Graph-based Ontology Analysis in the Linked Open Data

Lihua Zhao
Principles of Informatics Research Division
National Institute of Informatics
Tokyo, Japan
lihua@nii.ac.jp

Ryutaro Ichise
Principles of Informatics Research Division
National Institute of Informatics
Tokyo, Japan
ichise@nii.ac.jp

ABSTRACT

The Linked Open Data (LOD) includes over 31 billion Resource Description Framework (RDF) triples interlinked by around 504 million *SameAs* links (as of September 2011). The data sets of the LOD use different ontologies to describe instances, that cause the ontology heterogeneity problem. Dealing with the heterogeneous ontologies is a challenging problem and it is time-consuming to manually learn big ontologies in the LOD. The heterogeneity of ontologies in the LOD can be reduced by automatically integrating related ontology classes and properties, which can be retrieved from interlinked instances. The interlinked instances can be represented as an undirected graph, from which we can discover the characteristics of instances and retrieve related ontology classes and properties that are important for linking instances. In this paper, we retrieve graph patterns from several linked data sets and perform ontology alignment methods on each graph pattern to identify related ontology classes and properties from the data sets. We successfully integrate various ontologies, analyze the characteristics of interlinked instances, and detect mistaken properties in the real data sets. Furthermore, our approach solves the ontology heterogeneity problem and helps Semantic Web application developers easily query on various data sets with the integrated ontology.

Keywords

graph pattern, linked data, ontology alignment

1. INTRODUCTION

The Linked Open Data (LOD) is a collection of machine-readable structured data connected by *owl:sameAs* [1], which refers to related or identical instances in diverse data sets [2]. In the LOD cloud, data sets are linked with *owl:sameAs* at an instance level, but few links exist at a class or property level. Although the built-in properties owl:equivalentClass and owl:equivalentProperty are designed to indicate that two classes or properties refer to the same concept, there are only few links at a class or property level [9]. The data sets in the LOD are published in the form of RDF triples, mainly categorized into seven domains: cross-domain, geographic, media, life sciences, government, user-generated content, and publications.

The data providers publish data according to the Linked Data principles and also provide links to other data resources [2]. The LOD cloud has been growing rapidly over the past years and currently contains 295 data sets with over 31 billion RDF triples (as of Sep. 2011). However, there is no standard ontology for all the data sets, but all kinds of ontologies that cause the ontology heterogeneity problem.

Another critical problem in the LOD is that the publishers sometimes make mistakes when converting data into RDF triples. For example, some properties are used with different combination of words and instances are described with general ontology classes rather than specific classes. We should correct these mistaken data, but it is time-consuming and infeasible to manually inspect large ontologies of linked data to find out the mistakes.

In order to tackle with the above problems, we propose a semi-automatic approach to inspect how the instances are linked at a class-level in the real data sets, and what kind of properties are useful to find identical instances from different data sets. One of the commonly used methods to overcome the ontology heterogeneity problem is the ontology alignment, which finds corresponding mappings between ontologies [15]. Ontology integration is defined as a process that generates a single ontology from different existing ontologies [4]. Since the instances are linked by *owl:sameAs*, the links and instances consist of undirected graphs and there are various graph patterns for linking different types of instances. By analyzing the graph patterns, we can observe how different types of instances are interlinked.

In this paper, we extract graphs that consist of interlinked instances and perform ontology alignment method to retrieve related classes and properties that are critical to link the instances. We integrate related classes and properties for each graph pattern and aggregate all of them for the whole data sets we applied for our experiments. Furthermore, from the integrated ontology, we can find out misused properties and recommend standard classes and properties for an instance.

The remaining of this paper is organized as follows. In Section 2, we discuss some related work and the limitation of the methods. In Section 3, we present our approach in details and the experiments of our approach are discussed in Section 4. The comparison between our approach and related work is discussed in Section 5. We conclude and propose our future work in Section 6.

2. RELATED WORK

The authors in [11] introduced a closed frequent graph mining algorithm to extract frequent graph patterns from the linked data. Then they extracted features from the entities of the graph patterns to detect hidden *owl:sameAs* links or hidden relations in geographic data sets such as the U.S. Census, Geonames, DBpedia, and World Factbook. They applied a supervised learning method on the frequent graph patterns to find useful attributes that link instances. However, their approach only focused on geographic data and did not discuss further about what kind of features are important for finding the hidden links.

The analysis of the basic properties of the SameAs network, the Pay-Level-Domain network, and Class-Level Similarity network are discussed in [6]. They compared the five most frequent types to examine how the data publishers are connected. However, only considering types is not enough to detect related instances, which normally contain many data type properties.

A debugging method for mapping lightweight ontologies is introduced in [13]. They applied machine learning method to determine the disjointness of any pair of classes, with the features of the taxonomic overlap, semantic distance, object properties, label similarity, and WordNet similarity. Although their method performs better than other ontology matching systems, their method is limited to the expressive lightweight ontologies.

