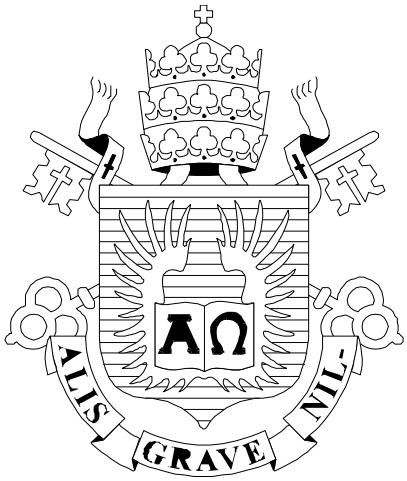

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 12/11

Complex Matching of RDF Datatype Properties

Bernardo Pereira Nunes Alexander Mera
Marco Antonio Casanova Karin Koogan Breitman
Luiz André Portes Paes Leme

Departamento de Informática – PUC-Rio

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 12/11 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena September, 2011

Complex Matching of RDF Datatype Properties

Bernardo Pereira Nunes[1], Alexander Mera[1], Marco Antonio Casanova[1], Karin Koogan Breitman[1], Luiz André Portes Paes Leme[2]

[1]Department of Informatics – PUC-Rio – Rio de Janeiro, RJ – Brazil
{bnunes, acaraballo, casanova, karin}@inf.puc-rio.br

[2]Instituto de Computação – UFF – Rio de Janeiro, RJ – Brazil
lapaesleme@ic.uff.br

Abstract. Property mapping is a fundamental component of ontology matching and yet hardly any technique goes beyond the identification of single property matches. However, real data often requires some degree of composition, trivially exemplified by the mapping of *FirstName, LastName* to *FullName*. Genetic programming offers an alternative, but the solution space is so large that the required computation effort would be prohibitive.

This paper proposes a two-phase instance-based technique for complex datatype property matching. In the first phase, the technique computes the estimate mutual information matrix of the property values to (1) find simple, 1:1 matches, and (2) compute a list of possible complex matches. In the second phase, it applies genetic programming to a much reduced search space to find complex matches. The paper concludes with experimental results that illustrate how the technique works and indicate that the technique obtains better results that those achieved by separately using the estimate mutual information matrix or genetic programming.

Keywords: Ontology Matching, Genetic Programming, Mutual Information.

Resumo. Mapeamento entre propriedades é um componente fundamental no alinhamento de ontologias. No entanto, existem poucas ferramentas que vão além de mapeamentos simples entre propriedades. Porém, dados reais muitas vezes requerem algum grau de composição, que vão de mapeamentos triviais como, por exemplo, da composição de *Nome* e *Sobrenome* para *Nome Completo*. Programação genética oferece uma alternativa, mas o espaço de soluções é tão grande que o esforço computacional necessário seria proibitivo.

Este trabalho propõe uma técnica, dividida em duas fases, para o mapeamento de propriedades complexas baseada em instâncias. Na primeira fase, calcula-se a matriz de informação mútua estimada dos valores das propriedades para (1) encontrar mapeamentos simples e biunívocos, e (2) computar uma lista de possíveis mapeamentos complexos. Na segunda fase, já com o espaço de busca reduzido na primeira fase, aplica-se programação genética para encontrar mapeamentos complexos. Por fim, este trabalho apresenta resultados experimentais que ilustram a técnica e indicam que ela leva a resultados melhores do que aqueles obtidos utilizando-se separadamente a matriz de informação mútua estimada e programação genética.

Palavras-chave: Mapeamento de Ontologias, Programação Genética, Informação Mútua.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

Ontology matching is a fundamental problem in many applications areas [6]. Using OWL concepts, by *datatype property matching* we mean the special case of matching datatype properties from two classes.

