

RDFBrowser – A tool to analyze RDF-based metadata

Bernhard M. Schueler
bernie42@uga.edu
bernie@uni-koblenz.de
Artificial Intelligence Center
The University of Georgia

Abstract

RDFBrowser is a tool for browsing RDF-based metadata including extensions like RDFS, OIL, DAML, but without using their special features. RDFBrowser allows for convenient browsing of the RDF-based metadata and for comparisons between files focusing on helping the user find semantic similarities and differences.

This report also includes a description of an advanced feature, which has not been implemented yet: Searching files for resources, which are likely to be synonyms, homonyms, and retrieving similar parts based on similarity of the underlying RDF-graph.

Keywords: *Semantic Web, RDF, graph isomorphism*

Content

1	Introduction.....	1
2	RDFBrowser.....	2
3	Implementation.....	4
3.1	Parsing.....	4
3.2	Knowledge base.....	4
3.3	Graphical User Interface.....	4
4	Unfinished: Analyzing the RDF-graph.....	4
4.1	Graph isomorphism.....	5
4.2	Algorithms.....	6
4.3	Approach for RDFBrowser.....	7
5	Conclusion.....	7
6	Installation.....	8
	Resources.....	9

1 Introduction

In order to supply machine readable information about the semantics of resources on the web standards like RDF, RDFS, DAML, OIL have emerged and are still emerging. The syntax of RDF is based on XML. The syntax and data model of RDFS, DAML, OIL is based on RDF. It seems reasonable to assume that further languages for representing metadata and ontologies, which might emerge in the future, will be based on RDF as well.

There are several tools available for building ontologies including *Protégé-2000* [PROT00] and *OntoEdit* [OE]. However there seem not to be many tools, which have the same focus as RDFBrowser. *XML spy* by *Altova* might share the ability to work with RDF-based descriptions using only RDF syntax. But it is not freeware. This made me choose this course project. For installation instructions see section 6.

The syntax of RDF, which is based on XML, is quite cumbersome for a human reader. Thus it is creating a need for tools to edit and view RDF-based metadata. RDFBrowser certainly cannot fulfill all of these needs since it only allows for viewing RDF files, not for editing them.

The underlying data model of RDF [RDF99] however provides two more readable representations of the data: As a graph, and as triples.

A RDF statement is a triple consisting of a resource *subject*, a resource *predicate* and a resource or literal *object*. Resources are identified by URIs.

A triple can be described by the following graph:

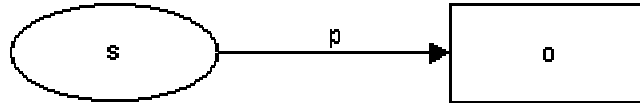


Fig. 1.1: A RDF triple, from [RDF99].

Since URIs are unique by definition, graph representations of triples can be joined together by having only one node for each resource in the RDF data. The graph in Fig. 1.2 represents a reified statement and consists of five triples, the subject of each being the resource in the middle.

RDFBrowser allows the user to browse the data based on the graph as well as based on triples. The compromise that I made here is that there is no graphical representation of the graph being shown. More about this in the next section.

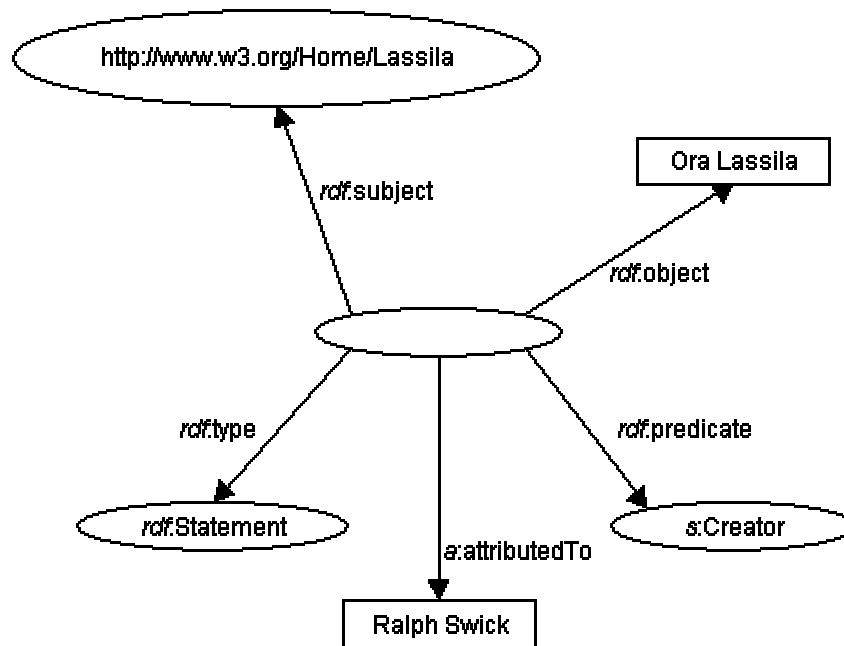


Fig.1.2: A RDF graph, from [RDF99].

