Master Thesis - Applied Computer Science
Albert-Ludwigs-Universität Freiburg im Breisgau

SUSI: Wikipedia Search Using Semantic Index Annotations

Björn Buchhold

26.11.2010

Albert-Ludwigs-Universität Freiburg im Breisgau
Faculty of Engineering
Supervisor Prof. Dr. Hannah Bast

Supervisor
Prof. Dr. Hannah Bast

Primary Reviewer
Prof. Dr. Hannah Bast

Secondary Reviewer
Prof. Dr. Georg Lausen

Date
11/26/10

Contents

Abstract

We present Susi, a system for efficient semantic search on the English Wikipedia. Susi combines full-text and ontology search. For example, for the query `penicillin scientist`, Susi recognizes that scientist is a type of person, and returns a list of names of scientists that are mentioned along with the word `penicillin`. We argue that neither full-text search alone nor ontology-search alone is able to answer these kinds of queries satisfactorily. Along with the list of entities matching the query (the name of the scientists in the example), Susi also provides excerpts from the text as evidence.

The data structure behind Susi is an index for the CompleteSearch search engine. This index is enriched by constructs derived from facts from the Yago ontology. The challenge was to do this in a way that keeps the index small and enables fast query processing times. We present an annotation style, specifically designed to eliminate index-blowup associated with adding semantic information. In our experiments on the complete English Wikipedia (26GB XML dump), Susi achieved average query times of around 200 milliseconds with an index blowup of only 42% compared to ordinary full-text search.

We also examine result quality by comparing the contents of hand-compiled Wikipedia lists like "*List of drug-related deaths*" against the output of Susi for corresponding semantic queries (`drug death person`). We come up with a simple typification of the kinds of errors that can occur. One of our findings is that the vast majority of false-positives is due to false omissions on the side of the Wikipedia lists, while the vast majority of false-negatives is due to omissions in the Yago ontology.

Zusammenfassung

Wir präsentieren Susi, ein System für effziente, semantische Suche auf der englischen Wikipedia. Susi kombiniert Volltext-Suche mit Suche in Ontologien. Für eine Anfrage `penicillin scientist`, zum Beispiel, erkennt Susi, dass scientist eine bestimmte Art Person ist, und findet entsprechend die Namen von Wissenschaftlern, die mit dem Wort `penicillin` zusammen genannt werden. Wir argumentieren, dass weder Volltext-Suche, noch Suche in Ontologien allein, diese Art von Anfragen zufriedenstellend beantworten können. Zusammen mit der Liste von Entitäten, die auf die Anfrage passen, präsentiert Susi außerdem Ausschnitte aus dem Volltext als Beleg.

Die Datenstruktur hinter Susi ist ein Index für die CompleteSearch Suchmaschine. Dieser Index ist um aus der Yago Ontologie abgeleitete Konstrukte erweitert, die die gewünschte, semantische Suche ermöglichen. Die Herausforderung war dabei, den Index klein zu halten und schnelle Antwortzeiten auf Anfragen zu ermöglichen. In unseren Experimenten auf der englischen Wikipedia (26GB XML dump), erzielt Susi durchschnittliche Antwortzeiten von 200 Millisekunden mit einem Index, der nur 42% größer ist als für herkömmliche Volltext-Suche.

Außerdem untersuchen wir, durch Vergleiche mit manuell erstellen Wikipedia Listen wie *„List of drug-related deaths“*, die Qualität der Antworten, die Susi für entsprechende, semantische Anfragen (`drug death person`) liefert. Wir stellen eine einfache Einteilung möglicher Fehler in Kategorien vor. Eine unserer Feststellungen ist dabei, dass die Mehrheit der false-positives auf fehlende Einträge in den Wikipedia Listen zurückzuführen sind, während die Mehrheit der false-negatives auf fehlende Einträge in der Yago Ontologie zurückzuführen sind.

1. Introduction

Imagine the following, simple question: *What scientists have been involved with penicillin?* While this seems to be a question that could easily be answered with the help of Wikipedia, imagine a query for a search engine that is supposed to fulfill this purpose. Traditional search in the Wikipedia documents is not able to reflect the semantics of our query. Finding the word *scientist* is not what we want. Instead, we are interested in instances of the class scientist.

We follow the idea of combining full text and ontology search. In this thesis we discuss the creation of our system Susi that enables semantic search on the English Wikipedia. Since the term semantic search is quite loose, we will discuss the exact problem in the following. Section 1.1 introduces the motivation for work on that topic and points out which aspects are being addressed. Subsequently, section 1.2 explicates the thesis' contributions. Section 1.3, finally, gives a survey at the structure of the rest of this document.

1.1. Motivation

The idea of a Semantic Web where computers are supposed to understand the meaning behind the information on the Internet has been around for years. Modern approaches usually involve the usage of some database that provides a formal representation of the semantic concepts of a domain of interest. There is also great work that accomplishes semantic search in this sense. Some of them are described further in chapter 3. But while the technology works great, a query's result can only be as good as the database.

In general, many resources carry their information in somewhere in the text. Humans can understand it well, but it starkly differs from formal, structured representations that are used by existing approaches. Although information and facts can be successfully extracted from full-text [Suchanek et al. (2007)], the amount and depth of extracted information is always limited. Any extraction tries to distinguish important facts from lesser important ones. Usually this evolves around a specialization on some domain of interest. The majority of unspecific information remains hidden.

Search engines are without doubt the most popular way to find certain pieces of information in a huge amount of text. However, they usually do not try to grasp the semantics of a query. They rather aim to deliver results based on the occurrences of

query words in documents[1]. While this concept clearly has proven itself in practice, we can still find some limitations.

Think of the query `penicillin scientist`. This query should return scientists that are involved with penicillin in some way. First and foremost its discoverer Alexander Fleming but also the Nobel laureates that accomplished its extraction and many scientists that are involved in one way or the other. For ordinary queries this task is typically solved quite well by search engines. However, the difficult part of this particular kind of query is understanding the word `scientist`. Important hits may evolve around documents where penicillin is mentioned close to an instance of a scientist. Usually this is a name but a pronoun that references a scientist is also very likely. Additionally, there may be a mentioning of some chemist, biologist or whatever specific kind of scientist.

Consider the following example text from figure 1.1:

> Alexander Fleming was a bacteriologist. He discovered penicillin by accident.

Figure 1.1.: Example Text Excerpt

While being obvious for a human reader, the information that this is a hit for a scientist that has something to do with penicillin, is not obvious for a computer who does not understand human language. First of all, this may be an excerpt of a huge document that mentions many different scientists in several contexts. Hence, one somehow has to preserve the fact that Alexander Fleming is mentioned very close to penicillin or ideally that the pronoun *"he"* refers to him. Secondly, one has to know that the particular Alexander Fleming, that is mentioned here, is a bacteriologist or biologist. And finally, it has to be clear that a bacteriologist or biologist also is a scientist.

Luckily, the required information on types of persons (or all entities in a wider sense) is manageable. It is much more likely that semantic ontologies exist that contain this kind of facts than that there are ontologies that cover basically everything that is present somewhere in the full-text. Hence, combining the power of full-text search with knowledge from semantic ontologies should enable new possibilities. Naturally, there are lots of challenging questions involved and only some of them can be dealt with at a time. Chapter 2 points out which of them are taken into consideration for this thesis, which approaches are chosen to realize them and discusses this choice.

[1] Modern search engines are able to find similar words in the sense of error-tolerant search [Celikik and Bast (2009)], handle synonyms and more. However, these refinements are not relevant for the following argument and not discussed further here.

1.2. Contributions

This thesis presents the power of the combination of ontology and full-text search. Contributions include:

- Susi, a fully operational system based on an index for the English Wikipedia that allows the CompleteSearch [Bast and Weber (2007)] search engine handing queries that feature semantic categories.
- Heuristics for simple entity recognition in Wikipedia pages.
- Development, implementation and comparison of different concepts for annotating entities with their semantic categories.
- An implementation that fits the modular architecture of the CompleteSearch search engine, including:
  - A parser for the wikipedia markup that recognizes and handles entities.
  - Tools that extract relevant knowledge from Yago [Suchanek et al. (2007); Suchanek (2009)] and combine it with Wikipedia entities.
  - Integration in the CompleteSearch engine by providing files that stick to the standards used by CompleteSearch.
- An evaluation framework for semantic queries. Both, general difficulties with semantic search and those specific to susi are identified and classified into groups of problems.

While we are able to present a demo version that works nicely, there are still many things to improve. This thesis contains an outlook on possible future improvements. Some of them already include concrete solutions that can be put into practice without much effort, others only give an outlook on more complex problems that should be addressed in the future. However, all of them do contribute towards semantic Wikipedia search.

1.3. Structure of this Thesis

This document is separated into six chapters. Each chapter is supposed to provide the answer to a specific question.

Why do we do all this? This first chapter has introduced the problem and pointed out why we expect interesting possibilities for semantic search that have not been researched yet. Also, we have seen the contributions we wish to make by the construction of our system susi and its discussion in this document.

What exactly do we do? In the second chapter, we define the scope of this thesis and hence the scope of Susi. We want to make clear what aspects we want to cover now, which ones are postponed and give reasons for that choice.

What have others done? Chapter 3 presents what other researchers have contributed to the field. We identify and briefly examine related work concerning semantic search and point out where our approach differs.

How do we do it? The fourth chapter is the main chapter of this thesis. We present our solution step by step and try to give a general understanding for all aspects relevant to Susi. Finally, we reconsider a simplified example and try to see all of the steps discussed orchestrated and working together in action.

How well did we do it? Chapter 5 contains an evaluation of our work. We examine both, quality and performance of Susi on an experimental basis. We identify problems within the current version and relate the observed issues to them.

How can we improve? The final chapter contains a discussion. After a short conclusion, we present possible future work as a list. Whenever possible, we also try to state solutions for the problems to be tackled in the future and give an estimation of how much effort is involved for which point.

Throughout this document, we try to establish the query `penicillin scientist` as what could be called a "running example". Whenever possible, we relate the current topics to this example and demonstrate their purpose by showing how they contribute to processing this query.

2. Scope of this Thesis

This chapter points out the scope of this thesis. First we give a reason for choosing the general approach of full-text search with additional semantic categories and describe why we implement this as index with semantic annotations. Secondly we discuss why this thesis is limited to the full English Wikipedia. Finally we explain why the YAGO ontology [Suchanek et al. (2007); Suchanek (2009)] has been chosen as source of the semantic knowledge that is used to enhance the index we build.

2.1. Choice of the general Approach

Since the term semantic search is quite inaccurate, numerous approaches arise from research in this area. Usually, they even aim to achieve different goals. Some of them are described in detail in chapter 3. In this chapter, however, we discuss which approach is followed by this thesis and why we chose it.

Recall our goal as it was discussed in *Motivation* (sec. 1.1). We want to solve queries of the form `penicillin scientist`, similar to the way various search engines solve such queries. That means that we want to receive a list of documents[1] that match our query words. Specifically, we want to receive documents that match the meaning of the words. As discussed earlier, a sentence that is a proper hit should contain the words penicillin and chemist, or penicillin and the name Alexander Fleming and so on.

In [Guha et al. (2003)] the authors distinguish between two kinds of search: *navigational search*, which is supposed to find a document that is relevant for the query, and *research search* that is supposed to gather information on an object denoted by the query. However, there is also a third, interesting kind of search. Consider the example above again. It is entirely conceivable to understand the query in a sense of: "*Which* scientist has something to do with penicillin?". Likewise, imagine a query `penicillin discovered scientist` to find out who actually discovered penicillin. If we simply regard our search results and additionally distinguish which exact word occurrence lead to a hit, i.e. the word "*biologist*", the name "*Alexander Fleming*" or the name *"Howard Florey*[2] *"*, we can easily provide answers to this kind

[1] for the sake of this argument it does not matter if the term "documents" refers to documents in a classical sense or if it only refers to some unit of words; a sentence for instance.

[2] *Howard Florey is a scientist invloded in the extraction of penicillin as a drug.*

of search, too. Doubtless, enabling this kind of search can be really useful and hence it is the main aim of this thesis.

Note, that our goal is to exceed simply giving an answer to the query. The matching sentence in the document acts as evidence for that fact.

In order to reach this goal, we need to know about the semantic relations between certain words on the one hand, and on the other hand, to comprise the knowledge about co-occurrence of words in our document collection. In the example above, this refers to facts like *"a chemist is a scientist"* or "*Alexander Fleming is a scientist*" and to the fact that the word "penicillin" co-occurs with some sort of scientist in our documents. We use two sources of input: A document collection with lots of facts (see *Limitation on the English Wikipedia* (sec. 2.2) for why the English Wikipedia was chosen for that purpose) and an ontology that provides the semantic information we need (see *Incorporation of the Yago Database* (sec. 2.3) for why YAGO [Suchanek et al. (2007)] was chosen). Naturally, we need to combine this information in a way that enables really efficient queries, since the amount of data will be huge. Basically, we can distinguish two approaches:

1. Adding the information from the the document collection's text to the ontology and keep the ontology in a well-suited data structure that allows efficient queries.
2. Constructing an index on the words in the document collection, just like common search engines do and somehow add the additional knowledge from the ontology.

As pointed out before, no valuable information should be lost and we do not want to restrict ourselves on a certain domain. Hence, we end up with a huge amount of data. The current version of the English Wikipedia, at the time of writing this thesis, has more than 1.4 billion word occurrences in about 130 million sentences. Assume the semantic ontology is represented in RDF[3] (Resource Description Framework), which is a very reasonable format that is used by many well-known ontologies, including YAGO. If we really were to model co-occurrence with all words that possibly matter, we would end up with billions of triples. This is quite large regarding that this year's Semantic Web challenge[4] encourages researchers to present useful end-user applications that use their so-called *"billion triples"* data set with 3.2 billion triples. Hence, we know for a fact that the resulting ontology would be of critical size.

