Building ontologies from folksonomies and linked data: Data structures and Algorithms

Technical Report - 22 May 2012

Andrés García-Silva[1], Jael García-Castro[2], Alexander García[3], Oscar Corcho[1], Asunción Gómez-Pérez[1]

[1] Ontology Engineering Group,
Facultad de Informática, Universidad Politécnica de Madrid, Spain
{hgarcia,ocorcho,asun}@fi.upm.es,
[2] E-Business & Web Science Research Group,
Universitaet der Bundeswehr, Muenchen, Germany
w31blega@unibw.de,
[3] Biomedical Informatics,
Medical Center, University of Arkansas, USA
agarcia@uams.edu,

Abstract. We present the data structures and algorithms used in the approach for building domain ontologies from folksonomies and linked data. In this approach we extracts domain terms from folksonomies and enrich them with semantic information from the Linked Open Data cloud. As a result, we obtain a domain ontology that combines the emergent knowledge of social tagging systems with formal knowledge from Ontologies.

1 Introduction

In this report we present the formalization of the data structures and algorithms used in the approach for building ontologies from folksonomies and linked data. In this approach we use folksonomies to gather a domain terminology. First in a term extraction activity we represent folksonomies as a graph which is traversed by using a spreading activation algorithm (see section 2.1). Next in a semantic elicitation activity we identify classes and relations among terms on the terminology relying on linked data sets (see section 2.2). During this activity terms are associated with semantic resources in DBpedia by means of a semantic grounding algorithm. Once terms are grounded to semantic entities we attempt to identify which of those resources correspond to classes in the ontologies that we are using. Finally we search for semantic relations between the previously identified classes on the ontologies. The classes and relations among them are used to create a draft version of a domain ontology.

2 From Folksonomies to Ontologies

We obtain domain knowledge from a general purpose folksonomy. Our approach uses pertinent domain resources (*i.e.* web pages) to define the search within the folksonomy. Those resources are called seeds; they are used in two ways: i) as starting points to traverse the folksonomy structure, and ii) as reference resources to search for similar resources in the folksonomy. Our approach takes as input a set of seeds in the form of appropriate domain resources and produces as output the corresponding domain ontology (see figure 1). More specifically, we are tapping into a convergent terminology obtained from folksonomies; we are making the semantics of this terminology explicit by associating them with linked data entities.

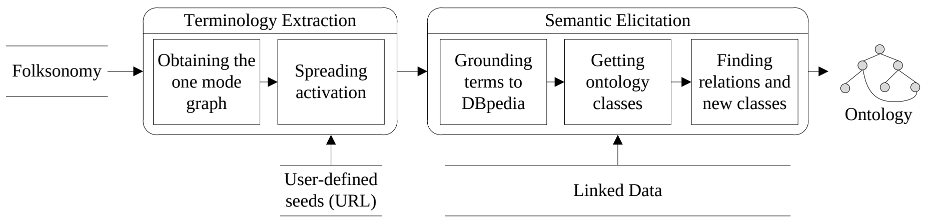

Fig. 1. Eliciting knowledge from folksonomies

We start identifying the appropriate terminology by traversing the graph defined by the folksonomy resources and tags. We use seeds to find similar folksonomy resources. From this process we obtain a set of relevant terms that are grounded to DBpedia. Then we look for classes that are related to those DBpedia resources. In addition, we look for relationships among those classes identified in the linked data set, *i.e.* existing ontologies in the linked data cloud from which we also discover new classes. As output we produce an ontology consisting of local classes that are directly linked to classes in the linked data set; these classes are related by relationships obtained from the linked data set.

2.1 Terminology Extraction

In social tagging systems (STS) users assign tags to resources. Thus, a folksonomy can be understood as a set of annotations $A \subseteq U \times T \times R$ where U, T and, R are sets whose members are users, tags and resources, respectively. A folksonomy can be represented as an undirected tri-partite hyper-graph $G = (V, E)$ where $V = U \cup T \cup R$, and $E = \{(u, t, r)|(u, t, r) \in A\}$. This hyper-graph can be reduced to two and one mode graphs [7].

We are particularly interested in the one mode graph $G' = (V', E')$ whose vertices V' are the set R of resources, and for which there is an edge between two resources r_i and r_j if there is at least a common tag assigned to both resources regardless of the user. Formally, $E' = \{(r_i, r_j) | \exists((u, t_m, r_i) \in A \wedge (u, t_n, r_j) \in A \wedge t_m = t_n)\}$[4]. We have chosen this graph because previous studies such as [5] showed that tags tend to converge around resources in folksonomies.

By browsing the graph G', we look for domain relevant resources and collect the tags used to annotate those resources as valid terms within the domain at hand. Our approach requires a set $S \subset R$ of user-specified seeds. A seed is a folksonomy resource considered highly pertinent to the domain. Thus, we can compare resources in G' with a seed in terms of shared tags; tags of highly similar resources are added to the domain terminology.

