Similarity-based Browsing over Linked Open Data

Michael Hickson[1], Yannis Kargakis[1] and Yannis Tzitzikas[1,2]

[1] Department of Computer Science, University of Crete, Greece
[2] Institute of Computer Science, FORTH-ICS, Greece
Email: {hickson,kargakis,tzitzik}@csd.uoc.gr

Abstract. An increasing amount of data is published on the Web according to the Linked Open Data (LOD) principles. End users would like to browse these data in a flexible manner. In this paper we focus on similarity-based browsing and we introduce a novel method for computing the similarity between two entities of a given RDF/S graph. The distinctive characteristics of the proposed metric is that it is generic (it can be used to compare nodes of any kind), it takes into account the neighborhoods of the nodes, and it is configurable (with respect to the accuracy vs computational complexity tradeoff). We demonstrate the behavior of the metric using examples from an application over LOD. Finally, we generalize and elaborate on implementation approaches harmonized with the distributed nature of LOD which can be used for computing the most similar entities using neighborhood-based similarity metrics.

1 Introduction

The last years a vast amount of structured data has been published as *Linked Open Data (LOD)*. However, in their current form, they cannot be directly exploited by end users, since better linking, browsing, presentation is required (*interaction* and *interfaces* is one of the main research challenges of LOD according to [4]). Our objective is to investigate generic methods for browsing and exploring such data sets. Context and motivation for our work was the design and development of an online movie exploration system based on Semantic Web technologies, whose data are fetched dynamically from the LOD cloud, and offers *similarity-based browsing* for bypassing the need for query formulation by end users.

In this paper, we motivate the need for similarity-based browsing, we identify related requirements, and we introduce a new similarity function for tackling them. In brief the proposed similarity between two RDF nodes is actually the Jaccard similarity coefficient evaluated over the nodes of the extended (radius bounded) neighborhoods (containing both instance and schema nodes) of the compared nodes. A distinctive characteristic of this metric is that each node that participates to an intersection or union operation of the Jaccard similarity coefficient, is weighted by a value based on its path distance from the compared nodes, for promoting close matches over distant ones. In a nutshell, the distinctive characteristics of the proposed similarity metric is that: (a) it is type independent (it can compute similarity between any pair of resources), (b) it can be applied within a single KB (thus different from the methods which have been proposed for ontology matching), and (c) it offers to the designer (or end user) the flexibility to choose the appropriate depth depending on his needs (on accuracy or computational complexity). Subsequently, we describe implementation approaches for computing the most similar entities and we analyze implementation approaches which are harmonized with the distributed nature of LOD. In particular we show how a similarity function can be *reversed* for enabling the computation of similar pages over the LOD without having to access the entire corpus. Such methods can be used not only for the introduced similarity metric, but for neighborhood-based similarity metrics in general.

The rest of this paper is organized as follows. Section 2 describes the motivation and application context of our work. Section 3 discusses related works. Section 4 introduces the least number of symbols and notations required for defining the similarity function. Section 5 introduces the similarity function and Section 6 demonstrates its merits over the running example. Section 7 discusses implementation approaches and shows how a similarity function can be *reversed*. Finally, Section 8 concludes the paper and identifies issues for further research.

2 Application Context

The context of our work is an application over the *Linked Open Data (LOD)* cloud. Our objective was to design and develop a system which allows the flexible exploration of *movie* information, based on information fetched from the LOD cloud. The distinctive characteristics of this system, called `MovieSim`, are:

- All information is fetched from the LOD cloud. This not only automates information updating, but enables the application to provide always up-to-date information.
- It links the available in the LOD structured information, and enriches it with links to external information (plain Web pages).

Specifically from LinkedMDB[3] the data are fetched in RDF format, from its available SPARQL Endpoint, while from Freebase[4] data cloud the data is fetched in JSON format through its provided API. Regarding the linking of the data extracted from each source we did not face any difficulty, since LinkedMDB provides for each of its entities a Unique Identifier, through FreeBase's link, that represents it in Freebase's data cloud.

Since most end users do not have the technical knowledge (or the willingness) to formulate explicit SPARQL queries, `MovieSim` provides a more user friendly interaction, namely (a) keyword-based retrieval and (b) similarity-based browsing.

To support *keyword-based retrieval* `MovieSim` periodically fetches information from LinkedMdb and indexes it with the help of LARQ (Lucene+ARQ)[5]. The

[3] http://www.linkedmdb.org/
[4] http://www.freebase.com/
[5] http://jena.sourceforge.net/ARQ/lucene-arq.html

availability of an index makes the evaluation of keyword queries very fast. We will not describe this functionality in detail since keyword searching over structured data is not the focus of this paper.

Similarity-based browsing aims at allowing users to explore the available information without having to formulate structured queries. Note that similarity-based browsing is mainly offered for browsing image and video databases (e.g. [5]), but (to the best of our knowledge) has not been applied over RDF data.

Regarding the presentation of information, `MovieSim` supports various kinds of Web pages, each one having a different role. Keyword search is supported through a search box, while the results of the query are viewed by a different kind of page. The essential category of pages contains page types for showing information about:

– actors,
– directors,
– editors,
– movies, and
– writers.

Each page type presents information which is dynamically fetched and linked. In addition, the system provides a general purpose page type to show information about entity types that do not fall in one of the previous categories. Below we present the information that we fetch for each supported type, from each individual source.

Movie	
attribute	source
Title	LinkedMDB
Runtime	LinkedMDB
Initial Release Date	LinkedMDB
Movie Actors	LinkedMDB
Movie Writers	LinkedMDB
Movie Directors	LinkedMDB
Movie Editors	LinkedMDB
Image	Freebase
Abstract	Freebase
Rating	Freebase
Tagline	Freebase
Genres	Freebase

Actor	
attribute	source
Actor Name	LinkedMDB
Films Acted	LinkedMDB
Image	Freebase
Abstract	Freebase
Birth Date	Freebase
Birth Place	Freebase
Nationality	Freebase

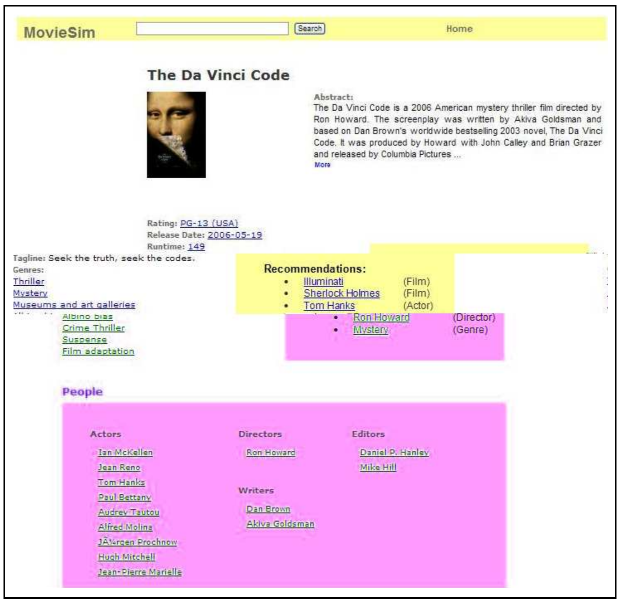
Director	
Director Name	LinkedMDB
Films Directed	LinkedMDB
Image	Freebase
Abstract	Freebase
Birth Date	Freebase
Birth Place	Freebase
Nationality	Freebase

Writer	
Writer Name	LinkedMDB
Films Writen	LinkedMDB
Image	Freebase
Abstract	Freebase
Birth Date	Freebase
Birth Place	Freebase
Nationality	Freebase

Editor	
Editor Name	LinkedMDB
Films Edited	LinkedMDB
Image	Freebase
Abstract	Freebase
Birth Date	Freebase
Birth Place	Freebase
Nationality	Freebase

General	
Title	LinkedMDB
Inbound Links	LinkedMDB
Outbound Links	LinkedMDB
Image	Freebase
Abstract	Freebase

While the user views the page of one entity he can continue browsing and exploring *similar entities*. The similar entities are computed using the similarity function that we will describe later on. Since the similar entities can be numerous and of different types, only the entities with the highest similarity should be suggested. Figure 1 shows a screenshot of the Web page produced for the movie `Da Vinci Code`.



MovieSim Search Home

The Da Vinci Code

Abstract:
The Da Vinci Code is a 2006 American mystery thriller film directed by Ron Howard. The screenplay was written by Akiva Goldsman and based on Dan Brown's worldwide bestselling 2003 novel, The Da Vinci Code. It was produced by Howard with John Calley and Brian Grazer and released by Columbia Pictures ...
More

Rating: PG-13 (USA)
Release Date: 2006-05-19
Runtime: 149
Tagline: Seek the truth, seek the codes.
Genres:
Thriller
Mystery
Museums and art galleries
Albino bias
Crime Thriller
Suspense
Film adaptation

Recommendations:
- Illuminati (Film)
- Sherlock Holmes (Film)
- Tom Hanks (Actor)
- Ron Howard (Director)
- Mystery (Genre)

People

Actors	Directors	Editors
Ian McKellen	Ron Howard	Daniel P. Hanley
Jean Reno		Mike Hill
Tom Hanks	**Writers**	
Paul Bettany	Dan Brown	
Audrey Tautou	Akiva Goldsman	
Alfred Molina		
JÃ¼rgen Prochnow		
Hugh Mitchell		
Jean-Pierre Marielle		

Fig. 1. Movie Page

Note that similarity-based browsing is actually an alternative (essentially complementary) approach to the *facet-based browsing* [26], which is supported by systems like: BrowseRdf [24], Humboldt, VisiNav [12], Longwell [25], Ontogator [20], /facet [14], Camelis2 [11]. Facet-based browsing also bypasses the query formulation effort. However, similarity-based browsing does not require from the user to select the relationship through which two entities are related. Instead, the similarity value actually quantifies several relationships (direct or path based) and offers an aggregated form of relevance.