Classical search engines and their indices, on the other hand, operate on even larger data sets without problems. Just think of the big web search engines as an extreme example. They create an index that allows to look up in which documents some

[3]The Resource Description Framework (RDF) Carroll and Klyne (2004) is a W3C specification for modelling facts. A single fact is represented as subject-predicate-object triple, e.g. "albert_einstein - isA - physicist". Conceptually, multiple facts form a directed, labeled graph. See Carroll and Klyne (2004) for further details.

[4]http://challenge.semanticweb.org/

word occurs within a few milliseconds. For queries with multiple words, they can intersect the document list. See chapter 4 for more detailed information on this topic. Consequently, we want to use such an index and somehow incorporate knowledge from our ontology.

The information from the ontology can be added to the index in various ways. We use annotations that simply are additional words written to the index. Details are discussed in chapter 4. Anyway, all techniques that are around for search engines obviously still apply for such an augmented index; in particular prefix search which is of high importance for our approach as discussed later. Still, the additional annotations may possibly blowup the index size and hence spoil everything. The index blowup is one of the main issues addressed by this thesis and the steps that provide a solution are outlined in chapter 4.

2.2. Limitation on the English Wikipedia

The previous chapter explains the need for a document collection. The English Wikipedia is great in both ways, rewarding to perform semantic search on, and comparatively easy to use.

First of all, it is a huge data set and full of interesting facts. After all, it is the largest encyclopedia ever assembled. Hence, being able to perform semantic search on this source is without a doubt really valuable. Secondly, from our perspective, several facts make it really easy to work with. Documents have a common structure that is a lot clearer than the structure of an arbitrary collection of web pages, for instance. There is always a title, headlines, paragraphs and the whole Wikipedia is available in XML[5] format. The size of this dump currently exceeds 26GB so we are able to deal with decent collection size. Last but not least, entity recognition in full-text is immensely simplified for correctly maintained Wikipedia pages, since the first mentioning of some entity is always supposed to be linked to the entity's article. This fact allows for simple heuristics to perform decently in recognizing entities, whereas more complex rules can be added to improve the recognition even further. More details on entity recognition are given in chapter 4.

2.3. Incorporation of the YAGO database

In addition to the document collection as main source of facts, we previously pointed out the need for an ontology that contains facts on the relations between words, e.g. "a chemist is a scientist", "a scientist is a person" or "a person is a living thing". This kind of knowledge is not easy to gather and requires an immense amount work that only humans can do. Fortunately there is a project called WordNet [Miller

[5]http://download.wikimedia.org/enwiki/

(1994)] that contains exactly this kind of information. WordNet is described further in chapter 4.

Since there is no alternative to WordNet at the time of writing this thesis, an ontology has to make use of it in order suit our needs. YAGO [Suchanek et al. (2007); Suchanek (2009)] does fulfill that criteria. Additionally, YAGO explicitly establishes the relation between WordNet's entities and Wikipedia categories and ultimately even contains "isA"-facts for entities, e.g. the fact that *"Alexander Fleming is a microbiologist"*. The actual content and structure of YAGO is described later in this document and not of importance here. Summing up, we want to make clear that YAGO offers everything our search requires from an ontology and its closeness to the English Wikipedia highly facilitates its incorporation.

3. Related Work

Considering semantic search in general, most research features search in ontologies in RDF [Carroll and Klyne (2004)] format. Usually this goes hand in hand with a query language. So in a way, all kinds of engines that enable a query language for RDF graphs can be seen as related work. RDF-3X [Neumann and Weikum (2008)] is presented exemplarily. Related work with exactly the same goal as this thesis, on the other hand, is quite rare. The only publication we know of, ESTER [Bast et al. (2007)], is discussed in the beginning of this chapter. Subsequently, we regard another approach. In [Suchanek (2009)] F. Suchanek describes the YAGO ontology and its automatic construction and growth. Although this is no search engine at all, it, first of all, served as an important foundation for this thesis. Secondly, enabling semantic search by querying ontologies requires automated construction of the later and hence projects like YAGO are inevitable for this approach and should be mentioned here.

3.1. ESTER

From all existing work, ESTER (Efficient Search on Text, Entities, and Relations) [Bast et al. (2007)] is the project that shares the most similarities with SUSI. The fundamental understanding of semantic search, i.e. how queries look like and what results are desired, goes hand in hand with the goal of this thesis. On top of that, the combination of YAGO and the English Wikipedia is also used for ESTER. In order to accomplish its goal, ESTER reduces the problem of semantic search to two operations: prefix search and joins. In [Bast et al. (2007)], the authors precisely carry out how ESTER works in general. This is not repeated in this document. Instead we consider an example query, examine how ESTER solves it and do some foreshadowing on which aspects are handled differently in this thesis.

Consider our example query `penicillin scientist`. ESTER uses an index with word-in-documents occurrences that allows efficient prefix-search. For `penicillin*`, the first part of the query, a simple look-up yields a list of pairs of words that start with the prefix `penicillin` and the documents in which they occur. For our example query, a slightly different query is actually processed, i.e. `penicillin person:*` (the reason for this extension is pointed out below) which returns the following list:

$$l_1 = \{(w, d) \mid w \text{ is a word that starts with the prefix "person:"; } d \text{ is a list of documents in which } w\text{and the word "penicilin" occur.}\}$$

The second part of the query is more interesting, since the additions for handling semantics come into play here. In general, Ester uses the following concept: Occurrences of entities are recognized in the text and special words, marking an occurrence of the recognized entity, are written to the index. Additionally, facts about each entity are located at a certain position in the document that describes the entity itself. For semantic queries, the entity occurrences are joined with the entity's facts.

So now let us examine this principle in practice and see how the second part of our query is handled. First of all, Ester has to decide if there is a semantic class that fits the query word `scientist`. Therefore, it launches a query of the form *`baseclass:scientist*`*. In this case, the query would be successful and the class `person` would be found. This class is used as the level of detail on which joins are performed. On a side note, wise choice of this base-classes is necessary to avoid very large lists when performing the joins. In a next step, the query `class:scientist - is_a - person:*` (where `-` is a proximity operator, which relates to the particular position in the documents where the facts about entities are stored) yields another list of word-in-documents pairs, such that:

$$l_2 = \{(w, d) \mid w \text{ is a word that starts with the prefix person: ; } d \text{ is a list of documents in which } w\text{occurs.}\}$$

One of those w's will be a word like `person:alexanderflemming`, that likely in fact co-occurs with the word `penicillin` in some documents. Other w's will be words like `person:aristotle`, that do not co-occur with `penicillin`. Likewise, l_1 contains numerous persons that co-occur with the word penicillin but do not necessarily have to be scientist. Hence, the final step in solving the query is a join of the lists l_1 and l_2 that uses w as join attribute. The resulting list, finally, is the desired answer to the semantic query.

The approach used in this thesis is quite similar to the ideas behind Ester. However, there are some significant differences. First of all, entity recognition in the text is done differently. Secondly, we regard sentences as documents instead of whole Wikipedia articles. This improves precision as pointed out in chapter 4. Nevertheless, the most interesting difference is that we aim to reduce the problem even further. Instead of using prefix search and joins, we show that we can solve this problem with prefix search only. This can easily be done by writing the facts always

next to an occurrence of an entity. Unfortunately, this blows up the index size and therefore Ester uses joins to avoid this phenomenon. The techniques carried out in chapter 4, however, can be used to significantly reduce the blowup and hence solve semantic queries with prefix search only without letting the index size explode.

3.2. YAGO, LEILA and SOFIE

As we have already discussed at several points in this document, Yago is an ontology and no search engine itself. However, huge parts of its construction are automated. This means that automatically retrieving facts from full-text, esp. the English Wikipedia is part of the research done for Yago. Hence, even though the goal is different, there is a relation between the work concerning the construction of Yago [Suchanek et al. (2007); Suchanek (2009)] and this thesis.

In his PhD thesis, titled *Automated Construction and Growth of a Large Ontology* [Suchanek (2009)], the author presents the concepts behind Yago. Instead of going into detail too much, let us focus on the overall structure. There are three building blocks:

Yago is the name of the ontology that is constructed. This is also the part that is used by this thesis' work to enrich full-text search with additional semantic knowledge.

Leila is a tool that extracts information from the text. It uses linguistic analyses to understand the structure of sentences rather than seeing them as sequence of words only.

Sofie validates new facts, ensuring that they do not contradict existing facts. This ensures that only decent facts are added to Yago.

Using the public demo, it is easy to convince oneself that this composition of tools works pretty well and hence Yago has been chosen as ontology used by Susi. As discussed in chapter 6, many of the concepts from [Suchanek (2009)] can be examined in order to improve the entity recognition. However, the construction of an ontology like Yago is a totally different goal than what this thesis aims to achieve. First of all the produced output obviously is different and secondly Yago limits itself on a fixed set of facts that are actually extracted and added to the ontology. For example there are facts like `isA` or `bornInYear`. Other than that, this thesis' search engine aims for a lot more arbitrary facts with the flavor of `hasSomethingToDoWith` that relates to co-occurrence of words.

3.3. RDF-3X

The most popular query language for Rdf is Sparql [Prud'hommeaux and Seaborne (2008)]. Since Rdf is the most common format to represent semantic ontologies, a

SPARQL engine can also be regarded as performing semantic search. RDF-3X [Neumann and Weikum (2008, 2009)] is such a SPARQL engine that is able to process even complex queries on large ontologies in milliseconds. In the following we briefly look at the basic ideas behind RDF-3X. Those ideas are based on cleverly constructed indices which might be useful for any kind of search engine.

Recall that RDF represents facts as subject-predicate-object triples. SPARQL supports conjunctions of triple patterns, which correspond to select-project-join queries in a relational engine. In order to solve a SPARQL query, it is therefore necessary to support retrieval of triples with zero to two variable attributes and joins between RDF triples. So first of all, the internal storage of the triples is an issue. While some approaches favor a table for each distinct predicate, RDF-3X uses a single table with three columns for subject, predicate and object. For that table, RDF-3X maintains indices for each possible permutation of subject, predicate and object. This enables fast look-up of patterns by simple range scans on the correct index whereas the ordering of the result is already known depending on the index used. Consequently, joins on any join attribute can also be performed efficiently.

Apart from that, RDF-3X speeds up complex queries with multiple joins by processing them in a clever way. First of all, statistical histograms help to predict the result size of a join. This allows ordering joins in a way that they can be performed more efficiently. Secondly, in [Neumann and Weikum (2009)], the authors present ways how RDF-3X speeds up joins even further, using methods of efficient list intersection that remember gaps in sorted join candidates.

In summary, we can see that RDF-3X enables very fast search in large ontologies by processing SPARQL queries. But as discussed earlier, this thesis aims for search also in full text and not only in an ontology. Additionally, SPARQL is a query language that is not as intuitive as a query to a search engine ideally is. Still, some concepts, like the join heuristics for instance and the general idea of building multiple indices, are interesting and can probably be useful for semantic search in full text as well.

3.4. DBpedia

DBpedia [Bizer et al. (2009)] is a community effort to extract structured information from Wikipedia and to make this information available on the Web. It is an ontology in RDFS format, that combines many concepts from other ontologies or directly extracted from Wikipedia. It is very similar to YAGO since it uses the class and type hierarchy from it. Many facts in DBpedia are obtained by extractions from the info boxes on Wikipedia pages.

There is also an application that uses DBpedia in order to offer a faceted Wikipedia search [Hahn et al. (2010)]. While this application provides great search on the relations from the DBpedia ontology with a sophisticated user interface, there are some notable differences to our work:

First of all, the search from [Hahn et al. (2010)] does not provide evidence. The facts are directly taken from the ontology and their origination is no longer known. Therefore, hits are the entities' Wikipedia articles and not all pages that contain a suitable fact. Secondly, semantic search is limited to the relations offered by the ontology. The only full-text that can be searched is the short abstract in the beginning of each entity's Wikipedia article. Additionally this search is limitted on the abstracts of the entities that are not filtered out by some semantic facet. Consequently, a query for `penicillin` faceted by the type `scientist` returns much less hits than Susi does for the same query, because all facts where a scientist is mentioned with penicillin outside of the abstract and all facts in a document other than the one for the scientist itself, are missed. Finally, the public demo of the application appears to have response times of tens of seconds for many queries. This may be acceptable for some use-cases, but is not what we want to focus on. However, the great user interface is an inspiration for future work.

4. SUSI

In the previous chapters, we have seen a motivation for semantic search, defined what exactly we want to achieve and had a look at research that is already present in this area. This chapter now covers the way that actually solves our problem. We present a system Susi (Wikipedia **S**earch **U**sing **S**emantic **I**ndex Annotations) that enables semantic search on a combination of the full-text of the English Wikipedia and the Yago ontology. We compare possible choices whenever there are multiple available. Especially, when it comes to supplying entities with semantic facts. we can distinguish several methods to add the desired information to the search index. By actually implementing them, we are able to provide concrete numbers for their application on the full English Wikipedia database. Hence, we can easily and precisely compare them.

However, at first, we want to recall necessary foundations. Afterward, we discuss all steps towards a semantic search engine individually. Finally, we reconsider our example query `penicillin scientist` and reconstruct how it is actually solved. This demonstrates the worth of every step presented previously.

4.1. Foundations

This first section briefly recalls the foundations for Susi. We only want to name the important building blocks behind the underlying search engine and describe the general idea behind those concepts. For more details please refer to the material in the references.

First of all, we have a look at the Hyb index that allows super-fast prefix search. Secondly, we regard WordNet which provides information on English words and their relation. A physicist, for example, is a scientist, a person, an organism and more. The effort put into the creation of WordNet is crucial for Susi. This section finally ends with a short description of CompleteSearch, a fully operational search engine that features the Hyb index and is the engine behind Susi.