In order to traverse the graph G' we use spreading activation [3] in combination with a breadth-first search strategy. Spreading activation is a graph search method initiated by a set of seed nodes weighted with an activation value. Each seed's activation value spreads through the linked nodes in the graph by means of an activation function. The spreading stops when a node activation value is below a specified threshold. When a node is activated more than once, *i.e.* it is reached by the spreading of different seeds, the node activation value can be added up.

The activation value for a node r_j (see equation 1) is calculated by estimating the rate of shared category names with r_i, being r_i the previous visited node from which we reached r_j.

$$a(r_j) = \frac{|\{cn \in CN | (u, r_j, cn) \in CI\} \cap \{cn \in CN | (u, r_i, cn) \in CI\}|}{|\{cn \in CN | (u, r_i, cn) \in CI\}|} \tag{1}$$

The activation function also depends on the activation value of the previously activated node r_i. Thus, $a(r_j) = a(r_j) + a(r_i) * \lambda$ where $0 \leq \lambda \leq 1$ is a real number representing a decay factor. If $a(r_j)$ is greater than a threshold h, then it is marked as activated and the search continues; otherwise the search stops.

Once all the seeds are processed we calculate weights for the category names of those activated nodes. We multiply the frequency of each category name by the node activation value (see equation 2). Then we gather all distinct category names used to classify the activated nodes and added up their weights. Finally the list of category names is sorted in descending order according to the aggregated weight. As output of this task we created a set of terms from the list of category names representing a valid domain terminology.

$$w(r_j, cn_k) = |\{(u, r_j, cn_k) | (u, r_j, cn_k) \in CI\}| \times a(r_j) \tag{2}$$

The spreading activation algorithm for collecting domain terms is presented in section 3.1.

2.2 Semantic Elicitation

We rely on linked data to identify the semantics of those terms extracted from the folksonomy. Linked data are published using RDF and typed links between

[4] Note that annotations can be made by different users u, however, to keep the notation simple we don't show this in the formalization.

data from different sources [1]. Throughout this report we refer to linked data offered by DBpedia [2], OpenCyc[5], UMBEL[6] and YAGO[9]. DBpedia contains knowledge from Wikipedia for close to 3.5 million resources; 1.6 million resources are classified in a cross domain ontology containing 272 classes. OpenCyc is a general purpose knowledge base; it has nearly 500.000 concepts, around 15.000 types of relationships, and approximately 5 million facts relating these concepts. UMBEL is a vocabulary and reference concept ontology designed to help content to interoperate on the Web. This ontology has 28.000 concepts and 38 types of relationships. YAGO is a knowledge base derived from Wikipedia and WordNet; its core version has over 2.6 million entities and around 33 million facts. These datasets are interlinked among them. DBpedia resources, and classes are connected to OpenCyc concepts using *owl:sameAs*, to UMBEL concepts using *umbel#correspondsTo*, and to YAGO concepts using *rdf:type* and *owl:sameAs*.

We are grounding the terminology to DBpedia resources; in this way we are leveraging the high degree of interconnection offered by DBpedia. Once the terms are grounded to DBpedia resources, we then tap into the DBpedia ontology and interconnected data sets together with their corresponding ontologies. By doing so, we are identifying terms corresponding to ontology classes. Similarly, we query the linked data set to look for relationships among the previously identified classes by browsing across the links between data sets. New classes can arise from relationships expanding throughout intermediate classes.

Grounding terms to DBpedia Associating the terminology with DBpedia resources is not a trivial task. A term can be used to refer to one or more DBpedia resources[7]. Thus, for terms associated with more than one DBpedia resource the grounding process requires disambiguating the meaning of the term in order to select the appropriate resource. In our case, terms are tags extracted from a folksonomy; thus, we can use tagging information as context to find the intended meaning of the tag. The fact that DBpedia resources are directly related to Wikipedia articles gives us, in addition to semantic information, textual information that can be consumed during the grounding process[8].

Our approach to ground tags, uses a vector space model representation to model an ambiguous tag as well as candidate DBpedia resources. By context we mean other tags used together with the ambiguous tag when annotating resources that have been activated in the spreading activation process. In other words, the context of an ambiguous tag t_i is the set $\{t \in T | (u, t, r_m) \in A \wedge (u, t_i, r_n) \in A \wedge r_m = r_n\}$. The components of the vectors are the most frequent terms of the Wikipedia articles related to each candidate DBpedia resource. Term Frequency and Inverse Document Frequency (TF-IDF) [8] is used as term weighting scheme to populate the vectors. Then we compare the vector representing the ambiguous tag with each of the vectors representing candi-

[5] OpenCyc home page: http://sw.opencyc.org/
[6] UMBEL home page: http://www.umbel.org/
[7] DBpedia encode this information using *dbpedia-owl:wikiPageDisambiguates*
[8] DBpedia resources and Wikipedia articles are associated with the *foaf:page*