Similarity-based browsing can actually be offered in the context of a facet-based browsing system. Specifically, a new facet can be defined which shows the most similar entities.

Figure 2 sketches the architecture of `MovieSim`. Its architecture is based on the *MVC* (Model View Controller) pattern, meaning that all business logic is implemented in Servlets and all communication and data transfer issues are dealt with the use of Java Beans (one for each entity type mentioned earlier). The presentation of data (page types) is specified using JSP pages in order to separate the presentation design from the application logic, making easier the extension and modification of the system.

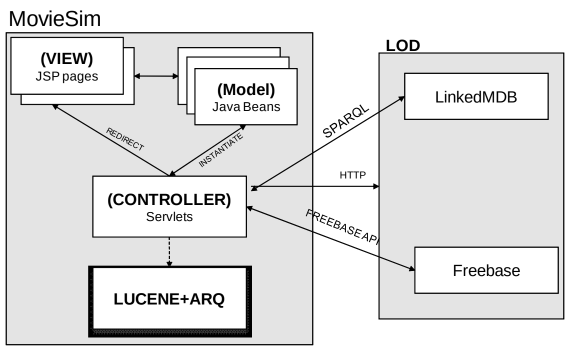


Fig. 2. The Architecture of `MovieSim`

3 Related Work on Similarity over RDF/S

Since we focus on similarity-based browsing, in this section we briefly review the related work that has been done. In general, with the rapid development of the Semantic Web, there has been an increased interest in developing methods for finding similarities between nodes in RDF/S graphs. There are several related works mainly for the problem of *ontology matching*. Below we list and comment in brief the more related works.

[29] presents a method for computing the similarity between two entities coming from two different OWL DL ontologies. The computation of similarity is based on the extraction of information encoded in each entity's description. The extracted components are then compared, taking into account the predefined meanings of OWL DL and RDF(S) primitives, to produce partial component similarity values, which are then combined using predefined weights under a variable weighting scheme.

[7] also proposes a similarity function for entity matching between different OWL ontologies .

There are also algorithms (again for the problem of ontology matching) which use the *edit* distance to find the lexical similarity between two entities, such as the MLMA+ algorithm [2] which, amongst other measures, makes use of the Levenshtein (Edit) distance [19].

Another algorithm (for ontology matching) is presented in [1] for finding similarities between two entities, of some given ontologies based on the combination of structural and lexical information provided by the ontology, which is divided into three stages. In the first stage each entity is lexically analyzed, based on information given from their labels and descriptions. The second stage involves the comparison of the entities based on the structure of the graph, while the third stage combines the results of the two previous stages and produces a final result that represents the similarity between the two entities.

Another related work aiming at identifying cases where the same objects are identified by different URIs in different datasets, in the context of LOD, is [22].

Finally, [27] proposes a metric for entity comparison in hierarchical ontologies (however that work exploits only hierarchical relationships and ignores properties).

Similar in spirit problem is that of *blank node matching* which aims at defining a mapping between the blank nodes of two KBs (related works include PromptDiff [23], Ontoview [18], CWM [3], RDFSync [30]).

To synopsize, most of the related works aim at finding similarities between entities of *different* knowledge bases. Therefore they mainly identify similarities between entities of the *same type*. Such approaches would not be convenient for our system, since we would have to design several class-specific similarity functions, i.e. similarity functions between movies and actors, directors and actors, writers and movies, and so on. For this reason, we decided to move towards a similarity computation method that is *type-independent* allowing the comparison of entities of the same or different types. At last we should note that the similarity function that we needed for our system, apart from being type-independent should exploit both the instance and the schema layer (for being able to compute similarities between entities which do not belong to the same classes).

4 Background (RDF definitions and notations)

An RDF Knowledge Base (KB) is defined as a set of RDF triples, denoted by K, each having the form (subject, predicate, object), for short (s, p, o). A KB K can also be viewed as a directed labeled graph $G = (N, E)$. The nodes of the graph are the URIs, the literals and the blank nodes that appear in the triples of K, while the edges of the graph are labeled arcs that connect the corresponding nodes.

We shall use as running example the KB that is illustrated at Figure 3. For the sake of completeness, even if the LOD dataset did not have an explicitly defined schema, we have created one (for capturing the general case of RDF/S KBs). Furthermore, we added some extra entities [6] apart from those fetched from LOD.

All resources which are instances of a class are vertically aligned with the class. Below we introduce some notations which are necessary for defining the similarity metric.

We shall use Pr to refer to the properties that occur in K. For a given resource u we shall use $ResFrom(u)$ (resp. $ResTo(u)$) to denote the resources which are pointed to by (resp. point to) resource u, i.e.

$$\begin{aligned} ResFrom(u) &= \{\ o \mid (u, p, o) \in K, p \in Pr\} \\ ResTo(u) &= \{\ o \mid (o, p, u) \in K, p \in Pr\} \end{aligned}$$

In our running example we have:
$ResFrom(\,SherlockHolmes) = \{England, GuyRitchie, JudeLaw, Mystery, SherlockHolmesBook\}$.

We define the classes and the superclasses of a resource u as:

$$\begin{aligned} Classes(u) &= \{\ c \mid (u, type, c) \in K\} \\ SuperClasses(u) &= \{\ c \mid (u, subClassOf, c) \in K\} \end{aligned}$$

[6] Specifically `DaVinciCode Book`, `Illuminati Book`, `Sherlock Holmes Book`, `Dan Brown`, and `Conan Doyele`.

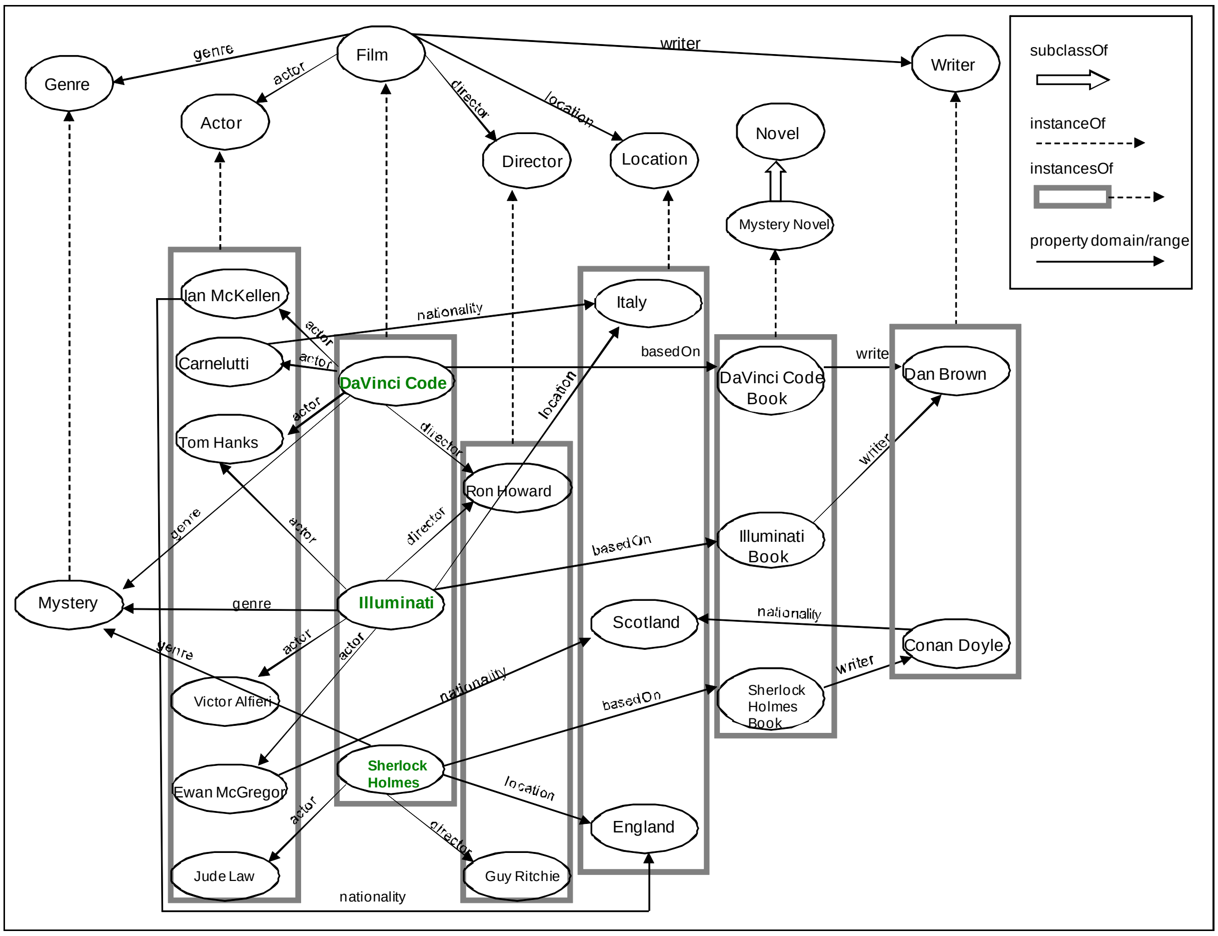


Fig. 3. The RDF graph G of our running example

For example in Figure 3 we have:
$Classes(IlluminatiBook) = \{MysteryNovel\}$ while $SuperClasses(MysteryNovel) = \{Novel\}$. Obviously if an element x is a class then, $Classes(x) = \emptyset$, while if x is an instance of a class then $superClasses(x) = \emptyset$.