The authors in [14] and [20] proposed constructing an intermediate layer ontology by an automatic alignment method on linked data. However the authors of [14] only focused on the class-level while the authors in [20] only considered property-level alignment. Considering both class and property level can better analyze the interlinked instances.

In contrast to the approaches adopted in the related research described above, our approach considers ontology alignment at both class and property level. Our method is applicable to linked data sets from any domain and large ontologies published in the LOD. Although we need minor manual revision on the automatically created integrated ontology, the ontology integration method successfully retrieves related ontology classes and properties that are critical for interlinking related instances.

3. OUR APPROACH

Figure 1 shows the architecture of our approach. The first step is the graph pattern extraction from SameAs Graphs constructed from the linked data sets in the LOD cloud. The second step is the <Predicate, Object> collection, which classifies them into five different types: Class, String, Date, Number, and URI. The next step is the related classes and properties grouping with ontology alignment methods. The

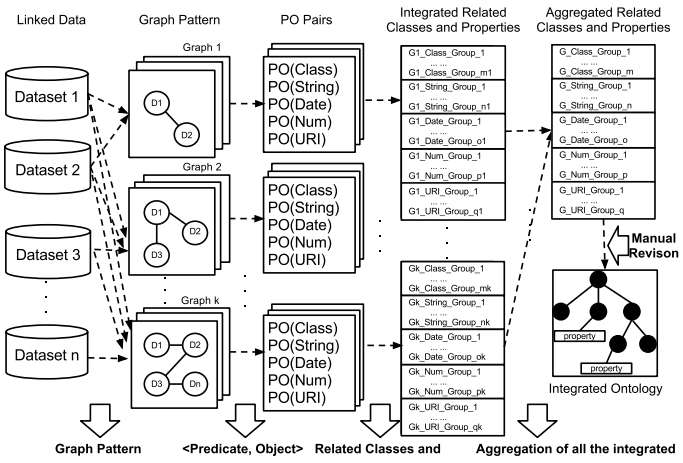

Figure 1: Architecture of our approach.

fourth step is the aggregation of all the integrated classes and properties to construct an integrated ontology and the manual revision on the integrated ontology is the last step. Through all these processing steps, we construct an integrated ontology from the linked data sets. In the following, we describe each step of our approach in details.

3.1 Graph Pattern Extraction

The instances which refer to the same thing are interlinked by *owl:sameAs* in the LOD cloud. We collect all the instances that have *owl:sameAs* (SameAs) links and construct graph patterns in order to observe the differences of the graph patterns by analyzing the integrated classes and properties. The terms of SameAs Triple, SameAs Instance, SameAs Graph and graph pattern are defined as follows:

Definition 1. ***SameAs Triple.*** A SameAs Triple is an RDF triple that contains the *owl:sameAs* predicate.

Definition 2. ***SameAs Instance.*** A SameAs Instance is a tuple SI = (U, T, L), where U is the URI of the instance that appears in a SameAs Triple <U, *owl:sameAs*, X> or <X, *owl:sameAs*, U>, T is the number of the distinct SameAs Triples that contain U, L is the label of the data set which includes the instance.

Definition 3. ***SameAs Graph.*** An undirected SameAs Graph SG = (V, E, I), where V is a set of vertices, which are the labels of data sets that include linked SameAs Instances, $E \subseteq V \times V$ is a set of edges, and I is a set of URIs of the interlinked SameAs Instances.

Here, we give an example of the SameAs Graph constructed with the interlinked instances of "Austria". The SameAs Graph $SG_{Austria}$ = (V, E, I), where V = {M, D, G, N}, E = {(D,G), (G,N), (G,M)}, I = {mdb-country:AT[1], db:Austria[2],

[1]mdb-country: http://data.linkedmdb.org/resource/country/
[2]db: http://dbpedia.org/resource/

Algorithm 1: SameAs Graphs extraction.

Input : $IndexSI$: Indexed SameAs Instances
Output: $SetSG$: A set of SameAs Graphs
Variable: $LinkedInst$: Linked instances of SI
begin
  $SetSG \longleftarrow \emptyset$
  for $SI \in IndexSI$ **do**
    if $SI.visited = false$ **then**
      $SG \longleftarrow \emptyset$
      $SI.visited \longleftarrow true$
      $SG \longleftarrow$ SearchGraph(SG, SI)
      $SetSG.put(SG)$
  return $SetSG$

SearchGraph(SG, SI)
begin
  $SG.V.put(SI.L)$
  $SG.I.put(SI.U)$
  $LinkedInst \longleftarrow \emptyset$
  for $X \in$ (<SI, *owl:sameAs*, X> $\cup$ <X, *owl:sameAs*, SI>) **do**
    if $X.visited = false$ **then**
      $LinkedInst.put(X)$
  for $X \in LinkedInst$ **do**
    if $X.visited = false$ **then**
      $SG.E \longleftarrow (SI, X)$
      $X.visited \longleftarrow true$
      $SG \longleftarrow$ SearchGraph(SG, X)
  return SG

geo:2782113[3], nyt:662210581613183736014}. The M, D, G, and N represent the labels of data sets LinkedMDB, DBpedia, Geonames, and NYTimes, respectively.