Very briefly, an *instance* of a datatype property p is a triple of the form (s,p,l), where s is a resource identifier and l is a literal. A *datatype property matching* from a *source* class S to a *target* class T is a partial relation μ between sets of datatype properties of S and sets of datatype properties of T. We say that a match $(\boldsymbol{A},\boldsymbol{B}) \in \mu$ is *m:n* iff $\boldsymbol{A}$ and $\boldsymbol{B}$ contain m and n properties, respectively. A match $(\boldsymbol{A},\boldsymbol{B}) \in \mu$ should be accompanied by one or more *datatype property mappings* that indicate how to construct instances of the properties in $\boldsymbol{B}$ from instances of the properties in $\boldsymbol{A}$. A match $(\boldsymbol{A},\boldsymbol{B}) \in \mu$ is *simple* iff it is *1:1* and the mapping is a simple translation; otherwise, it is *complex*.

In this paper, we introduce a two-phase, instance-based datatype property matching technique that is able to find complex *n:1* datatype property matches and to construct the corresponding property mappings. The technique extends the ontology matching process described in [9] to include complex matches between sets of datatype properties and is classified as instance-based since it depends on sets of instances.

In more detail, given two sets, $\boldsymbol{s}$ and $\boldsymbol{t}$, that contain instances of the datatype properties of the source class S and the target class T, respectively, the first phase of the technique constructs the Estimated Mutual Information matrix [8,9] of the datatype property instances in $\boldsymbol{s}$ and the datatype property instances in $\boldsymbol{t}$, which intuitively measures the amount of related information of the observed property instances. This phase possibly identifies simple datatype property matches. For example, it may detect that the *eMail* datatype property of one class matches the *ElectronicAddress* datatype property of the other class. The first phase may also suggest, for the second phase, sets of datatype properties that may match in more complex ways, thereby reducing the search space.

The second phase uses a genetic programming approach [7] to find complex *n:1*datatype property matches. For example, it may discover that the *FirstName* and *LastName* datatype properties of the source class matches the *FullName* datatype property of the target class, and return a property mapping function that concatenates the values of *FirstName* and *LastName* (of the same class instance) to generate the *FullName* value. The reason for adopting genetic programming is two-fold: it reduces the cost of traversing the search space; and it may be used to generate complex mappings between datatype property sets.

We also present an example of the technique using real-world data. The results show that the technique is useful in finding matches between sets of datatype properties with high accuracy, and that it improves the Estimated Mutual Information and the Genetic Programming approaches, when they are separately applied.

The problem of finding complex matches between sets of datatype properties should not be underestimated since the search space is typically quite large. The main contribution of this paper lies in proposing a two-phase technique that deals with the problem of finding complex matches by: (a) using the Estimated Mutual Information matrix (in Phase 1) as a pre-processing stage in which to limit the sets of properties that are candidates for complex matches; (b) adopting a genetic programming strategy to generate complex property mappings. Furthermore, we show empirical evidence that

the combination of both approaches, EMI and genetic programming yields better results than using either technique in separate.

As for related work, Wang et. al. [14] address two significant Web database schema matching problems: inter-site and intra-site. However, their approach does not handle complex matches over Web database schema properties. Nguyen et. al. [13] address the schema matching problem in Web forms using a hybrid approach, based on correlation and clustering. Because the approach is cluster-based, it does not perform data transformations, making it impossible to generate complex mappings. Dhamankar et. al. [3] describe the iMap system to semi-automatically discover *1:1* and complex matches between relational schemas. The iMap system is very similar to our approach, in the sense that it also transforms the matching problem into a search problem. Furthermore, it has a predefined module of functions, which can be integrated with our genetic programming approach. However, our technique features a first phase in which we reduce the search space, thus improving the accuracy and run time performance during the second phase of the technique, based on genetic programming.

Carvalho and Frade [2] propose a technique to identify replicas in two different repositories through a deduplication function, created using genetic programming. Using the same deduplication function, they address the schema matching problem. However, their approach is expensive and tries to find matches between all combinations of the available properties of both schemas. By contrast, the first phase of our technique reduces the search space of the genetic programming strategy used on the second phase, resulting in a faster and more accurate algorithm, as already pointed out. A recent technique, proposed by Blanco et. al. [1], is based on redundancy and, for a specific domain, searches the Web for similar data and uses the data found to create matches between multiple sources. The technique contributes to deduplication and data integration, but it does not deal with complex matches.