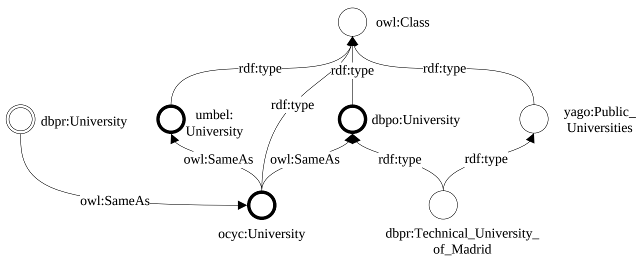

Fig. 2. RDF graph of data linked to dbpr:University. Bold edge nodes represent classes that are reachable following owl:sameAs relationships

date resources by using the cosine as similarity function. The candidate whose vector is the most similar to the ambiguous tag is the correct sense for that tag. Details of this procedure can be found in [4]. This grounding produces a set *Grounding* of tuples relating terms with DBpedia resources. The output of this task is the set of unique DBpedia resources identified in the grounding: $D = \{dbpr|(term_i, dbpr) \in Grounding\}$.

The semantic grounding algorithm for anchoring terms to DBpedia resources is presented in section 3.2.

Getting ontology classes At this point we have connected terms to the linked data cloud using DBpedia resources. We aim at benefiting from the DBpedia ontology as well as from linked ontologies to extract semantic information related to resources in D. However, there is not a direct way to know whether a resource is equivalent to an ontology class.

For instance, the resource *dbpr:University*[9] is not directly related to the class *dbpo:University*[10] (see figure 2). In order to understand this equivalence we ought to navigate through linked data sets following the next path: *dbpr:University owl:sameAs ocyc:University*[11] *owl:sameAs dbpo:University*. We can also identify *ocyc:University* as a class from the OpenCyc dataset. Moreover, if we continue exploring the linked data set we find that *ocyc:University owl:sameAs umbel:University*[12], and that *umbel:University* is also a class in the UMBEL ontology. For identifying classes we query the linked data set in order to find paths of equivalent relationships[13] and of variable length connecting the source resource with a target entity defined as a class.

Following a similar approach to [6] we use SPARQL queries to look for relationships linking the source resource s and a class c. We define the path length L as the number of relationships found in the path linking s with c. For $L = 1$ we

[9] The *dbpr* prefix refers to `http://dbpedia.org/resource/`
[10] The *dbpo* prefix refers to `http://dbpedia.org/ontology/`
[11] The prefix *ocyc* refers to `http://sw.opencyc.org/2009/04/07/concept/en/`
[12] The prefix *umbel* refers to `http://umbel.org/umbel/sc/`
[13] We are using the phrase 'equivalent relation' in a broad sense of similarity.

look for a relationship $relation_i$ linking s with c. As we do not know the direction of $relation_i$, we search in both directions: 1) s $relation_i$ c, and 2) c $relation_i$ s. For $L = 2$ we look for a path containing two relationships and an intermediate resource *node* such as: s $relation_i$ *node*, and *node* $relation_j$ c. Note that each relationship may have two directions and hence the number of possible paths is $2^2 = 4$. For $L = 3$ we have three relationship placeholders and the number of possible paths is $2^3 = 8$. In general, for a path length L we have $n = \sum_{l=1}^{L} 2^l$ possible paths that can be traversed by issuing the same number of SPARQL queries on the linked data set.

We ensure that the relationships in the path will be relationships such as *owl:sameAs* or *umbel:correspondsTo*. In order to guarantee that c is a class, we make sure its type is *owl:Class*. An example query is shown next:

```
SELECT  ?class
WHERE{<dbpr:University> ?relation1 ?node1. ?node1 ?relation2 ?class.
        ?class <rdf:type> <owl:Class>
  FILTER(?relation1=<owl:sameAs>||?relation1=<umbel:correspondsTo>
```

Listing 1.1. SPARQL query

Thus for each *dbpr* in D and a given value of L we pose n SPARQL queries following the above pattern to find related classes. As a result a set of tuples associating DBpedia resources *dbpr* with ontology classes *oc* are created. We create a local class *lc* for each DBpedia resource *dbpr* related to at least one class *oc*. Each local class *lc* is associated, by means of a *owl:sameAs* relationship, with the classes *oc* identified for the DBpedia resource. Hence, in this task we create an ontology O and then add the local classes *lc* along with their relationships with the classes *oc* to O. From the example, we create a local class *University* and assert that it is *owl:sameAs* to *dbpo:University*, *ocyc:University* and *umbel:University*. Henceforth, when a *oc* class is related to a local class *lc* by means of a *owl:sameAs* relationship, it is called a member class.

The algorithm for identifying classes in linked data sets out of DBpedia resources is presented in section 3.3.

Finding relationships and new classes In order to look for relationships among the local classes *lc*, we take advantage of their member classes *oc* to search for relationships among them in the linked data set. We propose to carry out a pairwise search for relationships among the member classes of all local classes. In order to benefit the most from the linked data graph, we need to look for variable length paths of relationships. Thus, we follow a similar strategy to the one mentioned above for finding classes for DBpedia resources. The only difference is that in this case we have a concrete source and target of the path; the source oc_i and the target oc_j are such that both classes are members of different local classes lc_i and lc_j. Classes found in a path linking oc_i and oc_j are added as local classes to the ontology O. In addition, for each relationship found between oc_i and oc_j we create relationships between lc_i and lc_j and add them to the ontology O .