4.1.1. Prefix Search and the HYB Index

The Hyb index [Bast and Weber (2006)] is an index for search engines that allows fast prefix search. Thus, it enables auto-completion search as known from various

applications. But prefix search is also important for other features. For example, synonyms can be supported. As an example consider the words `novice` and `beginner`. Instead of simply writing the words themselves, one can write something like `syngrp1:novice` and `syngrp1:beginner` to the index. A prefix-query for `syngrp1:*` should now find occurrences of any of the words. Likewise, for semantic search it is very useful to have a powerful tool like prefix search. E. g., we have seen in chapter 3 that ESTER is build around two operations, one of them being prefix search. On top of that, SUSI even reduces semantic search to prefix search only.

However, a big challenge for prefix search is that multiple words match a query and it is therefore required to merge the document list of all those words in order to provide the result. The HYB index stores precomputed doc lists in a very clever way. Just like an encyclopedia may be split across multiple books, the HYB index divides the possible prefixes in multiple blocks and stores each of them independently. Depending on the query, multiple blocks can be read and united, or the doc-lists read can be filtered so that they fit a more precise query. Note that sufficiently small block sizes will lead to negligible costs for filtering.

Block	List of `word - document` occurrences								
ADD - AZZ	Word	apple	add	apple	aristotle	angel			
	Document	1	47	51	62	120			
B - BYG	Word	boy	band	blue	boy	back	big	bang	big
	Document	2	30	30	30	47	402	507	507

Table 4.1.: HYB Example

Table 4.1 illustrates this principle of dividing the index into several block. Note that this is only an example and there are in fact numerical IDs stored for words and documents for obvious efficiency reasons. The division into blocks overcomes the problem that storing all those precomputed lists leads to a much larger index. In [Bast and Weber (2006)], the authors mathematically show that the HYB index does not use more space than a conventional inverted index. Hence, the HYB index is a very powerful way to enable prefix search in an efficient way.

4.1.2. WordNet

WordNet [Miller (1994)] is a semantic lexicon for the English language developed at the Cognitive Science Laboratory of Princeton University. It associates words to certain classes and provides relations such as subClassOf. This relation is exactly what we need for our semantic search, since it contains the fact that a scientist also is a person. Conceptually, the subClassOf relation in WordNet spans a directed acyclic graph with a single root node called entity. Although we only make indirect use of WordNet through YAGO, most of the interesting facts for our work are actually

inherited from WordNet which therefore deserves to be mentioned here in addition to YAGO.

4.1.3. CompleteSearch

CompleteSearch [Bast and Weber (2007)] is a fully functional search engine that supports a variety of features. It uses the HYB index which has been discussed previously and implements all necessary components to allow searching by end-users. There are multiple research projects (e.g. [Celikik and Bast (2009)]) in different areas of search engines that blend into CompleteSearch. The same thing is true for SUSI. CompleteSearch's modular architecture allows substituting certain components and keeping the rest. Hence, we can provide a special index for the English Wikipedia (plus some minor modifications to the user interface) that will enable semantic search, while no changes are necessary to the basic operations of the search engine itself.

4.2. Enabling Semantic Wikipedia Search

As discussed above, SUSI is build into the CompleteSearch engine. Actually, the semantic Wikipedia is nothing else but a database built from a document collection to search on. Semantic information is added during construction and treated as if it was part of the original documents. For CompleteSearch, such a database consists of three parts. The search index, the vocabulary and a structured version of the original text that allows displaying excerpts for all hits. Since the vocabulary can directly be found in the index, the interesting parts are the construction of the file for the excerpts and especially of the search index itself.

4.2.1. Index Construction

Recall the HYB index that has been discussed in the section on necessary foundations. Although the HYB index structures its data into several blocks, the smallest units of information remain postings. Such a posting is basically a word-in-document pair, while CompleteSearch has additional support for scores and positions. So while the actual index consists of multiple blocks, the input to index construction is just a set of postings. In the following, we can imagine the search index as a set of lines.

Note that in practice, word-ids are used instead of the actual words and the input is passed in binary format. This is absolutely crucial to performance but for the sake of all arguments and explanations we can always imagine the readable format from table 4.2. For SUSI, we want to see each sentence (or row in a list or table) as document. This leads to the following behavior of the search engine. A hit for

word	document	score	position
this	1	0	1
example	1	0	2
is	1	0	3
simplified	1	0	4

Table 4.2.: The index as set of postings.

all query words is a sentence that contains all of this words, supposing this is a fact in the Wikipedia that the query attempts to find. Actually, using sentences is just a very basic heuristic for determining the scope of facts. For future work we might consider trying to find better units by linguistic analysis. See chapter 6 for further notice.

Positions momentarily only play a minor role and scores are currently even unused. However, they may be inferred soon to realize new features.

4.2.2. Enabling Excerpts

In addition to the index, we also need to produce a file that contains the necessary information for providing excerpts for each hit. For the basic search features, this is quite easy. We simply write a line with `<doc-Id> <TAB> u:Wikipedia URL <TAB> t:<title> <TAB> H:<original text>` into a file. The CompleteSearch engine is then able displays excerpts that belong to a hit in a nice format, provide a link to the corresponding Wikipedia article and also highlights the query words.

However, Susi contains artificial words that contain the semantic additions. Highlighting those requires special effort, because the artificial words should not be visible and the corresponding entities should be highlighted instead. Fortunately, the CompleteSearch excerpt generator already supports special rules for artificial words. This is best explained with an example. Consider the string `^^A^^B^^C^^^D`. The letters `A`, `B` and `C` (preceded by `^^`) now will not be visible in the excerpts. It is D (preceded by `^^^`), that is displayed in their place. If a query now contains any of the first characters, the `D` will be highlighted instead by the excerpt generator. So for Susi, we can simply hide our artificial words for the semantics and highlight the corresponding entity instead.

The following screen-shot is taken from the current version of Susi. It shows a query for the entity Alistar Sinclair that is supposed to co-occur with the words geometric and algorithm.

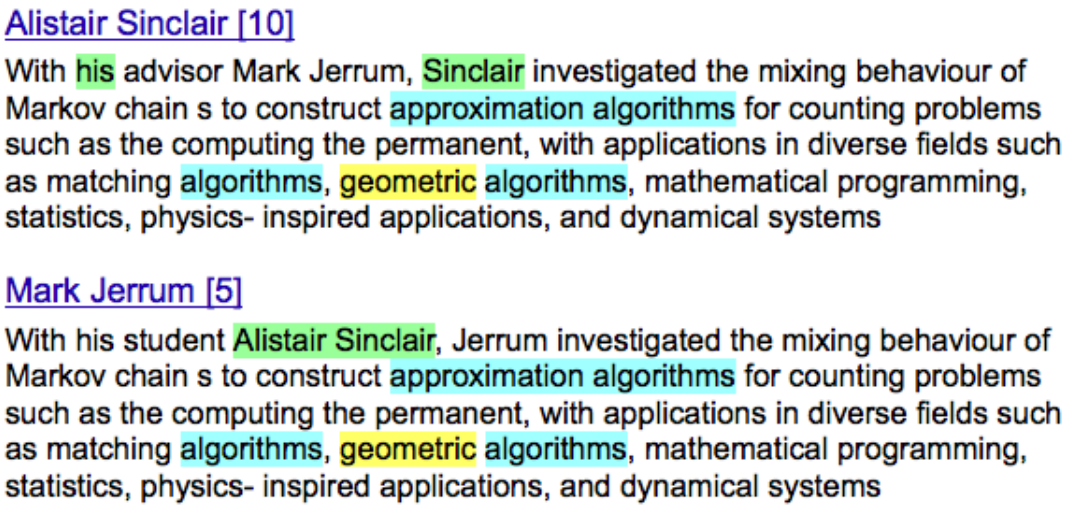

Figure 4.1.: Highlighting Alistair Sinclair, geometric and algorithm in the excerpts.

Not only does figure 4.1 demonstrate how well the highlighting of words like "his" works, it also shows that SUSI properly identifies entities. As a fun fact, we also get to see that even Wikipedia authors seem to like to copy from Wikipedia.

4.2.3. Implementation

Both, the index and the file with the excerpt information, are constructed while parsing the Wikipedia dump. The process takes several files as input which are constructed beforehand. For example, we use a redirect map that maps links to redirecting Wikipedia pages to a distinct entity name for every entity, a list of pronouns that are words which will reference some entity and many more. Apart from the information for parsing, the input also contains the data extracted from YAGO. The format of this data is discussed later in this section.

The software that builds index and excerpts file consists of three classes. The structure is very simple but still applies a clear separation of concerns:

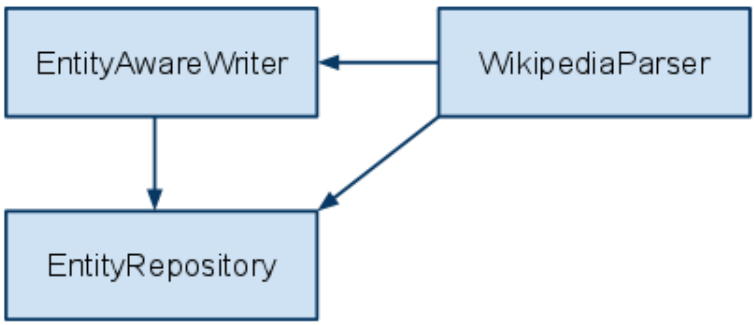

Figure 4.2.: The Semantic Wikipedia Parser

Figure 4.2 illustrates this concept. The class `SemanticWikipediaParser` (`SWP`) handles parsing the markup and recognizes, words, sections, links (to entities), tables, math and more. According to what is found the `SemanticWikipediaParser` calls methods of the `EntityAwareWriter` (`EAW`).

Deciding what is actually written to the output lies in the responsibility of the `EntityAwareWriter`. It provides methods that handle entity occurrences through links, redirects, entities with renaming, beginnings of new documents, new sentences or new sections to give some examples. The EAW then reacts accordingly. Additionally, it also decides on the format of the things written to the index and the excerpts file. Just think of the rules for highlighting words in the text that reference an entity in place of it. However, the exact things to write for some words depend on the context. For example, reacting on the word "he" will be different in the beginning of a document about tomatoes where no person has been mentioned yet and in a document about Albert Einstein, where the pronouns will likely reference a person. The word "apple" will require different things written to the index if it is found in a document Apple Computers or an article on Isaac Newton where an apple falling from a tree is mentioned.

This context information is stored in the EntityRepository (ER). The EntityRepository knows entities that previously occurred in the same section, the document entity, the last entity that occurred at any point in time and entities that relate to "active" (the scope depends on settings passed to the parser call) section headings. Whenever the EAW is supposed to write some word, it can query the EntityRepository to determine if the word might refer to an entity and which entity this would be.

4.3. Wikipedia Markup Parsing

Building a search index for the English Wikipedia, obviously requires access to the content of the Wikipedia in a proper format. Fortunately, dumps are being made

and provided for download on a regular basis. These dumps are available in XML format that sticks to a reasonably simple XSD[1]. The content of every Wikipedia page can be found in a single XML tag and formatted directly in Wikipedia markup. Hence, parsing the Wikipedia dump consist of two steps: Parsing the XML for the page tags and subsequently parsing the markup of each page. While there is nothing special to parsing the Wikipedia XML, maybe except for the fact that using a SAX strategy is absolutely mandatory due to the sheer size of the data, parsing the markup is special and therefore discussed in the following.

There are some challenges to parsing the Wikipedia markup:

1. Wikipedia accepts messy pages. An error in the markup will result in some formatting issues or a link not working correctly but the page will still be displayed in general. Sometimes, minor mistakes will not even have an effect at all. Consequently, there are many pages with flawed markup in the dump and a parser may not rely on opening tags to eventually get closed and so on.
2. Wikipedia is nice to its writer. There are many ways of doing the same thing. For example, there are special templates for tables that are usually used, but HTML tables do work as well. Thus, a perfect parser would have to take all possibilities into account.

Due to the challenges listed above, the parser written for SUSI is not fully mature, yet. However, it properly processes at least almost everything. The remaining difficulties will have to be addressed in the future. The parser follows a very simple principle: Avoid going through the same passage more than once. It usually reacts to the start of some construct and behaves accordingly. There are lots of possible elements that can be found. A complete list is available online[2]. We will not discuss all of them here but examine a few examples instead:

Internal Links may be the most important construct to parse. These internal links point to some other Wikipedia page and thus to the entity that is described on that page. Internal links are the easiest and yet most reliable way to recognize entities. Usually they consist of the entity title in double brackets, e.g. `[[Albert Einstein]]` but variations are possible. `[[(Albert) Einstein]]` or `[[Albert Einstein | That physics guy]]` can be used to reference the same entity but display only *"Einstein"* or *"That physics guy"* instead. These two were just chosen as example. More similar variations are possible and a parser has to react accordingly.

Sections and Section Headlines Sections and their headlines have to be found because they are important for entity recognition. See the section on entity recognition for further details. Apart from that, we can supply the parser with a list of section headlines that indicate sections that will always be skipped during parsing. One example is *"References"* which are usually ignored.

[1] http://www.mediawiki.org/xml/export-0.4.xsd
[2] http://en.wikipedia.org/wiki/Help:Wiki_markup and linked documents

Math and similar Structures are ignored altogether. Usually they do not contain any meaningful text that could be of use for Susi.

Wikipedia Templates are usually skipped since there are so many of them and even user defined ones are possible. However, some really important ones, like tables have to be parsed in order not to lose valuable information.

While those examples and more cases are already implemented in the parser, there are still others to be added in the future. Those, that might have been of use at some point already, are listed in chapter 6.

