than 600 unsatisfiable classes in FMA-NCI, LogMap only fails to detect one. It can therefore be assumed that if LogMap succeeds in repairing all found unsatisfiabilities, it likely that there are none remaining.

Table 5.1 describes in detail the steps followed by the repair algorithm currently implemented by LogMap. At each iteration of the LogMap core, a data structure keeps track of the *active mappings*, i.e. the set of mappings that have been found in the last discovery step and still need to be cleaned. The implemented version of Downing and Gallier has been extended to identify the subset of active mappings that are involved in the proof that a class is unsatisfiable. Thus, the diagnosis algorithm limits the search to mappings only from this 'conflicting' set.

Given an unsatisfiable class with respect to the propositional theory formed by the projected input ontologies and the established mappings, a *repair plan* is a subset of the active mappings such that, if they are removed from the theory, the class becomes satisfiable. Note that there may be zero, one or more possible repairs for a given class, and therefore it is needed to establish criteria to choose among them if several are found. LogMap uses an heuristic approach for plan selection, implementing the following specifications (in descending priority order):

1. A plan with a small **size** is preferred over a larger one. Intuitively, this condition states that it is desirable to remove as few mappings as possible during repair.
2. A plan with a low **confidence value** is preferred over a higher confidence one. Mappings are assigned confidence values through lexical and semantic heuristics (such as measuring lexical similarity or the amount of mappings found in the local neighbourhoods of the mapped classes). This way, a plan is assigned a confidence value equal to the added confidence values of the mappings it contains. Intuitively, this condition states that it is preferable to select plans involving removal of low-confidence mappings.

One of the most expensive operations in the repair algorithm is calling the Dowling and Gallier algorithm. Thus, reducing the number of calls made to this procedure would improve the system's efficiency for the expansion stage. For each class, one call is performed to test satisfiability. Additionally, for each class reported as unsatisfiable, there is one extra call for each potential repair plan that is checked until one is selected. This thesis introduces an enhanced version of the current repair algorithm which attempts to reduce the number of calls to Dowling and Gallier (section 5.2), as well as an extended version of the later which additionally allows to filter the 'conflicting mappings' before the possible plans are checked (section 5.3).

5.2 Enhancing the repair algorithm

LogMap's current repair algorithm has been optimized aiming to reduce the number of calls to the Dowling and Gallier algorithm but maintaining the

```
Procedure Repair$_2$
Input: $List$: Ordered classes; $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_M$ Horn-propositional theories.
Output: $\mathcal{P}_M$: set of repaired mappings
 1: for each $C \in List$ do
 2:     $\mathcal{P}_C := \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_M \cup \{\mathsf{true} \rightarrow C\}$
 3:     $\langle sat, \mathcal{P}_{act} \rangle := \mathsf{DowlingGallier}(\mathcal{P}_C)$
 4:     if $sat = \mathsf{false}$ then
 5:         $repair\_size := 1$
 6:         $\mathcal{R} := \emptyset$
 7:         repeat
 8:             $potential\_plans := \{\mathcal{R} : \mathcal{R} \subseteq \mathcal{P}_{act}$ and $|\mathcal{R}| = repair\_size\}$
 9:             $ordered\_plans := \mathsf{OrderPlans}(potential\_plans)$
10:             repeat
11:                 $\mathcal{R}_{candidate}$ := next element in $ordered\_plans$
12:                 $sat := \mathsf{DowlingGallier}(\mathcal{P}_C \setminus \mathcal{R}_{candidate})$
13:                 if $sat = \mathsf{true}$ then
14:                     $\mathcal{R} := \mathcal{R}_{candidate}$
15:                 end if
16:             until ($\mathcal{R} \neq \emptyset$ or $ordered\_plans$ is empty)
17:             $repair\_size := repair\_size + 1$
18:         until $\mathcal{R} \neq \emptyset$
19:         $\mathcal{P}_M := \mathcal{P}_M \setminus \mathcal{R}$
20:     end if
21: end for
22: return $\mathcal{P}_M$
```

Table 5.2: Enhanced repair algorithm. A call to OrderPlans returns a list of plans in ascending confidence order.

plan selection heuristics specified in section 5.1 (this premise was set so that the system's behaviour remains unaltered).