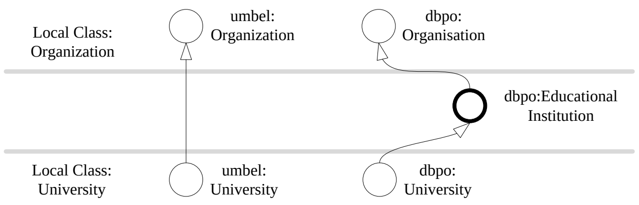

Fig. 3. Searching for relationships among members classes of University and Organization. Bold edge node represents a new class discovered in the process.

Figure 3 shows existing relationships between member classes of two local classes: *University* and *Organization*. The *University* class is defined as in the previous example. The *Organization* class has *umbel:Organization* and *dbpo:Organisation* as members. Looking for a path of length 1 between members of both local classes in the dataset we find: *umbel:University rdfs:subClassOf umbel:Organization*. Therefore we can assert, by creating a local relationship, that *University* is *rdfs:subClassOf* of *Organization*. Moreover, if we extend our search to paths of length 2 we find that *dbpo:University rdfs:subClassOf dbpo:EducationalInstitution*, and *dbpo: Educational Institution rdfs:subClassOf dbpo:Organisation*. In this case we have discovered a new class named *dbpo:EducationalInstitution* that we add to our ontology as a local class *EducationalInstitution*. Then we assert that *University* is *rdfs:subClassOf EducationalInstitution*, and that *EducationalInstitution* is *rdfs:subClassOf Organization*.

The algorithm for finding relations between classes in linked data sets is presented in section 3.4.

3 Algorithms

In this section we present the algorithms for each of the activities involved in the approach for building ontologies from folksonomies and linked data sets. We use pseudo-code for describing the algorithms and present the details of the main procedures and functions. Please note that the procedures and functions of each algorithm are presented in order of dependency. Non-dependent procedures and functions appears on top of the algorithm description while dependent ones are on the bottom part. When describing each algorithm we make clear the main procedure from which the reading of code can be started.

3.1 Algorithm for term selection

In this section we present the algorithm to collect the terms relevant to the domain. This algorithm consists of a main function *collectTerms* which uses a breath first search (BFS) and spreading activation over the graph. BFS is in charge of traversing the graph G' starting from an user-defined seed. Recall that G' is a graph whose vertices are resources, and there exists an edge between two resources if they share at least a category name. Vertices in this graph have as attributes the category names under which they have been classified. For each visited vertex the activation value is calculated using the *activationFunction*. We collect category names as relevant terms for those vertices with an activation value over a predefined threshold. The activation value depends on the rate of shared categories between the vertices, and on the activation value of the source vertex from which we reached the current vertex.

