UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Comparison of Concept Learning Algorithms

With Emphasis on Ontology Engineering for the Semantic Web

Diplomarbeit

Leipzig, January 17, 2008

Betreuer:
Prof. Dr. Klaus-Peter Fähnrich
Jens Lehmann

vorgelegt von

Sebastian Hellmann
geb. am: 14.03.1981

Studiengang Informatik

Abstract

In the context of the Semantic Web, ontologies based on Description Logics are gaining more and more importance for knowledge representation on a large scale. While the need arises for high quality ontologies with large background knowledge to enable powerful machine reasoning, the acquisition of such knowledge is only advancing slowly, because of the lack of appropriate tools. Concept learning algorithms have made a great leap forward and can help to speed up knowledge acquisition in the form of induced concept descriptions. This work investigated whether concept learning algorithms have reached a level on which they can produce result that can be used in an ontology engineering process. Two learning algorithms (YinYang and DL-Learner) are investigated in detail and tested with benchmarks. A method that enables concept learning on large knowledge bases on a SPARQL endpoint is presented and the quality of learned concepts is evaluated in a real use case. A proposal is made to increase the complexity of learned concept descriptions by circumventing the Open World Assumption of Description Logics.

Contents

1 Introduction and Preliminaries

Current advances of the Semantic Web have created many new problems, that need to be solved in order to fully use the advantages, the Semantic Web Vision of Tim Berners-Lee [9] has to offer. The standardization of formats is almost concluded and the theoretical foundations are fortified. Basic tools for editing and viewing of semantically annotated data have matured recently. and applications are being developed that further use the new possibilities.

One of the major problems that the Semantic Web is currently facing, though, is the lack of structured knowledge in form of ontologies. Existing data has to be converted to the new standards on a large scale to match the new paradigm and new ontologies have to be created. A widespread acceptance of ontologies as a way to represent information has yet not been achieved, which is due to the fact that ontologies are difficult to create and require a new way of thinking in concepts and graphs. For the forthcoming of the Semantic Web, applications are needed to ease the creation of ontologies with rich background knowledge to fully unlock the potential of Tim Berners-Lee's vision, in which machine reasoning plays a powerful role.

Concept learning in Description Logics has made major advances and the existing, implemented algorithms are now at the threshold to become applications that lower the complexity of creating background knowledge. This work investigates the above mentioned problems and tries to cover all aspects, that are important to step over that threshold. It not only compares the current approaches to concept learning, but tries to analyze problems in detail and also provides some solutions.

After the preliminaries in this section, we will identify problems that occur during the creation of ontologies in Section 2 and how concept learning algorithms can be used to tackle these problems 2.2. We will also give a solution how concept learning can be applied to large knowledge bases in Section 2.3. Thereafter we will examine the most prominent existing learning algorithms in Section 3 and provide a solution for general problems all approaches still have to overcome (Section 3.4). In Section 4, the currently implemented algorithms will be compared even closer with benchmarks and in the end (Section 5), we will give an outlook about how concept learning algorithms can be used in an ontology creation scenario.

1.1 Semantic Web: Vision, OWL, SPARQL

Since its creation in 1990, the World Wide Web has grown rapidly. This rapid growth comes along with new problems. The original World Wide Web was designed as a loose, distributed collection of documents, which are connected through Hyperlinks and can be reached via the Internet. Although it can be seen as a universal source of information, which is readily accessible by humans, it does not contain structured information, which can be efficiently analyzed by machines. The best example for the shortcomings of this design might be a Google search. Google is a highly optimized search engine with immense resources, yet it only presents a list of links with a short description, which has to be reviewed manually by the human, who ran the search. A special challenge for humans is also the correct guessing of keywords that might lead to relevant web pages. In 1998, Tim Berners-Lee created a document with the title *Semantic Web Road Map*. We can not find better words to describe the reasons for and the aim of the Semantic Web and thus prefer to quote the following sentences from the introduction:

> The Web was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help. One of the major obstacles to this has been the fact that most information on the Web is designed for human consumption, and even if it was derived from a database with well defined meanings (in at least some terms) for its columns, that the structure of the data is not evident to a robot browsing the web. Leaving aside the artificial intelligence problem of training machines to behave like people, the Semantic Web approach instead develops languages for expressing information in a machine processable form.
>
> This document gives a road map - a sequence for the incremental introduction of technology to take us, step by step, from the Web of today to a Web in which machine reasoning will be ubiquitous and devastatingly powerful.
>
> Tim Berners-Lee[1]

In 2001, Berners-Lee et al. further concretized these ideas in his often cited article *The Semantic Web* [9]. Now, almost 10 years later, these ideas have taken an even more concrete form and the standardization of the underlying formats (RDF, SPARQL, OWL) is about to be complete. Figure 1 shows the current architecture of the Semantic Web according to the World Wide Web Consortium (W3C)[2].

We will give a short description of the layers of the Semantic Web in Figure 1, that are the most important for this work and skip the more basic knowledge like the Resource Description Framework (RDF)[3]. These relevant layers include URIs and IRIs, the Web Ontology Language (OWL) and the SPARQL Protocol and RDF Query Language (SPARQL).

[1] http://www.w3.org/DesignIssues/Semantic.html

[2] http://www.w3c.org

[3] We refer the reader to http://www.w3.org/2001/sw/ which gives a complete overview of the current Semantic Web technologies

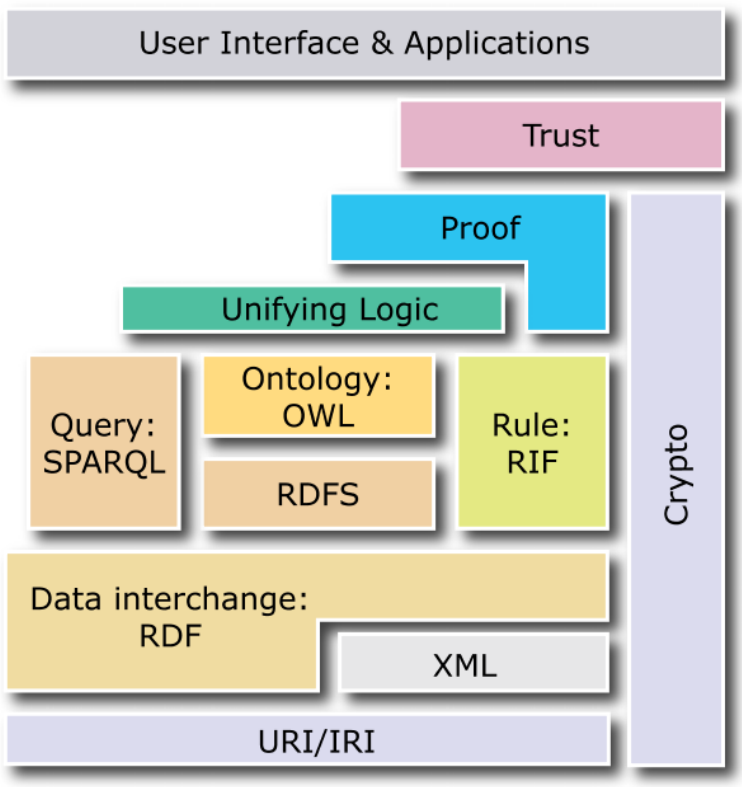


Figure 1: Latest "layercake" diagram, taken from http://www.w3.org/2001/sw/ .

URIs and IRIs[4] are well-known from the World Wide Web and are used in the Semantic Web as string representation of objects or resources in general. These objects can be virtually anything from real existing things like animals to Web pages to categories in a hierarchy or anything else[5]. They solve the problem of ambiguity of natural language, because it is possible to distinguish meaning. The term $Jaguar$ in natural language will always denote both meanings without a given context, a car or an animal, while the denotational meaning of the two URIs *http://www.cars.org/Jaguar* and *http://www.zoo.org/Jaguar* can have two different meanings, because they are different terms. Without going further into detail, we just want to mention the commonly used method to abbreviate IRIs (IRIs can contain Unicode characters, while URIs contain ASCII only). *http://www.w3.org/1999/02/22-rdf-syntax-ns#type* for example is normally abbreviated as *rdf:type*.

[4]Uniform Resource Identifier and Internationalized Resource Identifier, see http://www.w3.org/Addressing/

[5]recent proposals try to distinguish real world objects from Web pages with further conventions for URIs, cf. http://www.w3.org/TR/2007/WD-cooluris-20071217/ on this issue

OWL[6] has become a W3C Recommendation in February 2004. It is seen as a major technology for the future implementation of the Semantic Web. It is especially designed to allow shared ontologies and provide a formal foundation for reasoning. OWL is an extension of XML, RDF and RDF Schema and facilitates greater machine interpretability for applications. It comprises of three sublanguages with increasing expressivity. The three sublanguages are: OWL Lite, OWL DL, and OWL Full. Its vocabulary can be directly related to the formal semantics of Description Logics (cf. Section 1.2), but is extended to support practical issues like versioning and shared ontologies.

SPARQL[7] has become a W3C Candidate Recommendation in June 2007. SPARQL is the most important query language for RDF data and is developed by the Data Access Working Group of the W3C. Its syntax is similar to that of SQL and it can be used to query a RDF knowledge base[8] with triple and graph patterns, that contain query variables that are bound to matching RDF Terms (an RDF Term is a part of a triple). Example 1 shows how a simple query, that retrieves all instances for a class $Person$, might look like.

Example 1 (Syntax of a SPARQL query)
The query

```
SELECT ?instance
  WHERE { ?instance a <http://www.example.com/example#Person>  }
```

returns a list of instances that belong to the class Person. "a" is a built-in abbreviation for <rdf:type>, which is as mentioned above an abbreviation for: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>.

1.2 Description Logics

Description Logics is the name for a family of knowledge representation(KR) formalisms. They originated from semantic networks and frames that did not have formal semantics, which has changed. The most common representative is $\mathcal{ALC}$ (Attribute Language with Complement), which was first described in [33]. Although Description Logics are less expressive than full first-order logic, they have certain properties, that admitted them to become a popular knowledge representation formalism, especially in the context of the Semantic Web. The main advantages are that they usually have decidable inference problems and a variable free syntax, which can be easily understood. OWL (especially

[6]http://www.w3.org/TR/owl-features/
[7]http://www.w3.org/TR/rdf-sparql-query/
[8]Note that OWL is compatible with RDF (every OWL knowledge base is a RDF-S knowledge base is a RDF knowledge base)

the sublanguage OWL DL) is directly based on Description Logics and thus uses the same formal semantics (see below for a mapping). With the increasing popularity of the Semantic Web, the interest in Description Logics has also increased, which even resulted in the fact that DL knowledge bases are "nowadays often called ontologies" (as noted by Baader et al. [3]).

Description Logics represent knowledge using *objects* (also called *individuals*), *concepts* and *roles*, which correspond to constants, unary predicates and binary predicates in first-order logic. A knowledge base is normally divided in a $TBox$ and an $ABox$, whereas the $TBox$ introduces the *terminology* and the $ABox$ contains *assertions* about named individuals based on the terminology. Concepts can be interpreted as sets of individuals and roles denote binary relationships between individuals. For completeness, we included the Definition 1 for the syntax of constructing complex concepts and Definition 2 for the interpretation. Table 1 shows, how the interpretation function can be applied to complex concept descriptions. The definitions and the table were taken from Lehmann [23].

Definition 1 (syntax of $\mathcal{ALC}$ concepts)
Let N_R be a set of role names *and N_C be a set of concept names ($N_R \cap N_C = \emptyset$). The elements of the latter set are called* atomic concepts. *The set of $\mathcal{ALC}$ concepts is inductively defined as follows:*

1. *Each atomic concept is a concept.*
2. *If C and D are $\mathcal{ALC}$ concepts and $r \in N_R$ a role, then the following are also $\mathcal{ALC}$ concepts:*
   - *$\top$ (top), $\bot$ (bottom)*
   - *$C \sqcup D$ (disjunction), $C \sqcap D$ (conjunction), $\neg C$ (negation)*
   - *$\forall r.C$ (value/universal restriction), $\exists r.C$ (existential restriction)*

Definition 2 (interpretation)
An interpretation $\mathcal{I}$ *consists of a non-empty* interpretation domain $\Delta^{\mathcal{I}}$ *and an* interpretation function $\cdot^{\mathcal{I}}$, *which assigns to each $A \in N_C$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to each $r \in N_R$ a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.*

While Baader et al. [5] give an exhaustive description, we will only mention the basic terms we will use in this work.

Of special importance is the possibility to create concept hierarchies using the *subsumption* operators $\sqsubseteq$ and $\sqsupseteq$, which play an important role in ontology engineering, since they enable concept hierarchies. They create a quasi-ordering over the space of concepts, where $\top$ is always the most *general* concept, while all concepts subsumed by $\top$ ($C \sqsubseteq \top$) are more *specific*, see Definition 3.

construct	syntax	semantics
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
top concept	$\top$	$\Delta^{\mathcal{I}}$
bottom concept	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
exists restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{a \mid \exists b.(a,b) \in r^{\mathcal{I}}$ and $b \in C^{\mathcal{I}}\}$
value restriction	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \{a \mid \forall b.(a,b) \in r^{\mathcal{I}}$ implies $b \in C^{\mathcal{I}}\}$

Table 1: $\mathcal{ALC}$ semantics

Definition 3 (subsumption, equivalence)
Let C, D be concepts and $\mathcal{T}$ a TBox. C is subsumed by D*, denoted by $C \sqsubseteq D$, iff for any interpretation $\mathcal{I}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.* C is subsumed by D with respect to $\mathcal{T}$ *(denoted by $C \sqsubseteq_{\mathcal{T}} D$) iff for any model $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.*

C is equivalent to D (with respect to $\mathcal{T}$)*, denoted by $C \equiv D$ ($C \equiv_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and $D \sqsubseteq C$ ($D \sqsubseteq_{\mathcal{T}} C$).*

C is strictly subsumed by D (with respect to $\mathcal{T}$)*, denoted by $C \sqsubset D$ ($C \sqsubset_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and not $C \equiv D$ ($C \equiv_{\mathcal{T}} D$).*

Basic *reasoning* tasks, which aim at making implicit knowledge explicit, include, among others, *subsumption* to verify if one concept is a subconcept of another concept, *instance* checks to verify if an individual a is an instance of a concept C w.r.t. an $ABox$ $\mathcal{A}$ and *retrieval* to find all individuals that belong to a concept C, such that $\mathcal{A} \models C(a)$ for all individuals a.

Example 2 *The knowledge base $\mathcal{K}$ consisting of the $TBox$ $\mathcal{T}$ and $ABox$ $\mathcal{A}$ ($\mathcal{K} = \{\mathcal{T}, \mathcal{A}\}$) defines the application domain of a father and his two sons and another man. The $TBox$ defines the vocabulary and the relevant axioms, while the $ABox$ defines extensional facts about the domain.*
$\mathcal{T}$ = \{ $Male \sqsubseteq \top$, $Father$ = $Male \sqcap \exists\, hasChild.\top$ \}
$\mathcal{A} = \{$ $Male$ *(John_Senior)* , $Male$ *(Peter)* , $Male$ *(John_Junior1)* , $Male$ *(John_Junior2)* , $hasChild$ *(John_Senior, John_Junior1)*, $hasChild$ *(John_Senior, John_Junior2)* $\}$
Based on the knowledge base, we can infer by reasoning e.g. that $John\_Senior$ *is a* $Father$ *and also that* $Father$ *is a more specific concept than* $Male$ *(*$Father \sqsubset Male$ *).*

Description Logics adopt the Open World Assumption (OWA), which is different from Negation as Failure (NAF). Therefore, we can not deduct $\forall$ $hasChild.Male$ (John_Senior) from the knowledge base in Example 2, because the knowledge base does not explicitly contain the information that John_Senior only has male children; the knowledge is assumed to be *incomplete*.

The Web Ontology Language (OWL) can be directly mapped to certain extensions of $\mathcal{ALC}$. The two sublanguages (OWL Lite and OWL DL) are based on $\mathcal{SHIF}$ (D) (OWL Lite) and $\mathcal{SHOIN}$ (D) (OWL DL) and will often use the term ontology, DL knowledge base, OWL ontology as synonym. $\mathcal{SHOIN}$ (D) and OWL DL are of special interest since they allow for the most expressivity, while all reasoning tasks are still decidable. $\mathcal{SHOIN}$ (D) is shorthand for the combination of $\mathcal{ALC}$ and role hierarchies ($\mathcal{H}$), nominals ($\mathcal{O}$), role inverses ($\mathcal{I}$), unqualified number restriction ($\mathcal{N}$) and basic data types (D).

1.3 Learning Problem in Description Logics

Although some authors (cf. Section 3.2 or [20]) adapt a slightly different definition, we used the Definition in 4 in this thesis, because it was commonly used throughout the literature and describes the core problem. For a given knowledge base $\mathcal{K}$, a concept description C shall be found, based on assertional observations, i.e. a set of individuals, where each individual is either labeled "positive" or "negative" (supervised learning). If we add the axiom ($Target \equiv C$) to the $TBox$ of $\mathcal{K}$, resulting in $\mathcal{K}$ ', then the individuals labeled positive shall follow from $\mathcal{K}$ ' and negatives should not.

Definition 4 (learning problem in Description Logics)
Let a concept name $Target$, a knowledge base $\mathcal{K}$ (not containing $Target$), and sets E^+ and E^- with elements of the form $Target(a)$ ($a \in N_I$) be given. The learning problem is to find a concept C such that $Target$ does not occur in C and for $\mathcal{K}' = \mathcal{K} \cup \{Target \equiv C\}$ we have $\mathcal{K}' \models E^+$ and $\mathcal{K}' \not\models E^-$.