Table 5.2 shows in detail the the enhanced repair algorithm. The proposed modification is as follows: The former algorithm checks for all *potential plans* (i.e. mapping combinations) of a given size whether they are repair plans, and if several are found, then chooses according to their confidence values. The new version orders the potential plans according to confidence values into a queue, then iterates through it looking for the first element that is effectively a repair plan. If such a plan is found, it is selected and there is no need for extra calls to Dowling and Gallier for the remaining potential plans.

This method needs, therefore, to carry out tasks which the previous version did not, such as computing confidence values for every potential plan and ordering them. This operations, however, are comparatively inexpensive since confidence values are precomputed before the algorithm starts, and the saved calls to Dowling and Gallier are meant to compensate for their cost. Assessment results are given in section 5.4.

5.3 Mappings pre-filtering

The enhanced version of the repair algorithm described in section 5.2 introduces a new approach to plan evaluation and selection. However, all 'conflicting mappings' are still considered equal when obtaining potential plans. Another variation of the original algorithm has been designed extending the previous approach and allowing mappings to be filtered before combining them into plans. Repair exploration is optimized by checking first those plans containing only mappings which satisfy certain properties.

The pre-filtering extension allows application of two different filtering methods (combined or in a separate manner) to the conflicting mappings obtained from Dowling and Gallier:

- **Conflict likelihood:** Mappings less likely to be involved in a conflict are discarded (see section 5.3.1).
- **Confidence:** High-confidence mappings, which are less likely to be in the lowest-confidence repair plan, are discarded (see section 5.3.2).

Table 5.3 shows the modifications made to the previously enhanced algorithm (see table 5.2) to extend it with mapping pre-filtering. The new version filters the conflicting mappings obtained from Dowling and Gallier before entering the loop in which potential plans are checked. Then, the loop iterates twice for each size: The first time, filtering is enabled, and plans are formed only with filtered mappings. If no plans are found, then the loop is run a second time without increasing the size but disabling filtering, so plans are formed from all conflicting mappings. Also, in the second iteration, already tried plans are not checked again. If no plan is found in the second iteration, size is incremented and filtering enabled for the next iteration. Iterations over the size of potential plans are broken this way into two parts, first checking all plans of that size which contain only filtered mappings, and then checking the remaining ones.

In section 5.2 it was mentioned that the enhanced method (without filtering) complies with the specifications of the plan-selecting policy described in section 5.1 (selecting criteria were size and confidence). The method extended with filtering, however, does not fully comply with both of them. Small-sized plans are still selected over larger ones, but containing only filtered mappings is considered now a higher-priority criterion than confidence. Although it is untypical of the extended method to produce a plan that is not

Procedure Repair$_3$
Input: *List*: Ordered classes; $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_M$ Horn-propositional theories.
Output: $\mathcal{P}_M$: set of repaired mappings

```
1: for each $C \in List$ do
2:     $\mathcal{P}_C := \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_M \cup \{\mathsf{true} \rightarrow C\}$
3:     $\langle sat, \mathcal{P}_{act} \rangle := \mathsf{DowlingGallier}(\mathcal{P}_C)$
4:     if $sat = \mathsf{false}$ then
5:         $repair\_size := 1$
6:         $\mathcal{R} := \emptyset$
7:         $\mathcal{P}_{act'} := \mathsf{FilterMappings}(\mathcal{P}_{act})$
8:         $sat := \mathsf{true}$
9:         repeat
10:            if filter then
11:                $potential\_plans := \{\mathcal{R} : \mathcal{R} \subseteq \mathcal{P}_{act'}$ and $|\mathcal{R}| = repair\_size\}$
12:            else
13:                $potential\_plans := \{\mathcal{R} : \mathcal{R} \subseteq \mathcal{P}_{act}$ and $|\mathcal{R}| = repair\_size$
14:                                      and $\mathcal{R} \notin potential\_plans\}$
15:            end if
16:            $ordered\_plans := \mathsf{OrderPlans}(potential\_plans)$
17:            repeat
18:                $\mathcal{R}_{candidate} :=$ next element in $ordered\_plans$
19:                $sat := \mathsf{DowlingGallier}(\mathcal{P}_C \setminus \mathcal{R}_{candidate})$
20:                if $sat = \mathsf{true}$ then
21:                    $\mathcal{R} := \mathcal{R}_{candidate}$
22:                end if
23:            until ($\mathcal{R} \neq \emptyset$ or $ordered\_plans$ is empty)
24:            if $mappings = \mathcal{P}_{act}$ then
25:                $repair\_size := repair\_size + 1$
26:                $mappings := \mathcal{P}_{act'}$
27:            else
28:                $mappings := \mathcal{P}_{act} \setminus \mathcal{P}_{act'}$
29:            end if
30:            if filter then
31:                $filter := \mathsf{false}$
32:            else
33:                $repair\_size := repair\_size + 1$
34:                $filter := \mathsf{true}$
35:            end if
36:        until $\mathcal{R} \neq \emptyset$
37:        $\mathcal{P}_M := \mathcal{P}_M \setminus \mathcal{R}$
38:    end if
39: end for
40: return $\mathcal{P}_M$
```

