KEYWORD SEARCH INTERFACE FOR PATH QUERIES ON ONTOLOGY

by

SUJEETH THIRUMALAI

(Under the Direction of Amit P. Sheth & Lakshmish M. Ramaswamy)

ABSTRACT

Today's semantic web has a growing wealth of machine understandable metadata represented using markup languages like RDF, XML or OWL. There exists a plethora of query languages that aid is searching such data models. However, most real world searches involve queries expressed in natural language as it allows the user to get information without using complex formal query languages. This paper presents a search interface for path queries on ontologies, which accepts keywords and finds answers where each answer is a subgraph containing paths between nodes that match the keywords. Our approach for building such a system comprises of (1) a full-text search index for triples in the ontology (2) lexical and semantic query expansion to match user keywords to entities in the ontology, and (3) an algorithm which uses the Sparq2l path sequence indices to compute the answer subgraphs.

INDEX WORDS: Semantic Web, Path Query, Keyword Search, Ontology

KEYWORD SEARCH INTERFACE FOR PATH QUERIES ON ONTOLOGY

by

SUJEETH THIRUMALAI

B.E., University of Madras, India, 2003

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2007

© 2007

Sujeeth Thirumalai

All Rights Reserved

KEYWORD SEARCH INTERFACE FOR PATH QUERIES ON ONTOLOGY

by

SUJEETH THIRUMALAI

Major Professors:	Amit P. Sheth Lakshmish M. Ramaswamy
Committee:	Kang Li Prashant Doshi

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2007

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisors Dr. Amit Sheth & Dr. Lakshmish Ramaswamy for their valuable time and suggestions rendered during the course of this work. I am most grateful to Kemafor Anyanwu, whose guidance was pivotal in the completion of this work. Thanks are also due to my friends at UGA who have always been supportive and cared for my welfare. Finally, I am very thankful to my parents without whose love, support and encouragement none of this would be possible.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 MOTIVATION 3

1.2 PROBLEM DEFINITION 5

1.3 CONTRIBUTIONS 5

2 RELATED WORK & BACKGROUND 7

2.1 RELATED WORK 7

2.2 BACKGROUND 11

3 SYSTEM ARCHITECTURE 17

3.1 TRIPLE SEARCH INDEX 18

3.2 QUERY PROCESSING 21

4 EVALUATION 25

4.1 EXPERIMENT SETUP 25

4.2 ALGORITHM CORRECTNESS 32

5 CONCLUSION & FUTURE WORK 34

REFERENCES 35

APPENDICES

A SCREENSHOTS OF THE SEARCH INTERFACE 39

LIST OF TABLES

Page

1. 'Insider Threat' dataset general statistics 26

2. 'US Politics' dataset general statistics 26

3. Query Execution Times for Insider Threat dataset 27

4. Query Execution Times for US Politics dataset 27

5. Query Execution Time over Different Dataset 31

LIST OF FIGURES

Page

1. Motivating Scenario 4

2. Single source all path algorithm 12

3. Path Summarization 14

4. System Architecture 18

5. Multi Source Path Expression Algorithm 24

6. Insider Threat Dataset: Query Processing Time versus Number of Solved Nodes 29

7. US Politics Dataset: Processing Time versus Number of Solved Nodes 29

8. Comparing Query Execution Times over Different Datasets 31

9. Query: "Council Infiltration, Middle East" 39

10. Query: "Leader of, Philippines" 40

11. Query: "International fund, connected with, saddam hussein" 41

12. Query: "Mark Sanford, South Carolina" 42

13. Query: "Larry Craig, veto, Governor Quinn" 43

14. Query: "Arnold, registers, lawsuit" 44

Chapter 1

INTRODUCTION

Semantic Web (SW) [36] is as an extension to the current web where the content is machine understandable thus facilitating easy information integration. The Resource Description Framework (RDF) [1] is the core of W3C's SW language and technologies based on RDF provide the primary capabilities for building most SW applications. The current web has a growing availability of semantic metadata created from large repositories of information. Recent notable efforts like DBpedia [2] and Yago [3] extract structural information from Wikipedia [37], a large and popular community created encyclopedia, and use RDF to represent the information. In RDF, each resource has a unique identifier called the URI and statements are made using the resource URIs. RDF based technologies; especially an RDF query language can provide users to express complex queries against these repositories. An example of a complex query as given in [24] tries to "find soccer players with number 11 on their jersey, who play in a club whose stadium has a capacity of more than 40000 people and were born in a country with more than 10 million inhabitants". Such queries cannot be asked against Web-based data using traditional search engines. It is the representation of Web-based data in richer RDF form, and the expressiveness of RDF query languages which enables such a querying capability.

Several languages have been proposed for querying RDF. For example, the SPARQL query language [25] which just recently became a recommendation of the World Wide Web Consortium (W3C) is based on triple matching. However most real world searches, as done by common users, involve queries represented in natural language as this allows users to easily express their information needs without the knowledge of complex query languages, the underlying schema and the domain vocabulary. For example, for the query "Who is the author of "Introduction to Algorithms"?" the user needs schematic information on the relationship between the concepts *book* and *author* in order to construct the correct formal query. Hence, the main challenge towards providing user-friendly search on ontology is to provide an interface that accepts keywords and maps it to some internal graph representation of class, instance or relationship in the ontology. The mapped sub-graph is then used for processing the query. We propose a minimized NLP extension to full-text search indexing of the ontology to accomplish the goal of mapping user keywords to ontological entities. The triples are processed to include synonyms, derived words (stemming) & tokenized compound words (example: authorOf) in the search index.

There is an interesting class of query that is common especially in investigative applications where the user is interested in determining paths of associations between seemingly unrelated entities in the knowledgebase. While most languages support pattern-matching queries, there are few, like [5, 6, and 39] that provide support for such path queries. In the above mentioned languages, the searches are made using the node URI and the user has to adhere to a specified syntax for expressing the query. In this work, we propose a system that allows users to express path query search terms using

keywords. The answer to the keywords search is a set of subgraphs from the dataset where each subgraph includes paths involving entities that matched the keywords. A formal definition of the problem statement is presented in the following section.

1.1 MOTIVATION

Consider the scenario where the user is researching on the American Civil War. Suppose that he is looking for an association between the American Civil War and writings of the Greek historian Thucydides. The simple text search provided by Wikipedia using the keywords “American Civil War” “Thucydides” retrieves articles that have relevance scores not greater than 8.3%. As is common with such queries, the association might involve multiple entities which may be spread across the information web. Traditional search indices do not capture such link information.

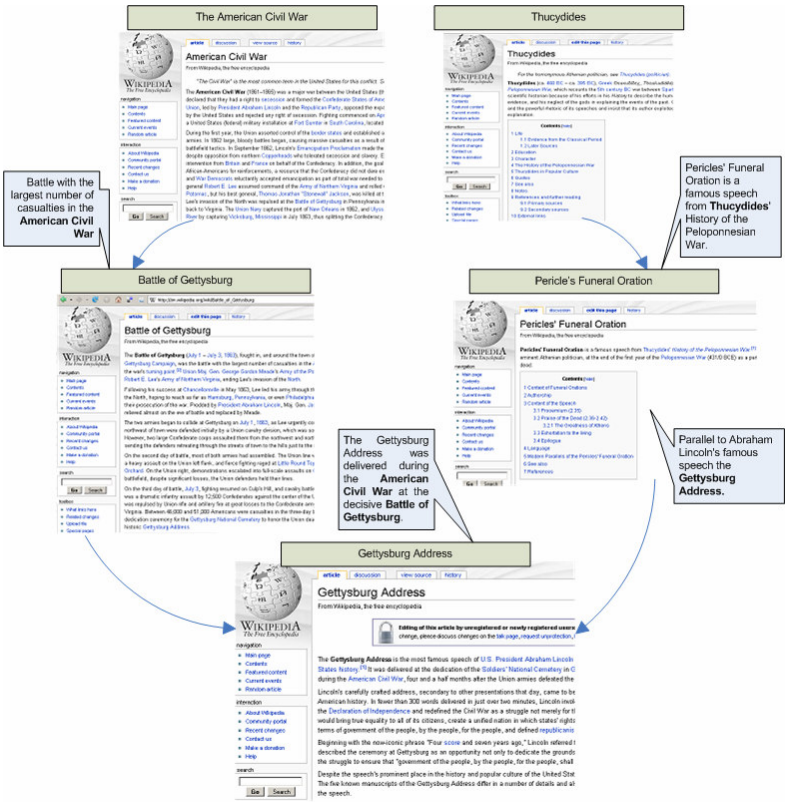


Figure 1: Motivating Scenario

This means that the user still has to sift through the result pages and traverse internal links to determine the association. Representation of facts from Wikipedia in RDF allows complex queries that make use of the structure of information to be constructed. DBpedia provides a SPARQL endpoint [41] where users can enter formal queries to search against structured information extracted from Wikipedia. Similarly, Panto takes in a natural language query and transforms it into an ontology query pattern. SPARQL based interfaces can only provide answer to pattern matching queries where the user has the knowledge of the relationships that connect the entities in the query. When the user does not know the relations involved in the association, the query belongs to the class of Path queries. Systems such as Panto cannot answer such queries.

1.2 PROBLEM DEFINITION

Formally, the problem we are trying to solve may be defined as follows:

Similar to [8], given a directed graph $G = (V, E)$ where each node $v \in V$ and edge $e \in E$ has a label (URI) associated with it, we are concerned with querying this graph using keywords. A keyword search query q consists of a list of n keywords (k_1, k_2… k_n). The answer to query q is the set of paths in G where the end point of each path is a node $v \in V$ that matched a user keyword based on one of the following criteria:

- There exists some keyword $k \in (k_1, k_2 \ldots k_n)$ that matches label of node v either lexically or on semantic query expansion. For example, the keyword "democratic convention" lexically matches the entity "democratic national convention". The keyword "national assembly" matches "national convention" through semantic query expansion of assembly into convention.
- Node v is the subject/object of the ontological triple whose predicate label matches some keyword lexically or on semantic query expansion. For example, the keyword "author" lexically matches the relationship "authorOf". The keyword "writer" matches the relationship "authorOf" through query expansion. In both cases, the node v that is the node associated through the predicate "authorOf".