Some notations for edges follow. We define the set of classification and inheritance links of a resource u and a class c as:

$$ClassLinks(u) = \{\ (u, c)\ |\ (u, type, c) \in K\}$$
$$SupLinks(c) = \{\ (c, c')|(c, subClassOf, c') \in K\}$$

The inbound and outbound property links of a resource u are defined as:

$$PropsFromLinks(u) = \{\ (u, o)\ |(u, p, o) \in K, p \in Pr\}$$
$$PropsToLinks(u) = \{\ (o, u)\ |(o, p, u) \in K, p \in Pr\}$$

Now we extend the above definitions to take as parameter a set (S) of resources, so we have:

$$\begin{aligned}
ResFrom(S) &= \cup_{u \in S} ResFrom(u)\\
ResTo(S) &= \cup_{u \in S} ResTo(u)\\
PropsFromLinks(S) &= \cup_{u \in S} PropsFromLinks(u)\\
PropsToLinks(S) &= \cup_{u \in S} PropsToLinks(u)\\
Classes(S) &= \cup_{u \in S} Classes(u)\\
SuperClasses(S) &= \cup_{u \in S} SuperClasses(u)\\
ClassLinks(S) &= \cup_{u \in S} ClassLinks(u)\\
SupLinks(S) &= \cup_{u \in S} SupLinks(u)
\end{aligned}$$

A *path over* G, is any sequence of edges of the form: $(A, P, C), (C, P', D), \cdots, (E, P'', u)$, where all predicates $(P, P', ..P'')$ are either properties in Pr or the predicate `type` or the predicate `subClassOf`.

We define the *distance* between two nodes A and B over G, denoted by $dist_G(A, B)$, as the length of the *shortest* path from A to B. If no path exists then the distance is assumed to be infinite.

5 Similarity Function

In this section, we will introduce and analyze, step by step, the proposed similarity metric, over the running example of Fig. 3. Suppose we want to compute the similarity between two nodes A and B of the RDF graph G. At first we define the subgraphs of A and B of *radius* k, denoted by:

$$\begin{aligned}
g_{A(k)} &= (N_k(A), E_k(A))\\
g_{B(k)} &= (N_k(B), E_k(B))
\end{aligned}$$

They consist of all nodes and edges that are visited if we start from A and B respectively, and traverse all links (properties, type, subclassOf) for depth up to k where the value of k is configured externally (and it will be discussed later on).

These graphs can be computed in an iterative manner. For instance, for defining $g_{A(k)}$ we start from $g_{A(0)} = (N_0(A), E_0(A))$ where $N_0(A) = \{A\}$ and $E_0(A) = \emptyset$. Subsequently, from
$g_{A(i-1)} = (N_{i-1}(A), E_{i-1}(A))$ we can compute
$g_{A(i)} = (N_i(A), E_i(A))$ (for all $1 \leq i \leq k-1$), as follows:

$$\begin{aligned}
N_i(A) = &\ N_{i-1}(A) \cup\\
&\ ResFrom(N_{i-1}(A)) \cup\\
&\ Classes(N_{i-1}(A)) \cup\\
&\ SuperClasses(N_{i-1}(A))\\
E_i(A) = &\ E_{i-1}(A) \cup\\
&\ PropsFromLinks(N_{i-1}(A)) \cup\\
&\ ClassLinks(N_{i-1}(A)) \cup\\
&\ SupLinks(N_{i-1}(A))
\end{aligned}$$

Each step of the iteration enriches the current set of nodes $N_{i-1}(A)$ with the nodes:

– which are classes of a node in $N_{i-1}(A)$ (since classes carry important information),
– the values of the properties that start from the nodes in $N_{i-1}(A)$ (they are actually attribute values),
– the superclasses of the nodes in $N_{i-1}(A)$ (for climbing up the subClassOf hierarchy)

The iterative expansion allows collecting values of complex attributes, as well as higher level superclasses (in this way we can detect similarities even between very "distant" entities which belong to different class hierarchies).

We should stress at this point, that one could adopt a different policy regarding how a subgraph expands. For instance, one could also expand the graph using properties which *point to* the current set of nodes (in that case $ResTo(N_{i-1}(A))$ would be added to $N_i(A)$ and $PropsToLinks(N_{i-1}(A))$ to $E_i(A)$). The decision is application or ontology specific. [16,17] have also made the observation that it is often not enough to use a single similarity measure to achieve good results, therefore a combination of features needs to be engineered or even learned. In our case we decided to take only the forward property direction since in most cases a property is more important for its origin than for its destination.

To better illustrate the construction of the subgraph, consider the graph G of Figure 3 and suppose that A = `DaVinci Code` and B = `Illuminati`. The subgraphs $g_{A(3)}$ and $g_{B(3)}$ are shown at Figure 4 and Figure 5 respectively (the latter depicts all subgraphs for $k = 0$ to $k = 3$).

Table 1 shows the distances $dist_{g_A}(A, u)$ and $dist_{g_B}(B, u)$ for various u nodes. The nodes for which both $dist_{g_A}(A, u)$ and $dist_{g_B}(B, u)$ are defined (i.e. both are different than ∞), actually belong to the intersection of the nodes of the two subgraphs, while the rest are nodes that belong only to one of the subgraphs.

After having constructed the graphs g_A and g_B , one could compute the similarity between A and B by applying the Jaccard similarity coefficient [15] over their node sets, i.e. between $N(A)$ and $N(B)$, as follows:

$$sim_k(A, B) = \frac{|N_k(A) \cap N_k(B)|}{|N_k(A) \cup N_k(B)|} \qquad (1)$$

In our example the intersection between $N_3(A)$ and $N_3(B)$ is illustrated (vertically aligned) at the center of Figure 6 where for reasons of space we do not show the schema level intersections.

Note that by considering the nodes at depth greater than 1, we can identify similarities between resources of different types. If resources of different types are compared (e.g. a film with an actor), they will rarely have the same properties in small depth (e.g. for $k = 1$) and therefore we will not get many (or any) intersecting nodes.

Obviously the similarity value obtained depends on the value of k. For example, for $k = 1$ we get:

$$sim_1(DaVinciCode, Illuminati) = \frac{4}{15} = 0.26$$

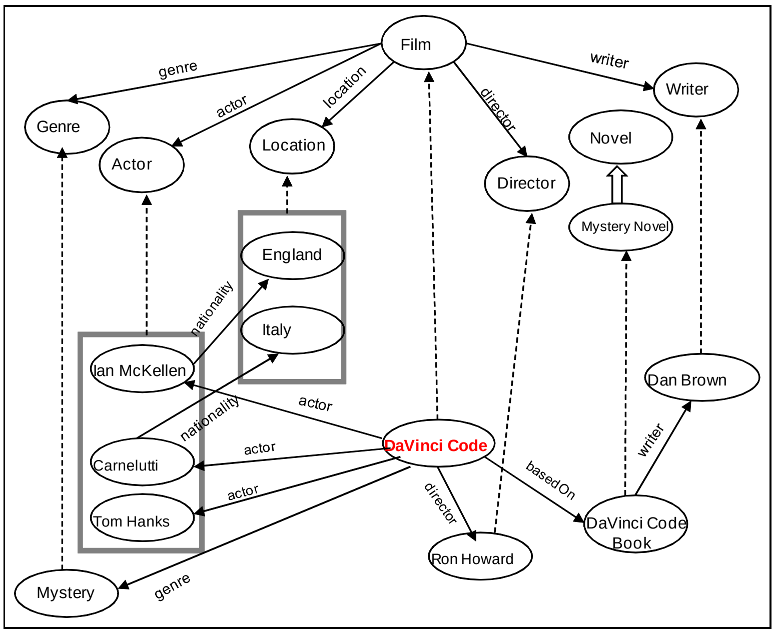


Fig. 4. $g_{A(3)}$ where $A =$ `DaVinci Code`

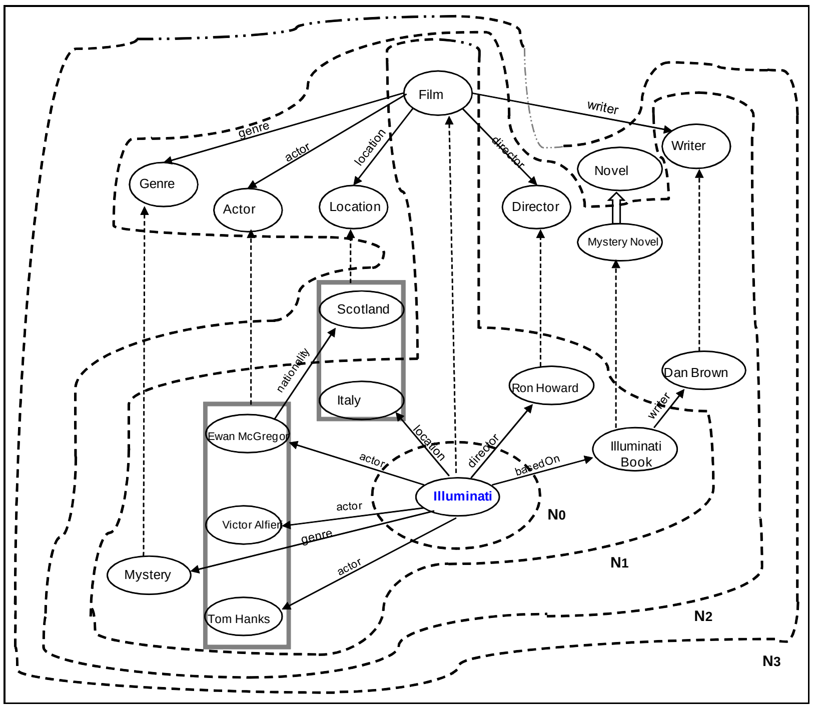


Fig. 5. $g_{B(3)}$ where $B =$`Illuminati`

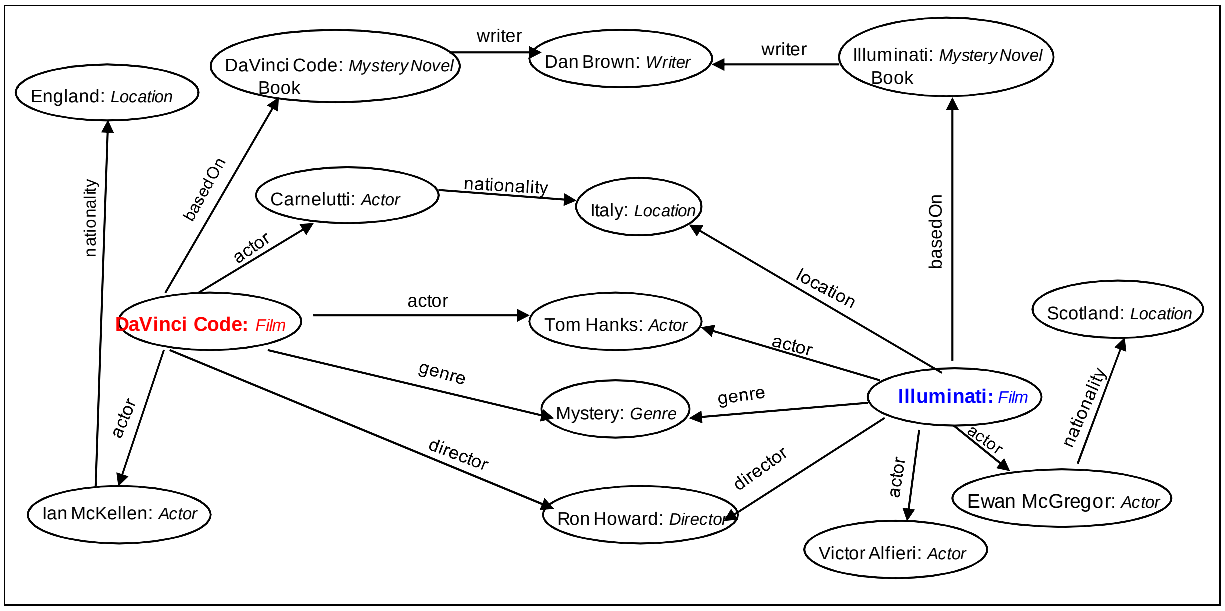


Fig. 6. Intersection between Illuminati and DaVinci Code Subgraphs

while for $k = 3$ we get

$$sim_3(DaVinciCode, Illuminati) = \frac{13}{21} = 0.61$$

However a shortcoming of this approach, is that a common node spotted at depth 1, is equally weighted as a common node of a larger distance. For this reason below we introduce a different similarity function which takes into account the values $dist_{g_A}(A, u)$ and $dist_{g_B}(B, u)$. We should clarify that this extension does not increase the computational cost of the similarity function since these distances are computed anyway during the construction of the subgraphs $g_{A(k)}$ and $g_{B(k)}$.

To understand the extension we shall first express function (1) in a different, but equivalent, manner:

$$sim_k(A, B) = \frac{\sum_{n \in (N_k(A) \cap N_k(B))} 1}{\sum_{n \in (N_k(A) \cup N_k(A))} 1} \tag{2}$$

This form makes evident that each element in the intersection or union contributes the value of one. Now we will introduce the new formula in which each element in the intersection or union does not contribute the value of one, but a value based on its average distance from nodes A and B.

Since the closest node is at distance 1 while the most distant is at distance k (or infinite) we shall use the expression $k+1-dist$ for giving to the closest nodes a contribution equal to k and to the more distant nodes a contribution equal to 1. If a distance equals ∞ we consider it as $k+1$. In this way the expression $k+1-dist$ yields a zero[7].

[7] This means that the cells of Table 1 that have an infinite value (∞) are actually considered to have the value $k + 1$, i.e. 4.

u	$dist_{g_A}(A,u)$	$dist_{g_B}(B,u)$
Genre	2	2
Actor	2	2
Film	1	1
Director	2	2
Location	2	2
Novel	3	3
Mystery Novel	2	2
Writer	2	2
Mystery	1	1
Ian McKellen	1	∞
Carnelutti	1	∞
Tom Hanks	1	1
Victor Alfiery	∞	1
Ewan McGregor	∞	1
Ron Howard	1	1
Italy	2	1
Scotland	∞	2
England	2	∞
DaVinci Code Book	1	∞
Illuminati Book	∞	1
Dan Brown	2	2

Table 1. Distances from A and from B

The proposed similarity function is defined as: $sim_k(A,B) =$

$$\frac{\sum_{n\in(N_k(A)\cap N_k(B))} \frac{(k'-dist_{g_A}(A,n))+(k'-dist_{g_B}(B,n))}{2}}{\sum_{n\in(N_k(A)\cup N_k(B)} \frac{(k'-dist_{g_A}(A,n))+(k'-dist_{g_B}(B,n))}{2}} \tag{3}$$

where $k' = k+1$.

If we apply (3) to our running example we now get:

$$sim_3(DaVinciCode, Illuminati) = \frac{29.5}{42} = 0.7$$

In brief, the proposed similarity between two nodes A and B is actually the Jaccard similarity coefficient evaluated over the nodes of the extended neighborhoods of the compared nodes. Each node of the neighborhoods is weighted so that the nodes closer to the compared nodes get a greater weight than the distant ones.

5.1 Properties of the Similarity Function

For any resource u, and for any positive integer k it holds: $sim_k(u,u) = 1$.

It is also clear that the metric is symmetric i.e. $sim_k(a,b) = sim_k(b,a)$.

Although in the examples that we have seen earlier it happens to hold: if $m > m'$ then $sim_m(a,b) \geq sim_{m'}(a,b)$, in the general case this does not hold. The reason is that for a high k we may have several non intersecting sets of nodes which increase the denominator of the similarity function.

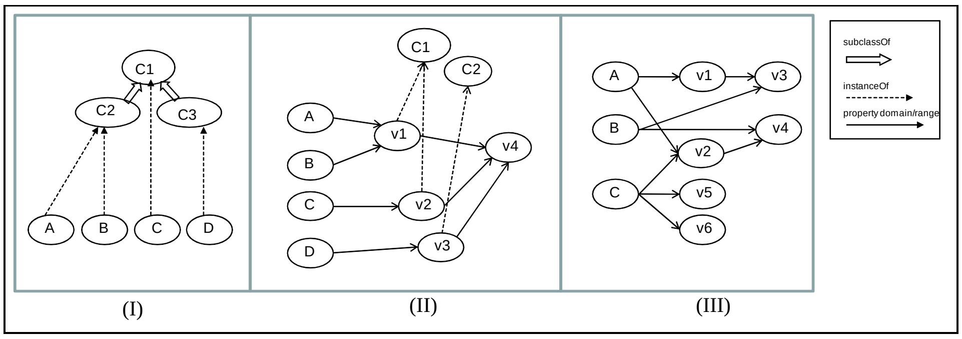


Fig. 7. Three examples

6 Examples and Analysis

6.1 Behavior

Table 2 shows the computed similarities between the films `DaVinci Code`, `Illuminati` and `Sherlock Holmes`, for $k = 1, 2, 3$. We observe that the most similar movie with `DaVinci Code`, is `Illuminati` (and not `Sherlock Holmes`) for all values of k from 1 to 3.

k	sim_k(DaVinciCode, Illuminati)	sim_k(DaVinciCode, SherlockHolmes)
1	0.53	0.30
2	0.67	0.54
3	0.70	0.58

Table 2. Similarity for different values of k

Let us now use some examples to justify the benefits of k values higher than 1, and to better understand the behavior of the similarity function. Table 3 shows the computed similarities between the nodes `A`, `B`, `C` and `D`, for $k = 1 \ldots 3$, for the example shown at Figure 7(I). We observe that for $k = 1$, `B` is the most similar to `A` since they are under the same class, while the similarity of `A` with `C` and `D` is zero. However for $k = 2$ the similarity of `A` with `C` and `D` is not zero, and `C` is more similar than `D`.