Table 5.3: Enhanced repair algorithm extended with mappings pre-filtering. A call to FilterMappings returns a subset of $\mathcal{P}_{act}$ containing only mappings presenting desired properties.

lowest-confidence, it is worth acknowledging this fact in order to account for minor differences in LogMap's results. Evaluation details are given in section 5.4.

5.3.1 Conflict likelihood filtering

Let $\mathcal{A}_1 \equiv \mathcal{B}_1$ and $\mathcal{A}_2 \equiv \mathcal{B}_2$ be two mappings where $\mathcal{A}_1$, $\mathcal{A}_2$are classes from $\mathcal{O}_1$ and $\mathcal{B}_1$, $\mathcal{B}_2$are classes from $\mathcal{O}_2$. For cases where $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and $\mathcal{B}_1 \perp \mathcal{B}_2$, the mappings are in direct conflict, and at least one of them will be included in any repair plan. For cases where $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and $\mathcal{B}_1$and $\mathcal{B}_2$belong to independent branches in the hierarchy, mappings are not necessarily in conflict but are likely to create one, and it is probable that one of them will be included in most repair plans.

The conflict likelihood filter performs the described check over all pairs of mappings in the conflicting set returned by Dowling and Gallier. Any mapping reported as likely to be involved in a conflict is included in the filtered output set.

5.3.2 Confidence filtering

Intuitively, mappings with high confidence scores are considered 'better mappings', and it is therefore preferable not to delete them. Additionally, the plan-selection policy prioritizes low-confidence plans, which typically will not consist of high-confidence mappings. Therefore, it is reasonable to check first if there is a plan without considering high-confidence mappings.

For this purpose, a very basic statistical analysis is performed on the confidence scores of the conflicting mappings in order to obtain their median and standard deviation. Then, a threshold is obtained from this parameters (e.g., $threshold = \mathsf{median}(confidences) - \mathsf{sd}(confidences)^1$), instead of being arbitrarily set. This way it is ensured that confidence values are only considered to be high if they are high with respect to most of the other conflicting mappings. Every mapping is compared against the obtained threshold, and then those with confidence values below it are included in the filtered output set.

Ontologies	**Repair1 (s)**	**Repair2 (s)**	**Repair3 (s)**
Anatomy	2	2	2
FMA-NCI	36	36	35
FMA-SNOMED	221	182	189
NCI-SNOMED	408	388	381

Table 5.4: Evaluation of enhanced versions of repair algorithms (execution times exclude the indexation stages). Algorithms labelled as follows: *Repair1* = Original repair (non-enhanced), *Repair2* = Enhanced version (no filtering), *Repair3* = Enhanced version with filtering.

5.4 Evaluation

Table 5.4 compares the obtained results from the evaluation of the different versions of the repair algorithm. Times were measured for the matching stages (computation of initial anchors plus expansion), excluding the indexation stages which are equal for all methods.

Results show that the enhanced version which order plans before testing for satisfiability (*Repair2*) succeeds in reducing execution times with respect to the original algorithm saving up to 18% of the original time for FMA-SNOMED. The version extended with mapping pre-filtering (*Repair3*) performs quite similarly to the one not including mapping pre-filtering.

In light of the shown results, it can be concluded that the enhanced repair with plan-ordering improves the original repair algorithm in LogMap. Regarding the extended version with mapping-filtering, it appears to be neither beneficial nor unfavorable with respect to its non-extended counterpart. It remains as possible future work to conduct research on whether a better adjusted filtering would improve the results.

[1]This is the threshold used in the current implementation. Using thresholds of the form $threshold = \mathsf{median}(confidences) - k \times \mathsf{sd}(confidences)$ allows experimenting with more aggressive ($k > 1$) or permissive ($k < 1$) filtering.))