1.3 CONTRIBUTIONS

The contributions of this work are given below:

- We present a full-text search index for ontology triples that provides matching capabilities based on semantic and morphological expansion of terms used for indexing the triple.
- Given a set of nodes based on text matches, we propose a method to construct the set of answer paths using the algorithm to solve the multi-source path expression problem.

The rest of the thesis is organized as follows. We survey related work and background information in Chapter 2. In Chapter 3, we introduce our system architecture and discuss the algorithm to solve the multi-source path expression problem. We discuss system evaluation in Chapter 4. Experiments over two datasets indicate that the query execution time is directly proportional to the number of entities involved in the query. Also, we note that the query execution time increases with the size of the dataset. Finally, we discuss future work and conclude in Chapter 5.

Chapter 2

RELATED WORK & BACKGROUND

2.1 RELATED WORK

Information search is one of the most popular applications with significant room for improvement. The availability of large amounts of structured, machine understandable information on the semantic web offers opportunities for improving traditional search. The Resource Description Framework (RDF) [1] is a powerful data model that it the core of W3C's Semantic Web architectural layers. It is a standard that provides the features for interoperability of data & machine understandable semantics for metadata. There exist several RDF query languages including RQL, RDQL [40], SeRQL, TRIPLE and SPARQL. However most real world searches, as done by common users, involve queries represented in natural language, such as English, that they are familiar with. This allows for users to express their information needs without the knowledge of the underlying schema or vocabulary of the ontologies.

The problem of natural language interfaces to knowledge bases has been extensively studied for years. [30, 29, 31] allow for keyword search over relational databases. [15, 13] provide a natural language interface to search over XML. [13] uses the tree structure of XML in translating the search keywords into XQuery [32] expressions. In this work, we present a keyword search interface over RDF ontology.

Such ontologies are directed acyclic graphs and hence the techniques used for XML cannot be applied.

Systems like [10, 33] are based on formal querying languages, a few allow the querying of Semantic Web repositories using keyword queries or rdf path fragments. Our system is different from the above systems, as it supports keyword search not only on literals, but also on related words of instances and relationships. The systems allow users to enter only a single keyword or literal per search, unlike our system which allows multiple keywords in a single search. The key feature of our system is that it supports searches on related words of instances and relations, unlike the direct or pattern based keyword searches. Additionally, our system displays search results in the form of paths.

Kowari [10] is a native RDF store that stores information using a RDF database. It allows users to query using iTQL RDF query language, which is similar to SQL. Sesame is a RDF database with support for RDF Schema inference and querying. It supports several query languages including SeRQL. Jena provides persistent storage of RDF using relational database. It provides SPARQL query language support for accessing parts of RDF/RDF or OWL and inference capabilities through SPARQL's inference engine. Swoogle [11] is a search and retrieval system for searching ontologies on the web. [11] uses a ranking scheme that utilizes relationship weights between Semantic Web Documents (SWD) to model the probability of being explored. Swoogle allows keyword searches on classes, literals or properties. The system uses a spread activation algorithm to find related instances or literals for a given set of concepts using a initial set of relationship weights.

QuizRDF [12] is another search engine that allows keyword searches on annotated documents. The searches in QuizRDF are limited to literals. Beagle++ [34] is a desktop search application that supports RDF path fragment queries and retrieves annotated desktop resources. It uses Lucene [19] to index RDF triples and paths. The system expects the user to have knowledge of the ontology. It takes path sequence queries such as *creator/affiliatedTo MIT* to find all documents whose authors are affiliated to MIT. [12] does not support searches on related words of ontological classes or relationships, unlike our system that supports both.

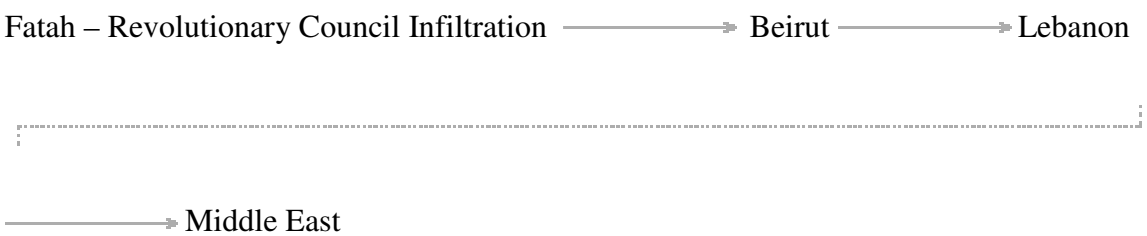
Similar approaches have been proposed for supporting keyword searches over relational and XML databases. However, these approaches often limit their search to the set of literal values i.e. leaves or terminal nodes, e.g. the title of a book or an author's name. The applications retrieve data by repeated joining of the data or tuples associated with the matched fields. [29] provides data and schema browsing through interactive displays. XRank [15] allows searches on XML elements or tags. Our system provides keyword searches on RDF documents and hence the challenges are different compared to keyword searches on database or XML documents.

The closest work that is related to our work is Panto [4]. It provides an interface that accepts general natural language queries and outputs SPARQL queries. They use the StanfordParser [35], WordNet [20] and string metrics algorithms to make sense of words in the natural language query and map them to entities (class, instance or relation) in the ontology. Then they translate the semantics of the query into a SPARQL query. The translation also supports features such as negation, comparative and superlative modifications.

Our work differs from Panto in the class of query handled. [4] translates the natural language query into its corresponding SPARQL query. When a user searches using SPARQL, he has the knowledge of the relationships that are involved between entities he is searching for. For example, a query like "Who are the tennis players from Moscow?" would translate into a SPARQL query as shown below:

```
Select ?player
Where {
        ?player     placeOfBirth   "Moscow".
        ?player     rdf:Type       tennis_player.
}
```

In our system, we try to answer path queries, where the user is looking for the relations that connect the entities in question. Consider the path:



Given the path above, [4] can answer the following queries:

- Where did the Fatah council infiltration take place?
- What is the capital of Lebanon?
- What council infiltration took place in Beirut?

However, [4] cannot answer a query such as:

- Which council infiltration took place in the middle east?
- How is Fatah Infiltration connected to Lebanon?

In the first query, we know the entities that are involved – Council Infiltration, Middle East. [4] can only solve queries in which the entities are directly connected through the specified relation. Hence, it will look for the triple that connects the two entities using the relations "took place". In the second query, the user has used the keyword connected to represent all possible relations (paths) between the entities involved. Our system identifies the entities that are involved in the query and determines all paths that include those entities.

2.2 BACKGROUND

In this section, we provide an introduction to Tarjan's algorithm [9] to solve the single source path expression problem; Labeling scheme proposed by SPARQ2L [5]; Lucene Search Engine; and WordNet English Lexicon. In our work, we propose an adaptation of Tarjan's algorithm in order to determine path expressions that include multiple sources. Understanding the single source algorithm is very important to analyzing our algorithm.

Tarjan Algorithm

Given a directed graph $G = (V, E)$ with a distinguished source vertex s, the single source path expression problem is to find, for each vertex v a regular expression $P(s, v)$ which

represents the set of all paths in G from *s* to *v*. [9] describes a decomposition method for computing these path expressions.

The input to the SOLVE [9] algorithm is a path sequence (P_i, v_i, w_i), $1 \leq i \leq l$ such that P_i is an unambiguous path expression of type (v_i, w_i). The notion of path sequence is based on an ordered graph i.e. every node has a unique number. In [5], an approach for labeling and indexing path sequences is described.

```
procedure SOLVE

begin

        Initialize:

                P(s, s) = Λ;

                for each v∈ V – {s}, P(s,v) = Ø

        Loop:

                for each path expression Pi do

                if vi = wi, then

                        P(s, vi) = P(s, vi)·Pi

                if vi ≠ wi then

                        P(s, wi) = [ P(s, wi) U [P(s, vi)·Pi] ]

end SOLVE
```

Figure 2: Single source all path algorithm

Sparq2l Labeling & Indexing

SPARQ2L [5] uses a concise representation of paths (called P-Expressions) instead of an enumerated listing. For example given the triples (x, P, y), (x, Q, y) and (y, R, z), the summary of paths between x and z can be represented as (P U Q · R). The system uses a binary encoding scheme to efficiently represent such regular expressions as opposed to a string representation.

[5] has a hierarchical labeling scheme based on 3 identifiers:

Component Identifier: unique number assigned to individual strong component during a depth first search on the graph.

Level Identifier: it is the depth of the strong component node in the optimal spanning tree.

Subgraph Identifier: identifies disconnected non-tree subgraphs and the dangling tree subgraphs.

Given a path query with source *s* and destination *d*, the labeling scheme has the **non-reachability property** that can be stated as follows:

- If s and d do not have the same subgraph identifier, then the query result is empty.
- If the level identifier of source is greater than that of the destination, the query result is empty.
- Any node with a level identifier lesser than that of *s* and greater than that of *d* cannot be a member of the result set.

The result of the SOLVE [9] algorithm for a source *s* is an array of path summaries. The array is indexed based on the node identifiers. For example, the array below summarizes the paths for a given source node. An entry of Ø indicates that there is no path between the source and that node. Otherwise, the P-Expression for the path is stored as part of the array element.