2 RDFBrowser

RDFBrowser allows the user to open files containing RDF data and view the triples of the RDF data model. The data is shown in a table containing the triples of the RDF data model. See Fig. 2.1 for a screenshot. I choose not to go for a graphical display of the graph in order to support all options described below, because a table offers a better overview over bigger amounts of data and because a table requires much less computation. And after all it is much easier to implement this way.

By selecting a resource in the table the user can browse through the data. Different triples will be shown in the table according to the selection.

There are 3 basic options for browsing:

- *Browsing*: The program browses through the RDF graph. The user can select a resource in the fields “subject” or “object”. If he/she chooses a subject the browser will display all triples in which this resource occurs as an object. If he chooses an object, the browser will display all triples in which this resource occurs as subject. Thus he can move along the edges in the graph. If a user selects a literal, which might occur as object nothing happens.
- *Show same*: The user can select a subject, predicate or object and the browser will display all triples in which the selected resource occurs at the same place. This could e.g. be useful to display all subclass relations expressed via the RDF-Schema [RDFS] property `rdfs:subClassOf`.

- *Show all occurrences*: The user can select a subject, predicate or object and the browser will display all triples in which the selection occurs. This is useful to obtain all statements about a certain resource.

The user can open different files and browse each of them with any of those options.

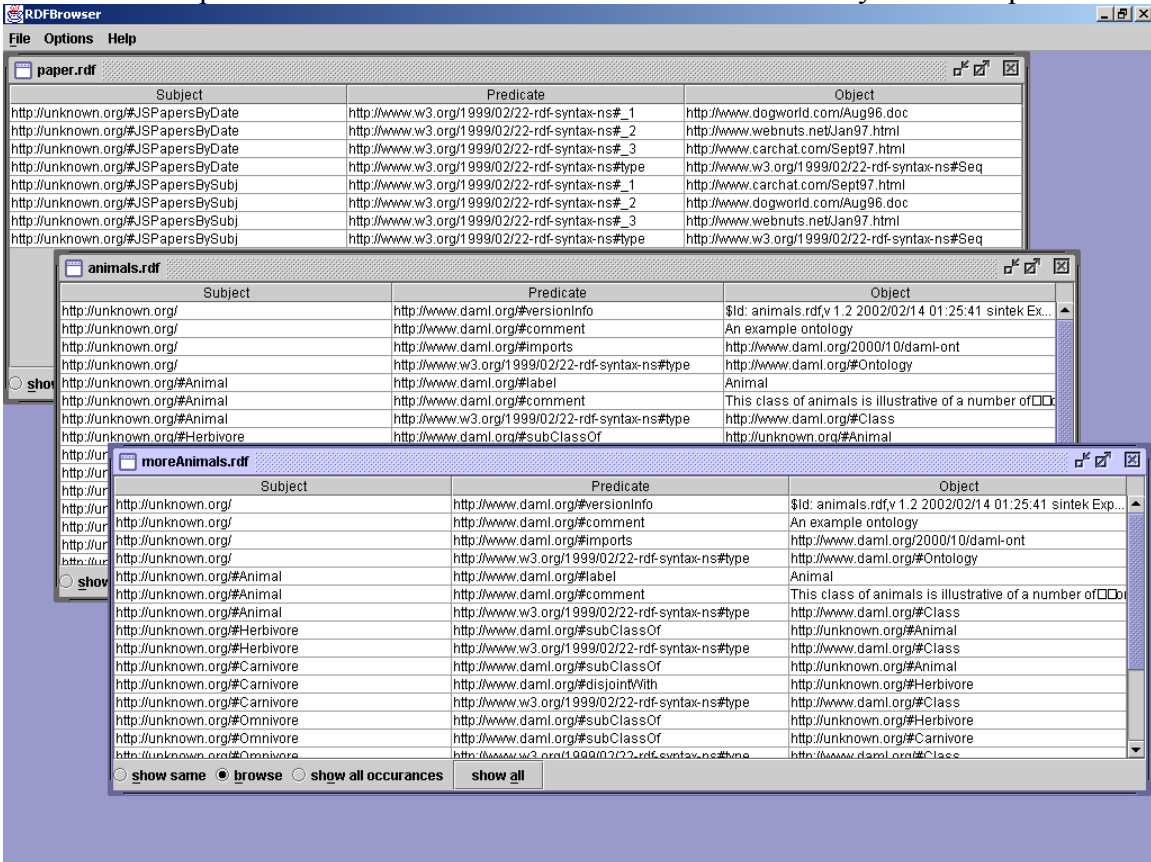


Fig. 2.1: Screenshot.