In order to collect all the SameAs Graphs in the linked data sets, we extract all the SameAs Instances and rank them based on the T, which is the number of distinct SameAs Triples. The ranked SameAs Instances are indexed in the $IndexSI$, from which we extract a set of SameAs Graphs $SetSG$ from the linked data sets with Algorithm 1.

In Algorithm 1, for each unvisited SI in the $IndexSI$, we create an empty SameAs Graph SG and construct a SameAs Graph using the function SearchGraph(SG, SI). We put the L and U of SI into the SG, and then search for the instances linked with SI and put them in the $LinkedInst$. For each unvisited instance X in the $LinkedInst$, we put the edge (SI, X) into the SG, and mark X as visited. Then we recursively search with SG and X, and assign the returned value to SG, until all the instances in the $LinkedInst$ are visited. The function SearchGraph(SG, SI) returns a SameAs Graph SG and all the SameAs Graphs constructed with the instances in the $IndexSI$ are stored in the $SetSG$.

Definition 4. We say two SameAs Graphs SG_i and SG_j have the same ***graph pattern (GP)***, if $SG_i.V = SG_j.V$ and $SG_i.E = SG_j.E$.

[3]geo: http://sws.geonames.org/
[4]nyt: http://data.nytimes.com/

Table 1: Type classification.

Type	Built-in Data Types
String	http://www.w3.org/2001/XMLSchema#string
Date	http://www.w3.org/2001/XMLSchema#date http://www.w3.org/2001/XMLSchema#gYear http://www.w3.org/2001/XMLSchema#gMonthDay
Number	http://www.w3.org/2001/XMLSchema#integer http://www.w3.org/2001/XMLSchema#float http://www.w3.org/2001/XMLSchema#double http://www.w3.org/2001/XMLSchema#int
URI	http://www.w3.org/2001/XMLSchema#anyURI

Table 2: PO pairs and types for $SG_{Austria}$

Predicate	Object	Type
rdf:type	owl:Thing	Class
rdf:type	db-onto:Place	Class
rdf:type	db-onto:PopulatedPlace	Class
rdf:type	db-onto:Country	Class
rdfs:label	"Austria"@en	String
db-onto:wikiPageExternalLink	http://www.austria.mu/	URI
db-prop:populationEstimate	8356707	Number
.....	.....	.....
geo-onto:name	Austria	String
geo-onto:alternateName	"Austria"@en	String
geo-onto:alternateName	"Republic of Austria"@en	String
geo-onto:featureClass	geo-onto:A	Class
geo-onto:featureCode	geo-onto:A.PCLI	Class
geo-onto:population	8205000	Number
.....	.....	.....
rdf:type	mdb:country	Class
mdb:country_name	Austria	String
.....	.....	.....
skos:inScheme	nyt:nytd_geo	Class
skos:prefLabel	"Austria"@en	String
nyt-prop:first_use	2004-10-04	Date

All the same SameAs Graphs consist a graph pattern, from which we can detect related classes and properties. The next step is to collect useful information from each graph pattern, which is described in details in the following section.

3.2 <Predicate, Object> Collection

An instance is described by a collection of RDF triples in the form of <subject, predicate, object>, where subject is the URI of an instance. Since a SameAs Graph contains linked instances, we collect all the <Predicate, Object> pairs of the instances as the content of the SameAs Graph. Hereafter, we use PO to represent the <Predicate, Object>.

We classify the PO pairs into five different types: Class, String, Date, Number, and URI. The Class is defined by the predicates rdf:type[5] and skos:inScheme[6]. The other four types of PO pairs can be identified from the object values with the built-in data types as listed in Table 1. Usually the data types of objects are followed by the symbol "^^". If the data types are not given expressively in the RDF triples, we analyze the object values in the following way:

Number: The value consists of all numbers.
URI: Starts with "http://".
String: All the other values that can not be classified.

Table 2 shows an example of the collected PO pairs and the types of PO pairs in $SG_{Austria}$. The first two columns list PO pairs and the last column lists the types of the PO pairs.

[5]rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
[6]skos: http://www.w3.org/2004/02/skos/core#

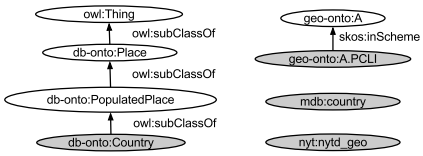

Figure 2: Collected classes from $SG_{Austria}$.

The content of $SG_{Austria}$ in Table 2 is used in the next step to discover related classes and properties.

3.3 Related Classes and Properties Grouping

In order to find related classes and properties for each graph pattern, we perform different methods on different types of *PO* pairs. In the following we describe how to discover related classes by checking subsumption relations and how to find related properties using ontology alignment methods.