Ø	Q · R	Ø	Ø

Figure 3: Path Summarization

Lucene Search Index

Lucene [19] search engine is a Jakarta open source project used to build and search indexes. It can index text documents and retrieve them based on various search criteria. It provides a basic framework which can be used to build a full-featured search engine. Lucene indexes using document objects. Thus, the text documents which are to be indexed have to be converted to document objects. Each document object consists of a set of field objects containing name and value pairs. The *name* is of type String and *value* can either be a String or a Reader object. Field class in Lucene provides various methods depending on whether the text in the *value* part of the field is tokenized, indexed or stored. Depending on the requirements some of the text information is tokenized, indexed or stored. A Lucene allows users to search on the values of these fields and this is done using an IndexSearcher object. All query terms are parsed using an analyzer, which is wrapped within the query object. Lucene provides four different analyzers to parse the

search terms in the query: the StopAnalyzer, WhiteSpaceAnalyzer, SimpleAnalyzer, and StandardAnalyzer. An analyzer takes in a stream of text and returns a set of tokens. Lucene tokenizes the queries depending on the kind of analyzer. The StopAnalyzer is used to split the terms and eliminate any stop words that exists in the query. The WhiteSpaceAnalyzer splits the query terms based on white space. The SimpleAnalyzer splits the text at non-character boundaries, such as special characters ('@','&' etc.). The StandardAnalyzer is the most sophisticated parser with rules for email addresses, acronyms, hostnames, floating point numbers, as well as the lowercasing and stop word removal. Lucene provides two important classes to build and search on a index. IndexWriter class is used to build the index and IndexSearcher class to search on the built index. Lucene provides tools to generate query objects called Query Parser.

The QueryParser class takes the search terms or queries and wraps them in a query object. This query object is later used by the *search* method in the IndexSearcher class. Later, the IndexSearcher returns the *Hits* object for the query. This *Hits* object is similar to a vector and contains the ranked list of document objects for a given query. For our use, we have implemented a PorterStemAnalyzer by extending Lucene's analyzer class and have used it to stem the words to its base forms to eliminate any stop words.

WordNet

The WordNet [20] is an online lexical reference system developed at the Cognitive Science Lab of Princeton University. Currently WordNet contains about 150000 words organized into 115,000 synsets of nouns, verbs, adjectives and adverbs. Each synset or set of words are related to other synsets by common relationships such as hypernym or

hyponym, and meronym or holonym, verb groups i.e. groups of related verb forms, synonyms or similar meaning words, derivational forms or morphological forms etc. There exist different groups of synonymous words that are grouped based on the sense of a particular word. For example, the word *faculty* has two synsets since it has different senses based on the usage context. WordNet can retrieve the different sets of related word information depending on the POS (Part of Speech) of the word. For example, the word *teaches* has related word forms such as verb groups, synonyms, derivational forms, and hyponyms. The derivational form of a word is given by adding the morphological suffixes. For example, derivational form of a word *write* is *writing*.

Chapter 3

SYSTEM ARCHITECTURE

.

The System has two execution phases: Pre-processing phase and the Query processing phase. During the pre-processing phase, the system builds indices that are later used while processing the keyword search query. Building indices offline predominantly helps in reducing the execution time of the query. Figure 5 shows the system architecture diagram. Before the user enters the query, the ontology is loaded and the system builds the necessary indices. We identify two kinds of indices that need to be built during the pre-processing phase: (1) Text search index for the ontological triples, (2) Labeling and indexing of graph's path sequence. We use Sparq2l system to build the path sequence indices as described in section 2.2. As described earlier, the query to a keyword search consists of a list of n keywords and the answer to the query is the set of paths in G where the end point of each path is a node $v \in V$ that matched a user keyword based on one of the following criteria:

- There exists some keyword $k \in (k_1, k_2 \ldots k_n)$ that matches label of node v either lexically or on semantic query expansion.
- Node v is the subject/object of the ontological triple whose predicate label matches some keyword lexically or on semantic query expansion

In order to achieve this, we first require an ontology text search index that can match user keywords to entity labels in the ontology.

3.1 TRIPLE SEARCH INDEX

In order to facilitate fast matching of user search terms to ontological entities, we build a search index for triples in the ontology. Consider the triple: <Thomas Cormen, *author*, Introduction to Algorithms>. The system creates an index for every triple in the dataset using its subject, predicate and object values in the least. Thus a search keyword *author* will retrieve the above triple based on its match of the predicate index.

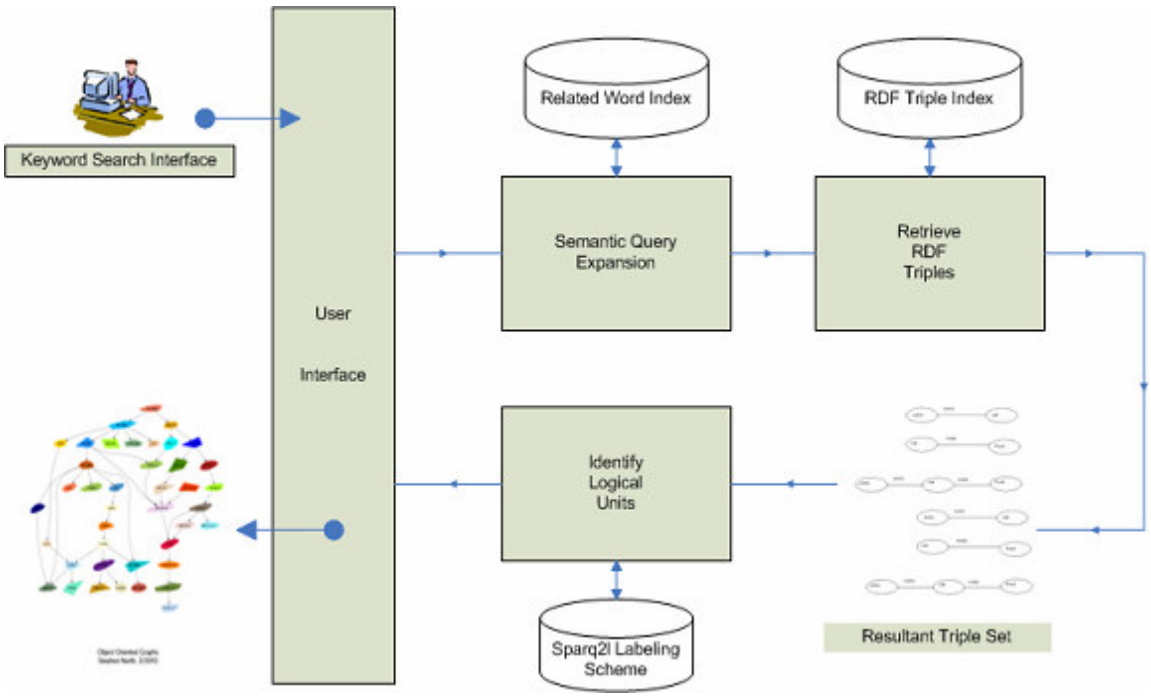


Figure 4: System Architecture

However, more often the labels in the ontology require lexical and semantic processing before creating the index. For example, it is very common to label relationships using compound words. The relationship *author* is more commonly represented using

compound words *author_of* or *has_author*. In such cases, it is necessary to tokenize the label and remove commonly occurring stop words.

Another common issue in information retrieval is vocabulary mismatch. For example, the user might have entered the keyword *writer* while the ontology has the term *author*. The main reason for such mismatches is the restricted vocabularies of knowledge bases. Traditional search engines do query expansion to overcome this problem. Query expansion is the technique of adding related terms to the original query to improve upon the problems of word mismatches. We use WordNet [20], a large lexical database of English, to expand the ontological term with its synonyms. Hence, while indexing *author*, its synonym *writer* is also indexed.

As part of the term expansion procedure, the system also indexes all derivational forms of a word. We use the porter stemming algorithm [16] to determine the root stem word for then determine derivational words by adding noun/verb suffixes to it. Hence, the ontological label *writer* will also be indexed using its derivations *write, writer* and *writings*.

Thus for every triple, we create an index using the following fields:

- Subject label & URI
- Predicate label & URI
- Object label, URI or value (in case of literal)
- Synonym
- Derived Words

We use Apache's Lucene to build the ontology triple index. As described earlier, the lucene index is made up of document objects and each document object consists of multiple design-specified fields. A field represents the value against which the index is queried. An example of fields for a document could be author, language, topic, data modified etc. A search of type *author:Thomas Cormen* will retrieve all documents authored by Thomas Cormen. In the case of ontology triples, each triple constitutes a document object and we define the following fields: subject label, predicate label, object label, subject URI, predicate URI, object URI/value (literal), synonyms & derived words. The search index is extensible in that if the triple needs to identified by additional properties (say class names), adding the new property as a document field would suffice the requirement without needing further changes to existing document structure.

Lucene provides four types of field objects that determine how the field values are indexed and stored. They are:

- *Field.Keyword* - The data is stored and indexed but not tokenized. This is most useful for data that should be stored unchanged such as a date.
- *Field.Text* - The data is stored, indexed, and tokenized. *Field.Text* fields should not be used for large amounts of data such as the article itself because the index will get very large since it will contain a full copy of the article plus the tokenized version. For the triple index, this field can be used to index the subject, predicate and object labels.
- *Field.UnStored* - The data is not stored but it is indexed and tokenized. Large amounts of data such as the text of the article should be placed in the index

unstored. Fields involving synonyms and derivational words of the ontological entities need not be stored as they do not need to be retrieved.

- *Field.UnIndexed* - The data is stored but not indexed or tokenized. This is used with data that you want returned with the results of a search but you won't actually be searching on this data. In our application, since we do not allow searching for the URI, there is no reason to index it but we want it returned to us when a search result is found.

If the user's keyword matches any of these keys, the corresponding triple is retrieved. The use of synonyms and derived words as search indices provides for semantic query expansion during the online query processing phase.

3.2 QUERY PROCESSING

As described earlier in the problem definition, the answer to the path query is a set of paths between nodes in the graph where the end points matched the user's keywords. Query processing consists of taking all the matched nodes as input and computing paths that connect them. [9] describes a method to solve the single source path expression problem, i.e. given a source node *s*, compute all paths between *s* and nodes in the graph. In our case, we have multiple matched nodes, each of which could potentially be the source of a path. We describe our MULTISOLVE algorithm later in the section.

