SemSearch: A Search Engine for the Semantic Web

Yuangui Lei, Victoria Uren, and Enrico Motta

Knowledge Media Institute (KMi), The Open University, Milton Keynes,
{y.lei, v.s.uren, e.motta}@open.ac.uk

Abstract. Semantic search promises to produce precise answers to user queries by taking advantage of the availability of explicit semantics of information in the context of the semantic web. Existing tools have been primarily designed to enhance the performance of traditional search technologies but with little support for *naive users*, i.e., ordinary end users who are not necessarily familiar with domain specific semantic data, ontologies, or SQL-like query languages. This paper presents SemSearch, a search engine, which pays special attention to this issue by hiding the complexity of semantic search from end users and making it easy to use and effective. In contrast with existing semantic-based keyword search engines which typically compromise their capability of handling complex user queries in order to overcome the problem of knowledge overhead, SemSearch not only overcomes the problem of knowledge overhead but also supports complex queries. Further, SemSearch provides comprehensive means to produce precise answers that on the one hand satisfy user queries and on the other hand are self-explanatory and understandable by end users. A prototype of the search engine has been implemented and applied in the semantic web portal of our lab. An initial evaluation shows promising results.

1 Introduction

One important goal of the semantic web is to make the meaning of information explicit through semantic mark-up, thus enabling more effective access to knowledge contained in heterogeneous information environments, such as the web. Semantic search plays an important role in realizing this goal, as it promises to produce precise answers to user's queries by taking advantage of the availability of explicit semantics of information.

For example, when searching for news stories about *phd students*, with traditional searching technologies, we often could only get news entries in which the term "phd students" appears. Those entries which mention the names of students but do not use the term "phd students" directly will be missed out. Such news entries however are often the ones that the user is really interested in. In the context of the semantic web, where the meaning of web content is made explicit, the semantic meaning of the keyword (which is a general concept in the example of phd students) can be figured out. Furthermore, the underlying semantic relations of metadata can be exploited to support the retrieving of

information which is closely related to the keyword. Thereby, the search performance can be significantly improved by expanding the query with instances and relations.

A number of semantic search tools have been recently developed [5, 4, 7, 2, 9, 6]. Our overview of the state-of-art semantic search tools reveals that while these tools do enhance the performance of traditional search technologies, they are however not suitable for naive users, i.e. ordinary end users who are not necessarily familiar with domain specific semantic data, ontologies, or SQL-like query languages. The semantic search engine we present here, SemSearch, provides several means to address this issue.

- SemSearch tackles the problem of knowledge overhead by supporting a Google-like query interface. As will be described in Section 4, the proposed query interface provides a simple but powerful way of specifying queries.
- SemSearch addresses the problem of existing semantic-based keyword search engines by supporting complex queries. It provides comprehensive means to make sense of user queries and to translate them into formal queries.
- SemSearch takes the focus of user queries into consideration when generating formal queries, thus being able to produce precise results that on the one hand satisfy user queries and on the other hand are self-explanatory and understandable by end users.

Thus, SemSearch makes it possible for ordinary end users to harvest the benefits of semantic search and other semantic web technologies without having to know the underlying semantic data or to learn a SQL-like query language. A prototype of the search engine has been implemented and applied in the semantic web portal of our lab[1]. An initial evaluation shows promising results.

The rest of the paper is organized as follows. We begin in Section 2 by investigating how current semantic search tools approach the issue of end user support. We then present an overview of SemSearch in Section 3. Thereafter, we explain the Google-like query interface in Section 4. We describe the major steps of the semantic search process in sections 5 and 6. In Section 7, we describe the implementation of SemSearch and the experimental evaluation. Finally, in Section 8, we conclude our paper with a discussion of our contributions and future work.

2 State of the art

In this section, we investigate how current semantic search approaches address user support. We have identified four categories of semantic search engines, according to the user interface they provide: i) *form-based search engines*, which provide sophisticated web forms that allow users to specify queries, in the format of choosing ontologies, classes, properties, and values; ii) *RDF-based querying languages fronted search engines*, which provide sophisticated querying languages

[1] http://semanticweb.kmi.open.ac.uk/semantic_searhing.jsp/

to support semantic search; iii) *semantic-based keyword search engines*, which enhance the performance of traditional keyword search techniques by making use of available semantic data; and iv) *question answering tools*, which exploit available semantic mark-up to answer questions asked in natural language format.

The SHOE search engine [5] is one of the first form-based semantic search engines. It provides sophisticated web forms that allow users to specify queries. Such forms however are only suitable for those users who are fairly familiar with the back-end ontologies and knowledge bases. Naive users have difficulties in understanding these forms. Further, they have difficulties in formulating queries using their own view on the information they aim to find.

The Corese search engine [2] is an example of RDF-based querying language fronted search engines. Other examples include the engines built in CS AKTive Space [8] and the SemanticWeb.org portal[2]. Such search engines usually provide a sophisticated querying language to support semantic data querying. However, in order to be able to ask queries with these search engines, end users will have to be fairly familiar with both the back-end ontologies and the provided querying language.

The TAP search engine [4] and the search engine presented in [7] are examples of semantic-based keyword search engines. The search process of such search engines often comprises two major steps: i) finding an instance match for the user keyword and ii) retrieving instances which are closely related to the instance match of the user keyword. Such search engines often provide comprehensive means to support the clustering of search results.