4.4. Entity Recognition

Semantic Search is about finding facts, facts about entities. So if we are looking at a system like Susi, where semantic search is performed on a combination of ontologies and full-text, entity recognition in the full-text is obviously necessary and even a key aspect. In general, this is a really challenging problem and solutions usually involve linguistic analysis of the text (done by Sophie [Suchanek (2009)] which is supposed to find facts for Yago) or analysis based on statistical co-occurrence (has been tried for Ester [Bast et al. (2007)]).

However, entity recognition is much easier on Wikipedia pages. At least if the pages live up to Wikipedia's standards, the first occurrence of any entity should always be linked to the page describing that entity. This allows for simple, yet excellently precise heuristics that abuse the structure of Wikipedia pages. The entity recognition performed for Susi follows exactly this idea.

- First of all, direct links to Wikipedia pages are always occurrences of the linked entities. Technically we have to take redirecting Wikipedia pages into account (e.g. the page *Einstein* redirects to *Albert_Einstein*), but a map with all redirecting pages is easily constructed by parsing the Wikipedia dump once.
- Secondly, further occurrences of an entity usually are not linked anymore. Additionally, those occurrences do not necessarily match the linked entity entirely. The most common example evolves around persons that are mentioned by their last name only. Consider a passage like: *"Einstein thought that..."*. If the entity *Albert_Einstein* has just been mentioned, It is clear that *Einstein* in fact refers to that entity.
- Finally, there are expressions referring to another entity. While those expressions are called anaphora in linguistics and their strict definition depends on theories, they are again best understood by some example. Again, the Wikipedia on Albert Einstein contains the following excerpt: *"He received the 1921 Nobel Prize in Physics for his services to Theoretical Physic..."*. Both, *he* and *his* do reference the entity. Anaphora handled by Susi are exactly those pronouns.

Now that we have seen the different kinds of words that may represent entity occurrences, we still have to associate them with the correct entities whenever we discover these words during parsing. Obviously, the correct choice always depends on the context of the current page. Therefore, we keep track of several entities when building the index for Susi:

- Entities inside the current section. This case is quite intuitive. If an entity has just been mentioned, it is a possible match. This can be seen in the following sentence: *Although Einstein had early speech difficulties, he was a top student in elementary school.* Obviously, the word *he* references the Einstein who has just recently been mentioned. Therefore we keep all entity occurrences in a map which has all words from their title as separate keys and the entity as value.
- The page entity. A Wikipedia page is always dedicated to one certain entity. This entity is always referenced by other expressions throughout the whole document.
- Entities referenced by section headlines. Just like the document entity, an entity in the headline of the current section can also be seen as present everywhere in the section. While this feature has been implemented for Susi, practice has shown that it is not really helpful and therefore it is currently deactivated, or to be precise, it has been set to a level where only the document entity is handled in a special way.

These cases fill the map of currently "near" entities. This map is cleared whenever new sections are discovered (the document entity remains in the map in the case) or if a new document starts. Additionally we always remember the last entity that occurred separately, because it may be more likely that this particular entity is referenced. The information can be used to associate word occurrences with entities and thus recognize entities in the text. In a nutshell, we can describe the strategy used by Susi for this association by the following algorithm:

Algorithm 1. *For every word occurrence, check if it should be treated as anaphora. For pronouns assume that he/his/him/she/her/they and so on, will always reference persons and it/its and so on will always reference non-person entities. Decide if the document entity is suited. Alternatively check if the last entity fits the type of the pronoun. For others words that exceed a minimum word length, check if any recent entity contains that word.*

This heuristic works surprisingly well, but there still are some problems that would require more sophisticated entity recognition strategies. Those are discussed in chapter 6 in the section of future work. Also chapter 5 deals with the evaluation of Susi and examines which failures can be blamed on entity recognition. In fact, there are not many failures although such a simple heuristic is used, so it works surprisingly well.

4.5. Incorporation of YAGO

This chapter discusses the main challenge during the development of Susi, i.e. associating entities with their semantic classes. For example, we want our search to know that Albert Einstein is a physicist and thus return occurrences of Albert Einstein as hits of queries for physicists. As we have discussed in previous chapters, Yago contains this kind of knowledge. It associates Wikipedia pages, and hence entities, with their semantic classes. Regarding these facts from Yago, we distinguish two categories:

Definition 1. Let *Classes* denote all semantic classes from Yago (which inherits them from WordNet) that some entity belongs to. They may range from interesting, specific ones like *"scientist"* to very generic ones like *"entity",* which is a class that every entity belongs to trivially.

However, these classes can play a special role for an entity:

Definition 2. *Leaves* is a term we use to describe special facts. Consider the fact that Albert Einstein is a physicist, again. This fact is directly contained in Yago. However, being a physicist implies several other things. We know that an entity that is a physicist also has to be a scientist, a person and so on. A *leaf* is a class that is directly associated with the entity and not implied by some other class.

Yago provides facts about the *leaves* in a designated file that directly associates them with entities. Therefore it is easy to use this kind of knowledge, although we have to consider special cases where Yago lists a leaf for some entity, that is already implied by another one. We will address this issue later on.

Concerning the implied classes, on the other hand, Yago provides a subclass relation called $subClassOf$. This relation is crucial for the ideas discussed in this section. We therefore introduce operators that should be intuitive and increase readability:

Definition 3. We write "$\subset$" (read: "subclass Of") when the following holds:
$(a \subset b) \Leftrightarrow (a, b) \in subClassOf$. For example we can write $scientist \subset person$.

Obviously, a subclass relation should be transitive. Accordingly, what we called implied facts earlier, actually means that we require the set of an entity's classes to be closed under the subclass relation.

Definition 4. We write "$\subset_c$" to denote that one class is a subclass of another, regardless of if they directly are in relation. Thus, we can use the following recursive definition:
$(a \subset_c b) \Leftrightarrow \exists a'.(a \subset a') \wedge (a' \subset_c b)$. This allows writing something like $scientst \subset_c entity$.

Technically we can easily construct a closed set of an entity's classes by the following algorithm:

Algorithm 2. *Start with the set of leaves for that entity which is directly available from* YAGO. *Continuously add all direct parent classes to the set until nothing new is added.*

In practice, we use a more complicated algorithm because we require the result to be of a very special format. This format can be used to annotate entities with their semantic classes in an efficient way. The next section presents how this is done in detail.

4.6. Entity Annotations

In the previous section we have seen what kinds of facts are supposed to be extracted from YAGO and have to be made available for search queries. Somehow those classes need to be associated with the entities they describe and the occurrences of those entities in the text. This section presents different ways of doing this and explains which approach has been chosen for SUSI.

4.6.1. Naive Annotation

The simplest way is writing all facts next to an entity. If we imagine that every occurrence of a scientist in the text is annotated with the word `scientist`, a query for that word, will find all occurrences of Albert Einstein amongst others. If we look for a simple way to find out which exact scientist is found in the hit for our query, we can just add the entities' name after each fact. This means we write a word `scientist:alberteinstein` instead. A prefix-query `scientist:*` will consequently deliver the entities that are scientists as last part of the query completions with hits.

This simple idea works perfectly fine, but unfortunately it requires lots of additional words to be written to the index. This causes a blow-up of the index that is not acceptable for really large document collections like the whole English Wikipedia (concrete numbers are presented later in this section). Thus, more clever ways are required to achieve the same thing with less space consumption.

4.6.2. ESTER style

The research work for ESTER [Bast et al. (2007)] features a neat idea to provide the semantic information with very low space consumption. As mentioned in the chapter on related work, ESTER writes all the facts available for an entity only once in the document describing that entity. Entity occurrences somewhere in the text are annotated with a single join attribute instead. Queries containing semantic categories like `penicillin scientist`, then are processed using joins.

This strategy is really space efficient but the additional join can be a costly operation that may slow down the query processing times.

4.6.3. Path-Style Annotations

The way of annotating entities with semantic facts that is actually used in the index of SUSI is yet different and supposed to be superior. The idea is based on the observation that many facts implicitly belong to an entity that is of a certain type. Just think of the entity Albert Einstein who is, amongst others, a physicist. This single fact implies that he also is a scientist, a person, an organism and many other things. A great example is a vegetarian which is implicitly all of: eater, consumer, user, person, organism, living thing, whole, object, physical entity, entity. Obviously, a single semantic class can carry lots of information about an entity.

In YAGO, the subclass relation spans a directly acyclic graph. However, this graph is very tree-like since the only violation of the tree properties is that there are classes that have two or more independent super-classes. As example, consider the class `woman`. A woman is both, an adult and a female. However, the classes `female` and `adult` are no subclasses of each other. For the sake of the following arguments, imagine the subclass relation to span a "broken" tree. Apart from the fact that some nodes have more than one parent, the model of a tree fits well. All types are really subclasses of some other type, except for `entity`. This is intended by WordNet but due to bugs in `YAGO`, some type entries seem to be missing in the current version (e.g. the type `health professional)`. Their subclasses (e.g. `doctor`) may end up without a parent. In this case, we simply add an entry to the subClassOf relation that makes the lonely facts direct subclasses of the root fact `entity`. This design decision leads to the following fact:

Fact 1. *The class* ***`entity`*** *is the only class that is no subclass of some other class. Thus, all other classes have super-classes and the chains eventually end at* ***`entity`****.*

We can make some statements on the subclass tree. Let a and b be classes and thus nodes in the tree:

- a is a parent of b iff $b \subset a$, respectively b is a child of a.
- a is an ancestor of b iff $b \subset_c a$, respectively b is a descendant of a.

The following figure now shows a tiny excerpt of the whole subclass-tree:

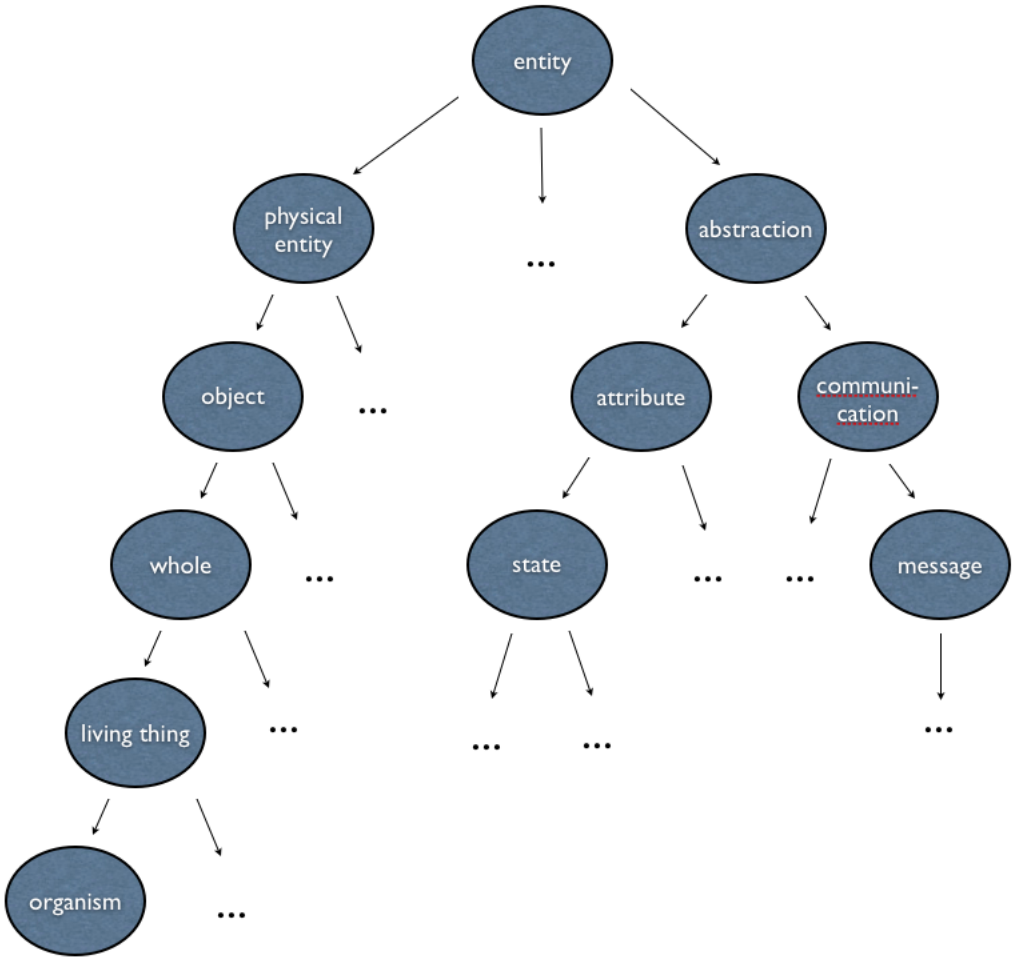

Figure 4.3.: YAGO Paths Example

The idea of path-style entity annotations is to write all paths from the root to the leaves for each class an entity belongs to. Consequently, it is no longer necessary to write all those facts that are implied by the fact as additional words. Consider figure 4.1 and let us say we recognize an entity (e.g. Bacillus Thuringiensis to pick one at random that is not associated with a subclass of organism, like persons or animals are) that is an organism. We now write the additional word :e:entity:physicalentity:object:whole:livingthing:organism:bacillusthuringiensis[3] instead of all those artificial words like organism:bacillusthuringiensis or livingthing:bacillusthuringiensis and so on. Examples involving persons lead to even longer paths so that it is hard to actually draw the tree for them. Assume we discover an occurrence of the entity Albert Einstein during parsing. If we now want to cover the fact that he is a physicist, we can write something like this to the index:

[3]The prefix `:e:` is only there for technical reasons.

word	doc	score	pos
:e:entity:physicalentity:object:whole:livingthing:organism:person:scientist:physicist:alberteinstein:	30	0	1
Albert	30	0	1
Einstein	30	0	1

For now, ignore the prefix `:e:` of the long word which is only there for technical reason. More interestingly, we can now use prefix search to find this word, that ends with alberteinstein, as a completion. Doing this we are free to choose which of the contained types we are looking for. The word is a viable completion for the queries `:e:entity:*` or `:e:entity:physicalentity:object:whole:livingthing:organism:person:*` and so on. The viability of this idea, in the sense that it really does save a significant amount of space compared to the naive way, has been confirmed experimentally:

Measurement	Leaves[a]	Paths	Facts / Naive Annotations
number of (entity, word) pairs	4,059,857	4,016,440[b]	16,281,766
number of entities	2,563,739	2,549,651[c]	2,397,667[d]
avg. facts per entity	1.58	1.58	6.79
median facts per entity	1	1	6
max facts per entity	23	21[e]	51
index blowup factor[f]	n.A.[g]	1.19	4.16

[a] We use the term *leaf* to denote an entities fact that has no subclasses that can also be associated with that entity.
[b] This number can be less than the number of leaves due to the removal of redundant facts.
[c] Some entities are merged due to case insensitivity. See future work on plans to fix this.
[d] The number of entities for the naive style is lower since entities that can only be associated with the generic fact *entity* are eliminated here, while they may or may not be contained in the paths. This does not matter during index construction, because the same thing is written to the index for entities without facts from YAGO or entities with the trivial fact only.
[e] This number can be less than the number of leaves due to the removal of redundant facts.
[f] Factor of size increase compared to not annotating entities at all. Compared number of index items instead of index file size.
[g] Building an index with the leaves only makes no sense, since we are no longer able process queries properly.