In Definition 5 some shortcuts are defined. We will also use the term *coverage*. A correct, learned concept definition covers all positive examples and does not cover the negative examples.

Definition 5 (complete, consistent, correct)
Let C be a concept, $\mathcal{K}$ the background knowledge base, $Target$ the target concept, $\mathcal{K}' = \mathcal{K} \cup \{Target \equiv C\}$ the extended knowledge base, and E^+ and E^- the positive and negative examples.

C is complete *with respect to E^+ if for any $e \in E^+$ we have $\mathcal{K}' \models e$. C is* consistent *with respect to E^- if for any $e \in E^-$ we have $\mathcal{K}' \not\models e$. C is* correct *with respect to E^+ and E^- if C is complete with respect to E^+ and consistent with respect to E^-.*

1.4 DL-Learner Framework

In this section we will give an overview of the DL-Learner framework, because we actively took part in the development during the course of this work. It recently became an Open-Source project and is available on Sourceforge[9]. Its core is divided in 4 components, which are knowledge sources like OWL-files or the SPARQL extraction module (cf. Section 2.3), reasoners, which e.g. use the DIG[10] reasoning interface (see [8] for details), learning problems and learning algorithms like the one based on refinement, which we present in Section 3.2.3. The DL-Learner also has a Web Service interface for integration in other applications like OntoWiki[11].

2 Automated Ontology Engineering

Fast, cheap and correct construction of domain-specific ontologies[12] is crucial for the forthcoming of the Semantic Web. Although the number is growing, only few ontologies exist[34] and tool support is still rare for other tasks than manually editing (see Swoop[13] or Protégé[14]). Maedche and Staab are calling it a knowledge acquisition bottleneck [28]. We will first give a short overview about problems when designing ontologies, then we will look at approaches for automatic ontology generation. Based on this we will show, why ontology engineering can greatly benefit from concept learning and how this can help to create more ontological knowledge faster and easier. Finally, we will give our own approach to enable concept learning on large, "unwieldy" ontological knowledge bases.

2.1 Current Problems of Ontology Engineering

Ontology creation is in general considered a difficult task. The errors that can be made are numerous and we will try to name the most important here.

The most prominent problem is probably the fact, that it is not yet clear, how knowledge can be lifted to an ontological representation. There does not exist a standard, which can tell what the best practice looks like (cf. [34] on this issue). The fact that the Semantic Web community[15] adopted RDF and OWL and therefore Description Logics as standard formalism might mislead into thinking there also exist best practises for ontology design. There is a big argument, ongoing since decades, about how to model existing data and the real world in ontologies. We refer here especially to the discussions in between the creators of the following TOP-level ontologies[16]: DOLCE, GFO, Cyc,

[9]http://dl-learner.org/Projects/DLLearner and http://sourceforge.net/projects/dl-learner

[10]http://dl.kr.org/dig/

[11]http://aksw.org/Projects/OntoWiki

[12]the discussion is concerned with Semantic Web ontologies, when we use the term *ontology* here, we mean an ontology based on OWL and DL, excluding generality, although some parts might scale to a wider scope.

[13]http://www.mindswap.org/2004/SWOOP/

[14]http://protege.stanford.edu/

[15]we refer here to the World Wide Web Consortium, as the major institution for standardization, http://www.w3.org/

[16]Instead of providing links for each entry we would like to refer to http://en.wikipedia.org/wiki/Upper_ontology_(computer_science), which gives an up-to-date link collection

WordNet and also there seems to be a discussion, if ontologies are needed at all to solve complex problems[17]. To give a brief example of how difficult it is to lift simple real world facts and processes to an ontological representation, we would like to mention a much discussed sentence: ”She drew a line with her fountain pen until there was no more ink left.” If we assume that there is exactly one point in time (modelled as real numbers) at which the ink is used up, then we have to admit that at this point there is still ink in the pen and at the same time there is no ink any more. If we assume that there are two time points, one which ends the interval of the pen still containing ink and one which starts the interval, where the pen is empty, then we have to admit that there are infinitely many time points in between, where there is an undefined pen-ink-level state. Without solving this intriguant matter here, we refer the reader to Herre et al.[18], chapter 5.1. (Time), for an interesting solution.

The next problem, which is one step further down in the hierarchy of problems, is the creation of domain specific ontologies. Although there still might be the same problems as in TOP-Level ontologies, normally they are one size smaller, because domain specific ontologies do not need to cover everything, but just a small domain of the world. Occurring problems can also be circumvented more easily by creating solutions that serve the practical purpose of the ontology, but still would not seem to be correct, i.e. matching the domain knowledge from an intuitive view (cf. the discussion about the "4-Wheel-drive" in Section 5).

The creation of an ontology from scratch requires immense resources[18]. Not only that existing data has to be converted to match the form[19], it has to be given ontological structure. Even the decision about simple matters can be quite complex. An example can be found in Section 5. We just name the problem whether an existing dataset shall be modelled as an individual or a concept or even a role. Numerous tools exist, which aim at supporting these decisions like for example the conversion of text [12] [20] or the transformation of existing relational databases [26].

Although those tool ease the burden of a knowledge engineer, the complex problems (like creating a subsumption hierarchy, defining concepts, ontology merging) are not well supported, which basically amounts to the necessity of manually post-processing the extracted ontology. For this post-processing step, there exist numerous requirements a knowledge engineer[21] has to meet. He needs to have at least some profession in logical formulas to understand the assumptions (e.g. the UNA and the OWA[22]) that are presupposed in Description Logics and he also has to know about the meaning of OWL constructs and their implications. Furthermore he not only needs to have knowledge about the domain, but also has to think in an analytic way, because he needs to analyze the structure of the

[17]remark: try a local search on Google or Yahoo, they work quite well already, although no ontologies are used.

[18]with resources we mean time and knowledge

[19]actually the simple conversion is the least problem

[20]http://olp.dfki.de/OntoLT/OntoLT.htm

[21]in practical terms this could also be a team of engineers, each specialized in a certain field

[22]Unique Name Assumption, Open World Assumption

domain in terms of categories, hierarchies and membership. Without branding him an out-spoken antagonist of ontologies, we would like to cite one of the founders of Google, Sergej Brin, on this matter. When asked about the Semantic Web and RDF at InfoWorld's 2002 CTO Forum, Sergey Brin said:

Look, putting angle brackets around things is not a technology, by itself. I'd rather make progress by having computers understand what humans write, than by forcing humans to write in ways computers can understand.

All these requirements currently encumber the establishment of the Semantic Web in the sense of Tim Berner-Lees Semantic Web Vision[9], because only a small group of people, i.e. mostly the people who research ontology engineering, can successfully create valuable ontologies, while the number of ontologies, which can be found in use in "real-world projects" is still relatively small (cf. the survey in [34]). Instance-based concept learning might be able to change this, because it simplifies the creation and enrichment of ontological knowledge; the difficulty of creating concept definitions is shifted to the simplicity of selecting a set of instances and either accepting a proposed solution or rejecting it to be refined again by a rerun of the learning algorithm. The more this process is automated, the less human intervention is necessary, thus saving time and knowledge needed to create and improve an ontology, making ontology more attractive for a broader audience.

2.2 Use Cases and Requirements for Tool-Support

We loosely define possible use cases here. The practical need to obtain ontologies has only surfaced recently with the rise of the Semantic Web. The requirements which have to be fulfilled by a concept learning tool have yet to be analyzed in detail. We mainly aim at providing suggestions, because it would lead in a completely other direction.

When interacting with a user, usability clearly depends on the knowledge required by the engineer. We assume now that it is easier to define groups of individuals than concepts, because the individuals already exist (or are easier to import from other data sources than e.g. a taxonomy as we show in Section 5) and the concept definitions either need to be created from scratch or have to be reinvented, resulting in the same effort (knowledge needed).

Using learning algorithms, the concept can be learned with a relative small number of examples and can be checked versus the other existing instances (cf. Section 4). This means that quality can be calculated (in its simplest form by manually counting the actual coverage and dividing by wanted coverage, i.e. precision) in numbers and, if necessary, the example set can be altered for iterative runs. This process is far more effective than manually editing an ontology. Providing proposals for concept definitions with the chance to edit them manually or refine them by rerunning the learning program together with a measure of how adequate the solution is, would greatly benefit the process of background knowledge creation.

A completely automatic tool support can be achieved by devising an algorithm that can choose example sets in a smart way, but this belongs more to the field of artificial intelligence, since the algorithm had to guess the meaning of the data (which is prepared by the semantic annotation). There are however certain scenarios, when such an automatic choice is indicated. If new individuals are added to an existing ontology, concept definitions can become inconsistent, covering individuals they should not cover or they could become incorrect, not covering individuals they should. This could be detected automatically and a learning tool can start redefining the concept, based on the previous individuals and the new ones, thus easing maintenance cost.

2.3 SPARQL Component

A significant use case where (semi-)automatic support for concept learning and maintenance of concept consistency could improve quality immensely, is the DBpedia[23] knowledge base. DBpedia extracted information from the Wikipedia MediaWiki templates and thus created a huge RDF knowledge base. In its new Version 2.0 it includes semantic information about 1,950,000 “things” with a total of 103 million RDF triples. Because of this immense size, manual editing has become unaffordable and automated techniques for ontology enrichment need to be used.

We will now give a short description about the existing structure of DBPedia, then we will present an algorithm that makes concept learning possible on large knowledge bases, before we think of possible uses.

The base data of DBpedia was extracted from the MediaWiki templates of Wikipedia, which contain structured information in a Wiki syntax for layout purposes. This layout structure was analyzed and mapped to a Semantic Web structure, which was accumulated in RDF-triples and made freely available like Wikipedia (the process is described in detail in Auer et al. [2]). The Wikipedia categories were used as concepts to group individuals. These categories were assigned by Wikipedia authors on an arbitrary, subjective base and contained many inconsistencies and only sporadic hierarchical information. Later the SKOS[24] vocabulary was used to create a SKOS-taxonomy on the Wikipedia-categories. Note, that SKOS relates instances and categories with its own vocabulary and thus differs from OWL classes.

In a next step YAGO[25] classes were added and assigned to the individuals as well as Word Net

[23]see http://www.dbpedia.org and for an overview of version 2.0 see http://blog.dbpedia.org/2007/09/05/dbpedia-20-released/

[24]http://www.w3.org/2004/02/skos/

[25]http://www.mpi-inf.mpg.de/ suchanek/downloads/yago/

Synset Links. YAGO recently has adopted a hierarchy. Currently, the two main hierarchies in DBpedia, i.e. YAGO and the Wikipedia categories, are only useful to a certain extent. Although YAGO is using the OWL vocabulary, its classes are very general like e.g. $Philosopher$, containing thousands of instances. On the other hand, the Wikipedia categories are very specific like $Ancient\_Greek\_Philosophers$ and still contain inconsistencies. Since they are not OWL classes, but SKOS concepts, it is difficult to combine both, which would result in a preferable hierarchy.

A point that is still missing is the enrichment of DBPedia with concept definitions to better cope with queries and with new individuals and changes (the original Wikipedia is parsed in regular intervals). Due to the immense size of the A-Box, reasoning, can not be conducted efficiently, but research is ongoing (see e.g. InstanceStore[26] or [16]). Although querying DBpedia with SPARQL, already yields new possibilities to retrieve structured information, complex background knowledge combined with proper retrieval algorithms can yet take it to a new level. A simple example, where a concept definition, can improve the structure, shall be given here:

Example 3 *Imagine there exists a concept like $Mother$.*
When using a SPARQL query to retrieve all mothers (all instances of the concept $Mother$), we would only retrieve all instances that are explicitly assigned to the concept $Mother$ with rdf:type.

```
SELECT ?instance WHERE { ?instance a yago:Mother }
```

But if the concept had a concept definition like $Mother = Woman \sqcap \exists hasChild.Person$, we could first retrieve the concept definition and then pose the following SPARQL query to retrieve all instances that match the definition (we use the YAGO classes here):

```
SELECT ?instance
  WHERE {
    ?instance a yago:Woman;
              dbpedia:hasChild ?c,
    ?c a yago:Person }
```

even consistency can be checked, by comparing the results of both sparql queries to this one ($Mother \sqcap Woman \sqcap \exists\ hasChild.Person$):

```
SELECT ?instance
  WHERE {
    ?instance  a yago:Mother;
               a yago:Woman;
```

[26] http://instancestore.man.ac.uk

```
                dbpedia:hasChild ?c,
    ?c a yago:Person }
```

which only retrieves correctly assigned instances.
Note: Since reasoning is so inefficient with such a large database like DBPedia, that it can hardly be conducted, we name the SPARQL queries explicitly here. Normally this would be done by a reasoner automatically. Also we will not retrieve instances that are merely assigned to subclasses of the above stated concepts, which is normally also done by a reasoner.

2.3.1 Circumventing the Problem of Reasoning on Large Knowledge Bases

The currently existing concept learning techniques heavily depend on deductive reasoning (instance checks as mentioned above are used by the DL-Learner, or subsumption checks by YinYang, see Section 3.2). Thus they can not be used efficiently on large knowledge bases. We will provide a method here that circumvents this problem and makes it possible to learn concepts. It served as a basis for Hellmann, Lehmann and Auer [17] submitted to the ESWC 2008, where it is used with the DL-Learner, exclusively. We will first describe the extraction algorithm in general and then use it on DBpedia to learn concepts with the DL-Learner.

Since the learning algorithm induces ontological knowledge, based on extensional information(facts), we use a set of instances as a starting point. The algorithm described in the following will then explore the RDF-graph with SPARQL in such a way that relevant information is extracted, i.e. information that provides a sufficient description of the facts and the corresponding concepts to apply the learning algorithm. The extracted information is filtered and corrected in a way that it complies with the requirements of OWL-DL. In the following we will adopt the RDF/OWL vocabulary, using the terms "classes", "properties" and "instances" instead of "concepts", "roles" and "individual", when it is more appropriate, especially in cases where OWL is used.

2.3.2 Selection of Relevant Information

Starting from the example set given for the learning problem, we start to explore the graph using SPARQL queries recursively up to a given depth. We will define a SPARQL query template with certain parameters, which is used to generate the actual queries in each recursion step.

Definition 6 (Special SPARQL query template using filters)
A SPARQL query of the form SQTF_E *(resource , predicateFilter, objectFilter, literals) takes as input a resource, a list of prohibited URIs for predicates, a list of prohibited URIs for objects and a boolean flag, indicating whether datatype properties should be retrieved. It returns a set of tuples of the form*

(p,o), where (p,o) are the resources (properties and objects) returned from the following SPARQL query on an endpoint E*:*

```
SELECT ?p ?o WHERE {resource ?p ?o .
  FILTER(generateFilter(predicateFilter,objectFilter,literals))}
```

We will simply use SQTF*(resource) when the context is clear.*

Example 4 (Example SPARQL query on DBpedia) *In this example we show, how we filter out triples using SKOS*[27] *and DBpedia categories, but leave YAGO*[28] *classes. Furthermore, FOAF*[29] *is allowed but websites are filtered out. The actual filters used are yet larger; we omitted some for brevity.*
$\text{SQTF}_{DBPedia}$*("http://dbpedia.org/resource/Angela_Merkel", predicateFilter, objectFilter, false) is resolved to:*

```
SELECT ?p ?o WHERE {
  <http://dbpedia.org/resource/Angela_Merkel> ?p ?o.
  FILTER (
    !regex(str(?p),'http://dbpedia.org/property/website')
    && !regex(str(?p),'http://www.w3.org/2004/02/skos/core')
    && !regex(str(?o),'http://dbpedia.org/resource/Category')
    && !isLiteral(?o) ). }
```

The filters are necessary to avoid retrieving information, that is not important to the learning process. The learning algorithm does not make use of datatype properties, thus literals will always be omitted. The predicate filter removes properties that are not important (in the case of DBpedia for example properties that point to web pages and pictures). The same accounts for the object filter. We filter `owl:sameAs` out by default, because it might result in the extracted fragment being in OWL Full (e.g. when it links instances to classes).

More namespaces and URIs can be added manually to the filter depending on the SPARQL endpoint. The choice is important since it determines which information is extracted. If the knowledge base makes use of different subsumption hierarchies like e.g. DBpedia, which uses YAGO classes and the SKOS vocabulary combined with its own categories, a special hierarchy can be selected by excluding the others (as e.g. in Example 4).

[27] http://www.w3.org/2004/02/skos/
[28] http://www.mpi-inf.mpg.de/ suchanek/downloads/yago/
[29] http://xmlns.com/foaf/0.1/

After having defined the filters for the respective SPARQL endpoint, the algorithm (see Algorithm 1) starts to extract knowledge recursively based on each instance (*SQTF(instance)*) in the example set. The objects of the retrieved tuples (p,o) are then used to further extract knowledge (*SQTF(o)*) until a given recursion depth is reached. The algorithm also remembers valuable information that is later used in the conversion to OWL DL, which we will describe in Section 2.3.3. To disburden the SPARQL endpoint, caching is used to remember SPARQL query results which were already retrieved. The clear bottleneck of the extraction algorithm is the SPARQL retrieval via HTTP.

The recursion depth greatly determines the number of triples extracted. A recursion factor of 1 means, that only the directly related instances and classes are extracted. A recursion factor of 2 extracts all direct classes and their super classes and all directly related instances and their direct classes and directly related instances[30](see Figure 2). If we use all existing instances as starting seeds with a sufficient recursion depth, the algorithm will extract the whole knowledge base with the exception of unconnected resources, which in most cases barely contain useful information.