AquaLog [6] and ORAKEL [1] are examples of ontology-based question answering engines. They make use of natural language processing technologies to reformulate natural language queries into ontological triples (e.g., in AquaLog) or into specific query languages (e.g., in ORAKEL). While these tools appear to be ideal for naive users, their performance on searching is heavily influenced by the performance of the used natural language processing techniques.

All the tools described above are able to enhance the search performance by making use of available semantic data and their underlying ontologies. With the partial exception of ontology-based question answering tools, state-of-art tools are however not suitable for naive users. One problem is **knowledge overhead**, which is requiring users to be equipped with extensive knowledge of the back-end ontologies and knowledge bases (e.g. form-based search engines) or specific SQL-like querying languages (e.g. RDF-based query language fronted search engines) in order to be able to formulate queries or to understand the search result.

Another problem is **the lack of support for answering complex queries** presented by current semantic-based keyword search engines. These search engines are often only able to accept one keyword as input and give back the semantic entities which are related to the keyword as results. Relation centered search that finds relations between multiple keywords is not supported. This greatly limits the scope of user queries. For example, current semantic-based

[2] http://semanticweb.org/

keyword search engines typically could not even handle simple queries where two keywords are involved such as *news* about *phd students*.

3 An overview of SemSearch

One major goal of this work is to hide the complexity of semantic search from end users and to make it easy to use and effective for naive users. To achieve this goal, we identified the following key requirements:

- **Low barrier to access for ordinary end users**. Our semantic search engine should overcome the problem of knowledge overhead and ensure that ordinary end users are able to use it without having to know about the vocabulary or structure of the ontology or having to master a special query language.
- **Dealing with complex queries**. In contrast with existing semantic-based keyword search engines which only answer simple queries, our semantic search engine should allow end users to ask complex queries and provide comprehensive means to handle them.
- **Precise and self-explanatory results**. Our semantic search engine should be able to produce precise results that on the one hand satisfy user queries, and on the other hand are self-explanatory. Thus, ordinary end users can understand the results (e.g. what they are and why they are there) without having to consult the back-end semantic data repositories or their underlying ontologies.
- **Quick response**. Our semantic search engine should provide quick response to user queries, thus encouraging ordinary end users to harvest the benefit of the semantic web technology. This requires that we make the mechanism of semantic search as simple as possible.

To meet these requirements, we chose the keyword-based searching route rather than the natural language question answering route, and deliberately avoided linguistic processing which is a relatively expensive process in terms of search. We overcome the limitation of current keyword-based semantic search engines by supporting a Google-like query interface which supports complex queries in terms of multiple keywords. Figure 1 shows a layered architecture of our semantic search engine. It separates end users from the back-end heterogeneous semantic data repositories by several layers.

- **The Google-like User Interface Layer**, which allows end users to specify queries in terms of keywords. As will be described in Section 4, the Google-like query interface extends traditional keyword search languages by allowing the explicit specification of i) the queried subject and ii) the combination of multiple keywords.
- **The Text Search Layer**, which makes sense of user queries by finding out the explicit semantic meanings of the user keywords. As will be described in Section 5, central to this layer are two components: i) a semantic entity

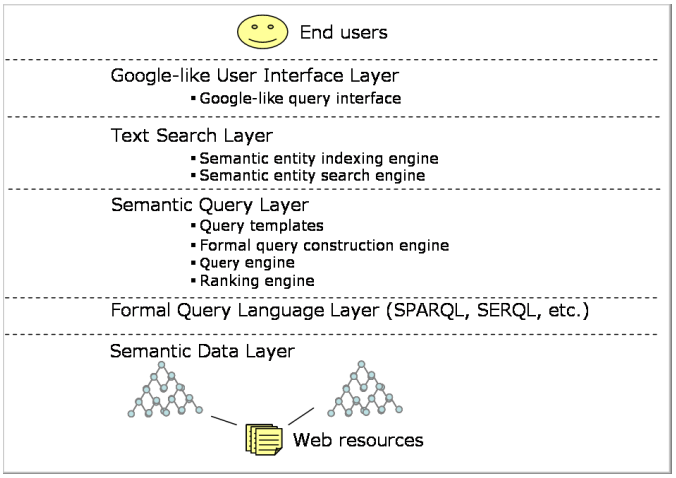


Fig. 1. An overview of the SemSearch architecture.

index engine, which indexes documents and their associated semantic entities including classes, properties, and individuals; and ii) a semantic entity search engine, which supports the searching of semantic entity matches for the user keywords.

- **The Semantic Query Layer**, which produces search results for user queries by translating user queries into formal queries. This layer comprises three components, including i) a formal query construction engine, which translates user queries into formal queries, ii) a query engine, which queries the specified meta-data repository using the generated formal queries, and iii) a ranking engine, which ranks the search results according to the degree of their satisfactory on the user query. The mechanism of formal query generation will be described in Section 6.
- **The Formal Query Language Layer**, which provides a specific formal query language that can be used to retrieve semantic relations from the underlying semantic data layer.
- **The Semantic Data Layer**, which comprises semantic metadata that are gathered from heterogeneous data sources and are represented in different ontologies.