Table 4.3.: Space Consumption of Entity Annotations

Table 4.2 compares the path-style annotations to the naive way. Note, that there are actually less paths per entity than there are leaves. This is related to the fact, that leaves are taken from YAGO which directly derives them from Wikipedia categories. This may lead to redundancy. The most common example is probably the class *person* which is present as leaf for many entities, while it is implied by the class that reflects the entity's occupation.

Since we now have to write much less words next to each entity compared to the naive way, the index blowup by adding the semantic facts is negligible. On top of

that, the HYB index enables very fast queries. Query processing times constitute one of the aspects that are evaluated in chapter 5.

Now that we have seen that this idea can effectively decrease the size of the index, we want to confirm that this strategy delivers correct results. Therefore we need to make the things, we have seen in the previous illustration, formal.

Definition 5. The operator $closure(class)$ denotes the smallest set that exhibits deductive closure under the subclass relation and contains $class$. This is simply the class itself and all implied classes. Formally this means that:
$closure(c) := \{c\} \cup \{x \mid c \subset_c x)\}$.
For example it holds that $closure(scientist) = \{person, organism, \ldots, entity\}$.

Obviously, this definition can be used to characterize the set of classes of an entity e by building the union over all closures of its leaves.

Definition 6. The classes of an entity e correspond to the union of the closed sets containing its leaves:
$classes(e) = \bigcup_{c \in leaves(e)} closure(c)$.
For example $classes(alberteinstein) = \{vegetarian, scientist, person, \ldots, entity\}$.

This set of classes should now be represented by the long words we write to the index. These words correspond to paths in the tree that is spanned by the subclass relation. Thus, we also use the term path to describe them.

Definition 7. A path is a word of the following form: It starts at the universal class `entity` and reaches to an arbitrary class that is some entity's leaf. We write a path as sequence of classes separated by colons. An example path could be `entity:physicalentity:object:whole:livingthing:organism:person`.

The operator $paths(class)$, denotes all possible paths from the universal root fact `entity` to the class that serves as argument for the operator. There may be multiple paths to a single class, because our tree does not really fulfill the tree properties and there are those few counter examples where a node has more than one parent. Formally we want that:

$$\begin{aligned} paths(c) = \quad & \{c_1 : c_2 : \cdots : c_n \mid c_n = c \wedge c_1 = \text{entity} \wedge \\ & \forall i, j < n.(i + 1 = j) \Rightarrow (c_j \subset, c_i)\} \end{aligned}$$

As the classes of a path p, we simply collect all classes that occur along it. We write $classes(p) = \{c_i \mid c_1 : c_2 : \cdots : c_n = p\}$.

The previous definitions can be used to show that the path-style annotations deliver the correct result.

Theorem 1. *The path-style annotations deliver the correct result.*

The desired annotations for an entity e are exactly all of its classes. According to definition 6, the set of those classes corresponds to the union of all closed sets containing a leaf of e. The path annotations written for e are exactly all paths leading to leaves of the entity. Thus, we can show the following lemma instead, to proof correctness of the theorem:

Lemma 1. *$Closure(c) = classes(paths(c))$. The set of classes found along all paths to a class c is exactly the smallest set containing c that is closed under the subclass relation ($closure(c)$).*

Proof. We show that $closure(c) = classes(paths(c))$ by showing containment between the two sets in both directions:

$closure(c) \subseteq classes(paths(c))$

The proof for this direction follows directly from the definition of a path. Consider an arbitrary path $p = c_1 : c_2 : \cdots : c_n$. The definition enforces that $\forall i \,.\, j = i + 1 \Rightarrow c_j \subset c_i$. Therefore $\forall i \,.\, c_n \subset_c c_i$ and hence $\forall i \,.\, c_i \in closure(c_n)$. Consequently, $closure(c) \subseteq classes(paths(c))$.

$closure(c) \supseteq classes(paths(c))$

The other direction is not as obvious. We need to recall fact 1 which states that entity is the universal super-class and all other classes are subclasses of entity. Now consider and arbitrary class c and the set of facts $closure(c)$. By definition 5, all classes in this set are either c itself or super-classes of c. $\forall c_i \in closure(c) \,.\, c_i = c \vee c \subset_c c_i$. Therefore, every class c_i is an ancestor of c in the tree. Fact 1 now ensures that every path from c to some c_i can be continued to the root class entity, since entity has to be an ancestor of c_i and hence such a path exists. Since the definition of $paths(c)$ collects all those possible paths between entity and c and $classes(path)$ collects all classes that occur along a path, we can be sure that $closure(c) \supseteq classes(paths(c))$. □

Improvements

We have shown that using all paths according to definition 7 delivers the correct result. However, our goal is to minimize the index-blowup caused by the entity annotations. Maintaining correctness is a necessity that has to be kept in mind, but not the ultimate goal. While we have shown that the paths neither lose real facts nor add wrong ones, we have not shown that they produce minimal overhead. In fact, the paths are not minimal at all. We will see that we can safely eliminate some of the paths and still retain the same expressive power. Those paths, that can be eliminated, arise from the following situation in the tree:

Looking at figure 4.2 and the paths from the root to the node labeled `G`, we see that these double diamond patterns may lead to paths that do not contain any new class. Four paths can be collected:

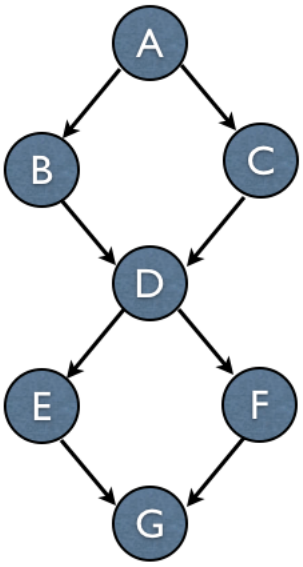

Figure 4.4.: Structure causing unnecessary paths

1. `A:B:D:E:G`
2. `A:C:D:E:G`
3. `A:B:D:F:G`
4. `A:C:D:F:G`

We see that while four paths are possible, the paths from 1. and 4. or from 2. and 3. alternatively already contain all classes. At first sight, it looks like one of the pairs might suffice for annotating an entity that as `G` as one of its leaves. However, we still want to be able to find all classes by prefix search and using one of the pairs of paths only, leads to the following problem:

Let us consider the pair of the paths from 1. and 4. (the argument is just as valid for the other pair). If an entity, that has a leaf `G`, was now annotated with the words `A:B:D:E:G:<entity>` and `A:C:D:F:G:<entity>`,we would have to query for `E` with the prefix query `A:B:D:E:*` and a prefix query with `A:C:D:E:*` could not be used. Unfortunately, this is hard to decide when all we know is that we want to query for `E`. The two possibilities `A:B:D:E:*` and `A:C:D:E:*` appear to be no equally suited. Hence, we need some kind of agreement which paths to use for queries. For the current version of Susi, we chose the following heuristic:

Algorithm 3. *Always query with the first path towards a class. Being first means that we apply the following ordering: Shorter paths take precedence over longer paths and equally long paths are ordered lexicographically. All paths that do not add any new class, can be dropped and hence can we reduce the number of paths without loosing any power of the search.*

```
for all c ∈ Classes do
  sort paths(c) short paths first, sort lexicographically on same length
  seen ← {} (HashSet of Strings)
  newPaths ← {}
  for all p ∈ paths do
    foundNewClass ← false
    classes ← classes(p)
    for all e ∈ classes do
      if e ∉ seen then
        seen ← seen ∪ {e}
        foundNewClass ← true
      end if
    end for
    if foundNewClass = true then
      newPaths ← newPaths ∪ {p}
    end if
    paths(c) ← newPaths
  end for
end for
```

Unfortunately, this heuristic does not necessarily lead to an ideally decreased index size. In fact, for the example above we can only eliminate one of the four paths and not two. This leaves still room for improvements and is also discussed in chapter 6 when we have a look at future work. However, for now we successfully eliminate some facts and we can safely claim that the path-style annotations always decrease the index size compared to naive annotations.

Lemma 2. *Using the path-style annotations with this heuristic for improvement, the index is always smaller than it is with naive annotations.*

Note 1. Note that the measurements from table 4.3 show that it is in fact significantly smaller (the blowup compared to no annotations drops from a factor of 4.16 to 1.15) and that the improvements discussed here only contribute slightly to that observation. The most benefits are obtained from the idea of using paths itself and caused by the fact that the graph spanned by the subclass relation is very tree-like. For a real tree, the path annotations would be even better and no improvements were necessary.

Proof. Consider any class other than `entity` (for the class `entity` the paths do trivially lead to the same result as the naive annotations). In this case, the first path will definitely contain multiple classes. I.e. `entity` and the class itself and possibly more. All further paths do add at least one new class. Since the naive way needs a posting for each class, we ca be sure that we need strictly less postings for all entities that do have leaves other than `entity` and exactly the same amount (1) for those that only belong to the class `entity`. □

Another improvement we make, is the elimination of unnecessary leaves. It may happen that YAGO associates an entity with multiple leaves, whereas one leave is actually already implied by another one. For that purpose we post-process the final map that associates entities with paths. In that map, we look at each entity's paths and eliminate those that are only a prefix of some other path. This is very simple but perfectly works for filtering out paths caused by unnecessary leaves.

4.7. Realization of User Queries

The last section has introduced very long words to represent paths in the tree of semantic classes. We have seen that prefix search can be used to query the index with those words efficiently. However, the queries were extremely long. Anyone who was about to use SUSI, would never want to type those extraordinarily long words for simple queries like our running example, `penicillin scientist`. On top of that, one would also have to know the correct prefixes and thus the paths in the tree. For queries for the class scientist, a path with no less than eight classes is necessary. This is not acceptable for any user.

Fortunately, CompleteSearch already has a powerful user-interface that is easily extended for SUSI. We add so-called translations for classes that provide the user-interface with the corresponding paths. Therefore we add a special document to the index that contains words with a prefix `:t:`. For scientist, for instance, we add a word of the form `:t:scientist::e:...:person:scientist.` Now, the CompleteSearch user interface can internally send additional queries for words with the `:t:` prefix. The returned completions then relate to possible paths.

At that point we distinguish two cases:

1. The `:t:` query has exactly one completion and hence a unique translation. For example, the query `scientist` only has the one meaning that we expect and hence the corresponding path the unique translation.
2. The `:t:` query has more than one completion / translation. The query `person` leads to that case, since person can mean the organism person that we expect but also refers to the grammatical category called person.

In the first case, the user-interface immediately replaces the word in the query by its translation. Hence, it transforms it into a semantic query and directly displays the result for the translated query. In the second case, the user-interface lets the user choose a suitable translation. Therefore it displays possible translations in a dedicated box and translations can be chosen by clicking them. For the displayed value in the box, it appends the last class on the path before the class itself, hence displaying entries like `person, the ORGANISM`.

This concept works well for the :t: queries that are supposed to find translations, but comes in just as handy for the real semantic queries. Since all paths annotating entities, do end with the entity name itself (`:e:...:scientist:alberteinstein:`),

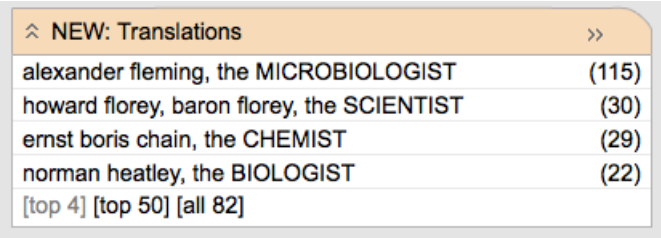

Figure 4.5.: Refinements by Instance

the same principle can be used to refine queries by instance, i.e. by actual entities. Consequently, Susi is able to provide the box from figure 4.5 for the query `penicillin scientist`.

Clicking on such a refinement will limit the displayed hits to those that are evidence for facts concerning this concrete entity.

4.8. Software Architecture

This section summarizes how Susi is built into the CompleteSearch search engine and what technology is used. The whole chain described here is defined and automated in a Makefile[4]. While all steps can be performed individually, they can also be processed together and required artifacts are created whenever they are needed.