There are also 2 global options:

- *Single File*: This option is the default. If this is selected the browser shows the behaviour just described.
- *Simultaneous*: This option lets the user browse simultaneously through all opened files. He selects an entry in one of the tables and whatever calculation is performed on this data will be performed on all open files. E.g. if subject “a” and the option *browse* is selected in one file the program will look for triples with “a” as subject in all files, which are currently opened.

There is an exception that I considered to be convenient: If there is no such triple in one file the display for that file won’t change. It would be more consistent to display an empty table in this case. On the other hand the user would have to reset that table to showing all triples in order to work with this particular one. And there isn’t much use in showing an empty table.

Browsing files simultaneously is very useful in order to explore ontologies. It works quite well with ontologies written using DAML, or OIL. A requirement is of course that the same URI are used at least for parts of the ontology. It is easy to find similarities and differences, as long as they are depicted in the URIs used.

In order to take the structure of the RDF graph into account for comparing RDF data a deeper analysis of the structure of the graph has to be performed. See section 4 for more on this topic.

3 Implementation

When considering how to implement RDFBrowser I focused on the following: For a browser a graphical user interface is inevitable. Further should the program run under different operating systems and could use only components, which are freely available. I wanted to use Prolog to store the triples and to work with it for at least two reasons:

- Prolog can easily be abused as a database,
- The concise syntax of the language and the abilities of the logic inference engine make Prolog systems a first choice for implementing prototypes.

Since I was working alone on this project it seemed important to avoid endless coding of basic tasks.

I decided to use the following setting: ARP for parsing the RDF input, AMZI! Prolog for storing the triples and inferencing with them, Java for the GUI.

3.1 Parsing

The RDF-based data is parsed using the ARP parser, by Jeremy Carrol. This parser is written in Java and depends on the XERCES Java parser. ARP provides means to extract the triples of the RDF data model out of RDF files.

3.2 Knowledge base

The triples of the RDF data model are assigned into a Prolog knowledge base. RDFBrowser uses AMZI! Prolog. It features a Logic Server for Window, Linux, Solaris, and HP/UX Editions. It also provides interfaces to use this server from within Java, C, C++, Delphi and more. Therefore this system fits the need to cooperate with parser and GUI.

3.3 Graphical User Interface

The choice of what to use to build a GUI was easy: There aren't many free software systems for that, which support different platforms. So I chose Java and its Swing library.

4 Unfinished: Analyzing the RDF-graph

As mentioned in section 2 the system as described so far depends on the use of the same URIs in different file in order to highlight semantic information such as similarities. It further depends on the user seeing those similarities and differences.

When reviewing ontologies interesting tasks are finding synonyms, homonyms and isomorphism.

Synonyms are different words, which have the same meaning. In the case of RDF data they can be seen as different resources used with the same meaning.

Homonyms are usages of the same word, in which they have a different meaning. In the case of RDF data they can be seen as a resources used with different meanings.

What does *meaning* mean in the case of a resource. Within an ontology the meaning of a resource is defined by its relations to other resources. Looking at the RDF graph those relations are adjacencies to other nodes.

Notice that looking for homonyms, synonyms with respect to an ontology defined in RDF-S, DAML or OIL based on the RDF graph, results might be less accurate if there is other RDF based data in the file.

Finding an isomorphism between two graphs means determining that they have the same structure. Assuming that in an ontology the meaning of a resource is defined by its relations to other resources finding synonyms means finding occurrences of different resources with the same relations to other resources. This is obviously related to finding isomorphism. More about this in the next section.

4.1 Graph isomorphism

An isomorphism between two graphs g_1 and g_2 is a bijection f between the nodes in g_1 and the nodes in g_2 such that for every pair of nodes n_i, n_j from g_1 $f(n_i)$ and $f(n_j)$ are adjacent iff n_i and n_j are adjacent.

Imagine having written an ontology or any RDF file and during the night someone nasty has changed all the names but has neither deleted nor added anything. This new RDF file is isomorphic to the old one.

The problem of finding isomorphism between graphs unfortunately is in NP. There can't be an algorithm, which answers the question whether there is an isomorphism in time (number of calculation steps) related by a polynomial function to the size of the input for the general case.