Consider the keyword search query *Amit Sheth Writings Semantic web*. We use the two criteria mentioned in the problem definition to determine the nodes to include in the resultant path set. The first criterion includes all nodes (instances) whose labels

directly matched the keyword either lexically or through semantic query expansion. In the example above, this would match entities *Amit Sheth* and *Semantic Web* and retrieve the triples that contain the matched entities as subjects or objects. In the case that a relationship label (*AuthorOf*) matches a keyword (*Writings*), we include all nodes that are associated with the relation. In order to filter out irrelevant nodes, we only consider already matched entities that have the matched property/relation. Hence, we would only include nodes that are objects of the triple (*Amit Sheth*, *AuthorOf*, ?o). Here, we say that the keyword has matched an ontology entity through semantic query expansion. While creating the index, the system handles the relationship label *AuthorOf* in the following manner. The final list of words used for indexing the relation *AuthorOf* contains the word *Writings*.

- Tokenize label and remove stop words {*Author}*
- Find synonyms of the word using WordNet {Author, Writer}
- Using porter stemming algorithm, determine the stem of each of the words in the list {Author, Write}
- Find derivatives from the stemmed word by adding noun/verb suffixes {Author, Writer, Writing, Writes, Writings}

The system determines subject/object matches first and only then handles matches at the predicate level. The main reason to do this is to easily filter out unwanted nodes before computing the paths.

At the end of the matching procedure using Lucene search index, we would have a list of nodes that are to be included in the resultant set of paths. The next step is to

determine the nodes that form the end points of the paths. In order to be able to use the underlying SPARQ2L system for path computation, the labels, as represented in sparq2l, of the matched nodes have to be determined. The multisolve algorithm iterates through this set and computes all the paths involving them. Applying the non-reachability property of the sparq2l labeling scheme, we can determine the order in which the set has to be iterated. We propose that sorting the nodes in increasing order of their IDs will result in the complete set of all paths between the matched nodes. As discussed in section 2, each node ID in sparq2l consists of 3 identifiers: strong component, subgraph & level identifier. The comparator between two nodes with IDs {scc1, sub1, lev1} and {scc2, sub2, lev2} is as follows:

- If scc1 is not equal to scc2 then the node with smaller scc ID is smaller. Else goto next step.
- If sub1 is not equal to sub2 then the node with smaller sub ID is smaller. Else goto next step.
- Node with smaller level ID is smaller.

Figure 6 presents the multi source path expression algorithm. We discuss the experiment and evaluation in the next section.

procedure MULTISOLVE

begin

Initialize:

M: Set of nodes matched using lucene search index

S: Set of nodes already solved in the algorithm

For each node s∈ M DO

P(s, s) = Λ;

for each $v \in V - \{s\}$, P(s,v) = Ø

End For

Loop:

Do until all nodes in M are marked

Mark smallest node s ∈ {M}−{S} and add to S

SOLVE(s)

For each node $a \in \{M\}-\{S\}$, Do

if $a \in$ path(s)

Mark *a* and add a to S

Add path(s) to result

End Do

end MULTISOLVE

Figure 5: Multi Source Path Expression Algorithm

Chapter 4

EVALUATION

4.1 EXPERIMENT SETUP

Implementation We have implemented our algorithm using Java 1.5 on a dual 1.8GHz AMD Opteron machine with 15GB RAM and running Linux 2.4. The version of Sparq2l used for path computation uses: Berkeley DB Java edition for storage and indexing; Colt library [18] to perform all matrix implementations. We used Brahms and SemDis API to parse the ontology and create a graph model which we used in creating triple indices and building indices for path expressions. To build the triple index, we use Apache Lucene [19] which is a high performance search engine library written in Java. To perform semantic term expansion, we use WordNet [20] which is a publicly available lexical database in English. The system is made web-based using the Apache Tomcat Java servlet container.

Datasets We used two datasets in our performance evaluation. The Insider Threat [21] dataset was developed at the LSDIS lab as part of the Insider Threat initiative at Advanced Research Development Activity (ARDA). The second is a real-world dataset that is populated using information on US political news. Tables 1 & 2 below show the properties of the datasets.

Table 1 'Insider Threat' dataset general statistics

INSTANCE LEVEL

Instances:	26442
Literals :	66047
Instance statements [instance - property - instance] :	34534
Literal statements [instance - property - literal] :	151313

SCHEMA LEVEL

Schema classes :	57
Schema class literals :	105
Schema properties :	75
Schema property literals :	45
Schema class statements [class - property - class] :	1
Schema class literal statements [class - property - literal] :	105

Table 2 'US Politics' dataset general statistics

General dataset statistics.

INSTANCE LEVEL

Instances:	471
Literals :	950
Instance statements [instance - property - instance] :	797
Literal statements [instance - property - literal] :	2345

SCHEMA LEVEL

Schema classes :	31
Schema class literals :	55
Schema properties :	32
Schema property literals :	21
Schema class statements [class - property - class] :	1
Schema class literal statements [class - property - literal] :	55

Table 3 Query Execution Times for Insider Threat dataset

Query String	Time (seconds)	Num. Paths	Max. Path Length	Num. Nodes
Council Infiltration , Middle East	6.57	3	3	2
abdul paktin, 2001 hijacking	6.58	2	3	2
oil training, middle east	6.58	3	2	2
REVU , robert fisher	9.44	2	2	2
janet wallace , unisys	9.48	1	1	3
james lawson ,stryker	9.54	1	1	3
Leader of , Philippines	10.42	2	2	3
norman phillips , lyondell	12.07	3	2	3
robert fisher, gap	14.66	3	3	4
al islamia , afghanistan	15.54	13	4	5
abdul paktin ,saddam hussein	17.17	3	4	8
International fund ,connected with ,saddam hussein	19.93	5	3	9
deborah pryce, ohio	21.53	1	1	8
barakaat , bank of england	62.45	39	4	23

Table 4 Query Execution Times for US Politics dataset

Query String	Time (Seconds)	Num. Paths	Max. Path Length	Num. Nodes
Arnold, registers, lawsuit	1.25	1	1	2
Mark Sanford, South Carolina	1.26	1	1	2
craig thomas, florida	1.27	5	7	3
obama, 2000 election	1.28	4	6	3
craig thomas, republican party	1.28	10	7	3
craig thomas, laura bush	1.29	5	6	3
christopher bond, patriot act	1.29	7	6	4
arnold, jeb bush	1.29	8	8	2
craig thomas, governors association	1.33	5	7	3
john mccain , oxley act	1.33	2	2	3
Larry Craig, veto, Governor Quinn	1.37	2	2	3
signed, patriot act	1.65	2	3	3
craig thomas, 2004 republican convention	1.67	8	7	3
obama , joseph biden	1.74	2	1	4
james talent, republican convention	2.61	17	7	7

We evaluate the performance of the multisolve algorithm by running path queries and measuring the query running time. For this experiment we randomly selected 20 different keyword queries for each dataset such that there exists at least one path between nodes in the query. Hence, we do not evaluate queries that do not result in any paths. Tables 3 & 4

show the result of our experiments on the Insider Threat and the US Politics datasets respectively.

The query execution time is in the order of seconds for the datasets chosen. We analyze the complexity of the algorithms involved and try to determine the factors affecting the running time of the query processing phase. The query processing phase consists of the following steps:

- Text search over the ontology to determine the nodes involved
- Sort the matched nodes in the order of increasing node IDs
- Compute paths that include the matched nodes

We build the search index for ontology using the Lucene search engine. It offers almost constant time lookup of documents in the order of milliseconds. We obtain average lookup times of 96ms for the US Politics dataset and 145ms for the Insider threat dataset. The runtime for the second step of sorting the matched nodes depends on the number of matched nodes. In our experiments, the most number of nodes that were to be sorted did not exceed 23. Hence, the algorithm chosen is not crucial for this step. We used the TreeSet data structure to sort the nodes based on their node IDs. The TreeSet comparator guarantees that the nodes are sorted at the time of insertion itself with a time complexity that is logarithmic on the number of nodes in the set.

Hence, we can deduce that the most time is spent in step 3 of the query processing phase. The multisolve algorithm iterates through the sequence of matched nodes and computes path expressions that include all of these nodes.

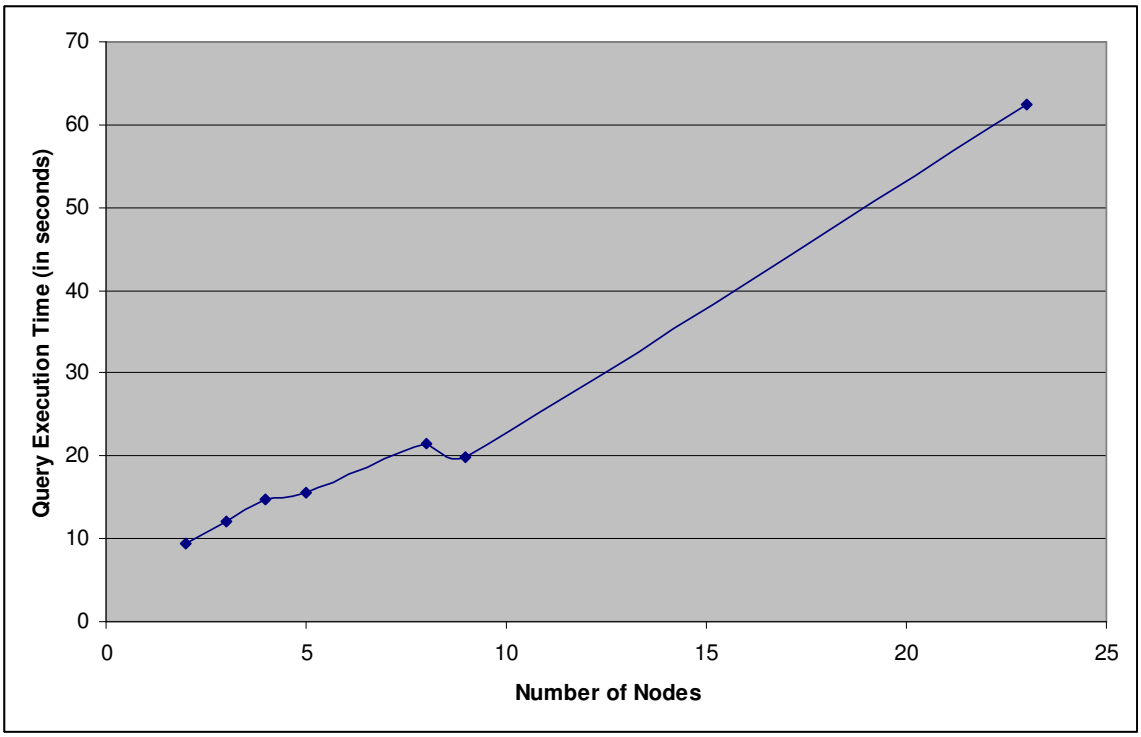


Figure 6 Insider Threat Dataset: Query Processing Time versus Number of Solved Nodes