Building Susi follows this overall steps:

- Generate the redirect map by parsing the Wikipedia dump once. Done by a simple SAX parser that fulfills exactly this purpose.
- Extract the facts from Yago and construct the paths according to the idea of the subclass tree as discussed above. This is done in three steps by scripts:
  - Extract the entity → leaves association from Yago into a map.
  - Construct the paths by starting with the root class entity and continuously appending all direct subclasses of the rightmost element to the right hand side. Also do improvements as discussed above.
  - Replace each leave in the entity → leaves map by all paths leading to that class. Again do improvements as discussed above.
- Now parse the Wikipedia dump and write the postings for the index and the file for the excerpts.
- Convert the postings for the index to binary format, sort them and generate the vocabulary, the Hyb index and compress the file for the excerpts.

[4] http://www.gnu.org/software/make/manual/make.html

- Launch the CompleteSearch server with suitable arguments and pass the index, the vocabulary and the excerpts file as data base.

Independent from this chain, there are several unit tests, generators for test data and the evaluation framework discussed in chapter 5. Of course, the CompleteSearch engine consists of many components itself but these are not discussed here.

4.9. Exemplary Execution of a Query

Now that we have discussed every aspect of Susi separately, we should be able to understand the whole way from the content of the text and our query to the final result. Recall the example query `penicillin scientist` once more. Additionally, we want to use the text from the very beginning of this document while we simplify it even further to save some writing in the following.

Alexander Fleming was Scottish. He discovered penicillin.

Figure 4.6.: Fictional Text Excerpt

For the text presented in figure 4.6, the postings in the index will look somehow[5] like this:

word	doc	score	pos
:e:entity:...:scientist:biologist:alexanderfleming	12	0	1
:e:entity:alexanderfleming:	12	0	1
alexander	12	0	1
fleming	12	0	1
was	12	0	1
scottish	12	0	1
:e:entity:...:scientist:biologist:alexanderfleming	13	0	2
:e:entity:alexanderfleming:	13	0	2
he	13	0	2
discovered	13	0	2
penicillin	13	0	2

Table 4.4.: Example Index Content

Now let us examine what happens when the query `penicillin scientist` is typed into the user-interface. First of all the user-interface will try to find translations for

[5] In fact, the entity Alexander Fleming is associated with more classes. However, we leave them out to keep things readable. Also penicillin will most likely be associated with a number of semantic classes, such as “drug”.

the word scientist. This will succeed and it will internally replace `scientist` by the prefix `:e:entity:...:scientist:*` , thus transforming it into a semantic query.

When the CompleteSearch engine processes this query it will find the word penicillin in a list of documents. This list will include document 13 because of the posting that is contained in table 4.4. Secondly, it will look for the prefix `:e:entity:...:scientist:*` which will return a number of completions - each with a list of documents where this particular completion is found. One of them should be `:e:entity:...:scientist:biologist:alexanderfleming` with a doc-list including document 13, too. Now, the doc lists for both query words are intersected, eliminating occurrences of scientists in documents that have nothing to do with penicillin and vice versa.

Thus, CompleteSearch will be able to tell that the completion `:e:entity:...:scientist:biologist:alexanderfleming` leads to a hit and that this hit is found in document 13. Hence, it is able to display Alexander Fleming, the biologist as a possible refinement by instance and to present an excerpt for our document 13 as hit. The internal representation of the excerpt of document 13, will contain the string `^^:e:entity:...:alexanderfleming^^^He discovered penicillin.` Consequently, the completion will match the hidden part and the word he is highlighted instead. The word penicillin matches right away and can be highlighted easily.

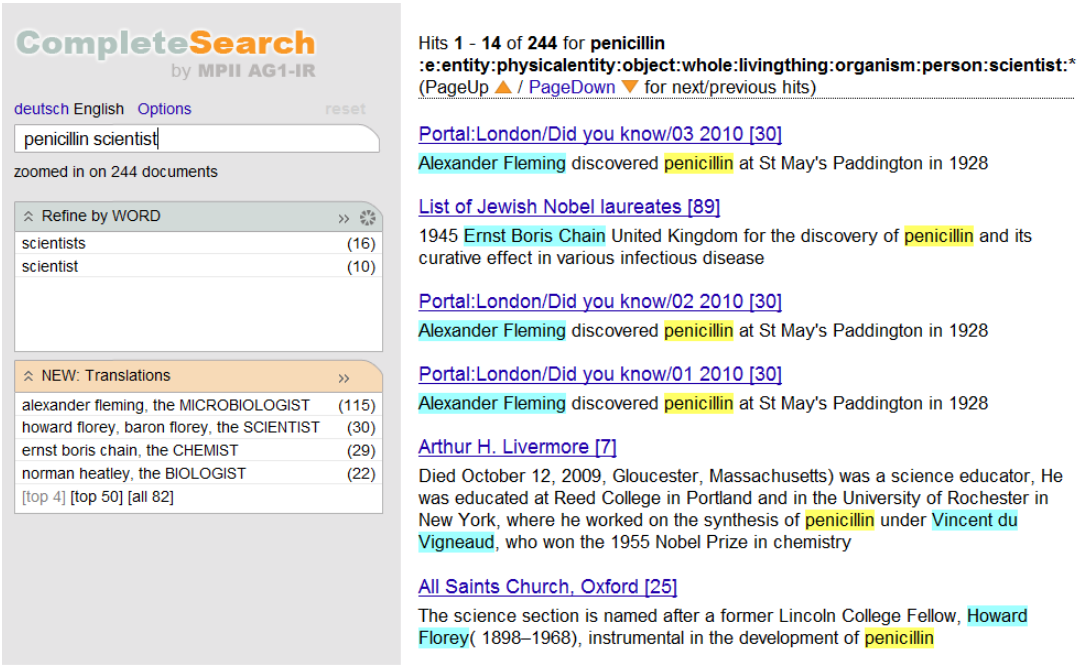

Figure 4.7.: Result returned for the query `penicillin scientist`

Like this, SUSI enables CompleteSearch to present Alexander Fleming as completion

and thus answer to the query. Additionally, the excerpt is presented as evidence and the words that made the query match are highlighted as can bee see in figure 4.7.

5. Evaluation

The evaluation of Susi has two different goals. First of all, we want to demonstrate the performance of Susi with respect to both, quality and efficiency. While it is always nice to present decent results, the current state of Susi makes the second goal much more important: We can examine any discovered deficits in detail, find out what causes them and react accordingly by fixing minor bugs or revising concepts that do not seem to work out as well as intended.

5.1. Quality

Examining the quality of a search engine usually evolves around the question: *Did the expected documents get returned*? Although Susi delivers something similar in form of the evidence provided for each fact, we rather want to answer the question: *Did the expected entities get returned?* This makes sense, because the evidence will always be the passage of the text that contains the found fact. As long as the entity really fits the fact, the evidence is likely to be correct as well.

We perform several queries, compare the result to a set of expected entities and take precision, recall and f-measure. Finally, we discuss the outcome. We examine queries with less good results in depth and try to determine the predominant reasons for misbehavior.

5.1.1. Experimental Setup

The quality measurements have been executed on a machine where no running version of Susi was available. Consequently, queries are processed over the network, although this had only practical reasons. Basically, the evaluation framework communicates with Susi via HTTP, receives a result in XML format, which is then parsed for a list of entities.

As expected entities, our ground truth, we chose Wikipedia lists. There are many Wikipedia Lists that are sometimes generated from external resources and often created by hand. Those lists may concern pretty much any topic. To name some examples, there is a list of bridges, a list of British monarchs or a list of plants with edible leaves. Each of the chosen lists is parsed by a dedicated Perl script. For most of them, the relevant entities can easily be matched by a regular expression. Note

that most existing approaches to semantic search could not even process all of those queries, because some lists evolve around very specific facts that usually are not reflected by relations in an ontology. Just consider the list of drug related deaths. It is very unlikely to list the cause of death for every person.

In order to evaluate a query we join both lists, the expected one and the one Susi returned, and mark each entity with one of the following:

True, which is used to mark entities that are in both lists. Everything works as expected for them.

False-Pos for "false positives", which is used for entities that are returned by Susi but not in the ground-truth.

False-Neg for "false negatives", which is used for entities that are in the ground-truth but not returned by Susi.

The queries that are supposed to recreate those lists are chosen by hand. Sometimes, we use multiple queries for the same list in order to compare them.

ID	Wikipedia List as Ground Truth	Query
Q1.1	Computer Scientists	computer ...:person:scientist:*
Q1.2	Computer Scientists	computer science ...:person:scientist:*
Q2.1	Treaties	...:writtenagreement:treaty:*
Q2.2	Treaties	treaty ...:writtenagreement:treaty:*
Q3.1	Regions of France	france ...:location:region:*
Q3.2	Regions of France	regions of france ...:location:region:*
Q4	Cocktails	...:mixeddrink:cocktail:*
Q5	Political Authors	political ...:communicator:writer:*
Q6.1	EMI Artists	emi :..:creator:artist:*
Q6.2	EMI Artists	emi :..:performer:musician:*
Q7	English Monarchs	english ...:ruler:sovereign:*
Q8	Saturated Fatty Acids	saturated fatty ...:compound:acid:*
Q9.1	Drug Related Deaths	drug death ...:organism:person:*
Q9.2	Drug Related Deaths	drug died ...:organism:person:*
Q9.3	Drug Related Deaths	...:substance:agent:drug:* died ...:organism:person:*
Q10.1	Presidents of the United States	united states ...:corporateexecutive:president:*
Q10.2	Presidents of the United States	united states elected ...:corporateexecutive:president:*

Table 5.1.: Quality Evaluation Queries

Table 5.1 contains all lists used for evaluation and the corresponding queries. Additionally, we assign IDs to the queries so that we can easily refer to them later. Note that queries contain our long, artificial paths and had to be abbreviated. The full ones can be found in the appendix.

Susi, or actually CompleteSearch, already includes some kind of ranking for the returned completions and hence the found entities. This rating simply reflects the number of different positions of the completion in the text that are matched by our search query. Consequently, we are able to examine the top-k entities returned. This is an interesting measure, because an entity at the bottom of the result list is usually not perceived at all or at least not as important as the top ones. For this document we included the measurements for top-500.

5.1.2. Measurements

For each query we collect the total numbers and calculate precision, recall and f-measure. With the current settings, Susi performs in the following way:

Query ID	True	False-Pos	False-Neg	Precision	Recall	F-Measure
Q1.1	295	3047	62	8.8%	82.6%	16%
Q1.2	248	1598	109	13.4%	69.5%	22.5%
Q2	762	863	143	46.9%	84.2%	60.2%
Q2.2	743	536	143	58.1%	82.1%	68%
Q3.1	26	47991	0	0.1%	100%	0.1%
Q3.2	26	5278	0	0.5%	100%	1%
Q4	88	83	68	51.5%	56.4%	53.8%
Q5	28	13236	19	0.2%	59.6%	0.4%
Q6.1	113	2420	256	4.5%	30.6%	7.8%
Q6.2	131	2633	238	4.7%	30.6%	7.8%
Q7	48	1412	11	3.3%	81.4%	6.3%
Q8	6	12	28	33.3%	17.6%	23.1%
Q9.1	50	863	164	5.5%	23.4%	8.9%
Q9.2	89	433	125	17%	31.6%	24.2%
Q9.3	89	918	125	8.8%	41.6%	14.6%
Q10.1	42	2255	1	1.8%	97.7%	3.6%
Q10.2	42	140	1	23.1%	97.7%	37.3%
Total	2826	83718	1512	3.3%	65.1%	6.2%

Table 5.2.: Quality Evaluation - All Completions
The statistics for each query are shown. True counts all entities that are in the list and also get returned by Susi. False counts those that get returned but are not in the list and missing counts those that are in the list but are not returned by Susi. Note, that the total statistics are influence by the extreme outliers. In the following section, we examine why some queries perform good, while others do not and determine common reasons for failure.

Query ID	True	False-Pos	False-Neg	Precision	Recall	F-Measure
Q1.1	160	309	197	34.1%	44.8%	38.7%
Q1.2	149	327	208	31.3%	41.7%	35.8%
Q2	299	201	606	59.8%	33%	42.6%
Q2.2	396	104	509	79.2%	43.8%	56.4%
Q3.1	22	276	4	7.4%	84.7%	13.6%
Q3.2	24	351	2	6.4%	92.3%	11.9%
Q4	88	83	68	51.5%	56.4%	53.8%
Q5	10	371	37	2.6%	21.2%	4.6%
Q6.1	69	407	300	14.5%	18.7%	16.3%
Q6.2	65	363	304	15.2%	17.6%	16.3%
Q7	45	453	13	9%	77.6%	16.2%
Q8	6	12	28	33.3%	17.6%	23.1%
Q9.1	23	230	191	9.1%	10.7%	9.9%
Q9.2	63	231	151	21.4%	29.4%	24.8%
Q9.3	42	272	172	13.4%	19.6%	15.9%
Q10.1	42	458	1	8.4%	97.7%	15.5%
Q10.2	42	140	1	23.1%	97.7%	37.3%
Total	1545	4589	2792	25.2%	35.6%	29%

Table 5.3.: Quality Evaluation - Top 500 Completions
The statistics for each query with top 500 retrieval are shown. Note that the top 500 completions do not necessarily reflect 500 distinct entities. For example scientist:biologist:aristotle and scientist:mathematician:aristotle are two completions describing the same entity. Apart from that, the principle is identical to the style of table 5.2.

5.1.3. Interpretation

In order to correctly interpret the results, we have to pay special attention to queries Q2.1 and Q4. For each of those queries, there is coincidentally a YAGO category that should directly reflect the Wikipedia list used as ground truth. However, the results are not close to 100% precision and recall at all. Consequently, we note that there has to be a mismatch. Further observation shows that sometimes Yago does not completely classify all entities, but on the other hand, the Wikipedia lists themselves may have lots of flaws. The most common issue is that manually created lists are far from being complete. Hence, many entities returned by SUSI are actually correct but recognized as false positives. Additionally, the Wikipedia lists may also contain false positives themselves. For example, the list of cocktails lists drinks that are not actual cocktails but would commonly be classified as liqueurs.