Figure 2 shows an overall diagram of the SemSearch search engine. It accepts keywords as input and produces results which are closely related to the user keywords in terms of semantic relations. The search process of SemSearch comprises four major steps:

- **Step1.** Making sense of the user query, which is to find out the semantic meanings of the keywords specified in a user query.
- **Step2.** Translating the user query into formal queries.
- **Step3.** Querying the back-end semantic data repositories using the generated formal queries.

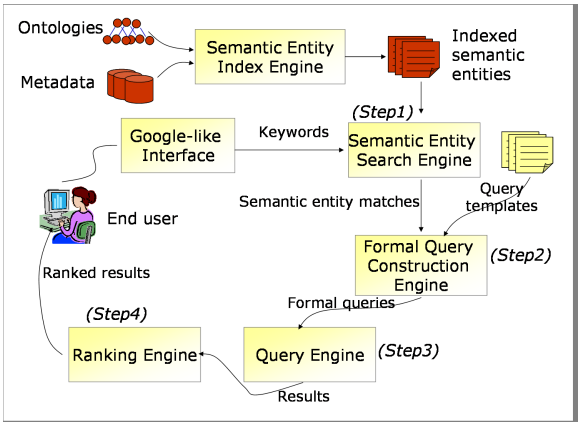


Fig. 2. An overall diagram of the SemSearch search engine.

– **Step4.** Ranking the querying results.

Step1 is carried out within the Text Search Layer. The rest of the steps are associated with the Semantic Query Layer. In the following sections, we will first describe the Google-like query interface (in Section 4). We will then come back to detail the first two steps of the search process (Step 1 in Section 5 and Step2 in Section 6). The two further steps will be briefly described in Section 7 when we describe the implementation of the search engine.

4 The Google-like query interface

The query interface of SemSearch extends traditional keyword search languages by allowing the explicit specification of i) the queried subject and ii) the combination of keywords. By the term *subject*, the user can explicitly tell the search engine the expected type of the search results. For example, when the user specifies the keyword "news" as the subject keyword, he or she expects the search results to be the instances of the class entity that matches against the keyword "news". By allowing the specification of how to combine multiple keywords, the search engine provides end users a simple way of expressing complex queries.

SemSearch uses three heuristic operators to support the specification of user queries. The first one is the symbol *":"*, which supports the specification of query subjects. The second one is the word "*and*", which indicates that all the keywords connected by the word are required to have connections with the search results. The last one is the word "*or* ", which supports the specification of optional keywords. Thus, a user query in SemSearch looks like "*subject:keyword1 and/or keyword2 and/or keyword3 ...*".

With this query syntax, the example of "news about phd students" can be easily specified as *news:phd students*, where the term *news* is the query subject and the term *phd students* is a required keyword. More complex queries in which

multiple keywords (except the subject keyword) are involved also can be easily specified. For example, when querying for projects in which both Enrico and John participate, the query can be specified as *project:Enrico and John*. If the user wants to know all the projects that Enrico *or* John are involved in, the query is *project:Enrico or John*.

In order to satisfy a query, each of the required keywords (please note that subject is a special required keyword) must be satisfied, which means that the search results must be *semantically* related to each of the required keywords. In the case when there are optional keywords, one of the optional keywords must be satisfied.

The SemSearch query interface provides a simple, flexible, and powerful approach for specifying user queries in semantic search. First, it overcomes the problem of knowledge overhead suffered in formal query fronted search engines and form-based semantic search engines, as it does not require end users to be familiar with any particular ontology, semantic data, or any special query language. Second, the query interface provides a more flexible way of specifying queries than the interface presented by form-based search engines. Indeed, it does not confine users to any pre-defined query subjects and values. Third, in contrast with current semantic-based keyword search engines which only accept one keyword as input, this query interface supports the specification of complex queries in the format of specifying multiple keywords and the expected type of results. Finally, this query interface is simpler than question answering tools as the search engine does not need to spend time calculating which of the keywords are in a user's query.

In this paper, we focus on the user queries in which there are at least two keywords involved, in order to better explain the distinctive features of our search engine. Regarding those queries that only comprise one keyword, the search engine develops an approach that is similar to the ones used in TAP and in [7] to find instances that are closely related to the semantic entity matches of the keyword.

5 Making sense of the user query

As mentioned earlier in Section 3, making sense of the user query is the first step of the search process in SemSearch, whose task is to find out the semantic meanings of the keywords specified in a user query so that the search engine knows what the user is looking for and how to satisfy the user query.

From the semantic point of view, one keyword may match i) general concepts (e.g., the keyword "phd students" which matches the concept *phd-student*), ii) semantic relations between concepts, (e.g. the keyword "author" matches the relation *has-author*), or iii) instances entities (e.g., the keyword "Enrico" which matches the instance *Enrico-Motta*, the keyword "chief scientist" which matches the values of the instance *Marc-Eisenstadt* of the property *has-job-title*). The ideal goal of this task is to find out the exact semantic meaning of each keyword. This is however not easy to achieve, as there may be more than one semantic

entity which matches a keyword. Thus, we relaxed the goal as finding out all the semantic entity matches for each keyword.

For the purpose of finding out semantic entity matches, we used the *labels* of semantic entities as the main search source. The rational for this choice is that from the user point of view labels often catch the meaning of semantic entities in an understandable way. In the case of instances, we also used their short literal values as the search source. So that when the user is searching for "chief scientist", the instance that has such a string as a value of its properties can be reached.