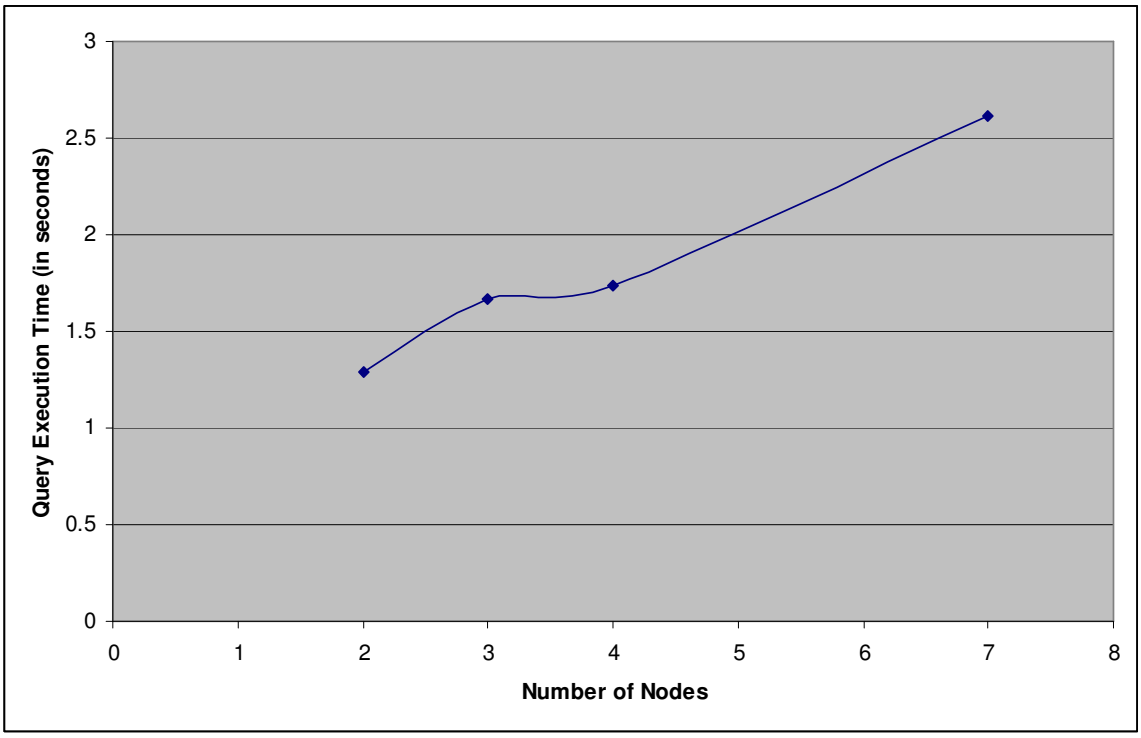


Figure 7 US Politics Dataset: Query Processing Times versus Number of Solved Nodes

The worst case scenario, in terms of runtime of the multisolve algorithm, is when all the matched nodes have to be solved using the Tarjan's single source path expression algorithm [9]. Suppose that the number of matched nodes is n and the number of path sequences in the graph (computed offline) is m, then the multisolve algorithm would scan the path sequence n times. Thus, the time complexity would be in the order of O(n.m). The value of m is dependent on the number of strongly connected components in the graph.

For a single source, the algorithm iterates through the entire path sequence and appropriately combines path expressions. Since we compute partial path expressions offline during the pre-processing phase, runtime is only affected by the method to combine partial path expressions. The underlying implementation in Sparq2l performs this operation in near constant time. This also implies that length of the path and number of paths does not affect the overall algorithm. Hence, the main factor that affects the runtime is the number of nodes that are considered as source nodes for computing path expressions. Figures 7 & 8 show the graph of query processing times against the number of matched nodes. It can be noted that the execution time increases with the number of nodes involved. However, in Figure 6 we note an anomaly. The runtime with 8 nodes is higher than with 9 nodes. For both of the queries, the number of path sequences read was the same. As mentioned earlier, SPARQ2L uses BerkeleyDB persistent storage to store the path sequences for a given graph. It is our educated guess that path sequences get cached during accesses to the database. Hence, the I/O access times could be smaller for a query with more number of nodes. This guess is to be validated and hence an issue to investigate in future work.

The length of the path sequence for a graph is directly proportional to the number of nodes in the graph. The query processing time directly depends on the length of the path sequence as the algorithm iterates through the entire path sequence for each source node. Figure 9 below shows the comparison between the query execution times of the two datasets used. The Insider Threat dataset has more number of nodes than the US Politics dataset and hence it takes longer to process queries, with same number of source nodes, over Insider Threat dataset.

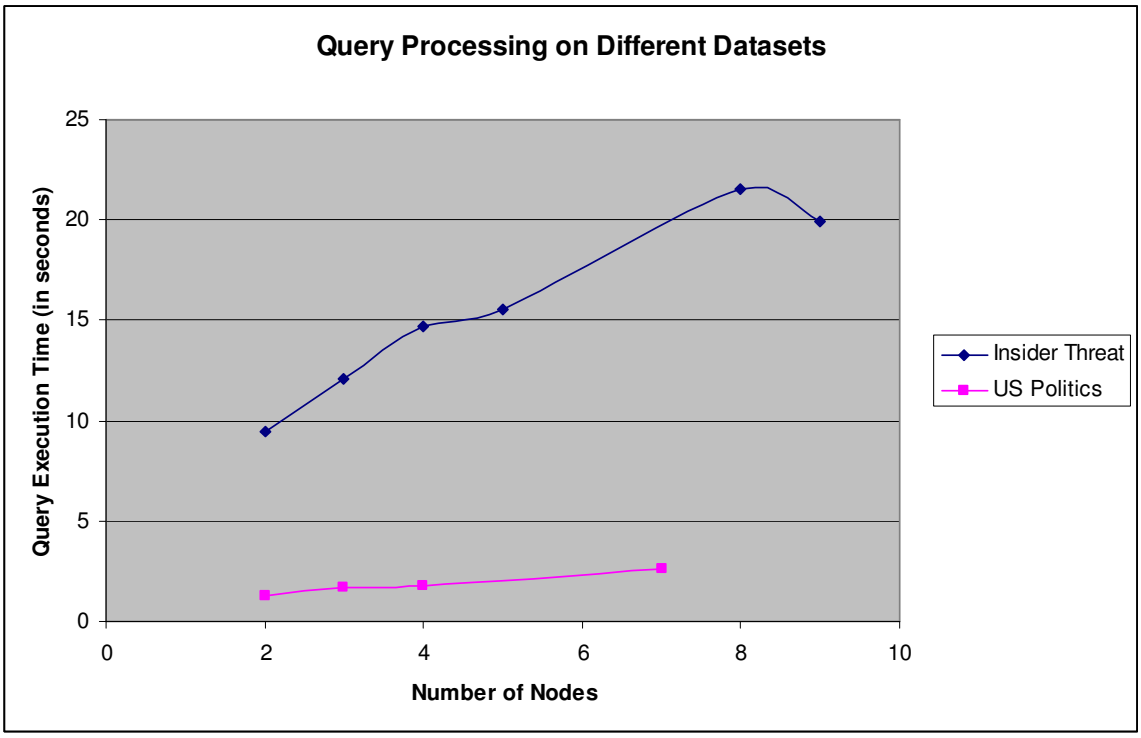


Figure 8 Comparing Query Execution Times over Different Datasets

Table 5 Query Execution Time over Different Dataset

Num. Nodes	US Politics - Query Execution Time (T1 seconds)	Insider Threat - Query Execution Time (T2 seconds)	Scale = T2 / T1
2	1.29	9.44	7.317829457
3	1.67	12.07	7.22754491
4	1.74	14.66	8.425287356

Table 5 shows the scaling of query execution times over the two datasets. The size of path sequence of the Insider Threat dataset is 26411 and that of US Politics dataset is 470 giving a size scaling of 56.2.

4.2 ALGORITHM CORRECTNESS

In section 2, we briefly described Tarjan's algorithm [9] to solve the single source path expression problem that given a source node and a path sequence will compute all paths between the source and any node *v* in the graph. [9] presents the algorithm and also discusses the proof of its correctness. [9] makes use of the single-scan-path-preserving property of path sequences and thus computes the path expression in a single scan of all the partial path expressions. In our scenario, we have multiple sources and the problem is to determine paths that include all these source nodes. Our main adaptation to [9] is in determining the order in which the source nodes are selected to compute the paths. We propose that selecting the nodes in the increasing order of their node IDs, determined based on the hierarchical labeling scheme; will ensure that no paths are missed.

The non-reachability property of the Sparq2l labeling scheme states that given two nodes *s* and *d*, there cannot be a path from *s* to *d* if any of the following is true:

- If *s* and *d* are not in the same subgraph
- If the level identifier of *s* is greater than that of *d*.

Thus, given a set S = {s1, s2 … s_n} of nodes, ordering them incrementally based on their node IDs would ensure that for any i and j such that nodeID(s_i) is less than nodeID(s_j),

the paths (if one exists) between s_i and s_j would be taken into consideration while computing the path expression. The single scan algorithm will solve si and correctly aggregate all the paths between s_i and nodes that appear later in the scan. Conversely, if node s_i is solved later than s_j, then the paths between s_i and s_j will not be considered in the result.