To demonstrate the potential of the similarity function to exploit commonalities in property paths, Table 4 shows the computed similarities between the nodes `A`, `B`, `C` and `D`, for $k = 1 \ldots 3$, for the example shown at Figure 7(II). We observe that for $k = 2$ `A` is more similar to `C` than to `D` because even though they do not have any direct value in common, `v1` and `v2` are under the same class `C1`, and `v4` is a common value at depth 2. Notice that the similarity between `A` and `D` is not zero for $k = 2$, due to the value $v4$.

k	sim_k(A, B)	sim_k(A,C)	sim_k(A,D)
1	1	0	0
2	1	0.60	0.33
3	1	0.625	0.40

Table 3. Similarity for different values of k over Fig. 7(I)

k	sim_k(A, B)	sim_k(A,C)	sim_k(A,D)
1	1	0	0
2	1	0.50	0.25
3	1	0.57	0.28

Table 4. Similarity for different values of k over Fig. 7(II)

It is also worth noting that the most similar entity can change as k changes. For instance, in the example of Figure 7(III), as we can see from Table 5, for $k = 1$ the most similar to A is the entity C, while for $k = 2$ (and higher) the most similar to A is the entity B.

k	sim_k(A, B)	sim_k(A,C)
0	0	0
1	0	0.40
2	0.60	0.46
3	0.625	0.47

Table 5. Similarity for different values of k over Figure 7(III)

6.2 Computational Complexity

Let d be the average number of edges which are adjacent to a node. For a node A, the number of nodes in the graph $g_{A(k)}$ is at most in $\mathcal{O}(d^k)$. This is therefore the cost of $sim_k(\cdot, \cdot)$.

6.3 On Selecting a value for k

One issue that plays an important role in the computation of similarity is the choice of the appropriate k. The choice can be made by the application designer (or even by the end user at run-time). By choosing a greater k more complexity is added to the computation of the similarity and this is the cost to pay for more accurate results in the sense that a wider part of the graph is taken into account. By choosing a lower k the computational cost gets decreased, but the results may not be as accurate as the user would like.

One method for selecting a k is to measure graph features of the RDF/S graph, e.g. the diameter of the graph.

6.4 Variations of the Similarity Function

As one may have noticed, the similarity function ignores the names of the properties.

The benefit of this choice is that the function can yield positive similarities also between objects that use different properties. For example consider the triples `(a, hasFriend, e)` and `(b, worksFor, e)`. The similarity function will return a positive value for $sim_1(a,b)$ although these entities have different properties. It would be zero if the property names were taken into account. However, the shortcoming is inability to promote matches also at the properties. For example, if we had another triple `(c, hasFriend, e)` then we would have $sim_1(a,c) = sim_1(a,b)$, although we would prefer $sim_1(a,c) > sim_1(a,b)$.

If we wanted to take into account the property names then we could *prefix* the nodes of the subgraphs which are reached from properties by the corresponding property name. In particular, instead of
$ResFrom(u) = \{\ o \mid (u,p,o) \in K, p \in Pr\}$, we could define
$ResFrom'(u) = \{\ p:o \mid (u,p,o) \in K, p \in Pr\}$,
where "$p : o$" is treated as one string. Clearly, with such a change, the new similarity function, denoted by sim', would yield $sim'_1(a,c) > sim'_1(a,b) = 0$.

One approach to reconcile the two approaches is to change the graph expansion step so that both $ResFrom(u)$ and $ResFrom'(u)$ are used for the definition of the nodes of the subgraphs. Specifically $N_i(A)$ can now be defined as:

$$\begin{aligned} N_i(A) = N_{i-1}(A) \cup & \\ & ResFrom(N_{i-1}(A)) \cup \\ & ResFrom'(N_{i-1}(A)) \cup \\ & Classes(N_{i-1}(A)) \cup \\ & SuperClasses(N_{i-1}(A)) \end{aligned}$$

In this way we will get $sim''_1(a,c) > sim''_1(a,b) > 0$.

6.5 Experimental Results

We created a bigger KB for testing the similarity function, i.e. for judging whether it returns intuitive results and for investigating how the value of k affects the results.
[*Setup of the KB*]
Our measurements were based on a KB that we created by extracting data from LinkedMDB, through Virtuoso's SPARQL Endpoint, with explicit queries. More specifically, we selected and downloaded 10 entities, that were quite relevant to each other. For each one of them we expanded their subgraphs for depth 3, and with the fetched information we created a KB on which our measurements were conducted. The entities that were chosen and their types are shown in Table 6.

The resulting KB contained: 16 classes, 70 properties, 3326 resources, 4301 property instances, and 4877 triples in sum.
[*Top-3 Results]*
We computed the similarity between every pair of these 10 entities for all $k = 1, 2, 3$. Table 8 shows the top-3 most similar entities for each entity.

Angels and Demons	Film
The DaVinci Code	Film
That Thing You Do!	Film
Original Sin	Film
Jude	Film
Catch Me If You Can	Film
Leonardo DiCaprio	Actor
Tom Hanks	Actor
Phil Alden Robinson	Director
Joe Dante	Director

Table 6. Selected (seed) entities

Entity	Top-3 more similar entities		
	sim_1	sim_2	sim_3
The Da Vinci Code	⟨ Angels and Demons, That Thing You Do!, Catch Me if You Can ⟩	⟨ Angels and Demons, That Thing You Do!, Catch Me if You Can ⟩	⟨ Angels and Demons, Catch Me if You Can, That Thing You Do! ⟩
Angels and Demons	⟨ The Da Vinci Code, That Thing You Do!, Catch Me if You Can ⟩	⟨ Tom Hanks, The Da Vinci Code, That Thing You Do! ⟩	⟨ Tom Hanks, The Da Vinci Code, That Thing You Do! ⟩
Tom Hanks	⟨ Leonardo DiCaprio, Phil Alden Robinson, Joe Dante ⟩	⟨ Angels and Demons, Leonardo DiCaprio, That Thing You Do! ⟩	⟨ Angels and Demons, Leonardo DiCaprio, That Thing You Do! ⟩
That Thing You Do!	⟨ Catch Me if You Can, Angels and Demons, The Da Vinci Code ⟩	⟨ Catch Me if You Can, Tom Hanks, Angels and Demons ⟩	⟨ Phil Alden Robinson, Tom Hanks, Angels and Demons ⟩
Original Sin	⟨ Jude, Angels and Demons, That Thing You Do! ⟩	⟨ Jude, Angels and Demons, That Thing You Do! ⟩	⟨ Jude, Angels and Demons, The Da Vinci Code ⟩
Jude	⟨ That Thing You Do!, Angels and Demons, Original Sin ⟩	⟨ Angels and Demons, Original Sin, That Thing You Do! ⟩	⟨ Phil Alden Robinson, Angels and Demons, Original Sin ⟩
Catch Me if You Can	⟨ That Thing You Do!, The Da Vinci Code, Angels and Demons ⟩	⟨ That Thing You Do!, The Da Vinci Code, Angels and Demons ⟩	⟨ Joe Dante, The Da Vinci Code, That Thing You Do! ⟩
Leonardo DiCaprio	⟨ Tom Hanks, Phil Alden Robinson, Joe Dante ⟩	⟨ Tom Hanks, Catch Me if You Can, Angels and Demons ⟩	⟨ Tom Hanks, Angels and Demons, Catch Me if You Can ⟩
Phil Alden Robinson	⟨ Joe Dante, Tom Hanks, Leonardo DiCaprio ⟩	⟨ That Thing You Do!, Catch Me if You Can, Angels and Demons ⟩	⟨ That Thing You Do!, Catch Me if You Can, Jude ⟩
Joe Dante	⟨ Phil Alden Robinson, Tom Hanks, Leonardo DiCaprio ⟩	⟨ Catch Me if You Can, Phil Alden Robinson, That Thing You Do! ⟩	⟨ Catch Me if You Can, The Da Vinci Code, Phil Alden Robinson ⟩

Fig. 8. Comparative Results for *sim*

We can observe that for some entities, the 3 most similar entities change when k changes. For example, the 3 most similar entities for `Tom Hanks` and $k = 1$, are:
⟨ `Leonardo DiCaprio`,
`Phil Alden Robinson`,
`Joe Dante` ⟩
while for $k = 2, 3$ they are:
⟨ `Angels and Demons`,

Entity	Top-3 more similar entities		
	sim_1	sim_2	sim_3
The Da Vinci Code	⟨ Angels and Demons, That Thing You Do!, Catch Me if You Can ⟩	⟨ Angels and Demons, That Thing You Do!, Catch Me if You Can ⟩	⟨ Angels and Demons, That Thing You Do!, Catch Me if You Can ⟩
Angels and Demons	⟨ The Da Vinci Code, That Thing You Do!, Catch Me if You Can ⟩	⟨ Tom Hanks, The Da Vinci Code, That Thing You Do! ⟩	⟨ Tom Hanks, The Da Vinci Code, That Thing You Do! ⟩
Tom Hanks	⟨ Leonardo DiCaprio, Phil Alden Robinson, Joe Dante ⟩	⟨ Angels and Demons, Leonardo DiCaprio, That Thing You Do! ⟩	⟨ Angels and Demons, Leonardo DiCaprio, That Thing You Do! ⟩
That Thing You Do!	⟨ Catch Me if You Can, The Da Vinci Code, Angels and Demons ⟩	⟨ Catch Me if You Can, Tom Hanks, Angels and Demons ⟩	⟨ Phil Alden Robinson, Tom Hanks, Angels and Demons ⟩
Original Sin	⟨ Jude, Angels and Demons, That Thing You Do! ⟩	⟨ Jude, Angels and Demons, That Thing You Do! ⟩	⟨ Jude, Angels and Demons, The Da Vinci Code ⟩
Jude	⟨ That Thing You Do!, Angels and Demons, Original Sin ⟩	⟨ That Thing You Do!, Angels and Demons, Original Sin ⟩	⟨ Phil Alden Robinson, That Thing You Do!, Angels and Demons ⟩
Catch Me if You Can	⟨ That Thing You Do!, The Da Vinci Code, Angels and Demons ⟩	⟨ That Thing You Do!, The Da Vinci Code, Angels and Demons ⟩	⟨ Joe Dante, The Da Vinci Code, That Thing You Do! ⟩
Leonardo DiCaprio	⟨ Tom Hanks, Phil Alden Robinson, Joe Dante ⟩	⟨ Tom Hanks, Catch Me if You Can, Angels and Demons ⟩	⟨ Tom Hanks, Angels and Demons, Catch Me if You Can ⟩
Phil Alden Robinson	⟨ Joe Dante, Tom Hanks, Leonardo DiCaprio ⟩	⟨ That Thing You Do!, Catch Me if You Can, Angels and Demons ⟩	⟨ That Thing You Do!, Catch Me if You Can, Jude ⟩
Joe Dante	⟨ Phil Alden Robinson, Tom Hanks, Leonardo DiCaprio ⟩	⟨ Catch Me if You Can, Phil Alden Robinson, That Thing You Do! ⟩	⟨ Catch Me if You Can, The Da Vinci Code, Phil Alden Robinson ⟩

Fig. 9. Comparative Results for sim''

`Leonardo DiCaprio`,
`That Thing You Do!`⟩.