Algorithm 1 Collecting domain relevant terms with spreading activation

```
        ▷ Breadth first search over the G' graph using s as a seed to drive the search
 1: function BFS($G', s$)
 2:     $actVertices \leftarrow Array[]$                              ▷ Array of activated vertices
 3:     $q \leftarrow Queue()$
 4:     $enqueue(q, s)$
 5:     $setState(s, ``visited'')$
 6:     while $q \neq empty$ do
 7:         $v_0 \leftarrow dequeue(q)$                                   ▷ Get vertex to process
 8:         $activationValue \leftarrow activationFunction(v_0)$                ▷ Calc Act. Value
 9:         if $activationValue \geq threshold$ then              ▷ if the vertex is activated
10:             $setActValue(v_0, activationValue)$
11:             $add(actVertices, v_0)$                          ▷ Collect activated vertices
12:             for all $v \in adjacent(G', v_0)$ do           ▷ for each adjacent vertex in G'
13:                 if $state(v) \neq ``visited''$ then
14:                     $setState(v, ``visited'')$
15:                     $setEdgeSrcVertex(v, v_0)$     ▷ Save the source vertex of the edge
16:                     $enqueue(q, v)$          ▷ enqueue the adjacent vertix for activation
17:                 end if
18:             end for
19:         end if
20:     end while
21:     return $actVertices$
22: end function
                                                ▷ Calc. activation function of the vertex
23: function ACTIVATIONFUNCTION($vertex$)
                                                   ▷ Get categories of the source vertex
24:     $srcVertex \leftarrow getEdgeSrcVertex(vertex)$
25:     $srcCategoryNames \leftarrow getCategoryNames(srcVertex)$  ▷ Array of categories
                                                  ▷ Get categories of the current vertex
26:     $vertexCategoryNames \leftarrow getCategoryNames(vertex)$  ▷ Array of categories
27:     $sharedCategoryNames \leftarrow 0$
28:     for all $categoryName \in vertexCategoryNames$ do
29:         if $categoryName \in srcCategoryNames$ then
30:             $sharedCategoryNames \leftarrow sharedCategoryNames + 1$
31:         end if
32:     end for
```

```
    $actVal \leftarrow \frac{sharedCategoryNames}{length(srcCategoryNames[])} + getActValue(srcVertex) * decayFactor$
    **return** $actVal$
**end function**
                                        ▷ Assign weights to every category name of each activated vertex
**procedure** CALCWEIGHTSFORACTVERTICES($actVertices[]$)
    **for all** $vertex \in actVertices$ **do**
        $categoryNames \leftarrow getCategoryNames(vertex)$
        **for all** $category \in categoryNames$ **do**
            $categoryFreq \leftarrow getClassificationFreq(vertex, category)$
            $weight \leftarrow getActValue(vertex) * categoryFreq$
            $setWeight(category, weight)$
        **end for**
    **end for**
**end procedure**
                                        ▷ Get a list of terms from categories of activated vertices
**function** GETCATEGORIES($actVertices[]$)
    $uniqueCategories \leftarrow Array[]$
                                        ▷ Group category names regardless vertices and sum their weights
    **for all** $vertex \in actVertices$ **do**
        $categoryNames \leftarrow getCategoryNames(vertex)$
        **for all** $category \in categoryNames$ **do**
            **if** $category \notin uniqueCategories$ **then**
                $newCategory \leftarrow category$
                $setWeight(newCategory, getWeight(category))$
                $add(uniqueCategories, newCategory)$
            **else**
                $existingCategory \leftarrow getCategory(uniqueCategories, category)$
                $weight \leftarrow sum(getWeight(existingCategory), getWeight(category))$
                $setWeight(existingCategory, weight)$
                $update(uniqueCategories, existingCategory)$
            **end if**
        **end for**
    **end for**
                                        ▷ sort categories in desc. order of weight and discard using the threshold
    $getTerms(sort(uniqueCategories, ``desc''), threshold)$
**end function**
                                        ▷ Get relevant terms from the activated vertices
**function** GETRELEVANTTERMS($actVertices[]$)
    $calcWeightsForActVertices(actVertices)$                ▷ weights per category
    **return** $getCategories(actVertices)$                 ▷ group categories
**end function**
```

```
                    ▷ Collect domain relevant terms from the G' graph using a list of seeds
69: function collectTerms($G'$, $seeds[]$)
70:     $allActVertices \leftarrow Array[]$
71:     for all $s \in seeds$ do                ▷ For each seed run the spreading activation
72:         $actVertices = BFS(G', s)$                      ▷ Breath first search over $G'$
73:         $add(allActVertices, actVertices)$                ▷ List of activated vertices
74:     end for
75:     $terms \leftarrow getRelevantTerms(allActVertices)$
76:     return $terms$
77: end function
```

3.2 Algorithm for Semantic Grounding

We present the algorithm 2 where the semantic grounding of contextualized terms to entities in knowledge bases is described. The main function is *SemanticGrounding* which receives as input the term to ground as well as the context where the term appears. The context consists of a list of terms. This function retrieves from the knowledge base the set of candidate meanings for the term. In the cases where there are more than one candidate a disambiguation function is used (*disambiguateTerm*). This function creates the vector space of candidates and their terms, as well as a vector for the query which represents the term context. Then, using the *search* function, vectors are compared so that the most similar candidate is retrieved.

Algorithm 2 Semantic Grounding of contextualized terms

```
                ▷ Create the matrix representing the vector space (candidates X allTerms)
 1: function createVectorSpace($candidates[]$, $allTerms[]$)
 2:     $vectorSpaceMatrix \leftarrow Matrix[length(candidates)][length(allTerms)]$
 3:     $i \leftarrow 0$
 4:     for all $candidate \in candidates$ do
 5:         $j \leftarrow 0$
 6:         for all $term \in allTerms$ do
 7:             $VectorSpaceMatrix[i][j] \leftarrow calcTFIDF(term, candidate, candidates)$
 8:             $j \leftarrow j + 1$
 9:         end for
10:         $i \leftarrow i + 1$
11:     end for
12:     return $vectorSpaceMatrix$
13: end function
                                ▷ Create the vector representing the term context
14: function createQueryVector($context[]$, $allTerms[]$)
15:     $queryVector \leftarrow Array[length(allTerms)]$
```

```
    $i \leftarrow 0$
    for all $term \in allTerms$ do
        if $term \in context$ then
            $queryVector[i] = 1$
        else
            $queryVector[i] = 0$
        end if
        $i \leftarrow i + 1$
    end for
    return $queryVector$
end function
                ▷ Select from the vector space the most similar candidate to the term context
function SEARCH(vectorSpaceMatrix[][],queryVector[], candidates[])
    $simCandidates \leftarrow Array[length(candidates)]$
    $i \leftarrow 0$
    for all $candidate \in candidates$ do
        $candidateVector \leftarrow vectorSpaceMatrix[i]$
        $simValue \leftarrow calcCosine(queryVector, candidateVector)$
        $simCandidates[i] \leftarrow candidate$
        $setSimValue(simCandidates[i], simValue)$
        $i \leftarrow i + 1$
    end for
    $sort(simCandidates, ``desc")$                ▷ sort by simValue in descending order
    if $simCandidates[0] \geq threshold$ then
        return $simCandidates[0]$                 ▷ Return the most similar candidate
    else
        return “”
    end if
end function
                ▷ disambiguate the term searching over the list of candidate meanings
function DISAMBIGUATETERM($termContext[], candidates[]$)
    $allTerms \leftarrow getTerms(candidates)$                ▷ Extract terms of the meanings
    $vectorSpaceMatrix \leftarrow createVectorSpace(candidates, allTerms)$
    $queryVector \leftarrow getQueryVector(context, allTerms)$
    return $search(vectorSpaceMatrix, queryVector, candidates)$
end function
                ▷ Get the semantic entity for the term meaning in the context
function SEMANTICGROUNDING($term, context[]$)
    $candidates \leftarrow knowledgeBase.getMeanings(term)$         ▷ Array of meanings
    if $length(candidates) = 0$ then
        return “”
    else if $length(candidates) = 1$ then
        return $candidate[0]$
    else
        return $disambiguateTerm(context, candidates)$
    end if
end function
```