In order to produce fast response, the search engine first indexes all the semantic entities contained in the back-end semantic data repositories, including classes, properties, and instances. It then searches the indexed repository to find out matches for keywords. Thus, two components are developed in the search engine, namely the semantic entity index engine and the semantic entity search engine. As it narrows the search sources to labels and short literals of semantic entities, the search engine is able to find out semantic entity matches for each keyword. These matches are the possible semantic meanings of keywords.

Please note that for the sake of getting quick response, we only use text search to find string matches for user keywords at the moment. We avoid using techniques like WordNet [3] based comparison to find matches. This might cost us some good matches, e.g., losing the match *table* if the user is searching for *desk*. But one to one comparison is time consuming and expensive in real-time scenarios. This is indeed a trad-off as well as a research challenge that we need to address in future.

6 Translating the user query into formal queries

In this step, the search engine takes as input the semantic matches of user search terms and outputs an appropriate formal query according to the semantic meanings of keywords. To achieve this task, the search engine needs to capture the focus of the user query (i.e., the type of the expected search results). As described earlier in Section 4, the subject keyword specifies the type of the expected search result. Thus, it is reasonable to expect that the queried subject is a general topic or concept (i.e. class). In the case when the subject keyword does not match any class, the search engine needs to figure out what the expected results are. This will be discussed in the following subsections.

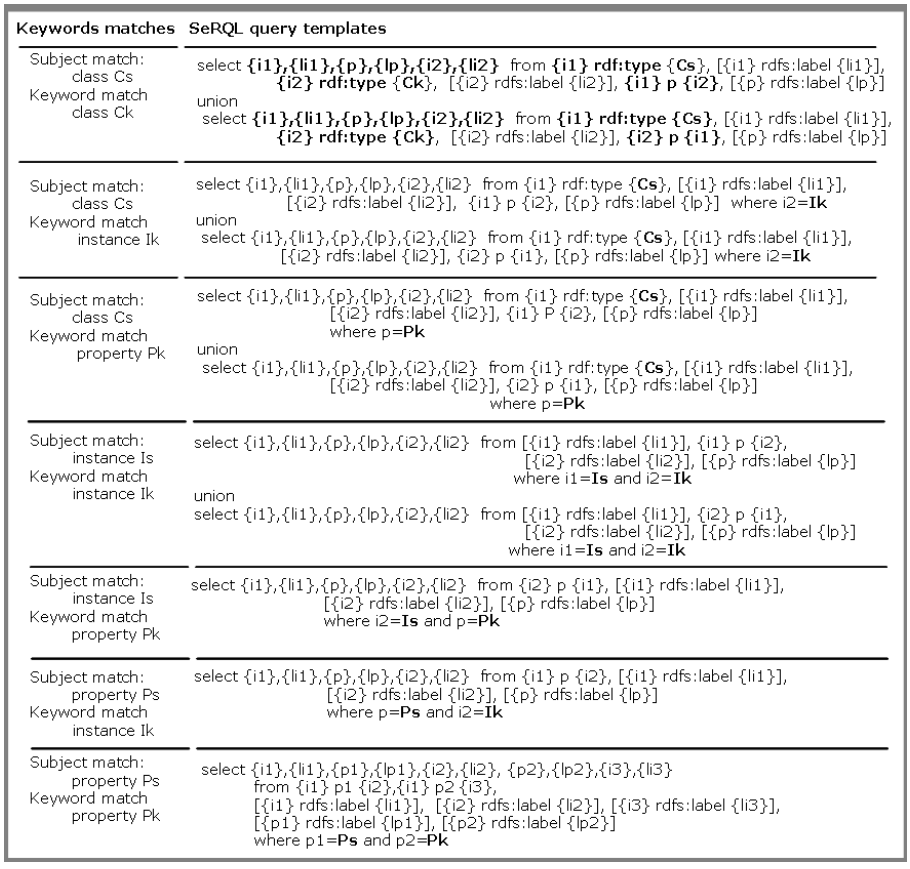
To better understand how to construct formal queries from user queries, we classify user queries into two types: i) simple queries which only comprise two keywords, and ii) complex queries where more than two keywords are involved. In the case of simple queries where the types of semantic entity match combinations are fixed, we developed a set of templates to support the formulation of formal queries. The situation is much trickier in the case of complex queries where there are many variables for keywords combinations.

In this section, we first look at the formulation of formal queries from simple user queries. We then investigate how to handle complex ones. As we used the

Sesame SeRQL language[3] as the formal query language in the prototype of the SemSearch search engine, we explain the mechanism using the same language (Please note the underlying approach does not confine itself to any specific query language).

6.1 Simple user queries

As described earlier, we define simple user queries as those which involve only two keywords. In this subsection, we first describe the query templates. We then describe how to instantiate the query templates to construct formal queries in the case of simple user queries.