Chapter 5

CONCLUSION & FUTURE WORK

This thesis presents a prototype system that allows keyword search for path queries on ontology. The contributions of this work are as follows:

- We present a full-text search index for ontology triples that provides matching capabilities based on semantic and morphological expansion of terms used for indexing the triple.
- Given a set of text matches, we propose a method to construct the set of answer paths using the algorithm to solve the multi-source path expression problem.

The paths retrieved by the system are not ordered. Hence, it could potentially lead to information overload. Semantic association ranking metrics such as those suggested in [22, 23] could be used to present only paths most relevant to user's context.

DBpedia, Yago are recent efforts to generate semantic metadata by extracting structured information form the web (Wikipedia). A keyword or natural language search interface to such knowledge bases would prove immensely useful as the end user need not be aware of the structure of the information. While this work is limited to handling keywords, it will be worthwhile to build a search interface that accepts queries in natural language.

REFERENCES

[1] Resource Description Framework: W3C Semantic Web Activity. RDF Core Working Group. http://www.w3.org/RDF/

[2] Christian Bizer, Soren Auer, Georgi Kobilarov, Jens Lehmann, Richard Cyginiak. DBpedia - Querying Wikipedia like a Database. WWW 2007.

[3] Fabian M. Suchanek, Gjergji Kasneci, Gerhard Weikum: Yago: A Core of Semantic Knowledge - Unifying WordNet and Wikipedia. WWW 2007.

[4] Chong Wang, Miao Xiong, Qi Zhou and Yong Yu. PANTO: A Portable Natural Language Interface to Ontologies. ESWC 2007.

[5] Kemafor Anyanwu, Angela Maduko, Amit Sheth. SPARQ2L: Towards Support For Subgraph Extraction Queries in RDF Databases. WWW 2007.

[6] RDFPath. http://infomesh.net/2003/rdfpath

[7] Alkhateeb, Jean-François Baget, Jérôme Euzenat, RDF with regular expressions, Research report 6191, INRIA Rhône-Alpes, Grenoble (FR), 32p., May 2007.

[8] He, H., Wang, H., Yang, J., and Yu, P. S. 2007. BLINKS: ranked keyword searches on graphs. In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data. SIGMOD 2007.

[9] Tarjan, R. E. “Fast Algorithms for Solving Path Problems”. JACM, Vol. 28, No. 3, July 1981, pp. 594-614

[10] Adams, T., Gearon, P., Wood, D., Kowari. A Platform for Semantic Web Storage and Analysis. XTech 2005.

[11] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R. Scott Cost, Yun Peng, Pavan Reddivari, Vishal C Doshi, and Joel Sachs. Swoogle: A Search and Metadata Engine for the Semantic Web. Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management, 2004.

[12] J. Davies, U. Krohn, and R. Weeks: QuizRDF: search technology for thesemantic web. In WWW2002 workshop on RDF & Semantic Web Applications,11th International WWW Conference WWW2002, Hawaii, USA, 2002.

[13] Yunyao Li, Huahai Yang, H. V. Jagadish. NaLIX: an Interactive Natural Language Interface for Querying XML. SIGMOD, 2005.

[14] Krys J. Kochut and Maciej Janik. SPARQLeR: Extended Sparql for Semantic Association Discovery. ESWC, 2007.

[15] Lin Guo Feng Shao Chavdar Botev Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. ACM SIGMOD, 2003.

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

[17] Maciej Janik, Krys Kochut. "BRAHMS: A WorkBench RDF Store And High Performance Memory System for Semantic Association Discovery", Fourth International Semantic Web Conference ISWC 2005.

[18] Colt Library for High Performance Scientific and Technical Computing in Java. http://dsd.lbl.gov/~hoschek/colt/

[19] Apache Lucene. http://lucene.apache.org/

[20] WordNet: An Electronic Lexical Database. http://wordnet.princeton.edu

[21] Boanerges Aleman-Meza, Phillip Burns, Matthew Eavenson, Devanand Palaniswami, Amit P. Sheth: An Ontological Approach to the Document Access Problem of Insider Threat. ISI 2005.

[22] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. Semrank: ranking complex relationship search results on the semantic web. WWW 2005.

[23] Boanerges Aleman-Meza, Christian Halaschek-Wiener, Ismailcem Budak Arpinar, Amit P. Sheth: Context-Aware Semantic Association Ranking. SWDB 2003.

[24] Sören Auer, Jens Lehmann: What have Innsbruck and Leipzig in common? Extracting Semantics from Wiki Content. ESWC 2007.

[25] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/

[26] Surajit Chaudhuri, Raghu Ramakrishnan, Gerhard Weikum: Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? CIDR 2005.

[27] Holger Bast, Ingmar Weber. The CompleteSearch Engine: Interactive, Efficient, and Towards IR& DB Integration. CIDR 2007.

[28] Andreas Harth, Stefan Decker. "Optimized Index Structures for Querying RDF from the Web". 3rd Latin American Web Congress, Buenos Aires – Argentina.

[29] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. ICDE 2002.

[30] Vagelis Hristidis University of California, San Diego. DISCOVER: Keyword Search in Relational Databases. VLDB 2002.

[31] Vagelis Hristidis, Yannis Papakonstantinou, Andrey Balmin. Keyword Proximity Search on XML Graphs. ICDE 2003.

[32] XQuery: An XML Query Language. www.w3.org/TR/xquery

[33] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. Proc. of 2nd International Workshop on the Semantic Web, May 2001.

[34] T. Iofciu, C. Kohlsch¨utter, W. Nejdl, and R. Paiu. Keywords and RDF fragments. Integrating metadata and full-text search in beagle++. In Proc. of the Semantic Desktop Workshop held at the 4th International Semantic Web Conference, 2005.

[35] Klein, D., Manning, C.D.: Accurate Unlexicalized Parsing. In: ACL. (2003) 423-430

[36] Semantic Web. W3C Semantic Web Activity. http://www.w3.org/2001/sw/

[37] Wikipedia. http://www.wikipedia.org

[38] RDFPath. http://logicerror.com/RDFPath

[39] PSPARQL Query Language. http://psparql.inrialpes.fr/

[40] RDQL - A Query Language for RDF. http://www.w3.org/Submission/RDQL/

[41] DBPedia SPARQL Endpoint. http://dbpedia.org/sparql

Appendix A

SCREENSHOTS OF THE SEARCH INTERFACE

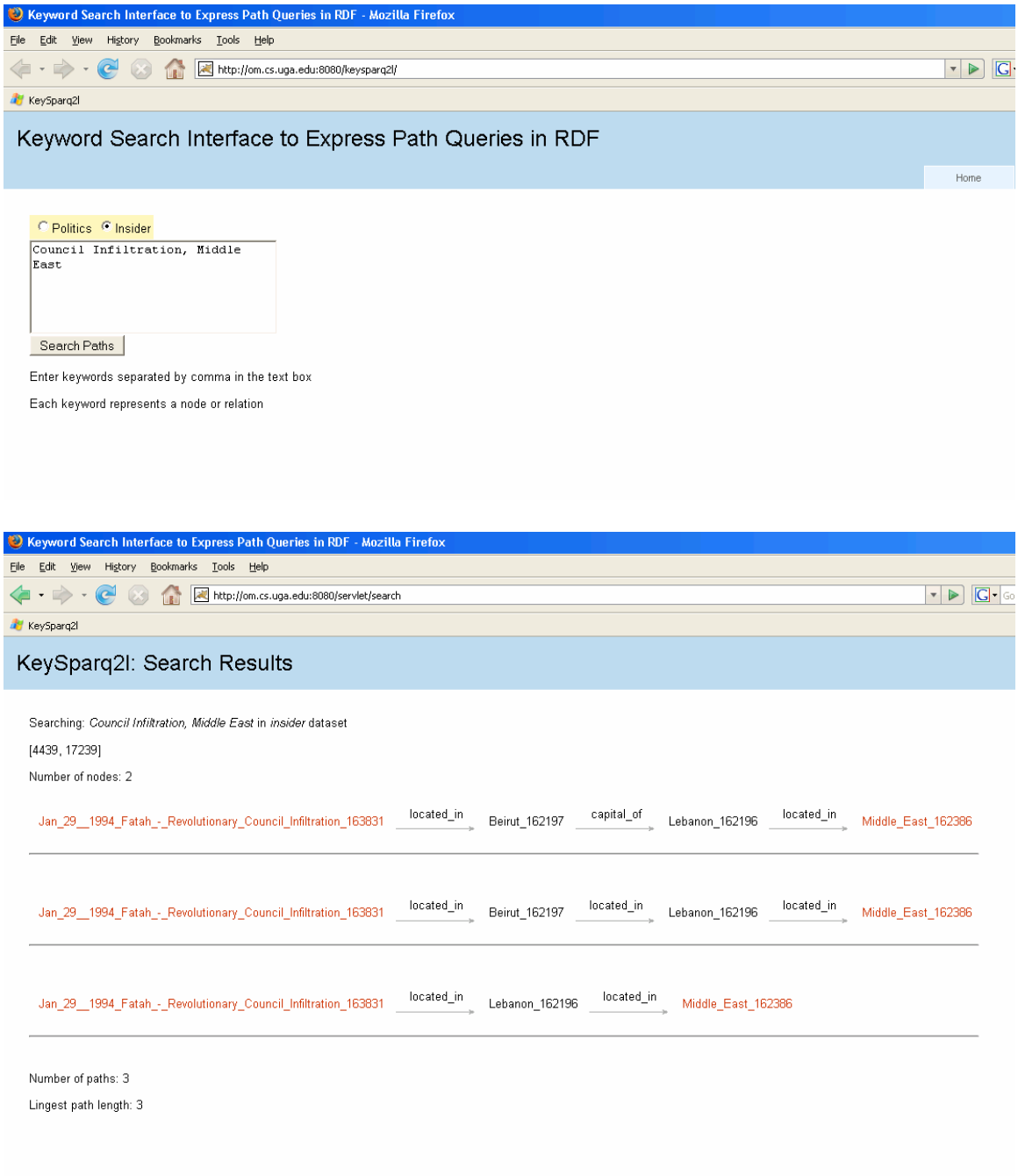


Figure 9 Query: "Council Infiltration, Middle East"