3.3 Algorithm for identification of classes

In this section we described the algorithm for identifying relevant classes from the set of semantic entities found in the Semantic Grounding. The main procedure is *IdentifyClasses* which receives as input the semantic entity and a predifined path length used to limit the number of relations in the SPARQL queries. This procedure uses a procedure (*generateQueries*) to generate dynamically the SPARQL queries to be posed against an SPARQL endpoint. Then, these queries are executed and the classes are extracted from the result sets.

The *generateQueries* procedure first creates the query (ASK) verifying whether the semantic entity is a class or not. Next it creates the list of queries per each of the possible values of i (1..PathLength). Recall that for each path length value i the number of queries to traverse all the possible paths is 2^i. The strategy to generate the queries per each path length value uses a queue of the resources involved in each query. The queue is created by the function *getResources*, and for each path length value the number of resources in the queue is $pathlength+1$. For instance for $i = 2$ we have three resources: the semantic entity, an intermediate node (node1), and the class variable.

The creation of each list of queries is delegated to an overload function *createQueries*. This is a recursive function which creates the queries by adding a query pattern at a time using a resource extracted from the queue in each recursive call. For instance, for $i = 2$, in the first call of the *createQueries* function two resources are dequeue and $2^2 = 4$ queries are created with the following query patterns: 1) $r_i \overset{rel1}{\rightarrow} r_j$, 2) $r_i \overset{rel1}{\leftarrow} r_j$, 3) $r_i \overset{rel1}{\rightarrow} r_j$, and 4) $r_i \overset{rel1}{\leftarrow} r_j$. Note that the direction of the relations is alternate according to the *setRelationDirection* function. In a second recursive call, another resource r_k is dequeue, and for each query already created we add new query patterns relating the last node of the existing query pattern to r_k. This results in the following query patterns: 1) $r_i \overset{rel1}{\rightarrow} r_j \overset{rel2}{\rightarrow} r_k$, 2) $r_i \overset{rel1}{\leftarrow} r_j \overset{rel2}{\rightarrow} r_k$, 3) $r_i \overset{rel1}{\rightarrow} r_j \overset{rel2}{\leftarrow} r_k$ and 4) $r_i \overset{rel1}{\leftarrow} r_j \overset{rel2}{\leftarrow} r_k$. Finally the SPARQL query containing these patterns are created by the function *createQuery* for which we do not provide more details since it performs a syntactical transformation between the aforementioned patterns and the SPARQL query syntax.