3.3.1 Related Classes Grouping

The ontology classes have subsumption relations such as owl:subClassOf and skos:inScheme. The two triples $<C_1$, owl:subClassOf, $C_2>$ and $<C_1$, skos:inScheme, $C_2>$ mean that the concept of C_1 is more specific than the concept of C_2. In order to identify the types of linked instances, we mainly focus on the most specific classes from the linked instances by tracking the subsumption relations. The classes and subsumption relations consist a tree, and the most specific classes are called a leaf nodes in the tree. A class which has no subsumption relation is considered as a leaf node.

From each SameAs Graph, we construct trees with the classes extracted from the *PO* pairs classified in the type Class as listed in Table 2. Then we group the most specific classes, which are the leaf nodes in the trees. For example, Figure 2 is a collection of classes extracted from $SG_{Austria}$, which are connected with the subsumption relations owl:subClassOf and skos:inScheme. The grey nodes, namely, mdb:country[7], db-onto:Country[8], geo-onto:A.PCLI[9], and nyt:nytd_geo are the leaf nodes in the Figure 2. Therefore, we can group these four classes, which are used for describing countries in different data sets.

3.3.2 Related Properties Grouping

We perform exact and similarity matching methods on the collected *PO* pairs to find out related properties, which are also used as predicates in the *PO* pairs. This is an extension of the similarity matching method introduced in [20].

- Group Predicates by Exact Matching

  The first step in the predicate grouping is to create the initial sets of *PO* pairs by exact string matching method. For each classified type of *PO* pairs, we perform a pairwise comparison of PO_i and PO_j and create the initial sets $S_1, S_2, \ldots, S_k$ by checking whether they have identical predicates or objects. Here, S is a set of *PO* pairs.

  For example, in Table 2, the predicates rdfs:label[10], mdb:country_name, skos:prefLabel, geo-onto:name, and geo-onto:alternateName have the same value "Austria", and the predicate geo-onto:alternateName has another object, "Republic of Austria"@en. Hence, these six *PO* pairs are grouped together to create an initial set. After creating initial sets by exact matching, we create an initial set for each *PO* pair that has not yet been grouped. For instance, nyt-prop:first_use[11] is in an initial set by itself because no predicate has the same object "2004-10-04" and no identical predicate exists in the data.

- Similarity Matching on *PO* Pairs

  Identical predicates of *PO* pairs that are classified into Date and URI can be discovered by exact matching. However, for the types of Number and String, the objects may be various in different data sets. For instance, the population of a country may be slightly different in diverse data sets and the values in String may have different representations for the same meaning. In order to find out related initial sets, we perform similarity matching on the *PO* pairs of two sets and merge them if the similarity of any two *PO* pairs is higher than the predefined similarity threshold.

  The string-based and knowledge-based similarity matching methods are commonly used to match ontologies at the concept level [7]. In our approach, we adopted three string-based similarity measures, namely, JaroWinkler distance [19], Levenshtein distance, and n-gram, as introduced in [10]. String-based similarity measures are applied to compare the objects of *PO* pairs that are classified in String. $ObjSim(PO_i, PO_j)$ is the similarity of objects between two *PO* pairs calculated as follows:

$$ObjSim(PO_i, PO_j) = \begin{cases} 1 - \frac{|O_{PO_i} - O_{PO_j}|}{O_{PO_i} + O_{PO_j}} & \text{if Number} \\ StrSim(O_{PO_i}, O_{PO_j}) & \text{if String} \end{cases}$$

  where $StrSim(O_{PO_i}, O_{PO_j})$ is the average of the three string-based similarity values and the term O_{PO} indicates the object of *PO*.

  We observed that the terms of predicates written in different languages or written in synonym can not be mapped in the exact matching step if the object values are not exactly the same. Therefore, knowledge-based similarity matching is required to group semantically similar predicates. We adopted nine knowledge-based similarity measures [16], namely, LCH, RES, HSO, JCN, LESK, PATH,WUP, LIN, and VECTOR, which are based on WordNet (a large lexical database of English [8]). The knowledge-based similarity measures are applied to compare the pre-processed terms of predicates, because most of the terms have semantic meanings that can be recognized as a concept.

  To extract the concepts of predicate terms, we preprocess the predicates of *PO* pairs by performing natural language processing (NLP), which includes tokenizing terms, removing stop words, and stemming terms using the porter stemming algorithm [17]. NLP is a

[7] mdb: http://data.linkedmdb.org/resource/movie/
[8] db-onto: http://dbpedia.org/ontology/
[9] geo-onto: http://www.geonames.org/ontology#
[10] rdfs:http://www.w3.org/2000/01/rdf-schema#
[11] nyt-prop: http://data.nytimes.com/elements/.

key method for the data pre-processing phase, in which terms are extracted from ontologies; this method helps improve the performance of ontology building [5].