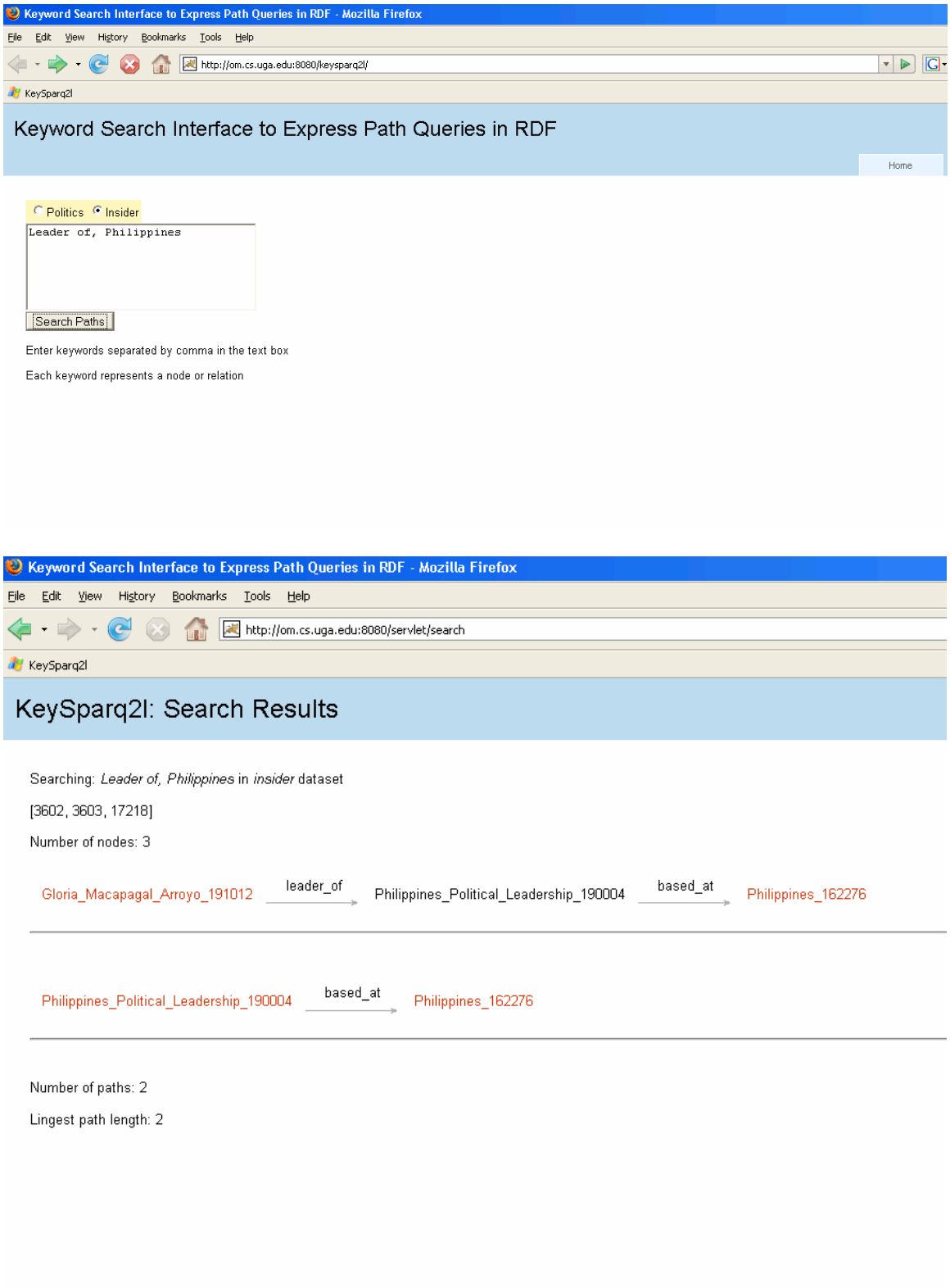


Figure 10 Query: "Leader of, Philippines"

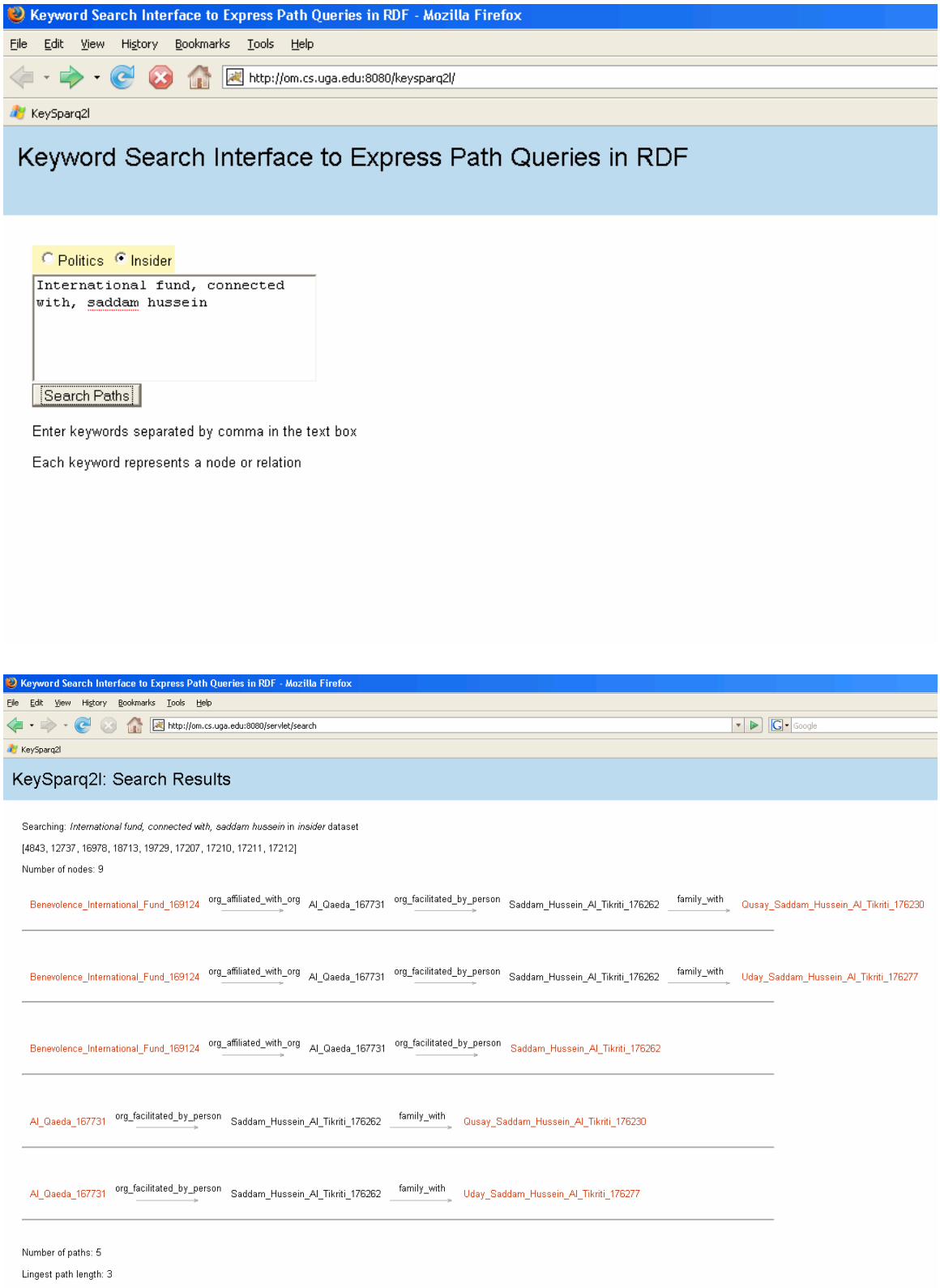


Figure 11 Query: "International fund, connected with, saddam hussein"