Keywords matches	SeRQL query templates
Subject match: class Cs Keyword match class Ck	select **{i1},{li1},{p},{lp},{i2},{li2}** from **{i1} rdf:type** {**Cs**}, [{i1} rdfs:label {li1}], **{i2} rdf:type** {**Ck**}, [{i2} rdfs:label {li2}], **{i1} p {i2}**, [{p} rdfs:label {lp}] union select **{i1},{li1},{p},{lp},{i2},{li2}** from **{i1} rdf:type {Cs}**, [{i1} rdfs:label {li1}], **{i2} rdf:type {Ck}**, [{i2} rdfs:label {li2}], **{i2} p {i1}**, [{p} rdfs:label {lp}]
Subject match: class Cs Keyword match instance Ik	select {i1},{li1},{p},{lp},{i2},{li2} from {i1} rdf:type {**Cs**}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], {i1} p {i2}, [{p} rdfs:label {lp}] where i2=**Ik** union select {i1},{li1},{p},{lp},{i2},{li2} from {i1} rdf:type {**Cs**}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], {i2} p {i1}, [{p} rdfs:label {lp}] where i2=**Ik**
Subject match: class Cs Keyword match property Pk	select {i1},{li1},{p},{lp},{i2},{li2} from {i1} rdf:type {**Cs**}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], {i1} P {i2}, [{p} rdfs:label {lp}] where p=**Pk** union select {i1},{li1},{p},{lp},{i2},{li2} from {i1} rdf:type {**Cs**}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], {i2} p {i1}, [{p} rdfs:label {lp}] where p=**Pk**
Subject match: instance Is Keyword match instance Ik	select {i1},{li1},{p},{lp},{i2},{li2} from [{i1} rdfs:label {li1}], {i1} p {i2}, [{i2} rdfs:label {li2}], [{p} rdfs:label {lp}] where i1=**Is** and i2=**Ik** union select {i1},{li1},{p},{lp},{i2},{li2} from [{i1} rdfs:label {li1}], {i2} p {i1}, [{i2} rdfs:label {li2}], [{p} rdfs:label {lp}] where i1=**Is** and i2=**Ik**
Subject match: instance Is Keyword match property Pk	select {i1},{li1},{p},{lp},{i2},{li2} from {i2} p {i1}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], [{p} rdfs:label {lp}] where i2=**Is** and p=**Pk**
Subject match: property Ps Keyword match instance Ik	select {i1},{li1},{p},{lp},{i2},{li2} from {i1} p {i2}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], [{p} rdfs:label {lp}] where p=**Ps** and i2=**Ik**
Subject match: property Ps Keyword match property Pk	select {i1},{li1},{p1},{lp1},{i2},{li2}, {p2},{lp2},{i3},{li3} from {i1} p1 {i2},{i1} p2 {i3}, [{i1} rdfs:label {li1}], [{i2} rdfs:label {li2}], [{i3} rdfs:label {li3}], [{p1} rdfs:label {lp1}], [{p2} rdfs:label {lp2}] where p1=**Ps** and p2=**Pk**

Fig. 3. The SeRQL query templates for two semantic entities.

[3] http://www.openrdf.org/

Query templates. The query templates are developed to describe how to retrieve relations between two semantic entities. As each semantic entity can be a class, a property or a instance, there are nine possible combinations in terms of formulating queries to find their relations. In the situation when the subject keyword does not produce a class match, but the other keyword does, we swap their position and treat the other keyword as the subject keyword. Thus, we are left with seven combinations. Figure 3 shows all these combinations and their templates.

Now let us investigate the first combination, which is when two keywords in a query both match classes. Suppose the subject keyword matches the class C_s and the other keyword matches the class C_k. The search results are expected to be the instances of the class C_s which have explicitly specified relations with the instances of the class C_k. For example, when querying for news about "phd students", the expected results are the news entries in which phd students are involved. Further, the search results are also expected to be self-explanatory, e.g., to motivate why certain news entries appear and others do not. Thus, along with the retrieving of news instances, the related phd stduents and the relations between students and news entries also need to be retrieved. Therefore, the search results of the query *news:phd students* are expected to be triples of *(news, relation, phd-sudent)*. Finally, in order to make the search results more understandable, the labels of each semantic entity also need to be pulled out to give an understandable explanation to users.

As shown in Figure 3, the query template of the combination described above is composed of two queries, which cover the relations from the instance of C_s to the instances of C_k in both directions. There are three pairs of variables in the query. The first pair, *i1* and *li1*, denotes the instances of the class C_s (i.e., URIs and labels), e.g. news stories in the example mentioned above. The second pair, p and lp, indicates the relations defined between i1 and i2, e.g. *has-author* or *mentions-person* in the example. The last pair is the variables *i2* and *li2* which represent the instances of the class C_k, e.g. phd students involved in the corresponding news stories.

As we can see from the code, we only take into account the direct relations that are explicitly specified in the back-end repository. The techniques for finding implicit relations like the ones used in TAP and in [7] have not been used. This is for the sake of simplicity and for the sake of improving response time.

There are six other combinations, which have also been shown in Figure 3. When the subject keyword matches a class, the search results are the instances of the matched class that have relations with the matches of other keywords.

In the situations when there are no class matches found for both keywords, the focus of user query varies according to the type of the semantic matches of keywords:

- When the subject keyword matches an instance and the keyword matches a property, the search results are the values of the matched property of the matched instance. For example, the results of the query *AKT:member* are the members of the project *AKT*.