Chapter 6

Conclusions and contributions

This chapter resumes the conclusions obtained after the finalization of the thesis from the obtained results and offers an account of the contributions of this thesis. The following sections address each of the stages of LogMap for which proposals were presented.

6.1 Lexical indexation

The proposed extensions for the lexical indexation stage consist on the use of WordNet synonyms and stemming algorithms:

- Application of the **WordNet** extension [1] has been found to produce a small increase in the recall of system. However, it presents major drawbacks such as a very significant drop of precision and a huge increase in computational costs for large ontologies, hindering the system's scalability. It is therefore not recommended for permanent integration in LogMap, although it remains as an available resource for the project which can be enabled for specific cases (e.g. small or non-specialized ontologies).
- It has been shown that enriching the lexical indices with **stemming** succeeds in improving the system's recall. However, the technique also involves a moderate loss of precision that is only compensated in specific cases, such as the OAEI benchmark Anatomy track (for which non-aggressive stemming improves the F-measure in 0.57%). A number of stemming algorithms have been integrated into LogMap, from

[1]The external tool *WordNetFetcher* was implemented for this extension.

which Porter2 [20] has been determined to be the most balanced. As a beneficial side-effect, aggressive stemming has been found to improve the system's performance by reducing execution times in up to 20% for NCI-SNOMED. It remains as a component in the system and its use is recommended for the above mentioned specific cases (or for those in which recall is prioritized over precision).

6.2 Structural indexation

Regarding structural indexation, an integrated library implementing the interval-labelling schema proposed by [24] has been built to substitute for the former one [23]. The library has been optimized for storing ontology hierarchies, and has been shown to produce a large efficiency boost for the task of structural indexation. For instance, SNOMED was indexed by the previous library in 961s, while the new takes 31s to do it (97% time reduction).

6.2.1 Participation in the OAEI 2011 Campaign

The Ontology Alignment Evaluation Initiative[2] [3] carries out annual campaigns for the evaluation of ontology matching tools, focusing primarily on concept matching although instance matching has recently been included as well. The initiative provides an assessment benchmark for ontology alignment, including test cases and associated gold standards.

Before the new interval-labelling schema library (written in Java) was implemented, LogMap relied on a Python library, run as an external process which communicated with the system via sockets. The fact that the new library integrates with the system into a single component allowed LogMap to qualify for entering the OAEI 2011 ontology matching contest.

Tables 6.1 and 6.2 compare LogMap to the top participating tools in the OAEI 2010 campaign for the Anatomy and Conference tracks. Precision, recall and F-measure scores are shown for both, and an incoherence score which measures the amount of logical errors present in the established mappings is shown for Conference. In the light of the good positioning of LogMap with respect to competing tools in these tracks (second for Anatomy and first for Conference), it is reasonable to expect good results on the 2011 edition taking place in the near future.

Publication of the participation of LogMap in the OAEI 2011 Campaign is pending [15].

[2] `http://oaei.ontologymatching.org/`

Systems	**Precision**	**Recall**	**F-score**
AgrMaker	0.903	0.853	0.877
LogMap	0.934	0.803	0.864
Ef2Match	0.955	0.781	0.859
NBJLM	0.920	0.803	0.858
SOBOM	0.949	0.778	0.855
BLOOMS	0.954	0.731	0.828

Table 6.1: Comparing LogMap with the top 5 tools in the Anatomy track of the OAEI 2010

Systems	**Precision**	**Recall**	**F-score**	**Incoherence**
LogMap	0.85	0.54	0.66	0%
CODI	0.86	0.48	0.62	0.1%
ASMOV	0.57	0.63	0.60	5.6%
Ef2Match	0.61	0.58	0.60	7.2%
Falcon	0.74	0.49	0.59	>4.8%
AgrMaker	0.53	0.62	0.58	>14.8%

Table 6.2: Comparing LogMap with the top 5 tools in the Conference track of the OAEI 2010. Incoherence degree for Falcon and AgrMaker is not complete due to a timeout in the mapping incoherence evaluation algorithm.