Consider the 2 example graphs in Fig. 4.1. Since they have a different number of nodes there can't be a complete bijective mapping but a partial one. This is called subgraph isomorphism.

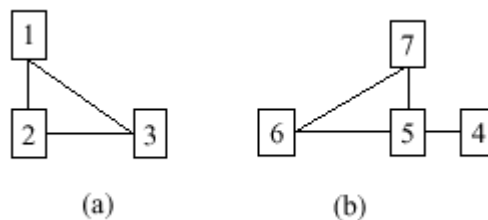


Fig. 4.1: 2 graphs.

The two graphs contain very few nodes. But the search space is already not that small. All possible maps can be arranged in the tree shown in Fig. 4.2.

Due to the size of the search space it is important to find ways to restrict it. In the following section I will introduce two basic algorithms for finding graph isomorphism. In section 4.3 I will discuss how the search for isomorphism exactly fits into finding candidates for homonyms, synonyms and similar subgraphs. I will propose ways to limit the search space for this application of the general problem.

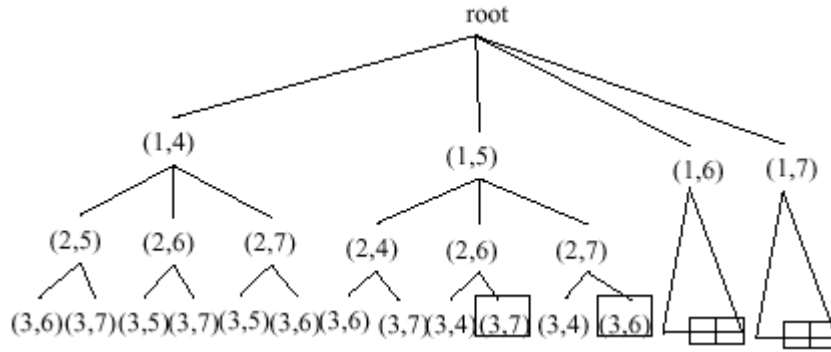


Fig.4.2: Possible mappings.

4.2 Algorithms

Two basic algorithms to find graph isomorphism are Ullmann's Algorithm [ULL76] and A* [NILS80]. They try to efficiently search and prune the search tree, Fig.4.2.1.

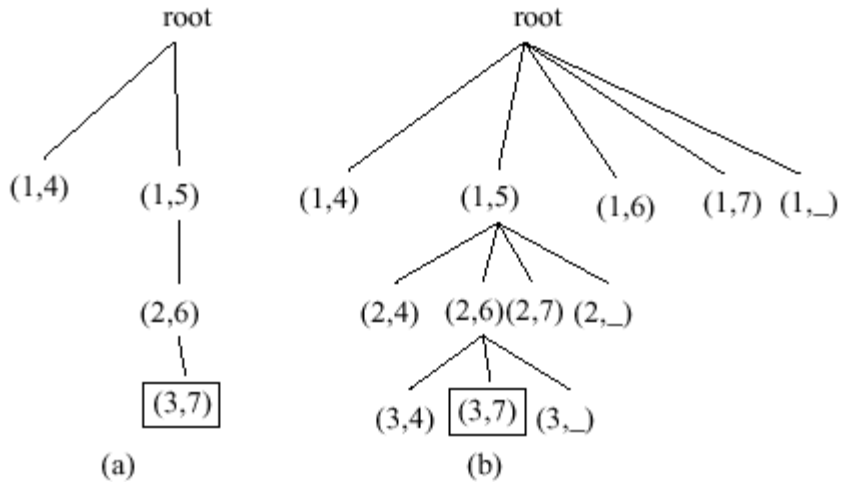


Fig.4.2.1: Ullman vs. A*.

Ullmann's Algorithm searches the tree in depth-first order. As soon as isomorphism condition does not hold in a branch that branch is cut of.

A* searches the tree in breadth-first order. Looking at all branches it decides which node should be expanded. The decision is based on a cost function, which determines a cost for each matching and estimates a cost for matching the unmatched nodes. That is were heuristics about the graphs come into play. A* also provides for inexact matches. Nodes labeled with (X,_) stand for not matching X to anything.

If inexact matches are considered searching in depth-first order is bears the danger of searching branches of infinite length if there is nothing to stop it.

4.3 Approach for RDFBrowser

What special properties might RDF graphs have that can be useful in limiting the search space?

Before even considering that the application in a browser has to be taken into account. The user might not want to start the search and come back 2 days later for results. Therefore he should be allowed to limit the search by determining the size of the subgraph the program should try to find a mapping for. Using RDFBrowser he might do that specifying a resource in one file and a depth until which adjacent nodes will be considered. This fits perfectly into the task of finding synonyms and homonyms.