With a depth of 1 the algorithm only extracts directly related resources, which is too scarce for sensible class learning. Depending on the average number of properties, a recursion depth of 2 or 3 represents a good balance between the amount of useful information and the possibility to reason efficiently. This choice as well as the choice of filterlists clearly depends on the SPARQL endpoint and has to be chosen manually.

By allowing some dynamic when decreasing the recursion counter, we can also extract more background knowledge. If the property we retrieved in the tuple (p,o) was e.g. `rdf:type` or `rdfs:subClassOf`, which indicates, that the respective object is a class, we could just not decrease the recursion counter and thus retrieve the whole hierarchy of classes that belongs to the instances; this will only slightly increase the size of the extracted triples, since the number of superclasses normally gets smaller. We can also avoid cycles by setting the recursion counter to 0, when we encounter properties like `owl:equivalentClass`, thus preventing infinite extraction. But if the object is a blank node, we will not decrease the recursion counter until no further blank nodes are retrieved[31]. Thus we can extract complex class definition.

The triples can further be manipulated during collection by user defined rules. There are vocabularies that are similar to OWL class hierarchies, but use different names. The SKOS vocabulary for example uses `skos:subject` instead of `rdf:type` and `skos:broader` instead of `rdfs:subClassOf`. We can convert those hierarchies to OWL DL during retrieval by replacing the strings as mentioned above (e.g. `skos:broader` is replaced by `rdfs:subClassOf`). We

[30] Remark: to enrich the fragment further, we also retrieve other types like *owl:SymmetricProperty* and super properties of the respective properties in separate SPARQL queries, a fact, we omitted here for brevity.

[31] The further retrieval of blank nodes requires a redefinition of the SPARQL query, which is merely a technical matter, not described here in detail

could also use this technique e.g. to convert tags or other structurally important instances to classes to enable class learning.

Algorithm 1: SPARQL Knowledge Extraction Algorithm

```
  Input: set $E$ of instances
  Input: parameters: recursion depth, predicateFilter, objectFilter, literals
  Output: set $T$ of triples
1 define function SQTF using (predicateFilter, objectFilter, literals)
2 $T$ = empty set of triples
3 foreach $e \in E$ do
4     $T = T \cup$ extract (recursion depth, e, "instance") ;
5 return $T$
```

2.3.3 OWL DL Conversion of SPARQL Results

The extracted knowledge has to be converted into OWL DL for processing, which means explicitly typing classes, properties and instances. Since the algorithm can not guarantee to extract all type definitions, if they exist, we chose another approach. We remove all type definitions during extraction and remember the type of the resource, based on the assumption that if the type of the subject is known, we can infer the type of the object by analysing the property. In a triple (s,p,o), we thus know e.g. that o is a class if s is an instance and p is `rdf:type`. We further know that o is an instance for all other properties. If it is the case that s is a class then o must also be a class, unless the SPARQL endpoint is in OWL Full, in which case we transform the property to `rdfs:subClassOf` if it is `rdf:type` and ignore all other properties, except properties from the OWL vocabulary like `owl:equivalentClass`. With the help of these observation we can type all collected resources iteratively, since we know that the starting resources are instances. Since DatatypeProperties are filtered out by default, all properties can be typed `owl:ObjectProperty`.

We thus presented a consistent way to convert the knowledge fragment to OWL DL based on the information collected during the extraction process. The extracted information is a clearly cut out fragment of the larger knowledge base. It contains relevant information, since we use an instance based approach to class learning, hence information can be considered relevant, if it is directly related to the instances. Due to its comparatively small size, deductive reasoning can now be applied efficiently, thus enabling to apply concept learning techniques. The fact that it is now OWL DL also renders all reasoning queries decidable.

Function `extract` (*recursion counter, resource,typeOfResource*)

```
if recursion counter equals 0 then
    return $\emptyset$
$S$ = empty set of triples;
resultSet = SQTF(resource);
newResultSet = $\emptyset$ ;
foreach tuple (p,o) from resultSet do
    // the function manipulate basically replaces strings after user defined rules
    // and evaluates the type of the newly retrieved resources
    newResultSet= newResultSet $\cup$ manipulate(typeOfResource,p,o);
create triples of the form (resource,p,o) from the newResultSet ;
add triples of the form (resource,p,o) to $S$;
foreach tuple (p,o) from the newResultSet do
    // the extended flag determines if all superclasses are extracted
    if p is rdf:type or rdfs:subClassof and the extended flag is true then
        $S$ = $S$ $\cup$ extract(recursion counter,o);
    // classproperty is a property from the OWL vocabulary like
    // {owl:equivalentClass,owl:disjointWith,owl:intersectionOf,etc.}
    else if p is a classproperty and o is not a blank node then
        $S$ = $S$ $\cup$ extract(0, o);
    else if o is a blank node then
        $S$ = $S$ $\cup$ extract(recursion counter, o);
    else
        $S$ = $S$ $\cup$ extract(recursion counter-1, o);
return $S$
```

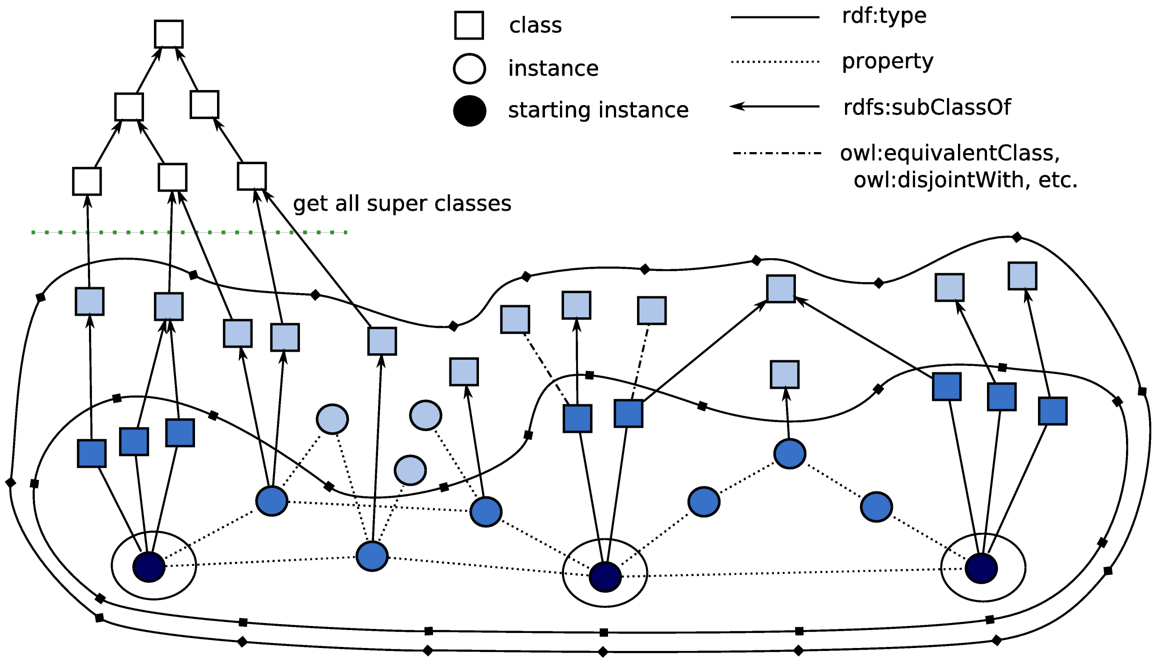


Figure 2: Extraction with three starting instances. The circles represent different recursion depths, where boxes mean recursion depth 1 and diamonds recursion depth 2.

2.3.4 Processing the Results

We showed that the DL-Learner can be used on large knowledge bases and we will now describe what can be done with the learned concept definitions. We will consider DBpedia only, because we especially designed the extraction algorithm for DBpedia. In [17] we also provided some concrete evaluation and Use Cases for detailed scenarios, which we omitted here, since we want to look at the possibilities from a broader perspective.

2.3.5 Semi-automatic Improvement of Background Knowledge

Because of the large number of classes in DBpedia, resources and know-how would be needed to substantially improve the background information. A similar problem exists in the Cyc project, which recently has also taken interest in DBpedia. To design concept definitions from scratch, experts had to look manually at the data and then in numerous steps refine their axioms. With the above described process, experts would have at the very least a first proposal for a concept definition, which in the best case they only needed to accept. By changing the original examples, they could easily modify the proposal by a new run of the algorithm. By changing the perspective from a skill-intensive, formal view to a lightweight one, the know-how needed to make a decision about correctness could be lowered further. Instead of regarding classes as concepts with a formal concept definition, classes could be simply seen as sets of instances, thus changing the decision about a correct and sensible concept definition to a decision about the now retrieved set of instances. This set would be based on a retrieval of instances according to the new concept definition and the only question that needs to be answered is now, whether all the shown instances are members of the concept. For a human, this question can be answered easily with the inherited knowledge of the world and intuition or at least with very few research. It could even go so far that this semi-automatic process could be conducted by a community effort similar to the original Wikipedia itself.

3 Comparison of Learning Algorithms

In this section we will consider different approaches to solve the learning problem in Description Logics. The application of refinement operators has produced useful results in Inductive Logic Programing and the most promising of the available algorithms try to apply this success to DL. We consider the learning algorithms of Lehmann [25] (DL-Learner) and Iannone [20] (YinYang) closest, because they are the most advanced and are already implemented for evaluation (cf. Section 4). They also incorporate gained experience from earlier approaches and directly work on Description Logics in OWL form, which makes them candidates for direct OWL tool support when designing and improving Semantic Web ontologies.

3.1 Desired Properties of a Concept Learning Algorithm

A concept learning algorithm takes as input a knowledge base and a list of positive and negative examples and outputs a concept definition, which poses a "good" solution to the learning problem as defined in Definition 4 in Section 1.2. We will now identify what a "good" solution is and we will later state what desired properties a learning program (tool) should have.

3.1.1 Solutions within the Training Data (Example Sets)

The learning problem in DL has not yet been proven to be decidable. As we will see later, certain design aspects of learning algorithms guarantee that a solution will be found if it exists. In this case a solution means a *correct* concept definition (like defined in Definition 5 in Section 1.2) with respect to the examples. If no solution exists or the algorithm is not complete, a "good" approximation should be found. Here, "good" depends on the number of positive and negative examples covered. A "better" solution therefore covers more positive examples, while it does not cover many negative examples. In other words the learning algorithm has to be able to find a solution while at the same time it should be robust with respect to inconsistent data. As far as we know, there is up to now no way to tell in advance, if a solution exists, which would be desirable, since it could lead to high runtimes with poor results, if none exists.

3.1.2 Scaling the Learned Concept Definition on the Test Data

The quality of the learned concept definition can be assessed further when being compared to the remaining relevant instances (in general called the test data), i.e. all instances that are not part of the example set $\{a|a \in N_I \wedge a \notin E\}$ and that should or should not belong to the target concept. We differentiate this from the above, because learning a concept definition for the training set only, could be wanted in a case where no prediction is necessary[32] and ontologies, where it is unlikely that more instances will be added later e.g. for an ontology about chemical elements. Hence, we will refer to scenarios, where the test data is taken into account as predictive scenarios (as opposed to simple scenarios). To assess the quality of a learned concept definition C, we can consider how "good" the solution is by looking at the test data and apply common machine learning measures like accuracy, precision, recall and F-measure.

3.1.3 Biases of Learning Algorithms

We can now define the following biases for learning algorithms:

Definition 7 (Minimum cross-validation error)
When learning concept definitions in DL the concept definitions with the minimum cross-validation error is preferred.

Definition 8 (Minimum description length)
When learning concept definitions in DL, in a case where both definitions are correct, the concept definition with the smaller description length is preferred (after Occam's razor [10])

Definition 9 (Correctness)
When learning concept definitions in DL the concept definition which is correct is preferred over one that is not correct.

[32]One could argue that the induced concept description can contain useful knowledge about the instances itself, e.g. if we learn the concept $Father = \exists\ hasChild.Person$, it would tell us something about the real existing world like that a father has at least one child.

3.1.4 Important Features of a Learning Program

Because we put special emphasis on the engineering process, we are also concerned about other features, which actually go beyond the scope of the learning algorithm itself, but have to do with the whole learning program. In the following we will give a list of features we consider useful for concept learning programs.

- can use RDF/OWL-files.
- can ignore concepts (enables the relearning of concepts).
- can learn online, thus avoiding a complete new calculation.
- can further refine concept definitions (improve found solutions).
- uses minimal amount of reasoner queries (performance).
- possibly selects instances smart (can automatically detect example sets for possible learning problems, fully automated concept learning)

3.2 Existing Algorithms

3.2.1 LCSLearn

Cohen and Hirsh [13] proposed a learning algorithm in 1994, which was based on the CLASSIC description logic, an early DL formalism. Their approach is simple and the definition of the learning problem is a little bit different. They assume the existence of most specific concepts (MSC)[33] for individuals and and take those MSC's as input examples. Then they simply create the target concept by joining all MSC's with the disjunction operator($\sqcup$). This generates extremely large concept definitions, some of them thousands of symbols long. The clear advantage is that this approach can be solved in polynomial time with respect to length of the MSC's. The disadvantages are obvious. Long concept definitions can not be handled or modified by a knowledge engineer anymore. They also lead to overfitting and do not scale well in a predictive scenario, leading mostly only to a solution within the training data itself. In [13] they also provide some evaluation, but upon request neither the data nor the algorithm were available to us for further investigation.

3.2.2 YinYang

YinYang[34] was implemented by Luigi Iannone and is currently in the process of redesign, which was not finished by the time of our evaluation. We will investigate the changes here only from a theoretical point of view [19; 20], while the evaluation in Section 4 is done with the previous version of 2005.

[33] the least concept description in the available description language that has this individual as an instance

[34] Yet another INduction Yields to ANother Generalization, http://www.di.uniba.it/ iannone/yinyang/

The learning process is close to an intuitive approach humans might follow, when trying to conceptualize instances. First the instances are raised to a concept level representation and then those concept representations are compared to each other, taking out parts that are responsible for inconsistency and adding parts to achieve completeness until a solution is found. We will investigate the approximation of the concept level representation now, which makes up the basis for the manipulation described thereafter.

The raising of individuals to a concept level representation is first mentioned in Dietterich [15] as the *Single representation trick*. The aim is to construct a concept representation that represents the individual, so that the individual is an instance of this concept and no other more specific concept exists that covers the individual. The general problem with these so called Most Specific Concepts (MSC) in DL is that they often do not exist and have to be approximated. It is also highly probable that in a sufficiently large knowledge base two instances exist, which have the same properties and realizations[35] and thus share the same MSC. In YinYang (version of 2005) an (upper) approximation is calculated according to the method proposed in Brandt[11], which is in some respect similar to the method in LCSLearn. In LCSLearn MSC's were calculated up to a user-given length k, which results in concept representation up to a length k (with k nested restriction on roles in CLASSIC DL). A short example for k = {0...3} (taken from [20] and slightly extended):

Example 5 (k-bound abstraction) *Let a knowledge base $\mathcal{K}$ = ($\mathcal{T}$, $\mathcal{A}$) be given with:*
$\mathcal{T}$ = {A , B , C , $\exists$ $R.\top$}
$\mathcal{A}$ = { *A(a), A(a), D(a), B(b), C(c), R(a,b), R(b,c), R(c,a)* }
$msc^k(a)$ *is the k-bound abstraction of a.*
For k = 0 the msc is a disjunction of all the concepts it belongs to:
$msc^0(a) = A \sqcap D$
a takes part in role R with b as a filler and b belongs to B.
Therefore: $msc^1(a) = A \sqcap D \sqcap \forall R.B$
b takes part in role R with c as a filler and c belongs to C.
Therefore: $msc^2(a) = A \sqcap D \sqcap \forall R(B \sqcap \forall R.C)$
For k >= 3 a cycle will occur, because $R(c, a)$ will be considered next and the abstraction of a will start again.
$msc^3(a) = A \sqcap D \sqcap \forall R.(B \sqcap \forall R.(C \sqcap \forall R.(A \sqcap D)))$

In the redesigned version of 2007 (cf. [20]), Iannone et al. propose an algorithm for the assessment of the size of k, so that it can be decided in advance, if their algorithm will find a solution of the learning problem. He states, that if for a k-bound abstraction of the example individuals, the least

[35]most specific classes that an individual belongs to

common subsumer(*lcs*) [36] is a possible solution of the learning problem, then the learning problem is well-posed and has a solution. Thus, the proposed algorithm starts with $k = 0$ and stops as soon as the *lcs* is a possible solution, taking the resulting k as the bound for abstraction when calculating MSC representatives. On the one hand this is useful since it creates an *a-priori* condition, if the algorithm is able to find a solution at all, but on the other hand it does not solve the problem of decidability of the learning problem as such, since it only accounts for the concept level representations of the example individuals. In Iannones et al. terms the learning problem is solved if the solution subsumes the MSCs of the positive examples while not subsuming the representation of the negative ones.

The employment of the MSC or an approximation thereof is critical for this algorithm. A major drawback of MSC's is that they do not(cf. [4] on this issue) exist for expressive languages like $\mathcal{ALC}$, which is the basis for OWL (which is even more expressive). This problem becomes obvious in example 5.4 shown in [20] which is repeated here in short:

$\mathcal{T} = \{ A \sqsubset \top, B \sqsubset \top , C \sqsubset \top , D \sqsubset \top \}$
$\mathcal{A} = \{ A(a_1)$, $(\forall R.(\neg A \sqcap B \sqcap C))(a_1)$, $A(a_2)$,
$(\forall R.(\neg A \sqcap B \sqcap D))(a_2)$, $A(a_3)$, $R(a_3,b)$, $A(a_4)$,
$R(a_4,a_5)$, $A(a_5)$, $(\neg B)(a_5)$, $A(b)$, $B(b) \}$
positive examples = $\{a_1, a_2\}$
negative examples = $\{a_3, a_4\}$
$msc^*(a_1) = A \sqcap \exists R.(\neg A \sqcap B \sqcap C)$
$msc^*(a_2) = A \sqcap \exists R.(\neg A \sqcap B \sqcap D)$
$msc^*(a_3) = A \sqcap \exists R.(A \sqcap B)$
$msc^*(a_4) = A \sqcap \exists R.(A \sqcap \neg B)$

For a complete example of the algorithm see below (Example 6), after the details of the methods are explained.