We also observed that for $k = 1$ for some entities we could not get any similar entity. Therefore higher values of k are beneficial.

[*Comparison with* sim'']

At Section 6.4 we described a variation of the similarity function, denoted by sim''. Table 9 shows again the top-3 most similar entities (as in Table 8) when using sim''. We observe that the results are quite similar to those of Table 8, in most times only the relative ordering of the three more similar entities differs.

[*Times]*

The average time to compute $sim_k()$ between two randomly selected resources, for $k = 2$ equals 3 milliseconds, while for $k = 3$ equals 32 milliseconds. All experiments were carried out in a computer with processor Intel(R) Core(TM)2 Duo @2.40GHz, 2 GB Ram, running Microsoft Windows 7 Ultimate.

7 Implementation Approaches

Here we discuss implementation issues.

[The Straightforward approach]
One could attempt to compute the similar entities at run-time during the construction of the page at hand. However, that would not be efficient in the sense that a lot of information would have to be fetched and processed. In particular, to compute the similar entities for an entity A we should compute the values $sim_k(A, x)$ for all possible resources x. The cost could be reduced by limiting the set of values that x may take. Specifically, we can first specify the classes of the possible similar entries, in our case the classes of *actors, directors, editors, movies, writers* (as we described at Section 2), and then download all information available only for these resources. In any case that would be unacceptably slow and inefficient for large KBs.

[The Single Repository (and Preprocessing) approach]
An alternative approach is to download and process the entire KB (e.g. as we did in the previous section). Since for each entity we need to show only the L (e.g. L=5) most similar entities, we can compute offline the L most similar entities for each entity of the classes of interest, and then store these L resources (e.g. in main memory) for immediate use at run time. Recall that current WSE (Web Search Engines) also compute off-line and store for each page the 20 most similar pages. This preprocessing can be done offline, before the deployment of the application, and it can be periodically redone as new information becomes available at LOD.

[A Similarity-Reversal approach]
An alternative and more challenging implementation approach is sketched below. One could attempt to "reverse" the similarity function, i.e. try traversing the graph around A and collect those entities which have high chances to be in the top-L most similar entities, and compute the similarities only for them. Such an approach does not require any preprocesssing and could be feasible at run time. Its feasibility also depends on how exactly the similarity function is defined. Below we will elaborate on such an approach. The presented approach can be applied to our similarity metric, as well as to other similarity metrics whose computation requires analyzing the subgraphs of the compared entities. The ultimate objective is to devise efficient top-k algorithms (in the spirit of [8,9]), appropriate for graph-based similarity measures. Nevertheless, such a method cannot be faster than the preprocessing method. On the other hand, the benefit of adopting such a method is that it does not require having access (or ability to store) the entire KB. We should note that [13] also proposes to query the Web of Linked Data by traversing RDF links during run-time since due to the openness of the LOD space it may not be possible to know in advance all data sources that might be relevant for query answering. We should stress at this point that our problem is more difficult since we do not want to evaluate a single SPARQL query but to find the most similar entities and this in general requires the evaluation of several queries.

7.1 On Reversing the Similarity Function

Consider an entity A and suppose that we want to compute the more similar entities to A. This requires computing the subgraphs of A as well as the subgraphs of the other entities of the KB. Below we will study this problem by considering one kind of graph expansion at a time.

• $ResFrom(\cdot)$-graph expansion.
Suppose the graph expansion is defined only by $ResFrom(\cdot)$. It is not hard to see that for each $x \in ResTo(ResFrom(A))$ it holds:
$ResFrom(A) \cap ResFrom(x) \neq \emptyset$.
Let $X_{rf}(A) = ResTo(ResFrom(A))$. Moreover if $x' \notin X_{rf}(A)$, then $ResFrom(A) \cap ResFrom(x') = \emptyset$. This means that the nominator of the similarity function is certainly greater than zero only for these entities.

• $Classes(\cdot)$-graph expansion.
For this expansion method, it is not hard to see that for each $x \in X_{cl}(A) = Instances(Classes(A)))$ it holds $Classes(A) \cap Classes(x) \neq \emptyset$.

• $SupClasses(\cdot)$-graph expansion.
Analogously, for each
$x \in X_{sp}(A) = SubClasses(SuperClasses(A)))$ it holds
$SuperClasses(A) \cap SuperClasses(x) \neq \emptyset$.

It follows from the above that all elements of $X_{\cup}(A) = X_{rf}(A) \cup X_{cl}(A) \cup X_{sp}(A)$, and *only these elements*, have certainly non zero similarity.

Let now discuss the case where $k > 1$. In general a value of k greater than one specifies a set of expansion *paths*. We can follow these expansion paths to get the nodes of subgraph for A, and then "reverse" the expansion paths and apply them to the ending nodes of the graph of A. This should be done with care, since although a path can have length 3 (i.e. $k = 3$), an ending node of the subgraph could be the result of an expansion of shorter length (e.g. of one), implying that reversed paths should be shorter too.

The application of these reversed paths, can give us the candidate entities. This is actually what we have described above for the case where $k = 1$. Below we describe in detail this process for any value of k.

Consider the set of strings $Directions = \{$ ResFrom, ResTo, Classes, Instances, SubClasses, SuperClasses$\}$. A graph expansion step over RDF/S can be specified by a subset of this set. For instance, the graph expansion used by the proposed similarity metric is specified by the set {ResFrom, Classes, SubClasses}. We can define the "reverse" of a direction as:

$$Rev(ResFrom) = ResTo$$
$$Rev(Classes) = Instances$$
$$Rev(SubClasses) = SuperClasses$$

For a subset $S \subseteq Directions$, we define $Rev(S) = \cup_{s \in S} Rev(s)$.

The Algorithm *getCandidateSimilar* (shown at Fig. 10) takes as input a node A, the value k, and a *policy* being a subset of $Directions$. It returns those objects which have high chances to be very similar to A (actually those whose similarity with A is certainly positive) assuming sim_k over subgraphs defined using the directions in *policy*.

```
Algorithm getCandidateSimilar
Input: $A$, $k$, policy
Output: A set of resources
(1) $R = \emptyset$;
(2) compute $g_k(A) = (N_k(A), E_k(A))$ w.r.t. policy
(3) For each $n \in N_k(A)$
(4)         let $d = dist(n, A)$
(5)         $R = R \cup traverse(Rev(policy)), n, d)$
(6) End for
(7) return $R$;
```

Fig. 10. Alg. for getting the resources which have "similar" subgraphs to A using sim_k

At line (2) the algorithm computes the subgraph of A according to the directions set in *policy*. The distance at line (4) has been computed during line (2). The invocation $traverse(dirs, n, d)$ starts from n and follows the links that correspond to the argument $dirs$, for up to distance d, and returns the encountered nodes. To make it more clear the set of nodes $N_k(A)$ (at line 2) can be computed by $N_k(A) = traverse(policy, A, k)$. Regarding the correctness of the algorithm, as explained earlier, only the elements in the returned R can have non zero similarity to A. After having run the algorithm, the next step is to compute $sim_k(A, r)$ for each $r \in R$ and return the more similar elements. Specifically, for each $r \in R$ we should get all information returned by $traverse(policy, r, k)$. With these information we can compute $sim_k(A, r)$. This can be done either by code or with queries. For instance, $sim_1(A, B)$, assuming that the subgraphs of A and B are defined only by $Classes(\cdot)$, can be computed with a query of the form[8]:

```
SELECT
   (count(distinct ?class1) as ?intersCard)/
   (count(distinct ?class2) as ?unionCard)
   as ?res WHERE {
    {
      A rdf:type ?class1.
      B rdf:type ?class1.
    } UNION{
        { A rdf:type ?class2. }
        UNION
        { B rdf:type ?class2. }
    }
}
```

The above query can be extended to capture also the rest graph expansion steps. However the case where $k > 1$ requires the formulation of much more complex queries. It is easier to do the required computation with a programming language than with a query language.

[8] To be more precise the division has to be casted using XSD data type.

We have just seen how we can collect only those elements with positive similarity to A, by first getting the subgraph of A, and then reversing the expansion paths that defined the subgraph of A.