$PreSim(PO_i, PO_j)$ is the similarity of predicates between PO_i and PO_j, which is calculated using the formula:

$$PreSim(PO_i, PO_j) = WNSim(T_{PO_i}, T_{PO_j})$$

where the term T_{PO} indicates the pre-processed terms of the predicates in PO and $WNSim(T_{PO_i}, T_{PO_j})$ is the average of the nine applied WordNet-based similarity values. For the WordNet-based similarity measures, we do not count on the term pairs that have no similarity value returned from the WordNet-based similarity measures.

$Sim(PO_i, PO_j)$ is the similarity between PO_i and PO_j calculated as follows:

$$Sim(PO_i, PO_j) = \frac{ObjSim(PO_i, PO_j) + PreSim(PO_i, PO_j)}{2}$$

If $Sim(PO_i, PO_j)$ is higher than the predefined similarity threshold, we consider that these two PO pairs are similar and merge two sets Sm and S_n which contain PO_i and PO_j, respectively. In this work, we set the default similarity threshold to 0.5. After comparing all the pairwise initial sets, we remove the initial set S_i if it has not been merged during this process and has only one PO pair.

- Refine Sets of PO Pairs

  The final step of related properties grouping is to split the predicates of each S_i according to the relation rdfs:domain [3]. Even though the objects or terms of predicates are similar, the predicates may belong to different domains. For further refinement, we determine the frequency of each pruned S_i in all of the data and keep any S_i that appears with a frequency that is higher than the predefined frequency threshold. In order to avoid losing important predicates, we set the frequency threshold as small as possible based on the number of SameAs Graphs in each graph pattern using the following formula:

  $$freq\_threshold_{GP_i} = \log(NumOfSG_{GP_i})$$

  where the $NumOfSG_{GP_i}$ is the number of SameAs Graphs in the graph pattern GP_i.

From the sets of PO pairs of each graph pattern, we collect the classes and properties from each set and construct integrated groups of classes and properties that are classified in Date, String, Number, and URI.

3.4 Aggregation of All the Integrated Classes and Properties

In this step, we aggregate the integrated classes and properties from all the graph patterns to construct an integrated ontology. During the aggregation, we keep the rdfs:domain information of properties, which infers which class the groups of properties belong to.

Then we construct an integrated ontology based on the integrated groups of classes and properties as follows:

- Select Terms for Integrated Ontology

  To perform automatic term selection, we pre-process all the terms of the classes and properties in each set by tokenization, stop words removal, and stemming. We keep the original terms because sometimes a single word is ambiguous when it is used to represent a set of terms. For example, "area" and "areaCode" have different meanings but may have the same frequency because the former is extracted from the latter. Hence, when two terms have the same frequency, we choose the longer one. The predicate ex-onto:ClassTerm is designed to represent a class, where the "ClassTerm" is automatically selected and starts with a capitalized character. The predicate ex-prop:propTerm is designed to represent a property, where the "propTerm" is automatically selected and starts with a lowercase character.

- Construct Relations

  We designed the predicate ex-prop:hasMemberClasses to link the integrated classes with ex-onto:ClassTerm, and use the predicate ex-prop:hasMemberDataTypes to link the integrated properties with ex-prop:propTerm.

  We use the relation ex-prop:hasMemberClasses and ex-prop:hasMemberDataTypes instead of the existing owl:equivalentClass or owl:equivalentProperty in order to reduce the number of triples to connect the related classes and properties. For example, if the number of integrated classes or properties in a set S_i is n, we need $n * (n - 1)$ triples to connect all the pairs of classes or properties using the owl:equivalentClass or owl:equivalentProperty. However, in our approach we only need n triples with ex-prop:hasMemberClasses and ex-prop:hasMemberDataTypes.

- Construct Integrated Ontology

  An integrated ontology is automatically constructed with the integrated sets of related classes and properties, selected terms of the ClassTerm and propTerm, and the designed relations ex-prop:hasMemberClasses and ex-prop:hasMemberDataTypes.

3.5 Manual Revision

The automatically integrated ontology includes related classes and properties from different data sets. However, not all the terms of classes and properties are properly selected, and there are some missing statements of rdfs:domain. Hence, we need experts to work on revising the integrated ontology by choosing proper terms, by adding domain information for each group of properties, and by amending each group of classes and properties.

4. EXPERIMENTS

In this section, we briefly introduce the experimental data sets used in our experiments. Then we analyze the graph patterns that are extracted from the linked instances, and analyze the characteristics of interlinked instances with the integrated ontologies at a class-level. The analysis at a property-level is also discussed, from which we can detect standard properties.

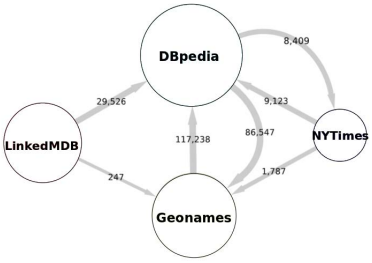

Figure 3: SameAs links between data sets.