Presented solution: $C_L = A \sqcap \exists R.(\neg A)$

After Iannones et al. definition of coverage of examples, $C_L = A \sqcap \exists R.(\neg A)$is a correct solution of the learning problem, since $C_L \sqsupseteq \{msc^*(a_1), msc^*(a_2)\}$, but the positive examples do neither follow from C_L $(C_L \nvDash_T a_1, a_2)$ nor from the MSC representatives $(msc^*(a_i) \nvDash_T a_i (i = 1, 2))$, which means the solution presented is actually incomplete. We considered that there maybe is an error in the paper only, since the method described above would yield different MSC's for a_1 and a_2

[36]the function $lcs(C_i)$ returns the least concept subsuming its arguments. According to Baader and Küsters [4] the *lcs* in DLs allowing disjunction is simply the disjunction of its arguments.

($msc^*(a_1) = A \sqcap \forall\, R\,(\neg\, A \sqcap B \sqcap C\,)$). An email to the author remained unanswered, though.

The core algorithm of YinYang consists of two interleaving methods, which alternately either generalize or specialize parts of the current concept definition. As the bases for this process, the algorithm uses the MSC's of the example instances, see Figure 3.

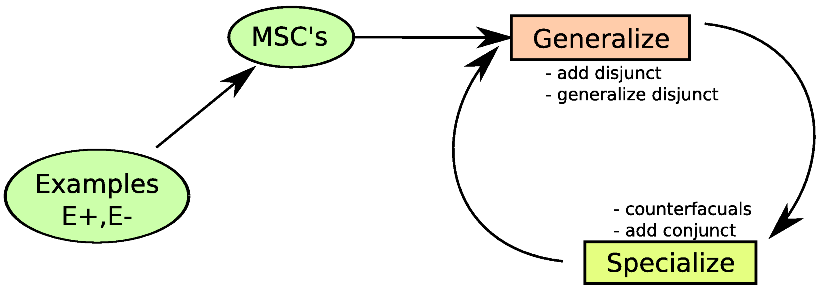


Figure 3: Interleaving methods of YinYang

Generalization At start, some positive MSC's are selected as a starting seed. These positives represent very special concepts and therefore need to generalized. They are selected after a heuristic, which chooses the ones, that cover the most positives, which are then merged with the disjunction operator. The resulting solution (called partial generalization $ParGen$) is further refined by an upward refinement operator, which uses the definition of the remaining positive MSC's to achieve a more general concept. The upward refinement operator either adds another disjunct to the partial generalization in a way that more positive MSC's are covered, while not covering more negatives (add disjunct), or tries to generalize one of the disjuncts of the partial generalization. This generalization is done by either dropping a conjunct from a disjunct (the concepts are in $\mathcal{ALC}$ normal form) or by generalizing one of the inner conjuncts with the help of the *add disjunct* method. This generalization is repeated until some negative MSC's are covered, then the specialization is called. See Example 6[37].

Example 6 (generalization in YinYang) *The same knowledge base is used as presented above. The example is taken form [20], we omitted some parts and added comments.*

generalize

ResidualPositives $leftarrow\ \{msc*(a_1), msc*(a_2)\}$

Generalization $\leftarrow \bot$

*/*The first msc* is selected as a starting seed*/*

ParGen $\leftarrow msc^*(a_1) = A \sqcap \exists\, R\,(\neg\, A \sqcap B \sqcap C\,)$

[37]We refrained from repeating the complete algorithm here, since it is very long and can be explained using only a fraction of space. It can be found in [20] in full.

```
CoveredPositives ← { $msc^*(a_1)$}
CoveredNegatives ← {}
/*the restriction is dropped to generalize the disjunct*/
ParGen ← $A$
CoveredPositives ← { $msc^*(a_1)$, $msc^*(a_2)$}
CoveredNegatives ← {$msc^*(a_3)$, $msc^*(a_4)$}
/*The current solution ParGen is too general now, that is why
it has to be specialized, by calling the specialize function*/
Call **specialize** (ParGen, CoveredPositives, CoveredNegatives )
```

Specialization The procedure used for specialization obtained basic changes in between the two versions of YinYang [19; 20]. In the previous version the concept was specialized with counterfactuals only. Counterfactuals are the result of a difference operator on DL concepts (e.g. $Counterfactual$ = A - B) and are investigated in [11] and [35]. In the new version the algorithm additionally uses a downward refinement operator. The basic idea of specialization with counterfactuals was to remove the part from the current solution, that is responsible for covering the negative MSC's, by subtracting the covered negative MSC's. The authors, however, discovered that counterfactuals are not correct with respect to value restrictions in a Logic that adopts the Open World Assumption like Description Logics (see [20] for details). If such a failure is detected, the algorithm[38] uses a non-deterministic downward refinement operator to either remove a disjunct from the partial generalization in a way that less negative MSC's are covered, while covering the same amount of positive MSC's (remove disjunct), or tries to specialize the disjuncts of the partial generalization (specialize disjunct). The two functions are only mentioned for completeness though, since they are defined analogously to the upward refinement operator. The authors propose a further downward refinement method for specialization, which is used in the new version of the algorithm, whenever counterfactuals fail. This specialization is done by adding a conjunct to the partial generalization (*ParGen*) to cover less negatives (*add conjunct*) (ρ(C) = $C \sqcap C'$).

The problem here is to find a suitable conjunct C'. The algorithm searches through the disjuncts of C = $\{C_1 \sqcup ... \sqcup C_i\}$ and evaluates the conjuncts within the C_is to find those that do not already subsume *ParGen* (else $C \sqcap C'$ = $\perp$), do not subsume the already covered negatives and subsume each of the already covered positives. If it finds such conjuncts they are joined with $\sqcup$ and added to C with $\sqcap$. If it does not find any suitable conjuncts, it substitutes *ParGen* with the `lcs` of the already covered positives and starts another generalization. The search for conjuncts is a costly task and thus

[38]The algorithm actually uses a downward refinement operator to create possible specialization from which the best one is chosen, how this is done though is left open.

is simplified with the disadvantage of rendering the method incomplete but tractable.

In the previous version which we used for our experiments YinYang had only one mode for generalization (*greedy*), where it tried to select the best (covering the most positives) positive MSC's to form the partial generalization. It also had only one method for specialization (counterfactuals), which only works properly in a setting adopting the CWA. The new version furthermore uses one more method for generalization, where *drop conjunct* is used to drop one inner conjunct at a time. Also there is a mode to change the ontology to adopt the CWA, which makes the use of counterfactuals more appropriate (cf. Section 3.4). In spite of these improvements, there are many issues which still need improvement. The complexity of approximating MSC's is polynomial in the size of k (see above) and could lead to an increased computing time, since it is not ensured that correct MSC's can be found at all. Furthermore the solutions YinYang produces tend to be quite long which normally leads to overfitting (low recall) and which render the resulting description unreadable by humans. The algorithm is thus unlikely to be useful for semi-automatic tool support for an ontology engineer like in Section 5. In a scenario, where humans are not involved, like automatic repair of an ontology, YinYang might successfully be used.

3.2.3 DL-Learner

The DL-Learner uses an approach completely different from YinYang and LCSLearn. The learning algorithm is an informed search algorithm using a heuristic over the space of possible solutions (space of states with goal states (one passing the goal test)) combined with a failsafe method that ensure completeness and guarantees that a solution, if it exists, is found.

It is currently implemented as a top-down algorithm, going from general to specific, but the other way is theoretically possible with a different heuristic. The operator responsible for node expansion (i.e. a refinement operator) makes good use of the partial-order (subsumption) and creates a search tree[39] with TOP (the supremum of the partial-ordered space) as its root. Then it starts to expand the nodes with best fitness (best heuristic estimate) up to a certain depth-limit here called horizontal expansion. If this limit is reached, nodes that are higher in the search tree and have not been expanded yet are expanded (failsafe), which makes sure that a solution, if it exists, is found. Certain techniques are applied to further prune the search tree.

We will first take a close look at the refinement operator. Throughout the literature[6; 7; 19; 25] we can find the following properties of refinement operators:

Definition 10 (properties of DL refinement operators)
An $\mathcal{ALC}$ refinement operator ρ is called

[39]Note that it does not matter if the tree is explicit or implicit (generated on the go).

- (locally) finite *iff* $\rho(C)$ *is finite for any concept* C.
- redundant *iff there exists a refinement chain from a concept* C *to a concept* D*, which does not go through some concept* E *and a refinement chain from* C *to a concept weakly equal to* D*, which does go through* E.
- proper *iff for all concepts* C *and* D*,* $D \in \rho(C)$ *implies* $C \not\equiv D$.
- ideal *iff it is finite, complete (see below) and proper.*
- optimal *iff it is finite, non-redundant and weakly complete.[19; 6]*

An $\mathcal{ALC}$ *downward refinement operator* ρ *is called*

- complete *iff for all concepts* C, D *with* $C \sqsubset_{\mathcal{T}} D$ *we can reach a concept* E *with* $E \equiv C$ *from* D *by* ρ.
- weakly complete *iff for all concepts* $C \sqsubset_{\mathcal{T}} \top$ *we can reach a concept* E *with* $E \equiv C$ *from* $\top$ *by* ρ.
- minimal *iff for all* C*,* $\rho(C)$ *contains only downward covers and all its elements are incomparable with respect to* $\sqsubseteq$.

The corresponding notions for upward refinement operators are defined dually.

In a recent paper by Lehmann [24], foundations for refinement operators have been thoroughly investigated. Lehmann proves that certain combinations of properties do not exist and provides a list of possible combinations.

Theorem 1 (properties of refinement operators (II))
Considering the properties completeness, weak completeness, properness, finiteness, and non-redundancy the following are maximal sets of properties (in the sense that no other of the mentioned properties can be added) of $\mathcal{ALC}$ *refinement operators:*

1. $\{$*weakly complete*, *complete*, *finite*$\}$
2. $\{$*weakly complete*, *complete*, *proper*$\}$
3. $\{$*weakly complete*, *non-redundant*, *finite*$\}$
4. $\{$*weakly complete*, *non-redundant*, *proper*$\}$
5. $\{$*non-redundant*, *finite*, *proper*$\}$

To design a learning algorithm using a refinement operator, one of the sets in Theorem 1 has to be chosen and a work around has to be found to cope for the properties not included in the choice. In case of this algorithm, the second set was chosen for the refinement operator, while the horizontal expansion provides finiteness and the search heuristic, among other things, helps to reduce redundancy. For the search tree this means, that the *completeness* of the operator provides that the node with a solution, if it exists, can be reached by expanding any node with a concept that subsumes the solution, while *properness* ensures that the concept of a parent node always strictly subsumes the concepts of its child nodes.

$$\rho'_{\downarrow}(C) = \begin{cases} \emptyset & \text{if } C = \bot \\ \{C_1 \sqcup \cdots \sqcup C_n \mid C_i \in M \ (1 \le i \le n)\} & \text{if } C = \top \\ \{A' \mid A' \in \text{nb}_{\downarrow}(A)\} \cup \{A \sqcap D \mid D \in \rho'_{\downarrow}(\top)\} & \text{if } C = A \ (A \in N_C) \\ \{\neg A' \mid A' \in \text{nb}_{\uparrow}(A)\} \cup \{\neg A \sqcap D \mid D \in \rho'_{\downarrow}(\top)\} & \text{if } C = \neg A \ (A \in N_C) \\ \{\exists r.E \mid E \in \rho'_{\downarrow}(D)\} \cup \ \{\exists r.D \sqcap E \mid E \in \rho'_{\downarrow}(\top)\} & \text{if } C = \exists r.D \\ \quad \cup \{\exists s.D \mid s \in \text{nb}_{\downarrow}(r)\} & \\ \{\forall r.E \mid E \in \rho'_{\downarrow}(D)\} \cup \ \{\forall r.D \sqcap E \mid E \in \rho'_{\downarrow}(\top)\} & \text{if } C = \forall r.D \\ \quad \cup \{\forall r.\bot \mid D = A \in N_C \text{ and } \text{nb}_{\downarrow}(A) = \emptyset\} & \\ \quad \cup \{\forall s.D \mid s \in \text{nb}_{\downarrow}(r)\} & \\ \{C_1 \sqcap \cdots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \cdots \sqcap C_n \mid & \text{if } C = C_1 \sqcap \cdots \sqcap C_n \\ \quad D \in \rho'_{\downarrow}(C_i), 1 \le i \le n\} & (n \ge 2) \\ \{C_1 \sqcup \cdots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \cdots \sqcup C_n \mid & \text{if } C = C_1 \sqcup \cdots \sqcup C_n \\ \quad D \in \rho'_{\downarrow}(C_i), 1 \le i \le n\} & (n \ge 2) \\ \quad \cup \{(C_1 \sqcup \cdots \sqcup C_n) \sqcap D \mid D \in \rho'_{\downarrow}(\top)\} & \end{cases}$$

Figure 4: definition of $\rho'_{\downarrow}$ from [25]

In the following, we will describe how the refinement operator works and then how the created tree structure is traversed and optimized.

The first part of the refinement operator is defined in detail in Figure 4. $\text{nb}_{\downarrow}(A)$ with $A \in N_c$ is the set of atomic concepts, which are directly subsumed by A with no intermediate atomic concepts ($\text{nb}_{\uparrow}(A)$ analogously) and the set M is defined as: All elements in $\{A \mid A \in N_C, \text{nb}_{\uparrow}(A) = \emptyset\}$, $\{\neg A \mid A \in N_C, \text{nb}_{\downarrow}(A) = \emptyset\}$, and $\{\exists r.\top \mid r \in mgr\}$ are in M. If a concept C is in M, then $\forall r.C$ with $r \in mgr$ is also in M. Which means M is the combined set of all most general atomic concepts, all negated most specific atomic concepts, all existential value restrictions on all most general roles(*mgr*) with $\top$ and all quantified value restrictions on all most general roles with the concepts from M.

The operator up to now is complete and infinite but not yet proper. Properness is achieved by a closure of the operator as proposed in Badea and Nienhuys-Cheng[7]. The closure of a refinement operator is achieved by consecutively applying the operator, thus refining the concept, until a proper result is reached. Lehmann [25] shows that the closure can be computed in finite time. Infiniteness is tackled by only allowing concepts up to a certain length, which is iteratively increased during the progression of the algorithm.

We will now take a closer look at the search process, how it traverses the search tree and eliminates redundant nodes. The elimination of redundant nodes is of great importance, since it is computationally expensive to assess a heuristic estimate for a node (see below), because expensive reasoner queries are needed. $\rho_{\downarrow}^{cl}$ is redundant as we saw above, but this redundancy is eliminated by removing nodes with concepts that are weakly equal(weak equality ($\simeq$) is similar to equality, but ignores e.g. the syntactic order of constructs) to concepts of other already existing nodes in the search tree, which can be done before evaluating the heuristic estimate. This is legal, because the refinement operator would expand these nodes in a "weakly equal" way ($\rho_{\downarrow}^{cl}(C) \simeq \rho_{\downarrow}^{cl}(D)$ if $C \simeq D$). Finding weakly equal concepts normally is computationally expensive because of the large size of combinatorial possibilities (e.g. $A_1 \sqcap A_2 \sqcap A_3 \simeq A_1 \sqcap A_3 \sqcap A_2 \simeq A_2 \sqcap A_1 \sqcap A_3 \simeq etc...$), but here the concepts are converted to *ordered negation normal form* which only takes a small amount of time compared to evaluating those redundant nodes.

The quality of a concept is evaluated by a function that assigns values according to example coverage. If not all positive examples are covered, then the value "tw" (too weak) is assigned. Since the algorithm performs a top-down search with a downward refinement operator, there is no need to further refine concepts with quality "tw" since it would only result in concepts, which are also "too weak". If all positive examples are covered, the quality is the negated number of negative examples covered 0...$|E^-|$ with 0 being the best since it is satisfies the goal condition and $|E^-|$ being the worst (e.g. $\top$). Note that the calculation of the coverage needs instance checks, which have a high complexity (EXPTIME for $\mathcal{ALC}$, NEXPTIME for $\mathcal{SHOIN}(D)$ and OWL-DL), thus pruning the search tree (in this case, by not expanding nodes with quality "too weak") which greatly benefits performance.

The heuristic function of the search algorithm uses of course the quality of concepts to determine the next node that should be expanded, but it also takes into account another factor which is responsible for the strong bias of this algorithm that shorter concept description are better. It is called the horizontal expansion and corresponds with the length of a concept definition, though it is not exactly the same. As we have seen above, the algorithm handles infinity by only considering created concepts of a refinement step up to a certain length. This means essentially that a node can be expanded several times, while it will always produce additional nodes (i.e. with longer concept description (or at least equal length)). The heuristic function works like this: If a node has better quality than any other node, expand it. If there are several nodes having the highest quality, expand the one with the smallest horizontal expansion, i.e. the node that has been expanded least often and thus is likely to have a shorter concept definition. Note that one thing is still missing. The described learning algorithm up to now would not be complete, since if there existed a node with a high quality (e.g. -1), it would straight go into that branch of the tree and would only expand this. Therefore minimal horizontal expansion is introduced, which expands nodes with a low horizontal expansion up to a minimum length which is constantly increased throughout the run of the algorithm. This guarantees that a solution will be found, which is proven in [25].