6.3 Mapping repair and discovery

Two enhanced versions for the repair algorithm of LogMap are presented in this thesis: the first one (*Repair2*) implements an optimization for the plan-selection method and the second one (*Repair3*) extends it by pre-filtering involved mappings. The proposed optimization was shown to significantly reduce execution times of the matching stages[3] of the algorithm, up to 18% for FMA-SNOMED. However, the mapping filtering feature was found not to improve the system's performance beyond these results. The possibility of considering different parameters for the filtering method remains open for further research.

[3]Excluding lexical and structural indexation

References

[1] University of Oxford Information Systems Group, Department of Computer Science. Logmap project. Technical report available at: `http://www.cs.ox.ac.uk/isg/projects/LogMap/`.

[2] Ernesto Jiménez-Ruiz and Bernardo Cuenca Grau. Logmap: Logic-based and scalable ontology matching. In *10th International Semantic Web Conference (in press)*, 2011.

[3] J. Euzenat, C. Meilicke, H. Stuckenschmidt, P. Shvaiko, and C. Trojahn. Ontology Alignment Evaluation Initiative: six years of experience. *J Data Semantics*, 2011.

[4] Silvana Castano, Alfio Ferrara, Stefano Montanelli, and Gaia Varese. Ontology and instance matching. In *Knowledge-Driven Multimedia Information Extraction and Ontology Evolution*, pages 167–195, 2011.

[5] Christiane Fellbaum. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication)*. The MIT Press, 1998.

[6] F. Hamdi, B. Safar, N.B. Niraula, and C. Reynaud. Taxomap in the oaei 2009 alignment contest. In *Fourth International Workshop on Ontology Matching*, 2009.

[7] Isabel F. Cruz, Cosmin Stroe, Michele Caci, Federico Caimi, Matteo Palmonari, Flavio Palandri Antonelli, and Ulas C. Keles. Using agreementmaker to align ontologies for oaei 2010. In *Fifth International Workshop on Ontology Matching*, 2010.

[8] Yves R. Jean-Mary, E. Patrick Shironoshita, and Mansur R. Kabuka. Asmov: Results for oaei 2010. In *Fifth International Workshop on Ontology Matching*, 2010.

[9] Yves R. Jean-Mary, E. Patrick Shironoshita, and Mansur R. Kabuka. Ontology matching with semantic verification. *J Web Semantics*, 7(3):235–251, 2009.

[10] Quentin Reul and Jeff Z. Pan. Kosimap: Use of description logic reasoning to align heterogeneous ontologies. In *23rd International Workshop on Description Logics*, 2010.

[11] Quentin Reul and Jeff Z. Pan. Kosimap: Ontology alignments results for oaei 2009. In *Fourth International Workshop on Ontology Matching*, 2009.

[12] Jan Noessner and Mathias Niepert. CODI: Combinatorial optimization for data integration results for OAEI 2010. In *Proc. of OM Workshop*, 2010.

[13] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for working with OWL 2 ontologies. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions, OWLED*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[14] Giorgos Stoilos, Giorgos B. Stamou, and Stefanos D. Kollias. A string metric for ontology alignment. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 624–637, 2005.

[15] Ernesto Jimenez-Ruiz, Anton Morant, and Bernardo Cuenca Grau. LogMap results in the OAEI 2011. In *To be presented in ISWC Ontology matching workshop*, 2011.

[16] Olivier Bodenreider, Terry F. Hayamizu, and et al. Of mice and men: Aligning mouse and human anatomies. In *AMIA Annu Symp Proc*, pages 61–65, 2005.

[17] Olivier Bodenreider. The Unified Medical Language System (UMLS): integrating biomedical terminology. *Nucleic acids research*, 32, 2004.

[18] Ernesto Jiménez-Ruiz, Bernardo Cuenca Grau, Ian Horrocks, and Rafael Berlanga. Logic-based assessment of the compatibility of UMLS ontology sources. *J Biomed. Sem.*, 2, 2011.

[19] M.F. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14(3):130–137, 1980.

[20] M.F. Porter. Snowball: A language for stemming algorithms. `http://www.snowball.tartarus.org/texts/introduction.html`.

[21] C.D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.

[22] JB Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[23] Victoria Nebot and Rafael Berlanga. Efficient retrieval of ontology fragments using an interval labeling scheme. *Inf. Sci.*, 179(24):4151–4173, 2009.

[24] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec.*, 18:253–262, 1989.

[25] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On labeling schemes for the Semantic Web. In *Proc. of WWW*, pages 544–555. ACM, 2003.

[26] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, pages 267–284, 1984.