[*Top-L Algorithm*]
The above algorithm can be extended to become a top-L algorithm, in case we are interested in finding only the L more similar entities. Let's start from the case where $k = 1$ and suppose that the cardinality of the set $X_{\cup}(A)$ is high. Since we are interested in finding the L most similar to A entities, we can adopt a different, more efficient, evaluation approach, specifically we can avoid collecting all elements that will be fetched at line (5) of the algorithm $getCandidateSimilar$. The idea is to collect at first those elements in $X(A)_{\cap} = X_p(A) \cap X_{cl}(A) \cap X_{sp}(A)$. Clearly, the elements in $X(A)_{\cap}$ will have a positive summand for each part of the similarity function, and thus have high probability to contain the L most similar entities. If they are more than the desired number of objects L, i.e. if $|X_{\cap}(A)| \geq L$, then we can rank them and present the L most similar entities. The benefit of this method, in comparison to collecting the elements of the entire $X_{\cup}(A)$ (i.e. line (5)), is that the elements of $X_{\cap}(A)$ apart from being less, they can be fetched efficiently, specifically with one query.

For instance, the set $X_{rf}(A)$ can be computed by the following SPARQL query:

```
SELECT  ?y
WHERE {  A  ?p1 ?x.
        ?y  ?p2 ?x.
        FILTER ( ?p1 != rdf:type &&
                 ?p2 != rdf:type)  }
```

Note that if we wanted to use $ResFrom'$ instead of $ResFrom$, then we would have to use the query:

```
SELECT ?y
WHERE { A ?p ?x.
       ?y ?p ?x.
       FILTER ( ?p != rdf:type ) }
```

The set $X_{cl}(A)$ can be computed by the following SPARQL query:

```
SELECT ?y
WHERE{  A  rdf:type ?x.
        ?y rdf:type ?x.}
```

The set $X_{sp}(A)$ can be computed by the following SPARQL query:

```
SELECT ?y
WHERE{  A  rdfs:subClassOf ?x.
        ?y rdfs:subClassOf ?x. }
```

Now $X_{rf}(A) \cap X_{cl}(A) \cap X_{sp}(A)$ can be computed by the following SPARQL query:

```
SELECT ?y
WHERE{  A ?p1 ?x.
       ?y ?p2 ?x.
```

```
        A  rdf:type ?z.
       ?y rdf:type ?z.

        A  rdfs:subClassOf ?w.
       ?y rdfs:subClassOf ?w.

    FILTER ( ?p1 != rdf:type &&
             ?p2 != rdf:type) }
```

Note that the above query can give a non empty result only if A is at class level and thus can have superclasses.

If however the fetched elements are less than L, i.e. if $|X_{\cap}(A)| < L$, then we have to fetch more elements. We can start collecting those elements that belong in intersections of two of the above sets, i.e. the elements in $X_p(A) \cap X_{cl}(A)$, $X_p(A) \cap X_{sp}(A)$, and $X_{cl}(A) \cap X_{sp}(A)$.

For example, $X_{rf}(A) \cap X_{cl}(A)$ can be computed by the following SPARQL query:

```
SELECT ?y
WHERE{  A ?p1 ?x.
       ?y ?p2 ?x.

        A  rdf:type ?z.
       ?y  rdf:type ?z.

FILTER ( ?p1 != rdf:type &&
         ?p2 != rdf:type)
     }
```

If again the fetched elements are less than L, then we can collect those in $X_{rf}(A) \cup X_{cl}(A) \cup X_{sp}(A)$, i.e. run the original line (5). These elements can be fetched using the following query

```
SELECT ?y
WHERE {
        A ?p1 ?x.
       ?y ?p2 ?x.
        FILTER ( ?p1 != rdf:type &&
                 ?p2 != rdf:type)
      }
      UNION {
        A  type ?z.
       ?y  type ?z.
      }
      UNION {
        A  subClassOf ?w.
       ?y  subClassOf ?w.
      }
```

Essentially the main idea is the following. If the subgraph is defined by a set of directions *dirs*, then instead of reversing each one direction in isolation and getting the union, try reversing all directions at once. Then all directions except one, and so on. In other words, it is like starting from the top node of the Hasse diagram of the powerset of *dirs* $(\mathcal{P}(dirs), \subseteq)$ and then descend level wise. E.g.:

```
      {P,C S}         :level 1
      /  |  \
 {P,C} {P,S}{C,S}     :level 2
   | \ / | \/ |
   | / \ | /\ |
  {P}   {C}  {S}      :level 3
```

A	$\lvert Answer(q)\rvert$			
	$X_{rf\cup cl}(A)$	$X_{rf}(A)$	$X_{cl}(A)$	$X_{rf}(A) \cap X_{cl}(A)$
DaVinciCode	82	76	81	75
Tom Hanks	185	5	182	2
The Thing You Do!	82	75	81	74

Fig. 11. Measurements over the local KB

A	$\lvert Answer(q)\rvert$			
	$X_{rf\cup cl}(A)$	$X_{rf}(A)$	$X_{cl}(A)$	$X_{rf}(A) \cap X_{cl}(A)$
Americano	1,679,605	32,318	1,679,318	32,031
DaVinciCode	1,683,729	246,918	1,668,503	231,692
Illuminati	1,676,081	98,032	1,668,503	90,454
Tom Hanks	2,218,574	862,458	2,183,320	827,204

Fig. 12. Measurements over DBPEDIA

Below we report the number returned resources, for various entities and for various queries, including the query that returns the union of $X_{rf}(A)$ and $X_{cl}(A)$, denoted by $X_{rf\cup cl}(A)$, defined as:

```
SELECT  ?y
WHERE{    A ?p ?x.
        ?y ?p ?x.
        FILTER (?p1 != rdf:type &&
                ?p2 != rdf:type)
     } UNION {
          A rdf:type ?z.
        ?y rdf:type ?z.
     }
```

We did not manage to obtain reliable results for the above queries over the LinkedMDB SPARQL endpoint, since for some reason it does not return very big answers. Therefore at Table 11 we report some indicative (and quite predictable) results over the local KB. Even in this toy KB we can see how the resources are reduced while the required time does not increase a lot.

To get more realistic results, we tried the SPARQL endpoint of DBPEDIA[9]. If A is the movie `Americano`[10] then

$$\begin{aligned}
|X_{rf \cup cl}(A)| &= 1,679,605 \\
|X_{rf}(A)| &= 32,318 \\
|X_{rf'}(A)| &= 32,094 \\
|X_{cl}(A)| &= 1,679,318 \text{ (i.e. all films)} \\
|X_{rf \cap cl}(A) &= 32,031
\end{aligned}$$

Measurements for other entities are shown at Table 12. We observe some big reductions in the answer set (from millions to tens of thousands). However, even for the intersection query the returned answer is quite big; 32 thousands hits although much less than millions, are probably many for fast real-time interaction. One approach to tackle this problem is to try formulating even more restrictive queries which capture the desired characteristic of similarity function in a more accurate way. The extra condition(s) can be added to the query as extra graph pattern, or the query can be enriched with an appropriate `order by` clause. In the latter case the application can consume only the top hits of the ranked hits of the computed answer.

The general approach would be to enrich the query with aggregated counts or similarity functions aiming at reaching a query that directly returns ranked the top-L similar entities. However this is not always possible (depends on how the similarity metric is defined), and in some times this approach is expected to be less efficient than getting through queries the information that is needed and then rank the entities using programming language code. Of course the availability of LOD SPARQL endpoints which support extended versions of SPARQL would be useful. For instance, [17] investigates methods to integrate customized similarity functions into SPARQL. Among the proposed techniques, it seems that the, so called *virtual triple* approach, would be beneficial (shorter queries which are easier to write, optimization potential). However, the scenarios described are more simple in the sense that only on the direct neighborhood of the compared entities is taken into account, and similarity thresholds should be adopted (instead of a parameter L). This direction should be further researched. In general, there is a need for semantic query optimization techniques for similarity queries.

Another important point, which is independent of the query language, is that the refinement of the information that is available in the LOD cloud, i.e. the classification of the available resources to more *refined* classes, is expected to improve not only the quality of the computed similarities, but will make the computation of the similar entities more efficient. Specifically, if entity A were not classified

[9] http://dbpedia.org/sparql

[10] http://dbpedia.org/resource/The_Americano

only as `film`, but to more refined classes (e.g. `Thriller`, `Anti-war Film` etc), then $|X_{cl}(A)|$ would be smaller.

Above we have sketched a top-L version of the algorithm and identified evaluation approaches and difficulties, for the case $k = 1$. If k is greater than one, then one approach is to start from a $k' = 1$ and apply the above algorithm. If the fetched elements are less than L then move to $k' = 2$, and so on, until having fetched L elements or reached the original value of k (i.e. until $k' = k$). However, as we saw in the example of Figure 7(III), such an approach does not guarantee that the top-L similar entities with respect to sim_1 are the same with the top-L similar with respect to sim_k (nevertheless this approach could be used as an approximation).

Probably, the best feasible solution, for the time being, is to define, store and periodically update, *materialized views* accessible through LOD endpoints, which for each entity contain the set of most similar entities.

8 Conclusion

In this paper, we motivated the need for similarity-based browsing over entities which are semantically defined. This kind of browsing can be applied for various kinds of entities e.g. for movies, paintings, photographs, videos, restaurants, or even social entities (groups or individual persons). We introduced a similarity metric which is type-independent, meaning that it can find similarities between entities of different type (for example similarities between an actor and a movie), which is very convenient for similarity-based browsing. The way the similarity metric functions is somehow similar with the *spreading activation* retrieval method proposed for semantic networks [6]. The metric can also be configured (the radius k as well as the graph expansion policy) according to the characteristics of the corpus at hand (and the "affordable" computational complexity). We demonstrated the behavior and the benefits of this metric over a LOD-based application offering similarity-based browsing for movie information. We believe that this metric can also be useful in semantic search [10]. We do not argue that the graph expansion method adopted by the similarity function is the best for all occasions. Instead we have the impression that in many cases the selection of the graph expansion method should be application specific.