A nice property of RDF graphs is that only adjacent nodes need to be considered for this task since resources are described by URIs and thus any statement about the same resource will be adjacent in the graph.

URIs also provides that nodes with the same URI are mapped onto each other between different graphs.

This task is not about finding one perfect match. Finding inexact matches of subgraphs is important because ontologies written by humans will either be known to be the same or won't be the same. It might even be interesting to find isomorphism among metadata which isn't about the same domain. Then there are few commonly used URIs.

Considering these properties of the task an algorithm for finding a subgraph isomorphism of specified maximal size m and specified precision p could look as follows:

Input: A query subgraph q and a graph g to which an isomorphism is to be found, m , p .

- Put the nodes in q in list N ;
- In N find the nodes N_{URI} whose labels (URIs) also occur in g ;
- In N_{URI} find all nodes N_{URI+} with edges that occur in q and g and have the same label;
- Order the nodes N : 1st N_{URI+} , 2nd N_{URI} , then the rest;
- Search the space in depth-first order with the maximum depth m :
 - Try to find matches for the nodes in N (in the obtained order!);
 - If a node n can't be matched match $(n, _)$;
 - If there are more unmatched nodes among the ancestors than precision p allows prune.

If the query graph is specified by giving a node and a depth, to which adjacent nodes are to be considered, the given node is in the center of that graph.

If the algorithm returns an isomorphism with reasonable precision the node is a candidate for a synonym, if it isn't mapped to the same URI.

If not (even with very low precision), and the node occurs in both files it is a candidate for a homonym.

5 Conclusion

This paper has introduced some basic and advanced features a tool for reviewing or editing ontologies and RDF data might have. They seem to be very useful, because they are answering inevitable questions about ontologies. The usefulness of other tools might be measured by questioning whether they can answer those questions.

6 Installation

The Java Runtime Environment (JRE) 1.3.1 or newer needs to be installed on your computer. Hint: Any JDK contains a JRE.

Download RDFBrowser_Install.zip from

<http://www.arches.uga.edu/~bernie42/SemWeb/>. If that's not available contact me.

Unzip that file.

The following files need to be in you PATH:

Windows: amzi.dll, amzijni.dll. If you need help to include them in your path, look for "path command" in the Windows help. Hint: You could put the files in any directory, which already is in your path.

Linux: libamzi.so, libamzijni.so.

The following files need to be in the CLASSPATH (the JRE needs to access them):

amzi.jar, arp.jar, xerces.jar. Instead of amzi.jar you can also include the folder "amzi" in the CLASSPATH.

The easiest way to do so is to put the .jar-files in the directory

...jre\lib\ext\ of your JRE installation.

Further information on CLASSPATH:

<http://java.sun.com/j2se/1.3/docs/tooldocs/win32/classpath.html>.

Files for RDFBrowser:

GUI.class, RDFBrowser.class, FileBrowser.class, TripleTableModel.class and triple.xpl have to be in the same directory and in the CLASSPATH.

Hint: The current directory (in the command prompt) is automatically included in the CLASSPATH.

Run: Type "java GUI" at the command prompt. Notice: Therefore the java interpreter (Windows: jre/bin/java.exe) has to be in your PATH, else you have to call "...jre\bin\java GUI".

If this installation for some reason doesn't work (I didn't test the Linux version), you can obtain the third-party software at the following sites:

<http://www.hpl.hp.co.uk/people/jjc/arp/download.html>

<http://xml.apache.org/xerces-j/>

<http://www.amzi.com/download/freedist.htm>

Copyright: For copyright information about Xerces and ARP see "Xerces_LICENSE.txt" and "ARP_LICENSE.txt". amzi.dll, amzijni.dll, libamzi.so, libamzijni.so are parts of Amzi! Prolog + Logic Server, copyright 1994-2000 Amzi! Inc..

Resources

[NILS80]

N. Nilson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[OE]

Ontoedit. Copyright Ontoprise GmbH. <http://www.ontoprise.de> .

[PROT00]

Protégé-2000. Developed by Stanford Medical Informatics. Copyright 1998-2001, Stanford University. <http://protégé.stanford.edu> .

[RDF99]

W3C. *Resource Description Framework (RDF): Model and Syntax Specification*. <http://www.w3.org/TR/REC-rdf-syntax/>, 1999.

[RDFS02]

W3C. *RDF Vocabulary Description Language 1.0: RDF Schema*. <http://www.w3.org/TR/rdf-schema/>, Draft, April 2002.

[ULL76]

J. Ullman. *An Algorithm for subgraph isomorphism*. Journal of the ACM, 23:31-42, 1976.