Keeping this issue in mind, we still notice significant differences between the queries that cannot be related to flaws in the Wikipedia lists. Naturally, we want to examine

possible reasons for this observation. Therefore, we pick some queries and analyze them in detail. Fortunately, we do not only have total and relative numbers for our evaluation but also the particular entities that are classified as either True, False-Pos or False-Neg. Thus, we can look at both, false positives and false negatives, go through the Wikipedia documents, check YAGO and finally state as reason for this mismatch between the result from SUSI and the Wikipedia list. In order to establish a systematic system we define several classes of common errors. Subsequently, we can assign one of those classes to each incorrectly returned entity. We use the following classes:

L List Problem.
There is a problem with the Wikipedia list used as ground truth for that query. Either the entity should be in the list but actually is missing or it is in the list while it does not deserve to be.
Example: Thomas Hobbes missing in the Wikipedia list of political writers.

Y Yago Problem.
There is a mistake in YAGO. Usually this means that an entity misses a class it actually belongs to. This usually happens when Wikipedia articles are not assigned to categories, the source for huge parts of the classifications in YAGO.
Example: Bill Clinton not being classified as president.

A Abstract Entity Problem.
The entity is some abstract entity like *computer scientist* instead of a name of a concrete person. While the page entity *computer scientist* is a valid instance of *scientist* and of a person, lists usually only contain concrete persons or likewise concrete entities. Currently, SUSI does not distinguish between abstract and concrete entities.
Example: The entity *Journalist* is returned as political writer.

R Referral Problem.
The fact refers to some other entity but both are mentioned in the same sentence.
Example: John Doe saw on the television that Jane Smith discovered a cure for cancer. John Doe would accidentally get associated with the cure for cancer.

Q Query Problem.
There is no suitable way to express the precise semantics of a list as query. Often this is caused by imprecise YAGO categories.
Example: Only single persons are classified as musicians or artists. Bands are classified as groups at best, while group is a very general class that contains things entirely unrelated to musicians such as institutions or organizations and many more.

E Entity Recognition Problem.
Some passage in the text is recognized as wrong entity.
Example: Herman Einstein was a salesman and engineer. Herman, who is Albert's father, might mistakenly get recognized as Albert Einstein himself.

W Incomplete Wikipedia.
The entity in the list references an entity that does not have its own Wikipedia page.
Example: There is no page for *Tridecylic acid*, while it occurs in the list of saturated fatty acids. In the list, there is a link to a nonexistent page.

D Disambiguation Problem.
Example: The word wardrobe describes many things. A piece of furniture, a clothing and more. An occurrence of this word might get recognized as the wrong entity.

F Fact not Found.
The fact cannot be found in Wikipedia apart from the list itself. This means it is definitely not on the entity's page (confirmed manually) and probably not located on some other page either (hard to check manually).
Example: Victoria Beckham is in the list of EMI artists. However, no other point could be found where it is stated that she really released something under this label.

N Negation Problem.
Often negations contain the same words as the positive case, plus an additionally *not*.
Example: This fatty acid is *not* saturated.

S Synonymy.
The fact is expressed differently. This is a common problem for search engines in general and not specifically related to semantic Wikipedia search.
Example: He committed suicide vs he shot himself.

Z Others.
This label classifies problems that could not be assigned to any of the above categories.

These categories may also serve as basis for any further evaluation that is performed on a larger scale. For now, we can only examine a few example queries. First of all, we want to have a look at the list of political writers which led to poor values. Let us recall the statistics for this query.

Query ID	True	False-Pos	False-Neg	Precision	Recall	F-Measure
Q5	28	13236	19	0.2%	59.6%	0.4%

Table 5.4.: Statistics for Query Q5 (political writers)

Since there are actually quite few entities that Susi could not find, we are able to regard all of them (false negatives). Additionally, we want to look at the topmost 30 of the false positives:

False Positives	
Entity	Reason
Journalist	A
Adolf Hitler	L
Barack Obama	L
Writer	A
John F. Kennedy	L
Winston Churchill	L
John McCain	L
Thomas Jefferson	L
Julius Caesar	L
Niccolò Machiavelli	L
Mao Zedong	L
Mohandas Ghandi	L
Theodore Roosevelt	L
Saddam Hussein	L
Jimmy Carter	L
Thomas Hobbes	L
Columnist	A
Leon Trotsky	L
Leo Strauss	L
Hannah Arendt	L
Gordon Brown	L
William Godwin	L
Harry S. Truman	L
David Axelrod	L
Vladimir Tismaneau	L
George Washington	L
Al Gore	L
Plato	Z
Otto von Bismarck	L
Benjamin Franklin	L

False Negatives	
Entity	Reason
Alireza Jafarzadeh	F / S
Amartya Sen	Y
Andy Croft (Writer)	F / S
Charles E. Silberman	X
David Gautier	Y
Frank A. Capell	W
Gheorghe Alexandrescu	Y
Greg Palast	X
Jan Narvseson	Y
Jean Edward Smith	Y
John Locke	Y
John Rawls	Y
Józef Mackiewicz	X
Jürgen Habermas	Y
Karl Marx	Y
Lewis Gassic Gibbon	Y
Plato (Computer System)	Z
Roberto Quaglia	L
Thomas R. Dye	Y

Table 5.5.: Evaluation of Query Q5 (political writers)

Table 5.5 gives us some insight on what went wrong with query Q5 that is supposed to recreate the list of political writers. We can see that the false positives contain some abstract entities while most of them are writers that actually should be in the list. The entity Plato is a special case. As we can see the ground truth somehow contains the entity Plato, the Computer System. However, this is not an error in the Wikipedia list, instead there seems to be a problem with case insensitivity of the redirect map. The Wikipedia page PLATO in all upper case redirects to the page

of the computer system. Plato, on the other hand is the page for the philosopher as expected. Unfortunately, the redirect map redirects occurrences of Plato to the page of the computer system. This issue can easily be fixed as discussed in chapter 6, Future Work. Apart from that, the false positives only confirm how bad the hand-made Wikipedia lists can be and we should note, that abstract entities are in fact an issue. The false negatives, however, are more interesting. While there are some lesser known writers whose Wikipedia pages do not contain all necessary information, the predominant problem are YAGO categories. Many philosophers are not classified as writers although they published several books or articles.

As a second example, we want to examine the query for the regions of France. Especially query Q3.1 shows terrible precision. The first guess should be that the word region is ambiguous. While the Wikipedia list contains the 26 political regions of France, the area between two arbitrary villages, a city or even another country, can also be described as a region. In order to confirm this theory, we look at the top 20 false positives.

Entity	Reason According to the Categories
France	Q - The class *region* is too general.
Departments of France	A - Abstract entity.
Communes of France	A - Abstract entity
Paris	Q - The class *region* is too general.
Germany	Q - The class *region* is too general.
Italy	Q - The class *region* is too general.
Regions of France	A - Abstract entity, the list itself.
Spain	Q - The class *region* is too general.
United Kingdom	Q - The class *region* is too general.
Belgium	Q - The class *region* is too general.
England	Q - The class *region* is too general.
Switzerland	Q - The class *region* is too general.
Netherlands	Q - The class *region* is too general.
Canada	Q - The class *region* is too general.
Russia	Q - The class *region* is too general.
Austria	Q - The class *region* is too general.
Australia	Q - The class *region* is too general.
New France	Q - The class *region* is too general.
Sweden	Q - The class *region* is too general.
Poland	Q - The class *region* is too general.

Table 5.6.: Top 20 False Positives for Q3.1 (regions of France)

Table 5.6 clearly confirms our theory. The class region is too general and not limited on the political regions form the list. However, we observe a second phenomenon. The word France also causes problem. While the lists uses it to describe the region,

the word also describes the entity France. Hence, many regions - especially countries - that have some relation with the country France are contained in the result of our query. Either way, it is safe to say that the bad precision is caused by our ambiguous query.

Finally, we want to consider another example. In order to regard something new, we pick a query with many false negatives. Query Q6, which is supposed to recreate the list of EMI artists has the worst recall with only 30%. We examine 20 false positives ones and 20 false negatives for this query.

False Positives		False Negatives	
Entity	Reason	Entity	Reason
Musician	A	All-4-One	Q
Songwriter	A	Adrian Gurvitz	F
George Martin	L	Agustin Anievas	Y
Composer	A	Air Traffic	Q
Guiseppe Verdi	R	Airbourne (Band)	Q
John Lennon	Z	Al'Margir	Q
Alice (Italian Singer)	L	Alfie (Band)	Q
Franz Schubert	Ö	Anouk	Q
Sakis Rouvas	L	Apawk	Q
Emi Hinouchi	Z	Art Brut	Q
Richard Wagner	L	Iron Maiden	Q
Ringo Starr	Z	jaguares	Q
Brian Epstein	Q	Jake Hook	Y
Emi Maria	Z	Jane's Addiction	Q
Per Gessle	R	Victoria Beckham	F/L
Wolfgang Amadeus Mozart	L	Vincent Vincent	Y
Richard Strauss	L	Vodka Collins	Q
Giacomo Puccini	L	W.A.S.P	Q
Berlin Philharmoic	L	Wanda Jackson	F/L
Ludwig van Beethoven	R	Watershed	Q

Table 5.7.: Evaluation of Query Q6 (EMI artists)

Our results listed in table 5.7 give an obvious answer to why this query has a lower recall than others. When we want to express a class that describes artists or musicians, we cannot find a suitable YAGO category. All bands, ensembles, etc do not share a class with solo artists. This is why the query does not return groups and hence the recall suffers immensely. The false positives can also be related to an interesting issue. Many musicians are returned that were never signed at EMI during their solo career. However, they are part of bands that are. This leads to the Beatles being one of the false negatives, while their members occur as false positives.

Another issue is that composers get returned whose work has been performed by some EMI artist. This is a referral problem and caused by using mere co-occurrence in a sentence for identifying semantic facts.

In summary, we see that most analysis lead to a discovery that allows really easy fixes and consequently improves to quality of the results returned by Susi. Obviously, this makes further evaluation an important goal for future work.

5.2. Performance

Apart from quality, there is a second very important aspect to our evaluation. Performance and efficiency are absolutely crucial. We have already seen the path style annotations that are supposed to achieve better performance. While we have seen that they are in fact very space efficient, we now want to to measure and examine response times for queries in practice. Especially since fine-tuning some settings may have a huge impact on performance, we really want an evaluation of concrete queries in order to spot cases where existing heuristics seem to fail.

5.2.1. Experimental Setup

All our experiments were run on an Intel Xeon X5560 2.8GHz machine with 16 processors, 40 GB of main memory, and running Linux. For the evaluation, an existing framework has been used which is part of the CompleteSearch project. It provides detailed statistics for each query as well as a summary. The complete statistics can be found in the appendix.

The queries from table 5.8 have been used for this part of the evaluation. Note, that Q1 to Q10.2 denote the queries from the quality evaluation whose performance is evaluated, too.

Query ID	Query
P0.1	&:e:...:organism:person:*
P0.2	&:e:...:location:region:*
P1	search engine
P2	:e:entity:alberteinstein:*
P3	:e:...:eater:vegetarian:*
P4	:e:...organism:person:*
P5	penicillin :e:...:person:scientist:*
P6	computer science :e:...:organism:person:*
P7	:t:algorithm*
Q1.1 - Q10.2	See table 5.1

Table 5.8.: Queries for the Performance Evaluation

We have extended the queries by two warm-up queries. The & symbol marks queries that are not supposed to contribute to the statistics. Those warm-up queries fill a specific part of Susi's history that is always kept in the cache. In production mode, Susi will always have this caches filled, too. However, we also evaluate our queries with cold caches once, in order to gain more insight on cache effects and possible problems.

5.2.2. Measurements

With warm-up		
Query	Millisecs	Strategy
P1	109.7	scan 2 blocks
P2	5.5	scan 1 block
P3	694.6	filter from hist
P4	0.0	fetch from hist
P5	62.0	scan 2 blocks
P6	375.8	scan 2 blocks
P7	3.2	scan 1 block
Q1.1	50.6	scan 1 block
Q1.2	2.7	filter from hist
Q2.1	69.6	scan 2 blocks
Q2.2	41.6	scan 1 block
Q3.1	287.7	scan 1 block
Q3.2	864.9	scan 1 block
Q4	31.7	scan 1 block
Q5	366.9	scan 2 blocks
Q6.1	224.8	scan 2 blocks
Q6.2	276.0	scan 1 block
Q7	159.4	scan 2 blocks
Q8	48.4	scan 3 blocks
Q9.1	137.6	scan 2 blocks
Q9.2	103.3	scan 1 block
Q9.3	163.5	scan 2 blocks
Q10.1	1001.7	scan 3 blocks
Q10.2	56.4	scan 1 block
Avg	214.1	-

Without warm-up		
Query	Millisecs	Strategy
P1	109.9	scan 2 blocks
P2	5.4	scan 1 block
P3	42.2	scan 1 block
P4	19740.9	scan 155 blocks
P5	63,6	scan 2 blocks
P6	366.4	scan 2 blocks
P7	3.0	scan 1 block
Q1.1	48.0	scan 1 block
Q1.2	2.4	filter from hist
Q2.1	66.5	scan 2 blocks
Q2.2	37.5	scan 1 block
Q3.1	1643.1	scan 58 blocks
Q3.2	1816.1	scan 59 bocks
Q4	34.9	scan 1 block
Q5	353.6	scan 2 blocks
Q6.1	216.3	scan 2 blocks
Q6.2	268.9	scan 1 block
Q7	155.3	scan 2 blocks
Q8	46.5	scan 3 blocks
Q9.1	132.4	scan 2 blocks
Q9.2	99.7	scan 1 block
Q9.3	158.5	scan 2 blocks
Q10.1	950.5	scan 3 blocks
Q10.2	49.2	scan 1 block
Avg	1100.4	-

Table 5.9.: Performance Evaluation

5.2.3. Interpretation

The measurements presented above provide insight into the current state of Susi's performance. We see that most queries are processed within only tens of milliseconds. However, there are also queries that require about a whole second to finish which is not really acceptable. This behavior is most likely related to suboptimal block boundaries for the HYB index.