4.1 Experimental Data

We used the following four data sets in the LOD cloud to evaluate our approach.

DBpedia is a core cross-domain data set that describes over 3.5 million things including persons, places, music albums, movies, video games, organizations, species, and diseases. DBpedia has more than 232 million RDF triples and more than 8.9 million distinct URIs.

Geonames is a data set that is categorized in the geographic domain and contains more than 7 million unique URIs that represent geographical information on places.

NYTimes is a small data set that consists of 10,467 subject headings, where 4,978 are about people, 1,489 are about organizations, 1,910 are about locations, and 498 are about descriptors.

LinkedMDB is the Linked Movie DataBase, which contains high-quality interlinks to movie-related data in the LOD cloud as well as links to movie-related web pages. LinkedMDB consists of approximately 6 million RDF triples and more than 0.5 million entities.

Figure 3 shows the SameAs links connecting the above four data sets, plotted using Cytoscape [18]. In this figure, the size of a node is determined by the total number of distinct instances in a data set on a logarithmic scale. The thickness of an arc is determined by the number of sameAs links as labeled on each arc on a logarithmic scale.

4.2 Graph Patterns of Linked Instances

In this experiment, we analyze the graph patterns extracted with the experimental data sets to observe how the data sets are interlinked. We retrieved 13 different graph patterns from the SameAs Graphs as listed in Figure 4. The labels of nodes M, D, N, and G represent the data sets LinkedMDB, DBpedia, NYTimes, and Geonames, respectively. The number on the right side of each graph pattern is the number of SameAs Graphs, which is used to decide the frequency threshold for refining sets of PO pairs.

As shown in Figure 4, the top 3 most frequent graph patterns are GP_1, GP_2, and GP_3, which contain only two nodes, (Geonames and DBpedia), (LinkedMDB and DBpedia), and (NYTimes and DBpedia), respectively. Four graph patterns GP_4, GP_5, GP_7, and GP_8 contain nodes from Geonames, DBpedia, and NYTimes with different set of edges. The

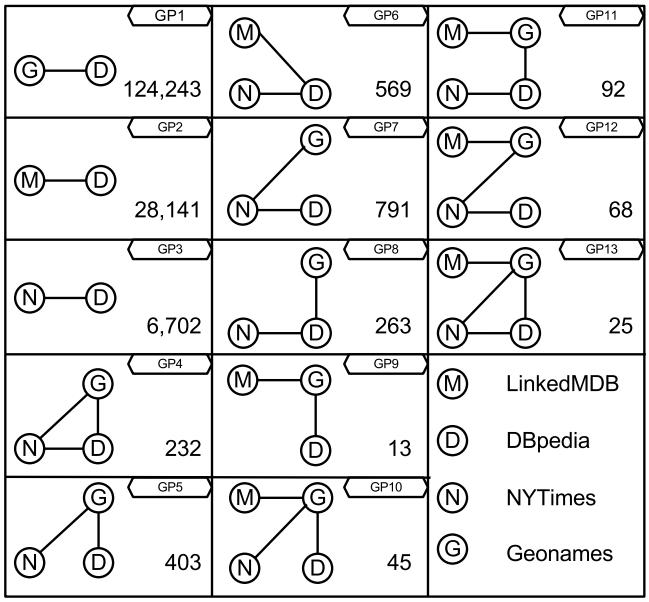

Figure 4: SameAs graph patterns.

Table 3: Characteristics of graph patterns.

Class Type	Graph Pattern
Actor	GP_2, GP_6
Person(Athlete, Politician, etc)	GP_3
Organization/Agent	GP_1, GP_3, GP_8
Film	GP_2
City/Settlement	GP_1, GP_4, GP_5, GP_7, GP_8
Country	GP_9, GP_{10}, GP_{11}, GP_{12}, GP_{13}
Place(Mountain, River, etc)	GP_1, GP_3, GP_7

other graph patterns with three nodes are GP_6 and GP_9, where GP_6 contains LinkedMDB, NYTimes, and DBpedia, while GP_9 contains LinkedMDB, Geonames, and DBpedia. The graph patterns GP_{10}, GP_{11}, GP_{12}, and GP_{13} contain all the nodes from four different data sets with different set of edges.

4.3 Class-level Analysis

In order to observe what kind of instances are interlinked in each graph pattern in Figure 4, we analyze the integrated classes in the third step of our approach. We classified the characteristics of graph patterns into seven different types: Actor, Person (except Actor), Organization/Agent, Film, City/Settlement, Country, and Place as shown in Table 3. The first column in Table 3 lists the class types and the second column lists the graph patterns which have the corresponding class type. In total, we created 48 integrated classes from the experimental data sets.