Algorithm 3 Algorithm for identifying classes using dynamic SPARQL queries.

```
        ▷ Give alternate directions for the relation to be included in the query patterns
1: function SETRELATIONDIRECTION($queryNumber, idRel, currentDirection$)
2:     if $mod(queryNumber, 2^{idRel-1}) = 0$ then
3:         if $currentDirection = “forward''$ then
4:             return $“backward''$
5:         else
6:             return $“forward''$
7:         end if
8:     end if
9: end function
```

```
                    ▷ Create a query containing a pattern for the input resources
10: function CREATEQUERY($resource, nextResource, queryNumber, idRel, direction$)
11:     $direction \leftarrow setRelationDirection(k, idRel, direction)$
12:     $rel \leftarrow$ “$?rel''$ $+ idRel$                        ▷ Name of the variable
13:     if $direction =$ “$forward''$ then
14:         $qry \leftarrow createQueryPattern(resource, rel, nextResource)$
15:     else
16:         $qry \leftarrow createQueryPattern(nextResource, rel, Resource)$
17:     end if
18:     return $qry$
19: end function
    ▷ Recursive procedure to create queries for each path length. Queries are created
    by adding relations between pair of nodes in each recursive call.
20: procedure CREATEQUERIES($queries, resourceQueue, currentPathLenght, idRel$)
21:     if $length(resourceQueue) = 0$ then         ▷ Stop condition of the recursion
22:         return $queries$
23:     end if
24:     $resource \leftarrow dequeue(resourceQueue)$                    ▷ Get a resource
25:     $direction \leftarrow$ “$forward''$                  ▷ Relation direction: $x \rightarrow y$
26:     if $length(queries) = 0$ then     ▷ Create the initial $2^{currentPathLenght}$ queries
27:         $nextResource \leftarrow dequeue(resourceQueue)$         ▷ Get another resource
28:         for $k = 1 \rightarrow 2^{currentPathLenght}$ do               ▷ For each query to create
                    ▷ Create a query with a pattern relating both resources in a direction
29:             $qry \leftarrow createQuery(resource, nextResource, k, idRel, direction)$
30:             $add(queries, qry)$
31:             $k \leftarrow k + 1$
32:         end for
33:         $idRel \leftarrow idRel + 1$
                              ▷ Recursive call to add another pattern to the queries
34:         $createQueries(queries, resourceQueue, currentPathLenght, idrel)$
35:     else                                  ▷ If the initial queries were already created
36:         $k = 1$
37:         for all $qry \in queries$ do
                    ▷ Query for relation between the last node of qry and the resource
38:             $qryPattern \leftarrow createQuery(lastNode(qry), resource, k, idRel, direction)$
39:             $qry \leftarrow mergeQueries(qry, qryPattern)$
40:             $k \leftarrow k + 1$
41:         end for
42:         $idRel \leftarrow idRel + 1$
                              ▷ Recursive call to add another pattern to the queries
43:         $createQueries(queries, resourceQueue, currentPathLenght, idrel)$
44:     end if
45: end procedure
```

```
                                   ▷ Create a queue of resources for queries of each path length
46: function GETRESOURCES($srcResource, targetClass, currentPathLenght$)
47:     $q \leftarrow queue()$
48:     $enqueue(q, targetClass)$
49:     for $j = 1 \rightarrow currentPathLenght - 1$ do
50:         $node \leftarrow queryPatternVariable(``node" + j)$        ▷ node1, node2, etc.
51:         $enqueue(q, node)$
52:         $j \leftarrow j + 1$
53:     end for
54:     $enqueue(q, srcResource)$
55:     return $q$
56: end function
                                                          ▷ Start the query generation.
57: function GENERATEQUERIES($srcResource, targetClass, pathLength$)
58:     $queries \leftarrow Array[]$
                                              ▷ First query if the srcResource is a class
59:     $queries[0] \leftarrow ``ASK\{"+srcResource + ``rdf : typerdfs : Class\}"$
60:     $relationCounter \leftarrow 1$                  ▷ used to name relations: rel1, rel2, etc.
61:     for $i = 1 \rightarrow pathLength$ do       ▷ For each i value generates all the $2^i$ queries
                              ▷ Get the (i+1) resources to be used in the queries of length i
62:         $resourceQueue \leftarrow getResources(srcResource, targetClass, i)$
                      ▷ create queries for the current path length (i) using resources in the queue
63:         $createQueries(queriesPerLength, resourceQueue, i, relationCounter)$
64:         $add(queries, queriesPerLength)$       ▷ Add the $2^i$ queries to the final lists
65:         $i \leftarrow i + 1$
66:     end for
67: end function
        ▷ Main procedure: given a semantic entity, and path length defining the number
    of links to traverse in the KB, we produce a set of related classes.
68: procedure IDENTIFYCLASSES($semEntity, pathLength$)
69:     $classes \leftarrow Array[]$
70:     $srcResource \leftarrow QueryPatternResource(semEntity)$ ▷ resource <semEntity>
71:     $targetClass \leftarrow queryPatternVariable(``?class")$            ▷ the class variable
72:     $queries \leftarrow generateQueries(srcResource, targetClass, pathLength)$
73:     for all $query \in queries$ do                          ▷ Every query is executed
74:         $result \leftarrow exec(``SPARQLEndPoint", query)$
75:         $add(classes, getClasses(result))$              ▷ get classes from the resultset
76:     end for
77: end procedure
```

3.4 Algorithm for Relation Discovery

We describe the algorithm used to discover relations between two classes from a knowledge base published as linked data. This algorithm poses SPARQL queries aiming at traversing all the possible paths (of a predefined length) linking the two input classes inside the knowledge base. The main procedure is *relationDiscovery* which receives as input the two classes and the maximum path length limiting the search of relations linking the two classes. This procedure first generate all the queries, using the function *generateQueries*, which traverse all the possible paths (of length lesser or equal to the predefined length) linking the classes. Next this queries are executed against the knowledge base SPARQL endpoint, and the result of each query is saved as an RDFGraph.

Each RDFGraph is traversed by the *BFS* function, which implements a breath first search, so that we collect the relations discovered in each query. The *BFS* function treats each RDF graph as undirected to reach all the connected nodes, and then obtains each relation, through the *getRelation* function, according to the direction of the edges linking the nodes. From the list of relations we get, using the *getClasses* function, the classes which are different from the input classes and collect them.