- When the subject match is a property and the keyword matches an instance, the results are the semantic entities which have the matched instance as the value of the matched property. For example, when query for *author:enrico*, the results are the instances (e.g. publications) that have the person *Enrico Motta* as authors.
- When both keywords match instances, the intension of the user query is often to find out the instances that have relations with both matched instances. For example, the results of the query *Victoria:Enrico* are the instances that have relations with both people, e.g., projects that both people participate in, publications that both people co-authored, etc.
- When both keywords match properties, the results are the instances that have both of the matched properties as relations to other instances. One example is the query *author:member*, which gives back persons who are both authors and project members.

Query formulation. In the context of simple queries where only two keywords are involved, the task of query formulation is relatively easy, which is to initiate the template that corresponds to the combinations of the semantic matches of the user keywords. As each keyword may match more than one semantic entity, often more than one query needs to be constructed. More specifically, if the subject keyword matches n_s semantic entities and the other keyword has n_k matches, there are n_s*n_k queries that need to be constructed. For example in the query *news:phd students*, two formal queries need to be constructed. This is because while the subject keyword matches the class *news-item* the keyword "phd students" matches two semantic entities, including the class *phd-student* and the instance *phd*. The more the generated formal queries, the slower the search process will become. This problem becomes more acute when there are many keywords involved in the user query. We will discuss how to reduce the number of formal queries in the following.

6.2 Complex user queries

We define complex queries as those which involve more than two keywords. As described earlier in Section 4, such keywords can be required or optional, and there is always at least one required keyword, which is the subject keyword. The Text Search Layer of the search engine finds semantic matches for the keywords. To construct formal queries, the search engine needs to combine the semantic matches together and construct sub-queries for each of the combinations. A key operational problem is that in real world situations there can be a large number of matches and hence even more combinations.

Combining different keywords. For keywords k_1, k_2, ..., k_n, suppose that the number of the semantic matches of the keyword k_i is n_i. There will be n_1*n_2*...*n_n (which can be respresented as $\prod_{i=1}^{n} n_i$) different combinations when considering all the keywords as required ones. Each combination of the matches corresponds

to a RDF-based formal query. Apart from considering all the keywords as required ones, the search engine also needs to investigate the combinations where one or more keywords are left out, in order to producing complete result sets to end users. Table 1 shows how to calculate the numbers of combinations possible for choosing different numbers of keywords from the keywords set. Note that because keyword k_1, the subject keyword, is always a required keyword in SemSearch, n_1 appears in all these calculations.

Table 1. Numbers of keywords combinations for keywords k_2, k_2, ..., k_n

Number of considered keywords	**Number of combinations**
n	$C_n = \prod_{i=1}^{n} n_i$
n-1	$C_{n-1} = n_1 * \sum_{i=2}^{n} \prod_{j=2, j\neq i}^{n} n_j$
n-2	$C_{n-2} = \frac{n_1 * \sum_{i=2}^{n} \sum_{j=2, j\neq i}^{n} \prod_{k=2, k\neq i, k\neq j}^{n} n_k}{2!}$
n-3	$C_{n-3} = \frac{n_1 * \sum_{i=2}^{n} \sum_{j=2, j\neq i}^{n} \sum_{k=2, k\neq i, k\neq j}^{n} \prod_{m=2, m\neq i, m\neq j, m\neq k}^{n} n_m}{3!}$
...	
2	$C_2 = n_1 * \sum_{i=2}^{n} n_i$
Total combinations	$\sum_{i=2}^{n} C_i$

The total number of keywords combinations is the sum of the combination number of choosing *n*, *n-1*,..., *2* keywords from the specified keywords, which is $\sum_{i=2}^{n} C_i$. This indicates that the total number of different combinations can get huge when i) there are many keywords involved and ii) some keywords are very generic and thus have many matches. For instance, imagine there are 3 keywords in the user query (including the subject keyword). Each keyword matches 3 different semantic entities. There will be 3*3*3=27 combinations for considering all three keywords as required ones and 3*(3+3)=18 combinations for only considering two. Thus the total number of combinations is 27+18=45. This indicates that rules are needed to reduce the number of matches for each keyword. In this context, we used several heuristic rules.

- **Rule1.** The subject keyword always matches class entities when there are more than two keywords involved in the user query.
- **Rule2.** Choose the closest entity matches of the keyword. This rule can significantly help us to reduce the number of entity matches. We are however concerned about losing useful matches.
- **Rule3.** Choose the most specific class match among the class matches.

Formulating formal queries For each combination of semantic matches, a formal query needs to be constructed. As mentioned earlier, we assume that the subject keyword always matches at least one class concept. Thereby, for keywords k_1, k_2, ..., k_n, we can expect that the main search results will always be the instances of the matches of the keyword k_1 (i.e. the subject keywords) that have relations

with other keywords. Furthermore, as described earlier in the last section, in order to allow the search results to be understandable, the relations and the related instances also need to be pulled out, which explains the search results.

A formal query in SeRQL comprises three building blocks: the *head* block, which describes what needs to be retrieved, the *body* block, which describes how, and the *condition* block, which expresses conditions. In addition, in order to cover relations of two entities in both directions, the query also comprises a *union* block, which covers all the possible relations between the involved keywords. The construction of all these blocks depends on the type (i.e. class, property, or instance) of the semantic entity match of each keyword contained in a combination of keywords' matches.