The algorithm is repeated here (cf. Algorithm 3, we added comments):

Algorithm 3: learning algorithm of the DL-Learner

```
    Input: $\mathit{horizExpFactor}$ in ]0,1]
 1  //user chosen value, if high the algorithm will go deeper in the tree before expanding
    horizontally
 2  $ST$ (search tree) is set to the tree consisting only of the root node $(\top, 0, q(\top), \mathit{false})$
 3  $\mathit{minHorizExp} = 0$
 4  while $ST$ does not contain a correct concept do
 5      //if nodes with equal quality exist, choose the one with the least horizontal expansion
 6      choose $N = (C, n, q, b)$ with highest fitness in $ST$
 7      expand $N$ up to length $n + 1$, i.e. :
 8      begin
 9          //checks for weak equality
10          add all nodes $(D, n, -, \mathit{checkRed}(ST, D))$ with
11          //achieves properness
12          $D \in \mathit{transform}(\rho_{\downarrow}^{cl}(C))$ and $|D| = n + 1$ as children of $N$
13          evaluate created non-redundant nodes
14          change $N$ to $(C, n + 1, q, b)$
15      end
16      $\mathit{minHorizExp} = \max(\mathit{minHorizExp}, \lceil \mathit{horizExpFactor} * (n + 1)) \rceil)$
17      while there are nodes with defined quality and horiz. expansion smaller $\mathit{minHorizExp}$ do
18          expand these nodes up to $\mathit{minHorizExp}$
19  Return a correct concept in $ST$
```

3.3 Comparison

After we investigated the inner workings of the algorithms, we will compare them directly in this section. The two main algorithms use completely different methods for learning concepts. One constructs the solution bottom-up and the other one top-down. For the following analysis, we will ignore the approach by Cohen and Hirsh [13] using the `lcs`, since YinYang is based upon their results and is much more advanced. Another approach that we will skip here was proposed by Badea et al. [7], who construct a complete and proper refinement operator for the $\mathcal{ALER}$ Description Logic. This approach was not implemented and it served as a basis for the top-down search and the refinement operator of the DL-Learner, which uses much more advanced methods and works on $\mathcal{ALC}$, which is directly relevant for OWL DL.

Both the DL-Learner and YinYang have, as stated in the respective papers, the same ambition to become tools for ontology engineering. Each adopts a different bias for their solutions. While the DL-Learner has a strong bias for short concept descriptions and correct solutions, YinYang tries to achieve a minimal cross-validation error, while ignoring concept length. The DL-Learner uses a complete and efficient algorithm to search top-down preferring short concepts. During the process, its current solution is always complete according to Definition 5 in Section 1.2, which makes it possibly to stop the algorithm at any time and still receive a solution that at least covers all the positive examples. Because of this behavior, there is also the risk of producing concepts which are too general, which tend to have a lower precision. This risk is reduced greatly though, through the use of the heuristic, which prefers consistent concepts. Yinyang on the other hand seems to completely ignore the length of the produced solution, which has the clear disadvantage that the solution can hardly be altered manually. During the run, it tries to alternately optimize both coverages, i.e. how many positive and negative examples are covered. While the amount of positive examples covered directly influences recall, both coverages are needed to achieve a high precision. If we consider a web scenario, where users are able to stop the algorithms, if they become impatient, the DL-Learner therefore would emit a short solution, which covers all positive examples, but not all negatives, while YinYang would return a long concept description, which would probably cover less negative examples, but not all positives.

We will briefly verify, if the programs possess the features mentioned in Section 3.1.4. Both learning programs can process RDF/OWL and have the ability to ignore concepts for relearning the definition. Both algorithms bear the possibility to learn online, although they did not implement it yet. Learning online becomes important when an existing concept becomes incorrect, e.g. when new instances are added. It is thus preferable to start from the existing concept definition instead of starting from scratch. YinYang was especially designed to work online (see [19; 20]). The existing incorrect concept definition can be used as partial generalization as a starting input. A major drawback is that the MSC's have to be recalculated. In case of the DL-Learner an upward refinement operator could be used until a more general concept is reached (similar to YinYang), which covers the newly added instances, then the downward refinement can be used combined with the top-down search.

Another useful feature is the ability to further refine a found solution. Once the DL-Learner found a solution, it is possible to search for further solutions, which could be requested by a knowledge engineer through interaction (cf. Section 5). We are not sure what will happen if YinYang tries to further refine concepts. YinYang heavily depends on the information contained in the MSC's. The algorithm stops, if all MSC's contributed to the solution, so there are none left to further improve the solution. Also the methods used for specialization seem to bear the risk that a further refinement of a solution could always decrease the quality (`lcs` substitution, if *counterfactuals* and *add conjuncts* fail).

3.4 Problems of the Open World Assumption

The Open World Assumption poses a problem, when checking example coverage for verifying the solution of a learning problem. The issue is mentioned in Badea et al. [7], who propose to close the knowledge base to allow example coverage checks in cases were examples would be covered under the Closed World Assumption (CWA). The next two examples show the problem of open knowledge bases when learning concepts. A problem that would not occur in Inductive Logic Programming when learning clauses in logic programs because of the CWA .

Example 7 *Consider the following knowledge base* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *with*
$\mathcal{T} = \emptyset$
$\mathcal{A} = \{A(a_1), A(a_2), A(b_1), B(b_1), A(b_2), B(b_2), R(a_1, b_1), R(a_2, b_2)\}$

and the example sets of the learning problem:

$E^+ = \{a_1, a_2\}$
$E^- = \{b_1, b_2\}$

With CWA in effect correct solutions would e.g. (there are more) be

$C = \forall R.B$ *or* $C = \neg B$

since $\mathcal{K}' \models E^+$ *and* $\mathcal{K}' \not\models E^-$ *with* $\mathcal{K}' = \mathcal{K} \cup \{Target \equiv C\}$

however with the OWA the positive examples do not follow from the learned concept. There could be more unknown roles of a_1 and a_2 with fillers not instances of B and they are not explicitly defined instances of the negated concept $\neg B$, meaning it is unknown, whether they belong to B. Note that the Unique Name Assumption (UNA) adds to this problem, because it prevents the learning of e.g. number restrictions. The concept $\leq 2R$ would not cover both positive examples since an interpretation could assign a_1 and a_2 to the same element $x^\mathcal{I} \in \Delta^\mathcal{I}\{a_1^\mathcal{I} = a_2^\mathcal{I} = x^\mathcal{I}\}$.

When considering OWL (Lite, DL, 1.1) as the target language, it is impossible to learn the following language constructs (if of course they are not defined already in a given knowledge base) without circumventing the problems stated above.

- Quantified value restriction
- Negated concepts
- Number restrictions

This directly restricts the usefulness of a learning algorithm for ontology engineering since the learned concepts will be less expressive. There are different ways to cope with these problems.

The knowledge engineer could be given a choice upon the presentation of the learned concept C containing an existential value restriction ($\exists R.D$) to transform $\exists R.D$ to a quantified value restriction($\forall R.D$). The new problem is now that all the positive examples and possibly more individuals will not be covered by the concept anymore. To fix this problem an even more invasive choice could be presented to an engineer, i.e. explicitly adding $C(a)$ to the $ABox$ for each $a \in R_{\mathcal{A}}(C)$ ($R_{\mathcal{A}}(C)$ for retrieval of all individuals that are instances of C) , thus forcing the assignment of individuals to the concept. In a second step, the chosen $\exists R.D$ could be transformed to $(\exists R.D \sqcap \forall R.D)$[40]. There are numerous possible examples where such a step would be sensible, e.g. a vegetarian meal can have numerous meatless ingredients, but should have at least one (a meal without any ingredients would be cheap to produce, but would not appeal to a hungry customer in any way) and all the ingredients should be without meat. One more proposal could be made to the engineer, in case it applies. The $\exists R.D$ could also be transformed to a number restriction $[< \mid > \mid =]nR$, which would result in an even stronger expression. In OWL DL qualified number restrictions do not exist, which would make the preservation of $\forall R.D$ mandatory, while it can be omitted in OWL 1.1 with $[< \mid > \mid =]nR.D$ (Note that all major reasoner already support qualified number restrictions). The proposed number n for these restrictions could be assessed in a simple way by counting the number of participations in certain roles by the concerning individuals. Some reasoners might make it necessary though to dissolve the UNA (with the owl:AllDifferent statement for example).

The proposed solution above solves the major problems that come with the OWA and the UNA, but it still is burdensome on the engineer, because he still has to make difficult decisions about the Ontology design. Thus a tool, in this respect, would only be semi-automatic in a way, that it takes tedious manual work away from the engineer, but does not aid him in the decision process, i.e. does not make proposals how he preferably should design the ontology.

The preferable way in our opinion should be, that the learning algorithm closes the knowledge base temporary to learn stronger statements, which then can be proposed to the knowledge engineer for

[40]Of course this could only be an option, if all the individuals do not have roles with individuals not belonging to the extension of D, because it otherwise would result in an inconsistent ontology.

his approval or rejection (relearn with an open knowledge base). Some authors have committed work [7; 20; 21] to this issue and we will analyze it here briefly while proposing a new way at the end which will be incorporated in the DL-Learner in future releases.

With respect to concept learning in DL, Badea and Nienhuys-Cheng [7] considered the **K** operator[41] from a theoretical point of view. The **K** operator alters constructs like $\forall$ in a way that they operate on a Closed World Assumption. The definition of such a operator is simple, because it has the desired properties by definition. The creation and implementation of such an operator is however not trivial. In a recent paper [21] of the developers of Pellet[42], they analyze how such an operator could be used in applications. The paper just considers many possible ways to incorporate such an operator, but in the end comes to the conclusion, that really none of those ways provide a best practice. Nevertheless on their homepage[43], they announce that they are working on the implementation of a **K** operator to provide CWA-based OWL reasoning.

Another approach is provided in the most recent paper about the YinYang algorithm[20]. They are adding axioms to the *ABox* (namely e.g. $\forall(a), a \in N_I$) to provide that the instance checks of the msc's contain the individual, they were created from. Since they define coverage (a learned concept covers examples) in a broader sense (solution is correct if it subsumes the msc's of positive examples (and not the msc's of negative examples)), they rather circumvent the problem than tackling it. Thus it might be argued, that the solution only is useful for the knowledge base with the added *ABox* axioms, but does not account for the original knowledge base.

The method we are pursuing is based on two assumptions. The first assumption is that from a practical view, a knowledge engineer (with domain knowledge) naturally rejects the Unique Name Assumption, meaning that he considers all instances different from one another, unless explicitly stated otherwise with e.g. owl:sameAs. The second assumption, we make, is that when providing choices to an engineer with domain knowledge, the choice if a number restriction applies and how big that number should be, is a choice that normally can be answered easily (e.g. a car has at least 4 tires, a hand has 5 fingers, the H in H_2O has two atomic connection to the oxygen). We are proposing a closure by adding number restrictions to individuals that mirror the amount of roles they take part in. Note that number restrictions do not make statements about values a role might take and do not make global assumptions. The way we apply them just reflects the actual state of the knowledge base. Instead of giving an exact definition, we just give a short example which suffices to explain the idea (since it is not that complicated).

Example 8 (closure with number restrictions) *We change a given* ABox*:*
$\mathcal{A} = \{A(a_1), A(a_2), A(a_3), A(a_4),$

[41]The K-estimation operator is originally named after Kripke.
[42]http://pellet.owldl.com/
[43]http://pellet.owldl.com/faq/closed-world/

$B(b_1), B(b_2), B(b_3), B(b_4),$
$R(a_1, b_1), R(a_1, b_2), R(a_1, b_3),$
$R(a_2, b_1), R(a_2, b_2), S(a_2, b_3)\}$
by adding the following axioms:
$= 3R(a_1)(a_1$ *participates three times in* $R)$
$= 2R(a_2)(a_2$ *participates two times in* $R)$
$= 1S(a_2)(a_2$ *participates one time in* $S)$

Because of the way the DL-Learner is implemented, it suffices to add the axioms to the DIG-code[44] that is sent to the reasoner, thus leaving the original knowledge base unchanged. Another statement, which is part of OWL-DL, is added to the DIG-code, namely owl:AllDifferent, which disposes of the UNA[45]. The addition of the axioms, now enables learning stronger statements containing e.g. the $\forall$ constructor. Note that now e.g. $\forall R.B(a_1)$ and $\forall R.B(a_2)$ can be inferred, i.e. they are true. The main difference now, compared to the above proposal, where the choice is up to the engineer, is that the bias is different. The learned concept containing e.g. an $\forall$ constructor is now a proposal, which means the learning algorithm (or more general speaking the tool supporting the engineer) considers it as a good solution, while before it was a mere choice after the liking of the engineer. If he accepts the proposal the above mentioned steps can be taken (assigning all concerning instances to the learned concept, etc.).

We, therefore, shifted the problem of the OWA, since now, the problem is not anymore that complex and stronger definitions can not be learned, it now all depends on how good the solutions are, the learning algorithm proposes.

4 Benchmarks

We used several benchmarks to compare the two implemented learning systems YinYang and DL-Learner. In Section 4.2, we tested both the algorithms for correctness, basically using the training data as test data. This resembles the use case, in which an engineer selects all available instances to learn a concept description during ontology creation. In Section 4.3, we evaluated the usefulness of the algorithms to make accurate predictions about instances not belonging to the training data. This normally occurs, if new instances are added to the knowledge base. The used benchmarks will be available with further releases of the DL-Learner at Sourceforge[46].

[44] http://dl.kr.org/dig/
[45] in contrast to a complete closure owl:AllDifferent is implemented in all major reasoners (FaCT++, Pellet)
[46] http://sourceforge.net/projects/dl-learner/

4.1 Some Remarks about the Experiments

Since the DL-Learner is open source we used the latest SVN version from the repository. For YinYang we had to use an older version, which was available for download from the authors homepage. In an email Luigi Iannone, the main developer of YinYang, stated that a release is scheduled in the next months; thus we could not use the improved version, which was used to produce the results in [20]. We used the newest version (1.5.1) of Pellet[47] a state-of-the art reasoner, which is freely available. We ran the programs from the console and measured the time they needed for a complete run. We did not subtract the times, they needed to start internal components or parse the ontologies, since we assume, that either the time needed for these tasks is marginal compared to the time needed for concept learning or is roughly the same for both algorithms. To validate the results we parsed the output and converted the learned concepts to an internal representation. The concept length is calculated by counting the number of atomic concepts, including $\top$ and $\bot$, and constructors as in Example 9.

Example 9 (concept length)
length 1: A
length 3: $A \sqcap B$
length 4: $A \sqcap \exists\, role.\top$
length 4: $A \sqcap \exists\, role.B$

We converted the internal representation to DIG code and queried the reasoner Pellet 1.5.1. for all instances of the given concept, which we then compared with the target data.

4.2 Simple Scenarios

As mentioned before, we will use the training data as test data in this section. We will measure correctness, time and concept length of the learned concept. Each subsection will give a brief overview about the used data and the results are summarized in the end.

4.2.1 Identifying Arches

The arch problem is small in terms of size and complexity of the background knowledge. It was first mentioned in [36] and used by [32] to evaluate the learning program FOIL, which learns Horn clauses. The axioms in the paper were converted to Description Logic axioms and one more negative example was added. The algorithms are presented 5 instances, of which two are arches and three are not.

[47] http://pellet.owldl.com/

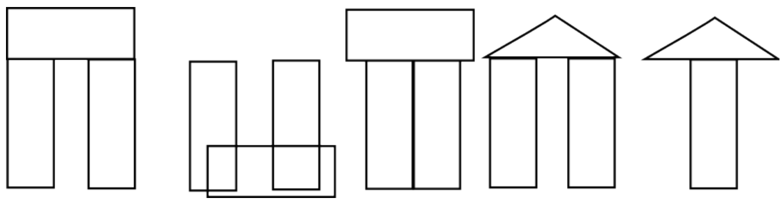


Figure 5: 5 constructions, of which 2 are arches

4.2.2 Trains

The trains problem originated from [30] and was converted to Description Logics by Lehmann [25] . The data describes different features of trains, e.g. which cars are appended to a train, whether they are short or long, closed or open, jagged or not, which shapes they contain and how many of them. The five trains on the left are the positive examples, the trains on the right negative examples.

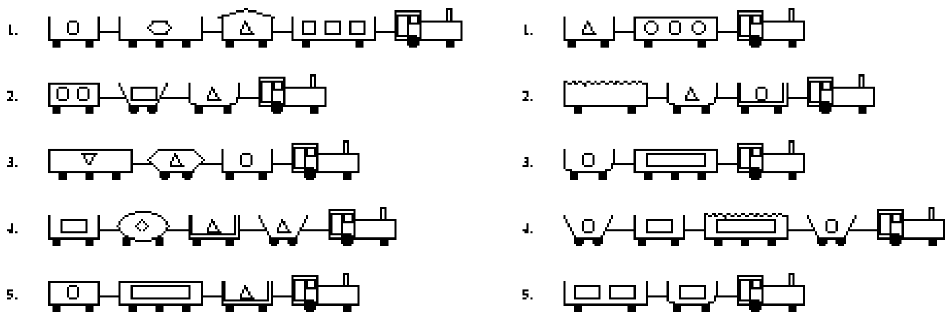


Figure 6: Ten trains

4.2.3 Summary

Table 2 presents the results of the experiments. Both algorithms succeed in learning correct concepts in adequate time. The obvious difference is the increased solution length produced by YinYang.