The type of instances shared by the four data sets is Country. The integrated class for Country is ex-onto:Country, which contains geo-ontoA.PCLI, geo-onto:A.PCLD, mdb:country, db-onto:Country, and nyt:nytd_geo. As we can see in Table 3, the graph patterns GP_9, GP_{10}, GP_{11}, and GP_{12} are subgraphs of GP_{13}. However, GP_{13} is not a complete graph and there are missing links between (M, N) and (M, D). Hence, with the ex-onto:Country, we can link missing links of countries among these four data sets.

Table 4: Predicates grouped in ex-prop:birthDate.

Property	Number of Instances
db-onto:birthDate	287,327
db-prop:datebirth	1,675
db-prop:dateofbirth	87,364
db-prop:dateOfBirth	163,876
db-prop:born	34,832
db-prop:birthdate	70,630
db-prop:birthDate	101,121

Furthermore, with the integrated classes, we can discover missing class information of linked instances. For instance, the db:Shingo_Katori is only described as a musical artist, but in fact he is also an actor and the DBpedia instance has a link to the mdb-actor:27092[12]. Hence, we should add the class db-onto:Actor to the instance db:Shingo_Katori, because all the instances linked with mdb-actor should be an actor unless it is a wrong linkage.

The main classes of each data set can be recognized from the integrated classes. The NYTimes is mainly categorized into person, organization, and place. The LinkedMDB is mainly categorized into movie, actor, and country. The main classes appeared in the graph patterns for the Geonames are A(country, administrative region), P (city, settlement), T (mountain), S (building, school), and H (Lake, river), where A, P, T, S, and H are the feature classes used in the Geonames ontology. The main classes of DBpedia linked with other data sets are person (artist, politician, athlete), organization (company, educational institute, sports team), work (film), and place (populated place, natural place, architectural structure).

4.4 Property-level Analysis

Only considering related classes from different data sets is not enough to detect identical instances. Hence, we analyze the properties of integrated ontologies to find out what kind of properties are critical to identify related instances from different data sets. We integrated 38 groups of properties from the graph patterns in Figure 4. However, 15 groups of the properties have no information about rdfs:domain; it is difficult to find which instances should be described with the properties without rdfs:domain. Hence, missing domain information is added to the groups of properties during the manual revision. The manual revision on the integrated ontology is light work, which only takes few hours.

If we want to link a person in different data sets, we can combine the classes which indicate a person with the properties such as the birth date, the place of birth, and the name, etc. However, there are various properties exist to describe the same kind of property. For example, we integrated 7 different properties that indicate the birthday of a person as listed in Table 4. However, only the property "db-onto:birthDate" is defined with the domain db-onto:Person and has the highest frequency of usage that appeared in 287,327 DBpedia instances. The second column in Table 4 represents the number of distinct instances use the property listed in the first column. From the definitions of the properties and the number of instances which contain the corresponding properties, we can assume that the properties except "db-onto:birthDate" are mistakenly used when the data providers publish the DBpedia data. The "db-onto:birthDate" is well defined with rdfs:domain and has the highest usage in the DBpedia instances. Therefore, we can suggest "db-onto:birthDate" as the standard property to represent the birthday of a person, and correct the other properties with this standard property.

[12]mdb-actor: http://data.linkedmdb.org/resource/actor/

Table 5: Links not included in related work.

Links between Data Sets	Graph Pattern
Geonames and NYTimes	GP_5, GP_7, GP_{10}, GP_{12}
Geonames and LinkedMDB	GP_9, GP_{10}, GP_{11}, GP_{12}, GP_{13}

Other than recommending standard properties, we also successfully integrated different property descriptions from diverse data sets. For instance, properties geo-onto:population, mdb:country_population, db-onto:populationTotal and other nine DBpedia properties are integrated into the property ex-prop:population. By combining the ex-onto:Country and ex-prop:population, we can detect the same country or countries with similar population.

5. COMPARISON WITH RELATED WORK

In this section, we compare our approach with the related research work introduced in [20]. In the related research, they used DBpedia as a hub data set, and collected all the SameAs links that contain instances of DBpedia. Hence, the authors were not able to collect the links between other data sets if they are not directly connected with DBpedia. The links that can not be found in their work, but can be collected with our approach are listed in Table 5. The links between Geonames and NYTimes in the graph patterns GP_5, GP_7, GP_{10}, GP_{12}, and the links between Geonames and LinkedMDB in GP_9, GP_{10}, GP_{11}, GP_{12}, GP_{13} are retrieved with our approach, but can not be retrieved in [20].

Furthermore, they did not identify the types of objects and considered all the object values as String during the predicates grouping step. Hence, even the numbers are compared with string-based similarity matching and it results in a wrong similarity value. For example, the similarity between 5000 and 4999 is 0 in [20], but with our approach the similarity is approximately 1. In our work, we classified the types of *PO* pairs into Class, String, Date, Number, and URI. Then we performed different methods to integrate related classes and properties from the *PO* pairs. For instance, we discover the most specific classes by tracking subsumption relations to find related classes. In order to detect related properties, we applied only exact matching on the types of Date and URI, and applied similarity matching on the types of Number and String in a different way. Furthermore, we analyzed the linked instances at both class and property level, while the authors in [20] only analyzed at a property level.