Finally, we discussed implementation approaches and we elaborated on a method which is "harmonized" with the distributed and open nature of LOD. The described method can be used for computing the L most similar entities according to similarity metrics which are neighborhood-based. Specifically we showed how a neighborhood-based similarity metric can be reversed to get a query which can collect only those entities whose similarity is certainly greater than 0. Furthermore we sketched possible top-L extensions of the algorithm.

Below we discuss some directions which according to our opinion are worth further research. Regarding similarity functions there is a need for test collections appropriate for comparative evaluation. Regarding algorithms, it is worth investigating top-K (or nearest K) algorithms appropriate for the LOD domain.

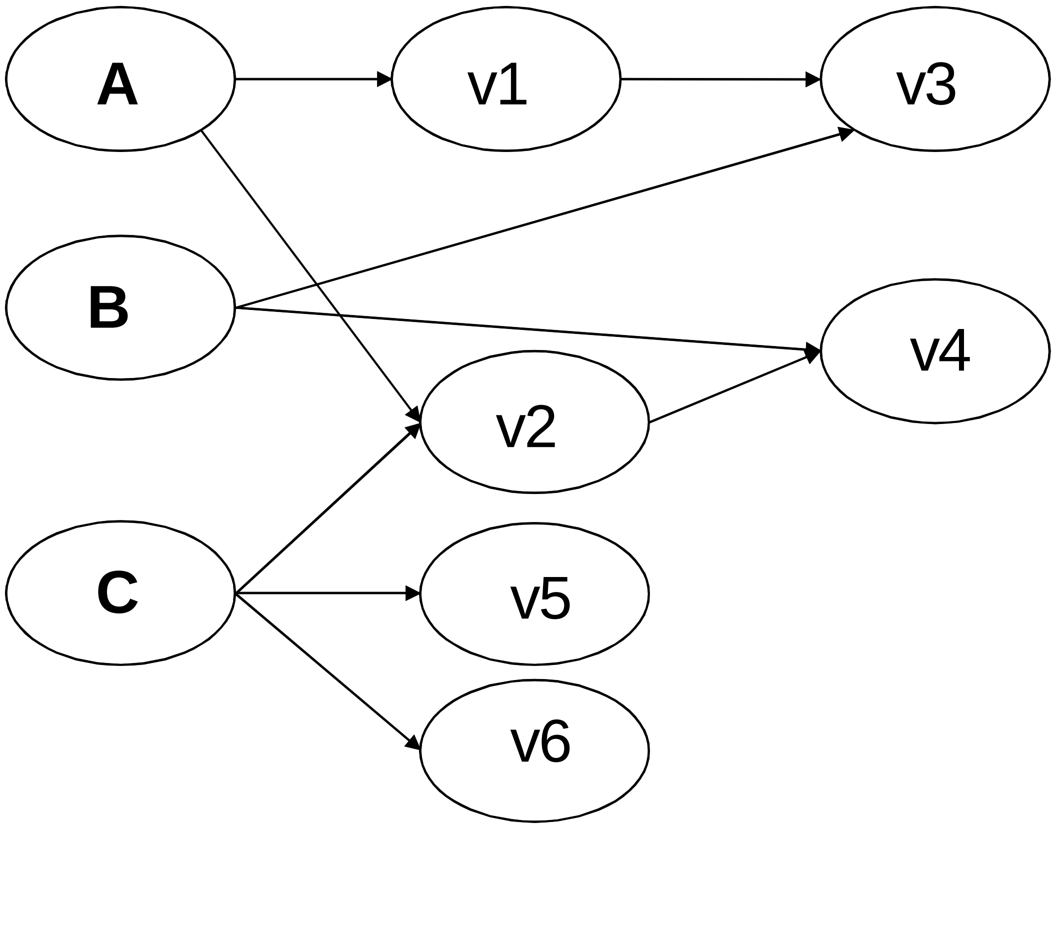
Regarding services for end users, a next step is to device methods for clustering the set of similar entities. Finally, as in web searching, log analysis can be exploited for improving the computation of similarities at application layer.

Moreover, we would like to note that as the number of sources increases, the need for ontology matching techniques (and lexical similarity functions) increases as well. In our application, and since we used two sources of information, we did not face this problem. In any case, the approach presented in this paper can be applied after applying entity matching approaches. A related issue is the management of the `sameAs` predicate. In brief, if two entities are related with such relationships, then they should be treated as equal by the similarity function. Another direction is to consider weighted triples, e.g. investigate a representation framework like Fuzzy RDF [28], and investigate similarity functions for such KBs (an extension of the faceted browsing for such sources is described at [21].

References

1. I. Akbari and M. Fathian. A novel algorithm for ontology matching. *Journal of Information Science*, 36(3):324, 2010.
2. A. Alasoud, V. Haarslev, and N. Shiri. An empirical comparison of ontology matching techniques. *Journal of Information Science*, 35(4):379, 2009.
3. T. Beners-Lee and D. Connoly. "Delta: An Ontology for the Distribution of Differences Between RDF Graphs", 2004. http://www.w3.org/DesignIssues/Diff (version: 2006-05-12).
4. C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.
5. D. Borth, C. Schulze, A. Ulges, and T. Breuel. Navidgator-Similarity Based Browsing for Image and Video Databases. *KI 2008: Advances in Artificial Intelligence*, pages 22–29, 2008.
6. P. R. Cohen and R. Kjeldsen. "Information Retrieval by Constrained Spreading Activation in Semantic Networks". *Information Processing and Management*, 23(2):255–268, 1987.
7. J. Euzenat and P. Valtchev. Similarity-based ontology alignment in OWL-lite. In *ECAI 2004: 16th European Conf. on Artificial Intelligence, August 22-27, 2004, Valencia, Spain*, page 333, 2004.
8. Ronald Fagin. "Combining Fuzzy Information From Multiple Systems". *Journal of Computer and System Sciences*, 58(1):83–99, 1999.
9. Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113, New York, NY, USA, 2001. ACM.
10. B. Fazzinga and T. Lukasiewicz. Semantic search on the Web. *Semantic Web*, 1(1-2):89–96, 2010.
11. S. Ferré. Conceptual Navigation in RDF Graphs with SPARQL-Like Queries. *Formal Concept Analysis*, pages 193–208, 2010.
12. A. Harth. Visinav: Visual web data search and navigation. In *Procs of the 20th Intern. Conf. on Database and Expert Systems Applications (DEXA '09)*, 2009.
13. O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In *Procs of the 8th Intern. Semantic Web Conference (ISWC '09).* Springer, 2009.

14. M. Hildebrand, J. Ossenbruggen, and L. Hardman. /facet: A browser for heterogeneous semantic web repositories. In *Procs of ISWC '06*, 2006.
15. Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
16. C. Kiefer, A. Bernstein, H. Lee, M. Klein, and M. Stocker. Semantic process retrieval with iSPARQL. *The Semantic Web: Research and Applications*, pages 609–623, 2007.
17. C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of isparql: A virtual triple approach for similarity-based semantic web tasks. *The Semantic Web*, pages 295–309, 2007.
18. M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. "Ontology versioning and change detection on the web". In *Procs of EKAW'02*, pages 197–212, Siguenza, Spain, Oct 2002.
19. V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.
20. E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - A Semantic View-Based Search Engine Service for Web Applications. In *Procs of ISWC '06*, pages 847–860, 2006.
21. N. Manolis and Y. Tzitzikas. Interactive Exploration of Fuzzy RDF Knowledge Bases. In *Procs of ESWC'11*, 2011.
22. Jan Noessner, Mathias Niepert, Christian Meilicke, and Heiner Stuckenschmidt. Leveraging terminological structure for object reconciliation. In *Procs. of ESWC'10*, pages 334–348, Heraklion, Crete, Greece, 2010.
23. N. F. Noy and M. A. Musen. "PromptDiff: A Fixed-point Algorithm for Comparing Ontology Versions". In *Procs of AAAI-02*, pages 744–750, Edmonton, Alberta, July 2002.
24. E. Oren, R. Delbru, and S. Decker. Extending Faceted Navigation for RDF Data. In *Procs of ISWC '06*, 2006.
25. E. Pietriga, C. Bizer, D. Karger, and R. Lee. Fresnel - a browser-independent presentation vocabulary for rdf. In *Procs of the Second InterN. Workshop on Interaction Design and the Semantic Web*, pages 158–171. Springer, 2006.
26. G. M. Sacco and Y. Tzitzikas. *Dynamic Taxonomies and Faceted Search: Theory, Practice, and Experience.* Springer, 2009.
27. V. Schickel-Zuber and B. Faltings. Oss: A semantic similarity function based on hierarchical ontologies. In *Proc. of IJCAI*, volume 7, pages 551–556, 2007.
28. Umberto Straccia. A minimal deductive system for general fuzzy rdf. In *Procs of the 3rd Intern. Conf. on Web Reasoning and Rule Systems (RR '09)*, 2009.
29. R.D.K. Thanh-Le Bach. Measuring Similarity of Elements in OWL DL Ontologies. In *Procs of AAAI2005 workshop on Contexts and Ontologies: Theory, Practice and Applications, Pittsburgh, Pennsylvanis, USA*, 2005.
30. G. Tummarello, C. Morbidoni, R. Bachmann-Gmur, and O. Erling. RDFSync: efficient remote synchronization of RDF models. In *Procs of the 6th International Semantic Web Conference (ISWC-07)*, pages 537–551. Springer, 2007.


A
B
C
v1
v2
v3
v4
v5
v6