4.3 Predictive Scenarios

In this section we tested the algorithms for their predictive qualities. We applied common measures like precision, recall, f-measure and accuracy on the results. The measures were calculated in the following way. We queried Pellet 1.5.1 and compared the results to the data according to the

	Correctness		Length		Time	
Experiment	DL	YY	DL	YY	DL	YY
Arches	100%	100%	6	24	8,82sec	6,11sec
Trains	100%	100%	4	12	3,69sec	6,88sec

Table 2: Results of the Arches and the Trains experiment

following formulas:

Definition 11 (Measures)

Definition of the sets:

$Relevant$	*all instances that should belong to the target concept.*
$Retrieved$	*all instances retrieved by the reasoner query.*
$Testdata$	*all positive and negative instances belonging to the* $Testdata$*.*
$Positives$	*all positive instances from the* $Testdata$*.*
$Negatives$	*all negative instances from the* $Testdata$*.*
$TruePositves$	$Retrieved \cap Positives$
$TrueNegatives$	$Negatives$ *without* $\{Retrieved \cap Negatives\}$

Definition of the measure:

Precision	$\|Relevant \cap Retrieved\| / \|Retrieved\|$
Recall	$\|Relevant \cap Retrieved\| / \|Relevant\|$
F-measure	$2 * (Precision * Recall) / (Precision + Recall)$
Predictive Accuracy	$\|TruePositves \cup TrueNegatives\| / \|Testdata\|$

4.3.1 Moral Reasoner

The benchmark was taken from the UCI Machine Learning Repository[48]. We converted it from Datalog to DL and it now consists of 18 defined concepts and 202 instances of which 102 belong to the target concept "guilty". The main aim of the problem is to learn rules whether people are to be judged guilty or not guilty. We conducted two experiments. In the first experiment we removed the axiom

$guilty = blameworthy \sqcup vicarious\_blame$
(formerly in Datalog:

[48] http://archive.ics.uci.edu/ml/ and for the data http://mlearn.ics.uci.edu/databases/moral-reasoner/

guilty(X) :- blameworthy(X).
guilty(X) :- vicarious_blame(X).)

and choose 40 instances (20, which originally belonged to the concept $guilty$ and 20 which did not) as example set. In the second experiment we used the same example sets and further removed the two intermediate concepts:
$blameworthy$, $vicarious\_blame$
The remainder of the instances served as test data. We discovered that YinYang produced different results at each run. We could not find an explanation for this behavior and thus just averaged the results over 6 runs.

Table 3 shows the results of the experiment. The DL-Learner scores high overall, needs less time and produces short solutions. It lost 3.77% in the precision for the complex experiment, because it retrieved 4 additional negative instances. This matches exactly the expectation of short solutions. They score high on recall and predictive accuracy, but have a tendency due to their generality to retrieve unwanted instances. Nevertheless it is just marginal in this case. The results YinYang produced on the other hand also match the expectation. It produced high precision, since it covers the positive examples, but does not scale well over the whole data. The recall shows how more or less exactly the 20 positive examples used for learning are retrieved out of 102 relevant. The high predictive accuracy is mostly achieved by excluding the target negatives, which count as $TrueNegatives$ in the calculation.

DL-Learner						
concept	precision	recall	f-measure	pred. acc	avg length	avg time
Moral simple	100%	100%	100%	100%	3	61.3sec
Moral complex	96.23%	100%	98.08%	97.48%	8	122sec
Total avg	98.12%	100%	99.04%	98.74%	5.5	91.65sec
YinYang						
Concept	precision	recall	f-measure	pred. acc	avg length	avg time
Moral simple	69.82%	25.33%	37.17%	49.69%	111.7	85.5sec
Moral complex	71.43%	19.61%	30.77%	50.84%	77.8	177.4sec
Total avg	70.63%	22.47%	33.97%	50.27%	94.75	131.45sec

Table 3: Results for the moral reasoner benchmark

4.3.2 Family Benchmark

Throughout the literature we can find many evaluations that use family trees for learning hypotheses. Family relations and concepts are well suited, because the initial knowledge is fairly simple and possible target hypotheses range from simple to difficult. Instead of using an existing example taken e.g. from FORTE[49] or [19; 32], we chose to create our own benchmark to be able to test a wide range of learning problems with oracled data. The ontology we created contains 257 instances, which belong to the concept $Person$ and to either $Male$ or $Female$. We furthermore used four properties to describe the relations between instances: $married$, $hasSibling$, $hasChild$, $hasParent$. We conducted two sets of experiments, one set with easier learning problems and one with a very complex problem. In the following, we will first describe the process in detail how the ontology was created.

Process of creation With the help of a script a randomized family ontology was created. Starting from one couple, we generated the family tree based on the following probabilities. In the first run the couple (first generation) has a 100% chance to have one child, an 80% chance to have a second child and a 60% chance for a third and so on. Each child, no matter what generation, has a 50% chance to be male or female. In the next run for the second generation each child of the first couple has an 80% chance of getting married and if this occurs, there is an 80% chance for the first child, 60% chance for the second child and so on. In the third generation, the children of the second generation only have a 60% chance of getting married and a starting chance of having one child of 60%. There are two factors that influence the size of the ontology, namely the diminishing factor, which was 20% in the above example and the number of families, meaning that one starting couple is used to create a family tree for each family. We defined 19 target concepts and collected the belonging instances during the creation process. A separate ontology was created based on the same instances, but containing more background knowledge, which was used for the second experiment. The basic ontology only contains three concepts, which are ordered as follows:

$Person \sqsupseteq Male$
$Person \sqsupseteq Female$

The ontology with extended background knowledge contains the same instances and 15 more concepts with the most complicated one, i.e. $Uncle$ left out (that presumably has the longest concept descriptions and thus is difficult to learn) It also contains a hierarchical order between the concepts like:

$Parent \sqsupseteq GrandParent$
$GrandParent \sqsupseteq GrandFather$
$Male \sqsupseteq GrandFather$

Table 4 shows detailed information about the number of instances for each concept.

[49]First Order Revision of Theories from Examples, http://www.cs.utexas.edu/users/ml/forte.html

Concept	**Instances**	**Percentage**
Person	202	100%
Male	104	51.48%
Female	98	48.51%
PersonWithASibling	72	35.64%
Father	60	29.70%
Mother	60	29.70%
Son	52	25.74%
Daughter	52	25.74%
Grandson	43	21.28%
Sister	42	20.79%
Uncle	38	18.81%
Granddaughter	37	18.31%
Grandfather	35	17.32%
Grandmother	35	17.32%
Brother	30	14.85%
Grandgrandson	24	11.88%
Grandgrandfather	17	8.41%
Grandgrandmother	17	8.41%
Grandgranddaughter	17	8.41%

Table 4: The table shows the number of instances for each target concept

Cross validation We divided the data for each instance randomly in 6 folds and used 5 for training and 1 for testing. The last target concepts did not have sufficiently enough instances for more folds. As a special tough test for the learning programs we decided not to include all negative examples in the folds, but only up to the size of the positive example set size.

Complexity of target concepts The complexity of target concepts vary heavily, which directly results in the big differences in accuracy and time. The simplest concepts are the ones that have the shortest definition like $Father$, $Mother$, Son, $Daugther$. The more difficult ones include $Grandgranddaughter$, $Grandgrandson$, etc. To give one short example, one of the shortest possible solutions for $Grandgrandfather$, which we could intuitively think of, ($Male \sqcap \exists\ hasChild.\exists\ hasChild.\exists\ hasChild.\top$) was found by the DL-Learner. It has a length of 6. YinYang on the other hand produced much longer solutions with an average length of 108.3, which we consider unreadable by a human. In a second experiment we tried to evaluate the solution for the most complex learning problem for the target concept $Uncle$, where we assume that the shortest possible solution is $Male$

$\sqcap$ ($\exists$ $hasSibling.\exists$ $hasChild.\top \sqcup \exists$ $married.\exists$ $hasSibling.\exists$ $hasChild.\top$ (A man that has a sibling that has a child or that is married to someone who has a sibling that has a child.). Actually that problem was so difficult to learn, that we faced some problems during the experiments, which we will analyze below in the Paragraph **Uncle**.

Result of the first experiment set The results for each of the 15 easier target concepts are summarized in Table 5. Both algorithms have a perfect score in the first 9 experiments with the DL-Learner being faster and producing shorter solutions. For the last 6 experiments the performance of the DL-Learner stays good to excellent with two exceptions for the concepts $Grandgranddaughter$ and $Grandgrandmother$. We already stated above, that we used, as an extra difficulty, positive and negative example sets of the same size, not including all negative examples in the folds. As we looked closer at the results, we discovered that the DL-Learner learned the correct concepts for $Granddaughter$ and $Grandmother$ in those two cases, which are superclasses of the two target concepts (every Grandgrandmother is also a Grandmother). Since there were not any Grandmothers or Granddaughter included in the negative example sets, the DL-Learner stopped, because the concepts were correct. As it is a top-down algorithm it thus produced a more general concept, which has a lower precision, a high recall and a good predictive accuracy , as confirmed by the experiment. YinYang behaves similar for those two concepts. It also learned the concept for $Grandmother$ instead of $Grandgrandmother$ and performs even poorer for the concept $Grandgranddaughter$[50]. In the other 4 from the last 6 more difficult experiments YinYang showed more weaknesses, which in some cases can even be considered horrible compared to the DL-Learner. It hardly reaches a precision of 30% for the concept $Grandgrandfather$ (only 50.53% for $Grandgrandson$) and it does not only produce very long concept description with sizes of e.g. 108.3 and 199.3, it also needs more than 20 times longer for some concepts ($Granddaughter$). The averaged scores reveal that YinYang compared to the DL-Learner has a lower total F-measure by 12.25%, a lower predictive accuracy by 7.65% and 7.6 times longer concept descriptions while needing 5.6 times longer.

Uncle As mentioned above, there seems to be a direct relation between target concept length and learning complexity. We originally planned on testing the algorithm with 3 target concepts of high complexity. The concepts were $Uncle$, $Aunt$, $Cousin$ (A $Person$ who has a $Parent$ that either has a $Sibling$ that has a $Child$ or who has a $Parent$ that is $married$ to someone who has a $Sibling$ who has a $Child$), whereas $Cousin$ had the highest complexity. We intended to test it twice, i.e. with the

[50]Remark: At first we assumed a flaw in our preparation of the folds, but we discovered that the target concept and the assigned instances and the instances in the folds were all correct, so that this outcome was produced by chance only. We refused to let the dice role one more time, because we do not consider it legal to tweak the data until the results appear pleasant.

DL-Learner Concept	Precision	Recall	F-measure	Pred. Acc.	avg length	avg time
PersonWithASibling	100%	100%	100%	100%	2	5.5sec
Brother	100%	100%	100%	100%	4	7.1sec
Sister	100%	100%	100%	100%	4	7.7sec
Son	100%	100%	100%	100%	4	7.6sec
Daughter	100%	100%	100%	100%	4	7.8sec
Father	100%	100%	100%	100%	4	9.3sec
Mother	100%	100%	100%	100%	4	10.5sec
Grandfather	100%	100%	100%	100%	5	11.3sec
Grandmother	100%	100%	100%	100%	5	11.3sec
Grandson	96.63%	100%	98.29%	98.61%	4.8	12.3sec
Granddaughter	100%	100%	100%	100%	5	11.4sec
Grandgrandson	100%	100%	100%	100%	6	25.6sec
Grandgranddaughter	45.95%	100%	62.96%	100%	5	7.9sec
Grandgrandfather	100%	100%	100%	100%	6	22.7sec
Grandgrandmother	48.57%	100%	65.38%	100%	5	7.5sec
Total avg	92.74%	100%	95.11%	99.91%	4.52	11.03sec
YinYang Concept	Precision	Recall	F-measure	Pred. Acc.	avg length	avg time
PersonWithASibling	100%	100%	100%	100%	8	19.1sec
Brother	100%	100%	100%	100%	8	13.1sec
Sister	100%	100%	100%	100%	8	14.2sec
Son	100%	100%	100%	100%	6	15.1sec
Daughter	100%	100%	100%	100%	6	15.2sec
Father	100%	100%	100%	100%	6	15.8sec
Mother	100%	100%	100%	100%	6	15.7sec
Grandfather	100%	100%	100%	100%	15.5	35sec
Grandmother	100%	100%	100%	100%	16	37.9sec
Grandson	85.15%	100%	91.98%	98.61%	22	94.7sec
Granddaughter	83.18%	40.09%	54.1%	68.33%	16.5	329.6sec
Grandgrandson	50.53%	100%	67.13%	91.67%	199.8	180.1sec
Grandgranddaughter	12.95%	56.86%	21.09%	58.33%	73.3	75.1sec
Grandgrandfather	27.81%	97.06%	43.23%	66.67%	108.3	49.8sec
Grandgrandmother	48.57%	100%	65.38%	100%	16	21.4sec
Total avg	80.55%	92.93%	82.86%	92.24%	34.36	62.12sec

Table 5: The table shows the results of a 6-fold cross validation

ontology containing only three defined concepts and with the ontology with larger background. We could not conduct the experiment, because we discovered that the bias for short concepts of the DL-Learner currently hinders it to learn long concept definitions. After we ran the DL-Learner on the first folds, we discovered that the runtime for one fold was approximately 15 hours with small background knowledge and 11 hours with large background, since it could use the hierarchy. The estimated time for all folds and all concepts would have been at least (3 * 6 * 15h + 3 * 6 * 11h) = 468 hours = 19.5 days. We therefore tested only all folds for one concept, namely $Uncle$ and only with the ontology with more background knowledge. The results are shown in Table 6, in which we also included two rows that show the results of a dumb learning algorithm that just returns $\top$ or $\bot$. Since the long time the DL-Learner needed (40,252.19 sec) is unacceptable, we also included the solution the DL-Learner produced, if it is stopped at approximately the same time that YinYang needed for a complete run (cf. Section 3.3). If we look at the results, we can see that although YinYang finished in acceptable time it produced a poor solution. The concept length is incredibly high and the F-measure is only slightly higher than what we retrieved for $\top$. Also, the predictive accuracy barely reaches 50%. The results we received, when stopping the DL-Learner, where as expected in Section 3.3, with a low precision and a high recall, but still significantly better than YinYang. The long time needed for a complete run of the DL-Learner makes the question necessary, *why* it needed that long and how it can be further optimized. A question we will analyze in the next section.

DL-Learner						
Concept	precision	recall	f-measure	pred. acc	avg length	avg time
Uncle stopped	44.19%	100%	61.29%	85%	6	465.95sec
Uncle	100%	100%	100%	100%	10	40,252.19sec
YinYang						
Uncle	40.76%	37.72%	39.18%	45%	246.2	505.1s
TOP and BOTTOM						
Uncle TOP	18.81%	100%	31.67%	50%	1	0sec
Uncle BOTTOM	0%	0%	0%	50%	1	0sec

Table 6: Results for the second experiment (concept $Uncle$) with more concepts defined explicitly in the background knowledge

4.4 Conclusions

In our experiments the DL-Learner clearly produced better results than YinYang. This might be due to the fact that we had to use an old version of YinYang, but we refrain here from speculating, what results a newer version of YinYang might produce. A clear advantage of the DL-Learner is the availability of the latest modifications, because it is freely available as Open-Source. Nevertheless, we clearly identified the limitations of both algorithms in our $Uncle$ experiment. Especially the DL-Learner, which we deemed more useful, based on our previous experiments, needed unacceptably long to compute a solution. This is due to the fact that it searches and evaluates all nodes up to a certain depth in the search tree (based on horizontal expansion). To find a solution for this problem is not trivial, because a change in the search process might lead to incompleteness in the algorithm. The use of other complete search algorithms like e.g. A* would certainly solve this problem, but it is difficult to choose an admissible heuristic. Neither the length of the concept nor the coverage of examples nor the horizontal expansion can be efficiently combined for a better heuristic.

5 Creating a Sample Ontology with Learning Support

In this section we propose an instance driven method for Ontology Engineering. The reason for this proposal is obvious. The realm of instances and individuals with their properties can be easily modeled by a human with domain knowledge only, while on the other hand modeling concepts requires more time, formal knowledge and also deep domain specific knowledge.

Imagine building a colour ontology, an issue that is still heavily argued about. Identifying different instances is easy, but the creation of background knowledge is still a matter of research. In case of e.g. a clothes company that needs to a certain extent ontological knowledge about colours, expenses for creating a complete colour ontology could not be justified. But even partial ontological knowledge comes at a price. As soon as a problem arises, e.g. a customer wants matching shoes for her red skirt, the need arises to find a simple solution.

The instance driven method we are proposing tries to simplify all problems a knowledge engineer might encounter by changing the view on the problem. Defining concepts is simplified to selecting example instances and creating background knowledge is simplified to assigning classes to instances.

In the following example we assume a knowledge engineer, which mostly has domain knowledge and only knows the basic principles of ontologies. His task is to design a domain specific ontology from scratch based on already existing instances from another data source.

5.1 Creating a Mercedes Benz Car Ontology

We chose the Daimler domain as an example because of its vast complexity. The Daimler AG offers its customers almost a free choice of configuration options when ordering a Mercedes Benz. Unlike other car producers which produce a large quantity of same car configurations, the Daimler AG assembles cars according to customer wishes. Estimates state that the factory in Sindelfingen alone can theoretically produce 10^{27} different cars. Creating an ontology to cope with this complexity is, given the actual procedures, a project which would consume an immense amount of resources. In this example we try to establish a new procedure, which could make it at least more affordable. The ontology we will create is by far not complete or exhausting, but we try to identify special steps, which can be eased by semi-automatic tool support. To put this procedure to practice much more research and case studies are needed. This section merely serves as a starting point.