Figure 4 shows a fragment of Java code for constructing SeRQL queries from combinations of semantic matches of keywords. In the head block, two pairs of variables (p_j, lp_j) and (i_j, li_j) are added for each keyword, which give back the relations of the search results and the keyword in question thus facilitating the self-explanation of search results. The body block specifies that the search results and each keyword must be explicitly connected in triples according to the type of the semantic entity match of each keyword. The union block covers the other direction that is not covered in the main query. These building blocks are derived from the templates of simple user queries described in section 6.1.

7 Implementation and experimental evaluation

A prototype of SemSearch has been implemented, which uses Sesame and Lucene[4]. Sesame provides a query language and a query engine for semantic data represented in RDF. Lucene provides a fast text search engine, which is used to build the semantic entity index engine and the semantic entity search engine contained in the Text Search Layer of SemSearch.

The prototype has been applied to the semantic web portal of our lab. Figure 5 shows a screenshot of the search results of the query example *news:phd students*. As described earlier, the search engine not only gives back the information that the user is looking for but also gives back explanations, which makes the search results much more understandable than those in state-of-art tools. The search results are ranked according to their closeness to the specified user keywords. The search engine takes two factors into consideration when ranking. One is the matching distance between each keyword and its semantic matches. The other is the number of keywords the search results satisfy. The search engine also provides support for search refinement. It provides a web form to allow the user to choose the meaning of the keywords and thus supports the user in reformulating a better search.

To assess the performance of the semantic search engine, we carried out an initial study in the context of the KMi semantic web portal. We used the questions that were gathered to evaluate AquaLog (a question answering tool

[4] http://lucene.apache.org/

```
for (int x = 0; x < composed_matches.length; x++) {
  String serql_head = "select distinct i, li ";
  String serql_body = "";
  String serql_where = "";
  String serql_union = "";
if (subject_match.isClass()) {
    serql_body = "from {i} rdf:type {<"+subject_match.getURI()+
            ">},[{i} rdfs:label {li}] ";
    serql_union = from_serql;
    for (int j = 0; j < composed_matches[x].length; j++) {
       SemEntityMatch keyword_match = composed_matches[x][j];
       if (keyword_match.isClass()) {
         serql_head += ",p"+j+",lp" +j+",i" +j+",li"+j+" ";
         serql_body += ",{i" +j+"} rdf:type {<" +keyword_match.getURI()+
                 ">},{i} p"+j+" {i"+j+"},[{i"+j+"} rdfs:label {li"+j+"}],"+
                 "[{p"+j+"} rdfs:label {lp"+j+"}] ";
         serql_union += ",{i"+j+"} rdf:type {<"+keyword_match.getURI()+
                 ">}, {i"+j+"} p" +j+" {i},[{i"+j+"} rdfs:label {li"+j+"}],"+
                 "[{p" +j+ "} rdfs:label {lp" + j + "}] "; }
       else
        if (keyword_match.isProperty()) {
         serql_head += ",p"+j+",lp"+j+",v"+j+",lv"+j+" ";
         serql_body += ", {i} p"+j+"{v"+j+"}, [{p"+j+"} rdfs:label {lp"+j+"}],[{v"+j+
                       "} rdfs:label {lv" + j + "}]";
          serql_where += "p"+j+"=<"+ keyword_match.getURI() +">";
           serql_union += ...;}
       else
         if (keyword_match.isInstance()) {
           serql_head += ",p"+j+",lp"+j+",i"+j+",li"+j+" ";
           serql_body += ", {i} p"+j+" {i"+j+"}, [{i"+j+"} rdfs:label {li"+j+"}],[{p"+j+
                         "} rdfs:label {lp" + j + "}]";
           serql_where += "i"+j+"=<"+keyword_match.getURI()+">";
           serql_union += ...;
 }
 if (serql_where.trim().length() > 0)
      serql_where = " where " + serql_where;
   //assembing a query
 String serql = serql_head+" "+from_serql+serql_where +" union "+ serql_head+" " +
              union_from_serql + serql_where;
              ...
}}
```

Fig. 4. A fragment of Java code for SeRQL query construction.

developed in our lab) [6] as the basis for experimental evaluation. Several reformulations of each search were attempted if necessary. For each search, we assigned a score that describes the performance of the search engine: i) 0 - no result; ii) 1 - could get a result with heavy analysis; iii) 2 - could get a result with moderate analysis; and iv) 3 - good result.

Taking into account only the questions for which an answer is possible, the average score was 2.1. The performance scores are only a qualitative assessment of how we felt the system answered the questions so these results are biased. However, based on these rudimentary performance measures the semantic search is answering questions reasonably well where data is available. In particular, SemSearch was able to answer a high proportion of the questions despite its simplicity. It is intuitive and simple to learn. The user doesn't need to have a full grasp of the ontology to get started (though they need to know something). This is an affordance of the way that results are presented in the interface in a way that informs the user about the terms in the ontology. That information can be used to help with search refinement. For example, the user might not remember relations like "generic-area-of-interest" but might remember that the