The integrated Mid-Ontology in [20] was constructed with properties only, without any relations and classes. However, in our approach, we also integrated classes of different ontologies and linked the integrated properties with the classes. Hence, it is easier to observe the characteristics of the interlinked instances with the integrated classes. Furthermore, the relations between properties and classes indicate what kind of core properties can well describe the instances and

help detecting related instances.

Moreover, many important links were not found in the related work, because they used only one common threshold to find frequent groups of related predicates. In our approach, the frequency threshold is calculated based on the number of SameAs Graphs in each graph pattern. Therefore, we can retrieve more properties that were not discovered in [20]. For example, the runtime of a movie extracted from the links between LinkedMDB and DBpedia are mdb:runtime, db-prop:runtime[13], db-onto:runtime, and db-onto:Work/runtime. This is retrieved with our approach, but can not be retrieved in [20]. With the approach introduced in [20], only 105 predicates are retrieved and classified into 22 groups, while with our approach 367 properties are integrated into 38 groups.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a graph-based heterogeneous ontology integration approach, which can discover the core classes and properties to link instances from various data sets. From the extracted graph patterns, we detect related classes and properties which are classified into different data types: String, Date, Number, and URI. Similar classes are integrated by tracking subsumption relations and different similarity matching methods are applied on different types of *PO* pairs to retrieve similar properties. We automatically integrate related classes and properties for each graph pattern, and then aggregate all of them to construct an integrated ontology. Minor manual revision by an expert is required to add missing domain information of properties and to correct groups of integrated classes and properties. Experimental analysis show that our approach can retrieve more information than in the research [20], and we can understand the characteristics of interlinked instances both at class and property level. Furthermore, with the grouped properties, we can recommend standard properties that should be used for publishing data and recommend revision of mistakenly used properties. By combining related classes and properties from various data sets, we can find missing SameAs links. The semi-automatically constructed integrated ontology can solve the ontology heterogeneity problem, and help Semantic Web application developers easily understand the relations between different ontologies without manual inspection.

In future work, we plan to apply our approach with more data sets publicly available in the LOD cloud. The scalability is a challenging problem when dealing with the huge data sets in the LOD. The most popular method to deal with big data is the MapReduce method, which is a framework for processing distributed large-scale data with a large number of computers [12]. We plan to extend our current research with the MapReduce method to solve both the ontology heterogeneity and data scalability problem.

7. REFERENCES

[1] T. Berners-Lee. *Linked Data - Design Issues*, 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[2] 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.

[3] D. Brickley and R. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, 2004. `http://www.w3.org/TR/rdf-schema/`.

[4] N. Choi, I.-Y. Song, and H. Han. A survey on ontology mapping. *ACM SIGMOD Record*, 35:34–41, 2006.

[5] P. Cimiano. *Ontology Learning and Population from Text: Algorithms, Evaluation and Applications*. Springer-Verlag New York, Inc., 2006.

[6] L. Ding, J. Shinavier, Z. Shangguan, and D. L. McGuinness. Sameas networks and beyond: Analyzing deployment status and implications of owl: sameas in linked data. In *Proceedings of the Ninth International Semantic Web Conference*, pages 145–160, 2010.

[7] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag, Heidelberg, 2007.

[8] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[9] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[10] R. Ichise. An analysis of multiple similarity measures for ontology mapping problem. *International Journal of Semantic Computing*, 4(1):103–122, 2010.

[11] N.-T. Le, R. Ichise, and H.-B. Le. Detecting hidden relations in geographic data. In *Proceedings of the 4th International Conference on Advances in Semantic Processing*, pages 61–68, 2010.

[12] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2010.

[13] C. Meilicke, J. Völker, and H. Stuckenschmidt. Learning disjointness for debugging mappings between lightweight ontologies. In *Proceedings of the Sixteenth International Conference on Knowledge Engineering and Knowledge Management*, pages 93–108, 2008.

[14] R. Parundekar, C. A. Knoblock, and J. L. Ambite. Linking and building ontologies of linked data. In *Proceedings of the Ninth International Semantic Web Conference*, pages 598–614, 2010.

[15] S. Pavel and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints), 2011.

[16] T. Pedersen, S. Patwardhan, and J. Michelizzi. Wordnet::similarity: Measuring the relatedness of concepts. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 1024–1025, 2004.

[17] M. F. Porter. An algorithm for suffix stripping. In *Readings in Information Retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., 1997.

[18] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Res*, 13(11):2498–2504, 2003.

[19] W. E. Winkler. Overview of record linkage and current research directions. Technical report, Statistical Research Division U.S. Bureau of the Census, 2006.

[20] L. Zhao and R. Ichise. Mid-ontology learning from linked data. In *Proceedings of the Joint International Semantic Technology Conference*, pages 112–127, 2011.

[13] db-prop: http://dbpedia.org/property/