5.1.1 Extracting Instances

Research has fostered numerous ways to convert conventional data sources to semantic enriched data. Some of those ways are very advanced and already produced amenable results. Besides the methods mentioned above (Section 2.3, [2] Wikipedia) several tools already exist. We will not go into further detail here, because our proposal builds upon those results. The interested reader is referred to [29], some existing tools are named here: D2R MAP[51] , Protégé Plug-In for Ontology Extraction from Text [12] [52]. Also, [26] provides a sound method to convert a relational database to an OWL ontology.

For our purpose we assume a simple relational database as existing data source, because it is the most common way to keep conventional data. The method for extracting the information from the data is straightforward and is only mentioned here for completeness. Every tuple in the database is extracted as one instance with foreign keys resulting in relations between instances. Classes were assigned according to table name and values in columns. The Tables 7, 8 and 9 show two different cars from the C-Klasse in the database.

The resulting A-Box can be found here:

C-Klasse (C-Klasse_Sportcoupe_1) .
Sportcoupe (C-Klasse_Sportcoupe_1) .
hasEngine (C-Klasse_Sportcoupe_1 , Engine_C_220_CDI_1) .
Engine (Engine_C_220_CDI_1) .
C_220_CDI (Engine_C_220_CDI_1) .
Diesel (Engine_C_220_CDI_1) .
R4 (Engine_C_220_CDI_1) .
hasTransmission (C-Klasse_Sportcoupe_1 , Transmission_Mechanic_Gear-6_Rear-Wheel-Drive_1) .
Transmission (Transmission_Mechanic_Gear-6_Rear-Wheel-Drive_1) .
Mechanic (Transmission_Mechanic_Gear-6_Rear-Wheel-Drive_1) .

[51] http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2rmap/D2Rmap.htm
[52] http://olp.dfki.de/OntoLT/OntoLT.htm

ID	Class	Type	Engine	Drivetrain
1	C-Klasse	Sportcoupe	1	1
1	C-Klasse	Limousine	2	2

Table 7: Car

ID	Label	Fuel	Cylinder
1	C_220_CDI	Sportcoupe	R4
2	C_350_4MATIC	Super	V6

Table 8: Engine

ID	Type	Drivetrain	Gear
1	Mechanic	Rear-Wheel-Drive	Gear-6
2	Automatic	4-Wheel-Drive	Gear-7G-TRONIC

Table 9: Transmission

Gear-6 (Transmission_Mechanic_Gear-6_Rear-Wheel-Drive_1) .
Rear-Wheel-Drive (Transmission_Mechanic_Gear-6_Rear-Wheel-Drive_1) .

C-Klasse (C-Klasse_Limousine_2) .
Limousine (C-Klasse_Limousine_2) .
hasEngine (C-Klasse_Limousine_2 , Engine_C_350_4MATIC_2) .
Engine (Engine_C_350_4MATIC_2) .
C_350_4MATIC (Engine_C_350_4MATIC_2) .
Gas (Engine_C_350_4MATIC_2) .
Super (Engine_C_350_4MATIC_2) .
V6 (Engine_C_350_4MATIC_2) .
hasTransmission (C-Klasse_Limousine_2 , Transmission_Gear-7G-TRONIC_4-Wheel-Drive_2) .
Transmission (Transmission_Gear-7G-TRONIC_4-Wheel-Drive_2) .
Automatic (Transmission_Gear-7G-TRONIC_4-Wheel-Drive_2) .
Gear-7G-TRONIC (Transmission_Gear-7G-TRONIC_4-Wheel-Drive_2) .
4-Wheel-Drive (Transmission_Gear-7G-TRONIC_4-Wheel-Drive_2) .

Instances were named after their classes for convenience only, just the id would have sufficed also.

The tables[53] were automatically (randomly generated) populated with legal configurations of cars according to certain rules (e.g. cars with 4 wheel drives can only have a 7G-TRONIC automatic gear). Since we did not have any electronic data, we created those rules manually, based on freely available brochures about the cars. The transformation from the table to the DL ontology is then straightforward as mentioned above. In total, 123 instances were extracted with 30 classes and 2 relations.

5.1.2 Creating a Subclass Hierarchy

We can now create a subclass hierarchy by finding all concepts C that contain at least all instances of a target concept E, i.e. the possible super concepts of E.

For a given set of instances, a solution would be to find all existing concepts that the instances belong to with the exception of the TOP-Concept, which would be a trivial solution, see Algorithm 4.

Algorithm 4: Finding super concepts for E

```
   Input: named class $E$
   Input: set of named classes $C$
1  $T$ = empty set of classes
2  foreach $c \in C$ do
3      $I$ = retrieve all instances belonging to $E$
4      $N = \{c \mid c(i), \forall i \in I , c \neq E \}$
5      $T = T \cup N$
6  return $T$
```

The engineer can then choose all superclasses for each concept from this list. In general the list shows only possible superclasses. The correct superclasses have to be identified by the engineer according to the domain knowledge. By applying the algorithm to the extracted instances and the knowledge base, the hierarchy shown in Figure 7 is created[54].

The resulting hierarchy is quite different from an intuitive approach. Several inclusions are not expected and would not result from an intuitive manual approach. The concept $4\text{-}Wheel\text{-}Drive$ (ger.: Allrad) is a subclass of the concept $Gear\text{-}7G\text{-}TRONIC$, which according to domain knowledge does not make perfectly sense. 7G-TRONIC is a special 7-speed automatic transmission which has at first glance nothing to do with the drivetrain (ger.: Antrieb). But all C-Klasse cars with a 4-wheel-drive

[53] Since this example was completely created from scratch, we actually did not make the effort to produce the tables just to extract the data from there. We created PHP-objects held in memory, which have the same built as the shown tables and exported them to RDF

[54] created with the DL-Learner

```
TOP
   C-Klasse
      Limousine
      Sportcoupe
      T-Modell
   Transmission
      Automatic
         Gear-5
         Gear-7G-TRONIC
            4-Wheel-Drive
               BOTTOM
      Rear-Wheel-Drive
         Gear-6
      Mechanic
   Engine
      Gas
         Super
            C_230
            C_280
            C_280_4MATIC
            C_350
            C_350_4MATIC
      Diesel
         C_200_CDI
         C_320_CDI
         C_320_CDI_4MATIC
      R4
         C_160
         C_180_Kompressor
         C_200_Kompressor
      V6
```

Figure 7: automatically created hierarchy

also have 7G-TRONIC transmission, thus rendering it a valid subclass. Almost exactly the opposite applies to the 5 and 6-speed transmissions. They are considered subconcepts of $Rear\text{-}Wheel\text{-}Drive$ (ger.: Heckantrieb), which is also valid, since all C-Klasse cars with 5 and 6-speed transmissions will have a rear-wheel-drive. This method thus proves useful, because it aides the engineer to design ontologies that might not reflect his view of the domain, but are formally correct, i.e. they match the data.

5.1.3 Identifying Disjoint Classes

To support the knowledge engineer further an algorithm can propose disjoint classes. In a strict hierarchy, subconcepts of a concept are normally disjoint. An easy algorithm can now test for all subconcepts of each concept, if this case applies and then present a decision to the engineer, which judges according to his domain knowledge. We will not present a formal algorithm here, because unlike above, numerous approaches already exist and are incorporated in tools already like in Protégé.

5.1.4 Learning Background Knowledge

Up to now, we automatically created an ontology that has instances, a subclass hierarchy and disjoint classes. We will now show, how the DL-Learner can be used to easily add complex background knowledge without the need to write formulas, but merely by selecting instances.

For the three most general concepts (C-$Klasse$, $Transmission$, $Engine$) we learned the following definitions, by choosing the 41 instances for each concept as positive examples and the remaining instances as negative examples; we *relearned* the concept.

$$\begin{array}{lll} C - Klasse & = & \exists hasTransmission.Transmission \sqcap \exists hasEngine.Engine \\ Transmission & = & Automatic \sqcup Mechanic \\ Engine & = & (\ Diesel \sqcup Gas\ ) \sqcap (\ R4 \sqcup V6\ ) \end{array}$$

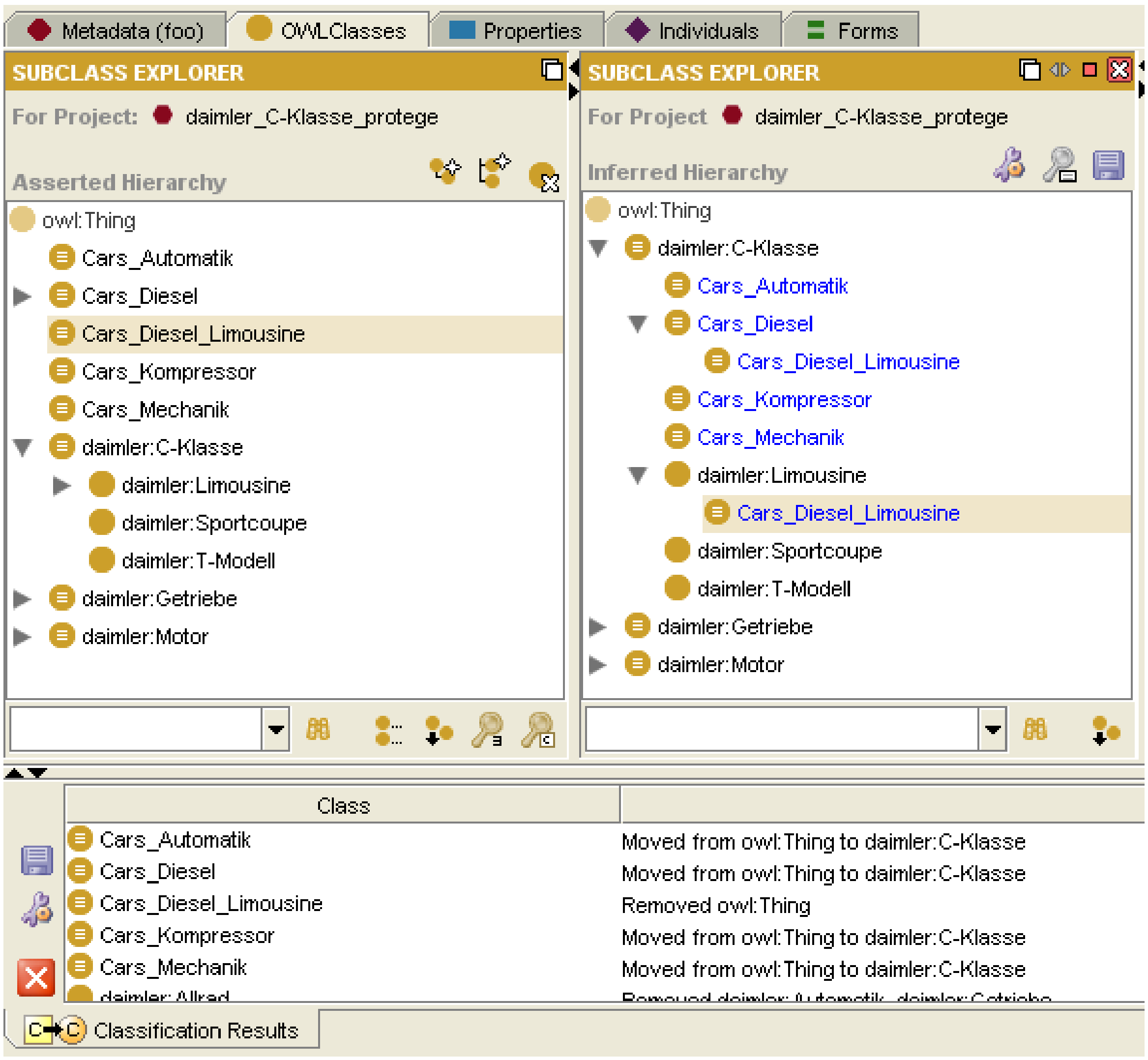
We added a total of 5 more classes and learned their definitions by selecting the appropriate instances as positive and negative instances. These newly added classes aim at providing structure to the ontology. They divide the 41 cars into subsets. The learned definitions are displayed here:

$$\begin{array}{lll} Cars\_Kompressor & = & \exists hasEngine.(C\_180\_Kompressor \sqcup C\_200\_Kompressor) \\ Cars\_Mechanic & = & \exists\ hasTransmission.Mechanic \\ Cars\_Automatic & = & \exists\ hasTransmission.Automatic \\ Cars\_Diesel & = & \exists hasEngine.Diesel \\ Cars\_Diesel\_Limousine & = & Cars\_Diesel \sqcap Limousine \end{array}$$

To validate the results we added the learned class definitions to the ontology and opened it with Protégé to use its ability to visualize inference. The Figures 8 and 9 show that the classes were included correctly in the hierarchy (Figure 8) and that the instances are correctly retrieved for each learned class (Figure 9).

5.2 Conclusions

We provided a process to create an ontology with minimal effort for a knowledge engineer. The most prominent result is, that the whole ontology was created completely without manual editing, which greatly eases the burden normally involved. We presented the ontology to Michael Herrmann, who currently writes his PhD thesis at the Daimler AG and discussed it in a meeting. We especially talked about the subclass problem, which we investigated in Section 5.1.2, where the concept 4-$Wheel$-$Drive$ was assigned a subclass of $Gear$-$7G$-$Tronic$. We came to the conclusion, that it definitely conflicts with domain knowledge and normally would not be considered a subclass, but in this case it might make sense, since it matches the data. The clear benefit is, that the problem is discovered and



Asserted Hierarchy

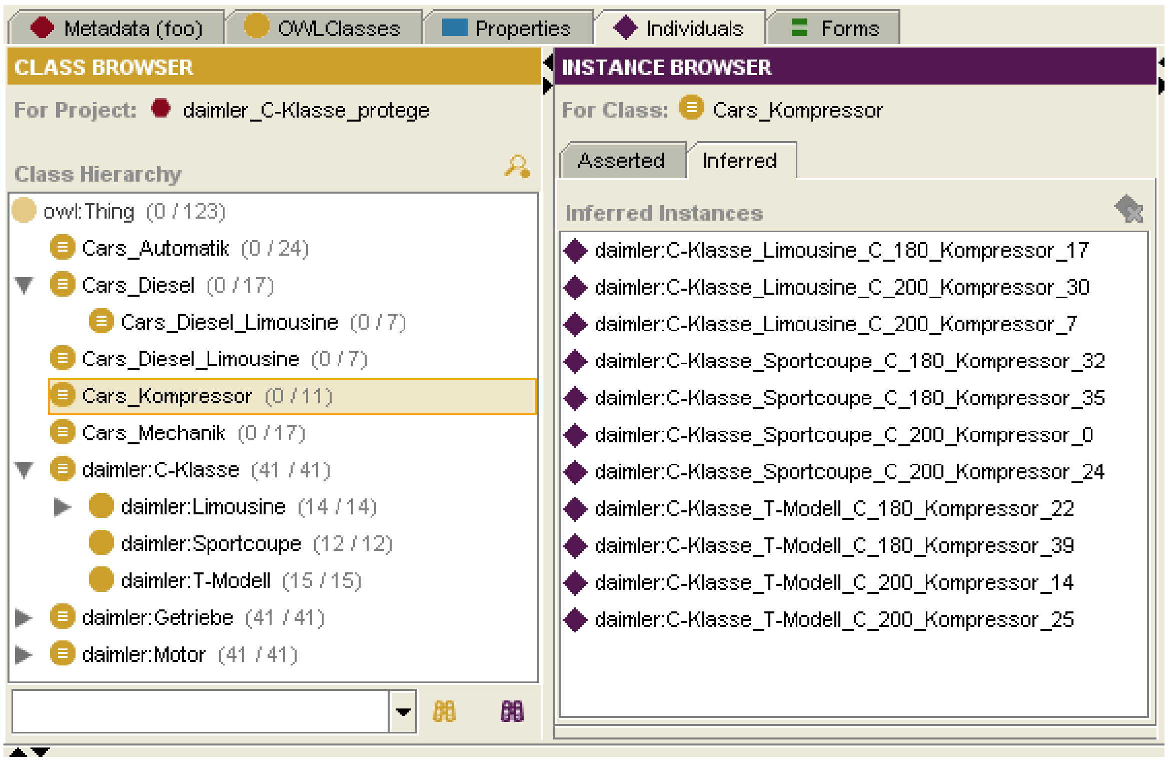
```
owl:Thing
  Cars_Automatik
  Cars_Diesel
  Cars_Diesel_Limousine
  Cars_Kompressor
  Cars_Mechanik
  daimler:C-Klasse
    daimler:Limousine
    daimler:Sportcoupe
    daimler:T-Modell
  daimler:Getriebe
  daimler:Motor
```

Inferred Hierarchy

```
owl:Thing
  daimler:C-Klasse
    Cars_Automatik
    Cars_Diesel
      Cars_Diesel_Limousine
    Cars_Kompressor
    Cars_Mechanik
    daimler:Limousine
      Cars_Diesel_Limousine
    daimler:Sportcoupe
    daimler:T-Modell
  daimler:Getriebe
  daimler:Motor
```

Class	
Cars_Automatik	Moved from owl:Thing to daimler:C-Klasse
Cars_Diesel	Moved from owl:Thing to daimler:C-Klasse
Cars_Diesel_Limousine	Removed owl:Thing
Cars_Kompressor	Moved from owl:Thing to daimler:C-Klasse
Cars_Mechanik	Moved from owl:Thing to daimler:C-Klasse
daimler:Allrad	Removed daimler:Automatik, daimler:Getriebe

Figure 8: The Protégé screenshot shows how the classes with the learned definitions are included in the hierarchy by inference.