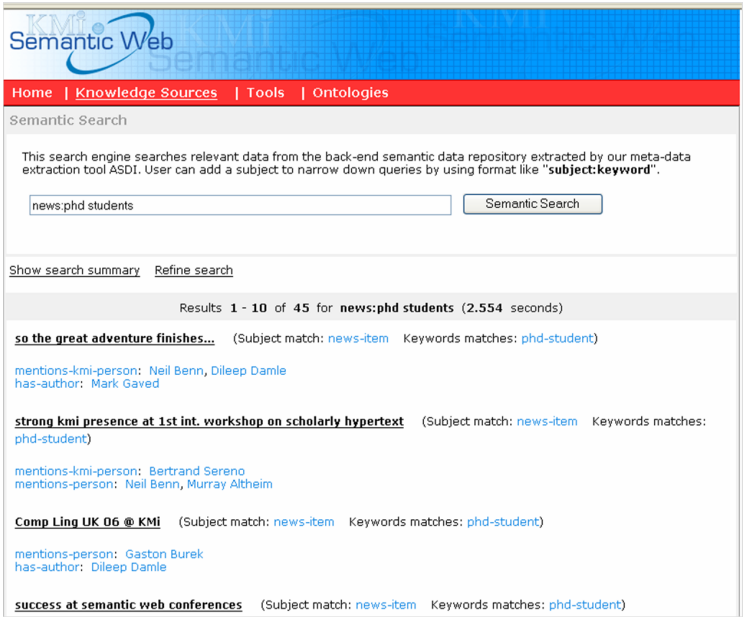


Fig. 5. A screenshot of the search results of the query example *news:phd students*.

word "area" is involved in a lot of relations about research topics and by browsing through the output of a search just for "area" can gather the information on the ontology vocabulary needed to formulate a better one.

8 Conclusions and future work

The core observation that underlies this paper is that, in the case of semantic search that promises to produce precise answers to user queries, it is important to ensure that it is easy to use and effective for ordinary end users who are not necessarily familiar with domain specific semantic data, ontologies, or SQL-like query languages. Our survey of state-of-art semantic search tools however reveals that current tools provide little support for end users.

In contrast with these efforts, our semantic search engine, SemSearch, provides several means to address this issue. First, SemSearch overcomes the problem of knowledge overhead by supporting a Google-like query interface. As described in Section 4, the proposed query interface provides a simple but powerful way of specifying queries. Second, SemSearch is able to produce precise answers for user queries by providing comprehensive means to make sense of user queries and to translate them into formal queries. In particular, as described in Section 6, the produced answers on the one hand satisfy user queries and on the other hand are self-explanatory and understandable by end users. Finally, SemSearch provides means (i.e., search refinement forms) to support end users in refor-

mulating better queries. A prototype has been implemented. The experimental study indicates a encouraging results.

Future work will focus on i) carrying out a more thorough evaluation which will help us to investigate the problems of each main component of the search engine and to improve their performance accordingly; ii) developing comprehensive means to perform semantic matching between keywords and semantic entities without compromising the performance of the search engine in terms of time; iii) developing more fine grained rules which on the one hand will help us to significantly reduce the number of keywords combinations and on the other hand will help us to identify and keep useful information in the reduction process; and iv) developing a powerful ranking mechanism, which guarantees the best results always staying on the top.

Acknowledgements

We wish to thank Marta Sabou for her valuable comments on earlier drafts of this paper. This work was funded by the Advanced Knowledge Technologies Interdisciplinary Research Collaboration (IRC) and the Knowledge Sharing and Reuse across Media (X-Media) project. AKT is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01. X-Media is sponsored by the European Commission as part of the Information Society Technologies (IST) programme under EC Grant IST-FP6-26978.

References

1. P. Cimiano. ORAKEL: A Natural Language Interface to an F-Logic Knowledge Base. In *Proceedings of the 9th International Conference on Applications of Natural Language to Information Systems*, pages 401–406, 2004.
2. O. Corby, R. Dieng-Kuntz, and C. Faron-Zucker. Querying the Semantic web with Corese Search Engine. In *Proceedings of 15th ECAI/PAIS, Valencia (ES)*, 2004.
3. C. Fellbaum. *WORDNET: An Electronic Lexical Database*. MIT Press, 1998.
4. R. Guha, R. McCool, and E. Miller. Semantic Search. In *Proceedings of the 12th international conference on World Wide Web*, pages 700–709, 2003.
5. J. Heflin and J. Hendler. Searching the Web with SHOE. In *Proceedings of the AAAI Workshop on AI for Web Search*, pages 35 – 40. AAAI Press, 2000.
6. V. Lopez, M. Pasin, and E. Motta. AquaLog: An Ontology-portable Question Answering System for the Semantic Web. In *Proceedings of European Semantic Web Conference (ESWC 2005)*, 2005.
7. C. Rocha, D. Schwabe, and M. de Aragao. A Hybrid Approach for Searching in the Semantic Web. In *Proceedings of the 13th International World Wide Web Conference*, 2004.
8. M.C. Schraefel, N.R. Shadbolt, N. Gibbins, H. Glaser, and S. Harris. CS AKTive Space: Representing Computer Science in the Semantic Web. In *Proceedings of the 13th International World Wide Web Conference*, 2004.
9. L. Zhang, Y. Yu, J. Zhou, C. Lin, and Y. Yang. An Enhanced Model for Searching in Semantic Portals. In *Proceedings of the 14th international conference on World Wide Web (WWW 2005)*, 2005.