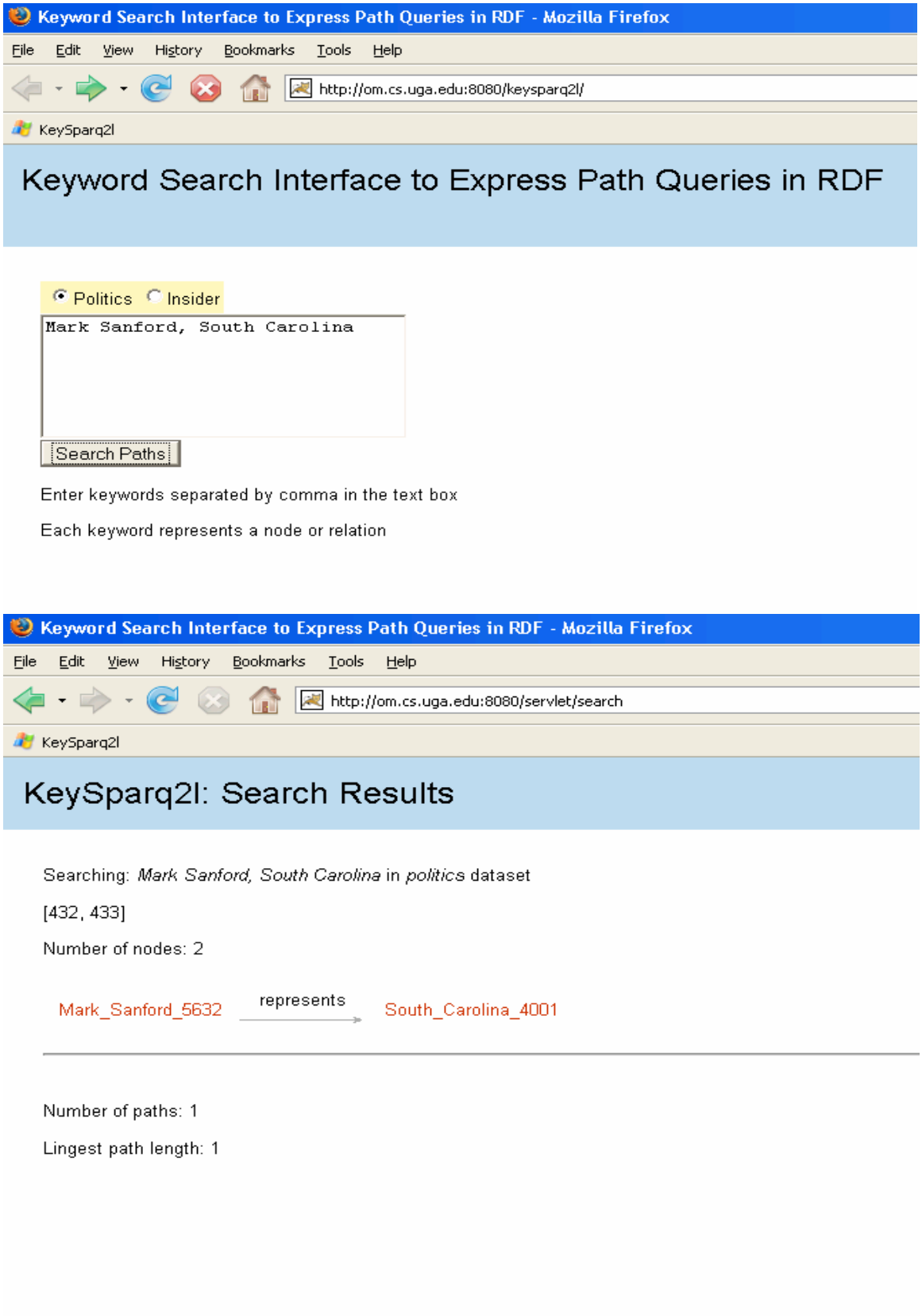


Figure 12 Query: "Mark Sanford, South Carolina"

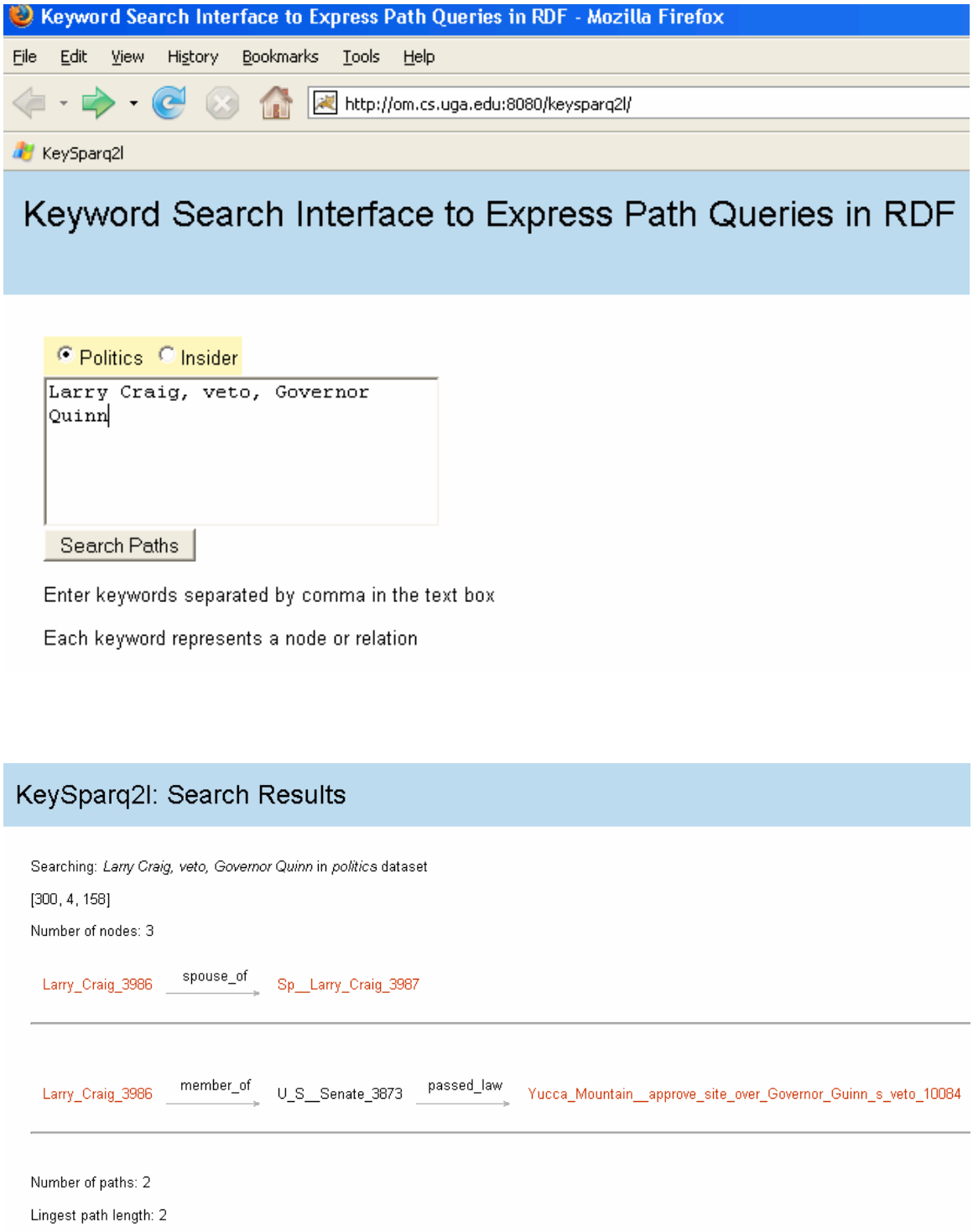


Figure 13 Query: "Larry Craig, veto, Governor Quinn"

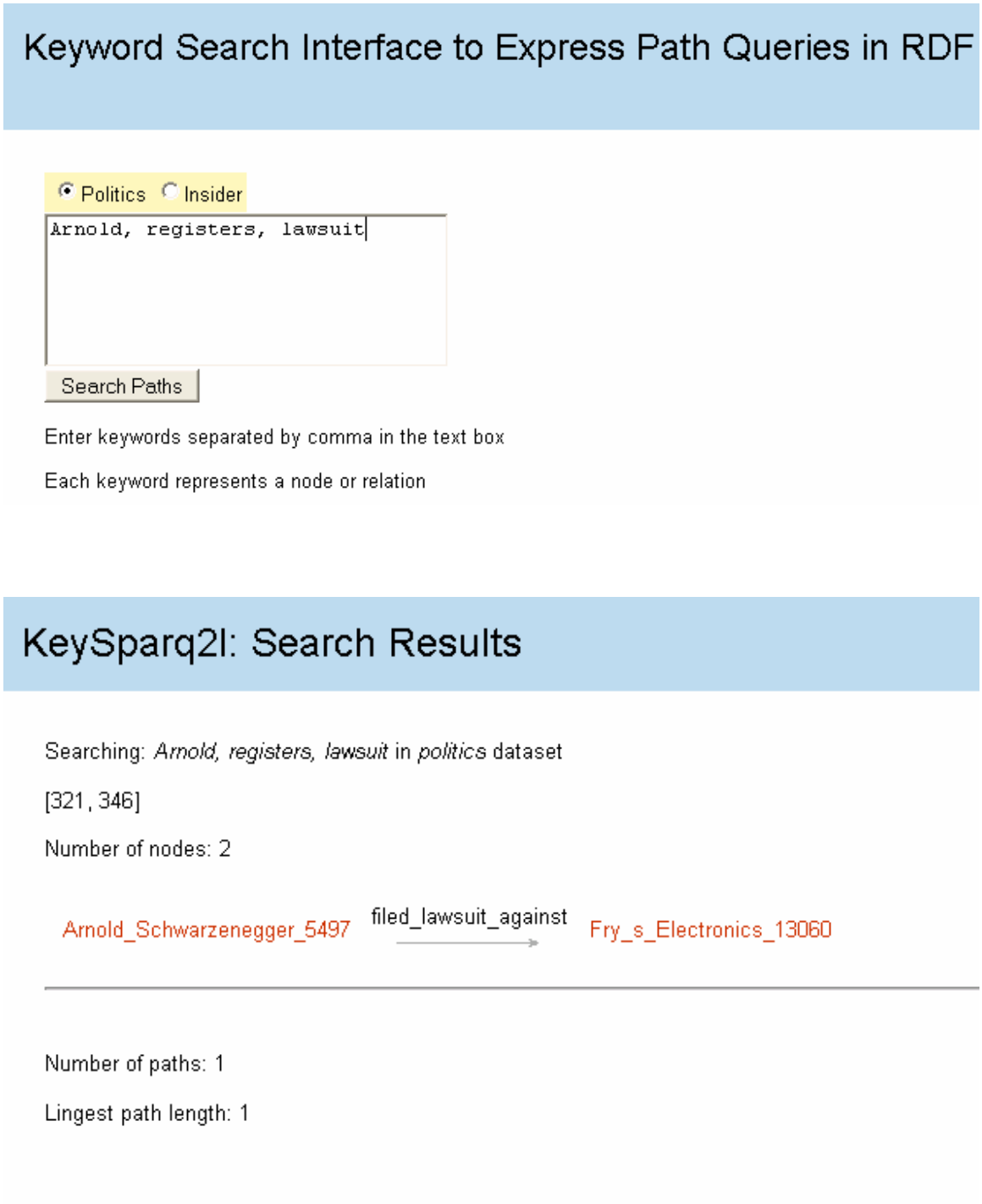


Figure 14 Query: "Arnold, registers, lawsuit"