For Project: daimler_C-Klasse_protege

Class Hierarchy

- owl:Thing (0 / 123)
  - Cars_Automatik (0 / 24)
  - Cars_Diesel (0 / 17)
    - Cars_Diesel_Limousine (0 / 7)
  - Cars_Diesel_Limousine (0 / 7)
  - Cars_Kompressor (0 / 11)
  - Cars_Mechanik (0 / 17)
  - daimler:C-Klasse (41 / 41)
    - daimler:Limousine (14 / 14)
    - daimler:Sportcoupe (12 / 12)
    - daimler:T-Modell (15 / 15)
  - daimler:Getriebe (41 / 41)
  - daimler:Motor (41 / 41)

For Class: Cars_Kompressor

Inferred Instances

- daimler:C-Klasse_Limousine_C_180_Kompressor_17
- daimler:C-Klasse_Limousine_C_200_Kompressor_30
- daimler:C-Klasse_Limousine_C_200_Kompressor_7
- daimler:C-Klasse_Sportcoupe_C_180_Kompressor_32
- daimler:C-Klasse_Sportcoupe_C_180_Kompressor_35
- daimler:C-Klasse_Sportcoupe_C_200_Kompressor_0
- daimler:C-Klasse_Sportcoupe_C_200_Kompressor_24
- daimler:C-Klasse_T-Modell_C_180_Kompressor_22
- daimler:C-Klasse_T-Modell_C_180_Kompressor_39
- daimler:C-Klasse_T-Modell_C_200_Kompressor_14
- daimler:C-Klasse_T-Modell_C_200_Kompressor_25

Figure 9: The Protégé screenshot shows how the instances are matched to the newly added classes.

presented as a choice to the engineer. The engineer therefore receives more options when designing an ontology and is not left alone with his opinion only. Besides this matter we also discussed the quality of the learned concept definitions. Michael Herrmann considered the learned concept definitions fairly simple, and concluded that an engineer with some skill in Description Logics could have defined them manually very fast without the proposed tool support. We share his opinion on this matter, but do not consider this fact as a failure of the tool support, but as a valuable suggestion for further improvement. As mentioned before in Section (2.2), the automatic selection of example sets is an issue still not solved, which would in this example further support the engineer in defining classes. Also the lack of more complex class definitions is directly related to the problem of the Open World Assumption with respect to concept learning (cf. Section 3.4). We agree that e.g. :

$$\begin{aligned} C - Klasse &= \forall hasTransmission.Transmission \sqcap \\ &\quad (= 1\ hasTransmission) \sqcap \forall hasEngine.Engine \sqcap (= 1\ hasEngine) \end{aligned}$$

or

$$Cars\_Diesel = \forall hasEngine.Diesel$$

would be stronger and more adequate statements, but could not be learned as of now. We nevertheless consider our approach successful, because we showed that concept learning can be used in an engineering process. This is important because it is the first step of evolution from an algorithm towards the creation of a useful application which can benefit the manifestation of the Semantic Web.

6 Related Work

Although we already related to significant other work in the respective sections, we would like to give a complete account with some additions here. Related work can basically be divided in two parts. The first part is only important for the extraction method in Section 2.3 and is concerned with other approaches for deductive reasoning on large knowledge bases and modularization/reuse, the second part gives an overview of the compared concept learning algorithms and other approaches for ontology enrichment different from concept learning.

In [16], Fokoue et al. (2006) describe an approach to produce a summary of the $ABox$. They argue that reasoning can be conducted efficiently on this summary and can be related to the original large knowledge base (although it has to be verified sometimes). While the approach successfully allows to spot inconsistencies, other inference methods like retrieval are not mentioned. Our method uses existing reasoners with full reasoning capacities and is efficient, because it reasons only over relevant parts, which successfully enables the use of applications that depend on reasoning [17]. It is further able to handle multi-domain ontologies in a remote scenario (over `HTTP`) and works without indexing and other preparation steps. The ability to relate inconsistencies spotted in the extraction to the original larger knowledge base, depends on the size of the extraction (recursion depth) and has not yet been investigated in detail. D'Aquin et al. (2007) [14] committed work about modularization of ontologies in general, mainly analyzing the problems and the criteria after which modularization should be conducted. They tested some existing tools for modularization and partitioning, which focus on $TBox$ axioms, while we devised an instance based method for extraction.

In this work we compared the concept learning algorithms proposed in [13; 19; 20; 25] in detail, whereas the latter three use refinement operators, which were first mentioned for Description Logics in Badea et al. (2000) [7] and later put on thorough theoretical foundations by Lehmann et al. (2007) [24]. Although our work is not concerned with the creation of a concept learning algorithm, we still consider Iannone et al. (2007) [20] and Lehmann et al. (2007) [25] closest to the essentials of our work, since both follow the goal to use concept learning as a tool for ontology engineering. Other interesting work that is concerned with tool support for knowledge engineers can be found in Baader et al. (2007) [3]. The approach is quite different, since it tries to adapt methods of Formal Concept Analysis to complete DL knowledge bases instead of learning concepts according to user-defined example sets. The proposed method for completion aims at asking a domain expert a minimal amount of questions about axioms that might not have been included in the knowledge base during the engineering/extraction process. The domain expert can then decide to include those axioms, which is similar to the acceptance of a learned concept definition as in our work. Without going into detail, we would also like to mention the work of Lisi et al. (2003) [27], which is concerned with refinement operators and concept learning for the hybrid language $\mathcal{AL}$-Log , which merges Datalog with Description Logics. Although their work might be valuable for expert systems, it surely does not match the paradigm of the Semantic Web, which has adopted OWL as its standard. We agree with the arguments given in Patel-Scheider et al. (2007) [31] on this issue and completely omitted these approaches from our analysis. The work of Maedche et al. [29] about *Ontology Learning for the Semantic Web* is complementary and looks at the process of ontology engineering from a broader perspective. Their approach for "learning ontologies" consists of four steps, which are: *import/reuse*, *extract*, *prune* and *refine*. In this context, concept learning can be sorted into the *refine* step. Our approach for extracting relevant knowledge from existing ontologies via SPARQL belongs to the *import/reuse* step after some modifications (cf. Section 8). For completeness, we would also like to mention the approaches of [22; 1], whereas the first expresses the idea to transform DL knowledge bases to feature vectors to be able to apply existing machine learning methods and the second uses a probabilistic Description Logic called YAYA. We did not consider both approaches for our investigations, because the achieved results are used for classification only and depend on internal classifiers, which can not be used to improve the original ontology.

7 Summary and Conclusions

We identified current problems of ontology engineering for the Semantic Web and connected the concept learning methods to the engineering process to show how some of these obstacles can be overcome. We further provided useful observations about how concept learning can be used for semi-automatic tool support or even to automatically enrich ontologies. Especially in combination with the proposed SPARQL extraction method, concept learning can be applied easily to very large ontologies once they are available over a SPARQL endpoint, which was not possible before. Furthermore, we investigated in detail the most promising approaches for concept learning, which are likely to become useful applications in the future (especially the DL-Learner). We identified theoretical weaknesses in YinYang, which occur during the creation of the MSC's and when the counterfactuals fail. The need for benchmarks for concept learning algorithms expressed in Lehmann et al. [25] is answered and thorough testing of the algorithms clearly revealed the necessity for further improval, before the threshold to becoming an application can be crossed. The created benchmarks are published in the Sourceforge project of the DL-Learner and might serve as the basis for the first concept learning repository for Description Logics. At the end, we tried to apply concept learning to a potentially real existing use case, the Daimler ontology with the result of identifying further weaknesses with respect to the value of learned concepts, but also showed that concept learning can be successfully used for tool support. We also provided a solution to learn concept definitions that could not be learned before without major disadvantages, because of the Open World Assumption.

8 Future Work

This thesis was created in the context of ongoing research by the AKSW group of the department of computer science, University of Leipzig, where it is part of a larger process.

The provided solution for closing an ontology for concept learning needs to be included in the DL-Learner framework for future releases. Based on the resulting ability of the DL-Learner to learn more complex concept definitions on the closed knowledge base, the car ontology can be recreated and extended. Once it reaches a sufficient size and quality, the created ontology could be evaluated by real domain experts, who work at a car producing company with the help of a questionnaire, which could become the first real use case for ontology engineering with concept learning support.

Based on the results of the experiments, the DL-Learner framework can be extended by another learning algorithm, that succeeds in learning the concept for $Uncle$ in reasonable time. Although it is not yet clear how the new learning algorithm will look like, the ability to directly test it on the benchmarks, will lower the time needed for improvement.

The availability of benchmarks will also enable a new comparison of algorithms as soon as the new version of YinYang will be published or any other approaches to concept learning surface.

The possibility of *Class Learning on SPARQL endpoints* [17], which is based on the extraction algorithm in this thesis, will eventually result in the DBpedia Navigator (implementation has already started), which will, combined with proper retrieval methods, enable users to navigate through sets of instances based on learned concept descriptions.

We also consider, the possibilities the extraction algorithm implies, as interesting. Based on the assumption that the function `extract` (cf. Section 2.3.2) can retrieve all important information for one instance under a certain aspect (defined by the filter *SQTF* and the recursion depth), it might be possible to create a semantic neighborhood of a single fact in a knowledge base. It would be interesting to show that this neighborhood, if it is well-defined, can be used to define and extract sensible packets of information from a knowledge base. These packets enable the sharing of Semantic Web knowledge based on instances. New domain ontologies can be created e.g. by defining starting instances and extracting sensible parts of one knowledge base like DBpedia and then relating those instances to another knowledge base, extracting more neighborhoods to merge them with the existing knowledge on a local level, thus easing reuse of structured knowledge throughout the Semantic Web instead of starting from scratch each time.

References

[1] Jordi Alvarez. A formal framework for theory learning using Description Logics. In James Cussens and Alan M. Frisch, editors, *ILP Work-in-progress reports*, volume 35 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2000. 54

[2] Sören Auer and Jens Lehmann. What have Innsbruck and Leipzig in common? Extracting semantics from wiki content. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 503–517. Springer, 2007. 11, 46

[3] Franz Baader, Bernhard Ganter, Baris Sertkaya, and Ulrike Sattler. Completing Description Logic knowledge bases using Formal Concept Analysis. In Manuela M. Veloso, editor, *IJCAI*, pages 230–235, 2007. 5, 53

[4] Franz Baader and Ralf Küsters. Non-standard inferences in Description Logics: The story so far. In D. M. Gabbay, S. S. Goncharov, and M. Zakharyaschev, editors, *Mathematical Problems from Applied Logic I. Logics for the XXIst Century*, volume 4 of *International Mathematical Series*, pages 1–75. Springer-Verlag, 2006. 23

[5] Franz Baader and Werner Nutt. Basic Description Logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors, *The Description Logic Handbook: Theory and Implementation and Applications*, pages 47–100. Cambridge University Press, 2003. 5

[6] Liviu Badea. Perfect refinement operators can be flexible. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 266–270. IOS Press, 2000. 26, 27

[7] Liviu Badea and Shan-Hwei Nienhuys-Cheng. A refinement operator for Description Logics. *Lecture Notes in Computer Science*, 1866:40–58, 2000. 26, 28, 31, 32, 34, 53

[8] Sean Bechhofer. The DIG description logic interface: Dig/1.1. Technical report, 2003. 8

[9] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001. 1, 2, 10

[10] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's Razor. In *Readings in Machine Learning*, pages 201–204. Morgan Kaufmann, 1990. 20

[11] Sebastian Brandt, Ralf Küsters, and Anni-Yasmin Turhan. Approximation and difference in Description Logics. In *KR*, pages 203–214, 2002. 22, 25

[12] Paul Buitelaar, Daniel Olejnik, and Michael Sintek. A Protégé plug-in for ontology extraction from text based on linguistic analysis. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2003. 9, 46

[13] William W. Cohen and Haym Hirsh. Learning the Classic Description Logic: Theoretical and experimental results. In Jon Doyle, Erik Sandewall Pietro Torasso, editor, *Proceedings of the 4th International Conference on*, pages 121–133, Bonn, FRG, May 1994. Morgan Kaufmann. 21, 31, 53

[14] Mathieu d'Aquin, Anne Schlicht, Heiner Stuckenschmidt, and Marta Sabou. Ontology modularization for knowledge selection: Experiments and evaluations. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *DEXA*, volume 4653 of *Lecture Notes in Computer Science*, pages 874–883. Springer, 2007. 53

[15] Thomas G. Dietterich, Bob L. London, Kenneth Clarkson, and Geof Dromey. *Learning and inductive inference*, volume III of *The Handbook of Artificial Intelligence*, chapter XIV, pages 323–512. William Kaufmann, 1982. 22

[16] Achille Fokoue, Aaron Kershenbaum, Li Ma, Edith Schonberg, and Kavitha Srinivas. The Summary Abox: Cutting ontologies down to size. In *ISWC*, pages 343–356, 2006. 12, 53

[17] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Class learning on SPARQL endpoints. In *ESWC 2008*, submitted. 13, 18, 53, 55

[18] Heinrich Herre, Barbara Heller, Patryk Burek, Robert Hoehndorf, Frank Loebe, and Hannes Michalek. General Formal Ontology (GFO): A foundational ontology integrating objects and processes. Part I: Basic principles. Technical report, Research Group Ontologies in Medicine (Onto-Med), University of Leipzig, 2006. 9

[19] Luigi Iannone and Ignazio Palmisano. An algorithm based on counterfactuals for concept learning in the Semantic Web. In Moonis Ali and Floriana Esposito, editors, *IEA/AIE*, volume 3533 of *Lecture Notes in Computer Science*, pages 370–379. Springer, 2005. 21, 25, 26, 27, 31, 40, 53

[20] Luigi Iannone, Ignazio Palmisano, and Nicola Fanizzi. An algorithm based on counterfactuals for concept learning in the Semantic Web. *Applied Intelligence*, 26(2):139–159, 2007. 7, 19, 21, 22, 23, 24, 25, 31, 34, 36, 53

[21] Yarden Katz and Bijan Parsia. Towards a nonmonotonic extension to OWL. In *Proceedings of OWL: Experiences and Directions Workshop. Galway, Ireland.*, 2005. 34

[22] Daniel Kudenko and Haym Hirsh. Feature-based learners for Description Logics. In *Description Logics*, 1999. 54

[23] Jens Lehmann. Concept learning in Description Logics. Master's thesis, TU Dresden, 2006. 5

[24] Jens Lehmann and Pascal Hitzler. Foundations of refinement operators for Description Logics. In *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP)*, 2007. 27, 53

[25] Jens Lehmann and Pascal Hitzler. A refinement operator based learning algorithm for the $\mathcal{ALC}$ Description Logic. In *Proceedings of the 17th International Conference on Inductive Logic Programming (ILP)*, 2007. 19, 26, 28, 29, 37, 53, 54

[26] Man Li, Xiao-Yong Du, and Shan Wang. Learning ontology from relational database. In *Proceedings of International Conference on Machine Learning and Cybernetics, Volume 6, Issue , 18-21 Aug. Page(s): 3410 - 3415*, 2005. 9, 46

[27] Francesca A. Lisi and Donato Malerba. Ideal refinement of descriptions in AL-log. In Tamás Horváth, editor, *ILP*, volume 2835 of *Lecture Notes in Computer Science*, pages 215–232. Springer, 2003. 54

[28] Alexander Maedche and Steffen Staab. Ontology learning for the Semantic Web. *IEEE Intelligent Systems*, 16(2):72–79, 2001. 8

[29] Alexander Maedche and Raphael Volz. The Text-To-Onto Ontology Extraction and Maintenance System. In *Workshop on Integrating Data Mining and Knowledge Management co-located with the 1st International Conference on Data Mining*, San Jose, California, USA, 11 2001. 46, 54

[30] Ryszard S. Michalski. Pattern recognition as rule-guided inductive inference. *Machine Intelligence*, 2(4):349–361, 1980. 37

[31] Peter F. Patel-Schneider and Ian Horrocks. A comparison of two modelling paradigms in the Semantic Web. *Journal of Web Semantics*, 2007. 54

[32] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990. 36, 40

[33] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements,. *Artificial Intelligence*, 48:1–26, 1991. 4

[34] Elena Paslaru Bontas Simperl and Christoph Tempich. Ontology engineering: A reality check. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 836–854. Springer, 2006. 8, 10

[35] Gunnar Teege. Making the difference: A subtraction operation for Description Logics. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the 4th International Conference (KR94)*, San Francisco, CA, 1994. Morgan Kaufmann. 25

[36] Patrick H. Winston. Learning structural descriptions from examples. Technical report, Cambridge, MA, USA, 1970. 36

Acknowledgement

Ich wollte mich nur bedanken bei allen, die mich, hauptsächlich moralisch, da das Thema doch sehr speziell war, unterstützt haben.

Ich danke Seebi für seine Ideen, wie man den Rahmen so einer Diplomarbeit gut organisieren kann und für seine herzlichen Begrüßungen besonders in unachtsamen Momenten.

Dank an Micha, der, während ich beschäftigt war, immer noch andere wichtige Sachen im Blick gehabt hat auf unserem gemeinsamen Weg durchs Studium.

Besonders besonderer Dank an meinen Betreuer Jens, der mich über viele Irrtümer aufgeklährt hat, auch wenn ich es zuerst nie einsehen wollte.

Hanne, Dir möchte ich im Speziellen danken, Vielen Dank

"Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe".

Ort Datum Unterschrift