Just look at the query for American presidents which performed poorly. The reason is, that the current heuristic uses one block for each direct subclass of person. However, in the case of the class president, the content is located in a block for all leaders which has about 6 million entries. Subsequently, only words that are actual completions of president have to be filtered. In fact, we have examined this further and were able to conclude that there are 5,930,590 leader vs 1,343,830 scientist occurrences and 164,812 leader vs 36,782 scientist completions. So we can be pretty sure that this block is in fact too huge to deliver the desired performance results.

In conclusion, we can say that it would be wise to divide the vocabulary further. Especially since no queries required scanning lots of blocks, it is save to say that further division should not harm the performance. However, fine-tuning is postponed for now and counted towards future work instead.

The second thing we notice concerns query P3 which queries for vegetarians. Being a subclass of person, the entries can be filtered from history. In this particular case, however, the cold queries show that actually reading the corresponding block from disk instead of filtering, is faster by order of magnitude. CompleteSearch is configured based on the expectation that filtering form history is always faster than reading blocks from disk and decompressing them. However, the different nature of the artificial words used by Susi, render this expectation wrong under some circumstances. Consequently, this is another issue that should be regarded in the future.

6. Discussion

This final chapter concludes the work done for Susi. Additionally, it features a list of pieces of future work. For some of the issues, we can already propose possible solutions. On top of that, we try to judge the necessary effort involved whenever possible.

6.1. Conclusion

We have introduced an approach to semantic search that combines full-text with ontology search and presented its application in form of our system Susi. Susi enables search that directly finds facts in the English Wikipedia and provides excerpts from the text as evidence. While existing approaches to semantic search restrict themselves to search on a fixed set of attributes or relations, being able to access all information in the text obviously exceeds those restrictions.

Our evaluation has shown that we already achieve query processing times of fractions of a second. However, there are still some outliers that leave room for improvements. Queries that take over half a second are not really acceptable. Fortunately, those queries can easily be accelerated by fine-tuning such as choosing more suitable block boundaries for the HYB index.

On the other hand, quality evaluation has show that precision and recall are not entirely satisfying, yet. However, a huge portion of unexpected outcomes can be related to mistakes and especially incompleteness of the Wikipedia lists that have been used as ground truth. Still, there are several issues that spoil precision and recall and that should be tackled in order to improve the quality of the results delivered by Susi. Especially distinguishing abstract and concrete entities should possibly help a lot. Fortunately, detailed examination of our results has shown that for each query with bad statistics there usually is one distinct problem to blame it on. This observation allows focusing on issues whose resolution will really improve the system.

6.2. Future Work

Throughout this whole thesis we have presented all relevant aspects to the creation of Susi and identified several issues that currently require further improvements.

We now want to summarize future work as a list. Note, that we try to state possible solutions and estimate the effort for them. The list items are ordered by decreasing priority.

1. Add case sensitivity to the redirect map. Wikipedia URLs and the resulting redirects are case sensitive, too. For example, PLATO redirects to the entity PLATO_(Computer_System) while Plato points to the philosopher's article. This issue has high priority because it is a real bug and no failure of a heuristic and hence is pretty easy to fix. Estimated effort: 1-2 days, since the parser has to be modified in order to preserve the case of recognized entities a bit longer, too.
2. Stop parsing Wikipedia pages called Template:.. . Those pages do not contain interesting facts and spoil both, the result itself and especially the excerpts. Estimated effort: A few minutes plus rebuilding the index.
3. Add a ranking to the excerpts. While the completions are already ranked by frequency and deliver a really convenient output, the excerpts are ranked by document ID. Consequently, displaying excerpts for lowly ranked completions happens a lot. Frequently, this even involves list-like documents or Wikipedia template pages that deliver really messy excerpts. Estimated effort: 1 week or more, since a whole new concept of relating their ranking to the rankiong of the completions has to be established.
4. Distinguish abstract entities (physicist) from concrete ones (Albert Einstein). This could be done by using different prefixes and adapting the UI accordingly. The entities can actually be distinguished very well. YAGO uses different files for entities that are derived from WordNet and for those that are extracted from Wikipedia articles. Abstract entities are usually contained in WordNet and hence the ones extracted from articles can be assumed to be concrete entities. There is already a script that is able to flag abstract or concrete entities that has been used for measurements. For the current version of SUSI, it is not included in the make-chain. Estimated time: 1 day for changes to the created index, multiple days or weeks for the necessary UI remake to support the distinction.
5. Quality evaluations at a larger scope. With help of Amazon Mechanical Turk or similar systems, we could evaluate a lot more queries and gain even more insight.
6. Forbid some queries in the UI. For example, typing `:e:entity` can be deadly for the system because it is a prefix of half of the index which is just too much to process. So just like the UI prevents prefix queries that consist of only one letter, those queries should be avoided as well and a prefix for `:e:entity:alberteinstein:` should only be processed as soon as it reaches a certain length or probably the second colon. Estimated effort: 1 day.
7. Add more relations from YAGO or some other ontology like DBpedia. While it is nice to express some semantic relations by mere co-occurrence, useful

realtions are already present in existing ontologies. Using them should be a source of highly precise facts.

8. Prefixes for semantic classes may currently have more completions than there are actual entity occurrences that fit. For example the prefix ...organism:person: will have more multiple completions for each occurrence of Albert Einstein - e.g. `...physicist:alberteinstein`, `...vegetarian:alberteinstein` and so on. We have taken measurements and for the prefix *person* the current version has 2.59 times more completions than there are actual occurrences. Consequently, this is not critical at the moment but still a huge possible source of inefficiency of the path-style annotations. Thus, the path-style annotations still leave room for improvements.
9. Refine the scope of facts. Simply choosing sentences, items in lists and rows in tables works for many cases but at the same time ignores many constructs in human language. Consequently, some facts are lost while false ones are added.
10. Experiment with the pronoun recognition. If both, the document entity and the last entity from the text, match the type of the pronoun it is hard to decide which one to choose. Currently, no real experiments have been done. One cold easily experiment with the order for a few hours. However, in the long run, linguistic analysis may be far superior, but infering them involves much more work.
11. Rework the decision when to filter query results from history and when to scan blocks from the disk. For the query `...:person:user:eater:vegetarian:*` scanning only takes about 10ms whereas filtering may take more than 500ms, if the CompleteSearch engine filters from the huge result set for the query `...:person:*`. The block that would be scanned (user:*), on the other hand, is really small.
12. Noun phrase recognition. For example, an entity *therapy center* should be recognized as one thing so that it does not match the sole word *therapy* at some other point. For names, on the other hand, matching only first- or last name is sometimes desired. However, sometimes one also has to take care not to confuse relatives or siblings. While this sounds important, almost no concrete case occurred during evaluation where an error could be related to this or the following phenomenon.
13. Disambiguation between abstract entities. Since abstract entities are not related to Wikipedia articles in YAGO, it is hard to associate the right one with an occurrence in the text. For example, there are five WordNet entities for wardrobe meaning the piece or furniture, the clothing and so on. While they are distinguished in WordNet and YAGO, it is hard to tell which one to choose if a Wikipedia entity called wardrobe is discovered.

Apart from the improvements from the above list, there is also more to address in the future. While the above list contains pieces of future work that contribute

further to the current goal behind Susi, we can also extend the goal by some facets. First of all, we may want to extend the ontology part of the search. Currently, Susi only uses information about the semantic classes of entities while Yago actually also provides a lot more relations such as `bornIn`, `hasGdp` and so on. Including some or all of those relations can be used to restrict entities not only by class but also by attributes.

Apart from that, the creation of a user-interface dedicated to semantic search is an interesting project for the future. The current UI has been created for traditional search applications. The extensions made to support semantic queries are possible due to its modular architecture that evolves around multiple independent boxes. However, most of those extensions are mere hacks and a system dedicated to semantic queries would be a lot more suited.

Either way, there are lots of topics that can be addressed in the future. Querying Susi already is real fun and delivers interesting results, but there is still so much to do until all semantic queries can be processed perfectly.

Acknowledgments

I am heartily thankful to my supervisor, Hannah Bast, whose guidance and support from the initial to the final level enabled me to develop an understanding of the used technology and the subject itself.

I also want to thank all friends that helped me a lot by proofreading this document, Elmar Haussmann, Nils Schmidt, Daniel Schauenberg, Alexander Gutjahr, Linda Kelch, Florian Bäurle and Christian Simon.

Lastly, I offer my regards to all of those who contributed to CompleteSearch.

Björn Buchhold

A. Appendix

A.1. Full Queries of Quality Evaluation

Q1.1	computer :e:entity:physicalentity:object:whole:livingthing:organism:person:scientist:*
Q1.2	computer science :e:entity:physicalentity:object:whole:livingthing:organism:person:scientist:*
Q1.3	computer scientist :e:entity:physicalentity:object:whole:livingthing:organism:person:scientist:*
Q2.1	:e:entity:abstraction:communication:message:statement:agreement:writtenagreement:treaty:*"
Q2.2	treaty :e:entity:abstraction:communication:message:statement:agreement:writtenagreement:treaty:*
Q3.1	france :e:entity:physicalentity:object:location:region:*
Q3.2	region of france :e:entity:physicalentity:object:location:region:*
Q4	:e:entity:physicalentity:matter:substance:food:beverage:alcohol:mixeddrink:cocktail:*
Q5	political :e:entity:physicalentity:object:whole:livingthing:organism:person:communicator:writer:*
Q6.1	emi :e:entity:physicalentity:object:whole:livingthing:organism:person:creator:artist:*
Q6.2	emi :e:entity:physicalentity:object:whole:livingthing:organism:person:entertainer:performer:musician:*
Q7	english :e:entity:physicalentity:object:whole:livingthing:organism:person:ruler:sovereign:*
Q8	saturated fatty :e:entity:physicalentity:matter:substance:material:chemical:compound:acid:*
Q9.1	drug death :e:entity:physicalentity:object:whole:livingthing:organism:person:*
Q9.2	drug died :e:entity:physicalentity:object:whole:livingthing:organism:person:*
Q9.3	:e:entity:physicalentity:matter:substance:agent:drug:* died :e:entity:physicalentity:object:whole:livingthing:organism:person:*

Q10.1	united states :e:entity:physicalentity:object:whole:livingthing:organism: person:leader:head:administrator:executive:corporateexecutive:president:*
Q10.2	united states elected :e:entity:physicalentity:object:whole:livingthing:organism: person:leader:head:administrator:executive:corporateexecutive:president:*

A.2. Full Queries of Performance Evaluation

P0.1	&:e:entity:physicalentity:object:whole:livingthing:organism:person:*
P0.2	&:e:entity:physicalentity:object:location:region:*
P1	search engine :e:entity:alberteinstein:*
P2	:e:entity:physicalentity:object:whole:livingthing:organism:person:*
P3	:e:entity:physicalentity:object:whole:livingthing:organism: person:user:consumer:eater:vegetarian*
P4	penicillin :e:entity:physicalentity:object:whole:livingthing:organism:person:scientist*
P5	computer science :e:entity:physicalentity:object:whole:livingthing:organism:person:*
P6	:t:algorithm*

Bibliography

Bast, H., Chitea, A., Suchanek, F. M., and Weber, I. 2007. Ester: efficient search on text, entities, and relations. In *SIGIR*, W. Kraaij, A. P. de Vries, C. L. A. Clarke, N. Fuhr, and N. Kando, Eds. ACM, 671–678.

Bast, H. and Weber, I. 2006. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, E. N. Efthimiadis, S. T. Dumais, D. Hawking, and K. Jaervelin, Eds. ACM, 364–371.

Bast, H. and Weber, I. 2007. The completesearch engine: Interactive, efficient, and towards ir& db integration. In *CIDR*. www.crdrdb.org, 88–95.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R., and Hellmann, S. 2009. Dbpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web 7,* 3, 154 – 165. The Web of Data.

Carroll, J. J. and Klyne, G. 2004. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C. Feb. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

Celikik, M. and Bast, H. 2009. Fast error-tolerant search on very large texts. In *SAC*. 1724–1731.

Guha, R. V., McCool, R., and Miller, E. 2003. Semantic search. In *WWW*. 700–709.

Hahn, R., Bizer, C., Sahnwaldt, C., Herta, C., Robinson, S., Buergle, M., Duewiger, H., and Scheel, U. 2010. Faceted wikipedia search. In *Business Information Systems*, W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, W. Abramowicz, and R. Tolksdorf, Eds. Lecture Notes in Business Information Processing, vol. 47. Springer Berlin Heidelberg, 1–11.

Miller, G. A. 1994. Wordnet: A lexical database for english. In *HLT*. Morgan Kaufmann.

Neumann, T. and Weikum, G. 2008. Rdf-3x: a risc-style engine for rdf. *PVLDB 1,* 1, 647–659.

Neumann, T. and Weikum, G. 2009. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, U. Cetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, Eds. ACM, 627–640.

Prud'hommeaux, E. and Seaborne, A. 2008. SPARQL query language for RDF. W3C recommendation, W3C. Jan. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

Suchanek, F. 2009. Automated construction and growth of a large ontology. Ph.D. thesis.

Suchanek, F. M., Kasneci, G., and Weikum, G. 2007. Yago: a core of semantic knowledge. In *WWW*, C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, Eds. ACM, 697–706.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.