For the query generation, in the *generateQueries* function, we use an algorithm similar to the one used in the algorithm for identifying classes. That is we create the queries to traverse all the possible paths linking the two input classes using: 1) a queue of resources to be included in each lists of queries of a given path length, and 2) we use a recursive function which creates the queries for each path length by adding in each recursive call a query pattern involving a resource taken from the queue and a relation. In fact within the code of the *generateQueries* function we reuse the functions *getResources* and *createQueries*. For details of these algorithms as well as an explanation of how they work the reader is referred to the algorithm explained in section 3.3.

Algorithm 4 Algorithm for discovering relations using SPARQL queries

```
                                                  ▷ Start the query generation.
 1: function GENERATEQUERIES($srcResource, trgResource, pathLength$)
 2:     $queries \leftarrow Array[]$
 3:     $relationCounter \leftarrow 1$            ▷ used to name relations: rel1, rel2, etc.
 4:     for $i = 1 \rightarrow pathLength$ do     ▷ For each i value generates all the $2^i$ queries
          ▷ Get the (i+1) resources to be used in the queries of length i. This function is
          described in algorithm 3.3 line number 46
 5:         $resourceQueue \leftarrow getResources(srcResource, targetClass, i)$
          ▷ create queries for the current path length (i) using resources in the queue. This
          function is described in algorithm 3.3 line number 20
 6:         $createQueries(queriesPerLength, resourceQueue, i, relationCounter)$
 7:         $add(queries, queriesPerLength)$      ▷ Add the $2^i$ queries to the final lists
 8:         $i \leftarrow i + 1$
 9:     end for
10: end function
```

```
                                  ▷ create a relation based on the edge direction linking $v_0$ and $v$
function GETRELATION($RDFGraph, v_0, v$)
    $edge \leftarrow getEdge(RDFGraph, v_0, v)$                                              ▷
    $relationName \leftarrow getEdgeName(edge)$
    $subject \leftarrow getSourceNode(edge)$
    $object \leftarrow getTargetNode(edge)$
    return $setRelation(subject, relationName, object)$
end function
                                            ▷ Traverse the rdf graph as an undirected graph
function BFS($RDFGraph, srcClass, trgClass$)
    $relations \leftarrow Array[]$                                 ▷ Array of new relations
    $q \leftarrow Queue()$
    $enqueue(q, srcClass)$
    $setState(srcClass, “visited'')$
    while $q \neq empty$ do
        $v_0 \leftarrow dequeue(q)$                                 ▷ Get vertex to process
        for all $v \in adjacent(RDFGraph, v_0)$ do          ▷ for each adjacent vertex
            if $state(v) \neq “visited''$ then
                                        ▷ Define a relation using the edge direction information
                $add(relations, getRelation(v_0, v))$
                $setState(v, “visited'')$
                $enqueue(q, v)$                   ▷ enqueue the adjacent vertix for activation
            end if
        end for
    end while
    return $relations$
end function
    ▷ Get a list of rdf graphs containing the relationships of variable length found in
  the knowledge base for the two input classes.
procedure RELATIONDISCOVERY($srcClass, trgClass, pathLength$)
    $allRelations \leftarrow Array[]$
    $classes \leftarrow Array[]$
    $srcResource \leftarrow QueryPatternResource(srcClass)$
    $trgResource \leftarrow queryPatternVariable(trgClass)$
    $queries \leftarrow generateQueries(srcResource, trgResource, pathLength)$
    for all $query \in queries$ do                              ▷ Every query is executed
        $result \leftarrow exec(“SPARQLEndPoint'', query)$
        $rdfGraph \leftarrow getGraph(result)$       ▷ Create an RDF graph from the result
        $relations \leftarrow BFS(RDFgraph)$            ▷ BFS to traverse and get relations
        $add(allRelations, rdfGraph)$
    end for
            ▷ get the classes from the relations which are different from the input ones
    $classes \leftarrow getClasses(allRelations)$
end procedure
```

References

1. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.
2. C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantic*, 7(3):154–165, 2009.
3. F. Crestani. Application of spreading activation techniques in information retrieval. *Artificial Intelligence Review*, 11:453–482, 1997.
4. A. García-Silva, M. Szomszor, H. Alani, and O. Corcho. Preliminary results in tag disambiguation using dbpedia. In *Knowledge Capture (K-Cap 2009)-Workshop on Collective Knowledge Capturing and Representation-CKCaR*, 2009.
5. S. A. Golder and B. A. Huberman. Usage patterns of collaborative tagging systems. *Journal of Information Science*, 32(2):198–208, 2006.
6. P. Heim, S. Lohmann, and T. Stegemann. Interactive relationship discovery via the semantic web. In *Proceedings of the 7th Extended Semantic Web Conference (ESWC 2010)*, volume 6088 of *LNCS*, pages 303–317, Berlin/Heidelberg, 2010. Springer.
7. P. Mika. Ontologies are us: A unified model of social networks and semantics. *Web Semant.*, 5:5–15, March 2007.
8. G. Salton and M. J. Mcgill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.
9. F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *Elsevier Journal of Web Semantics*, 2008.