Wang et. al. [15] propose a flexible framework that is able to add new approaches for matching entities at the schema level and at the instance level. In this sense, our two-phase instance-based approach could be added to the framework and combined with schema-based matching approaches to discover complex matches. Other approaches adopt machine learning techniques, such as LSD [4], GLUE [5] and Semint [10,11]. Although most of the machine learning techniques shows good results, their accuracy sometimes depends on a non-trivial manual effort, which we avoid by adopting genetic programming.

The remainder of the paper is structured as follows. Section 2 summarizes basic results that we use in the next sections. Section 3 introduces the technique. Section 4 contains an example of the technique. Finally, Section 5 presents the conclusions.

2 Background

2.1 Vocabulary Matching and Concept Mapping

Following Leme et al. [9], we decompose the problem of OWL ontology matching into the problem of vocabulary matching and the problem of concept mapping. In this section, we briefly review these concepts and extend them to account for complex property matching. In what follows, let $\boldsymbol{S}$ and $\boldsymbol{T}$ be two OWL ontologies, and $\boldsymbol{V}_S$ and $\boldsymbol{V}_T$ be their vocabularies, respectively. Let $\boldsymbol{C}_S$ and $\boldsymbol{C}_T$ be the sets of classes and $\boldsymbol{P}_S$ and $\boldsymbol{P}_T$ be the sets of properties in $\boldsymbol{V}_S$ and $\boldsymbol{V}_T$, respectively.

An *instance* of a class c is a triple of the form $(s,\texttt{rdf:type},c)$, an *instance* of an object property p is a triple of the form (s,p,o) and an *instance* of a datatype property d is a triple of the form (s,d,l), where s and o are resource identifiers and l is a literal.

A *vocabulary matching* between $\boldsymbol{S}$ and $\boldsymbol{T}$ is a finite set $\mu \subseteq \boldsymbol{V}_S \times \boldsymbol{V}_T$. Given $(v_1,v_2) \in \mu$, we say that (v_1,v_2) is a *match* in μ and that μ *matches* v_1 *with* v_2; a *property (or class) matching* is a matching defined only for properties (or classes).

A *concept mapping from* $\boldsymbol{S}$ *to* $\boldsymbol{T}$ is a set of transformation rules that map instances of the concepts of $\boldsymbol{S}$ into instances of the concepts of $\boldsymbol{T}$.

In this paper, we extend vocabulary matchings to also include pairs of the form $(\boldsymbol{A},\boldsymbol{B})$ where $\boldsymbol{A}$ and $\boldsymbol{B}$ are sets of datatype properties in $\boldsymbol{P}_S$ and $\boldsymbol{P}_T$, respectively. We say that $(\boldsymbol{A},\boldsymbol{B})$ is an *m:n* match iff $\boldsymbol{A}$ and $\boldsymbol{B}$ contain m and n properties, respectively. In this case, a match $(\boldsymbol{A},\boldsymbol{B})$ must be accompanied by *datatype property mappings*, denoted $\mu[\boldsymbol{A},B_i]$, such that $\mu[\boldsymbol{A},B_i]$ is a transformation rule that maps instances of the properties in $\boldsymbol{A}$ into instances of the property B_i, for $i=1,\ldots,n$, where $\boldsymbol{B}=\{B_1,\ldots,B_n\}$. Using "//" to denote string concatenation, the following transformation rule

$$(s,fullName,v) \leftarrow (s,firstName,n),\ (s,lastName,f),\ v = n\ //\ f$$

indicates that the value of the *fullName* property is obtained by concatenating the values of properties *firstName* and *lastName*. We will use the following abbreviated form for mapping rules with the above syntax:

$$\mu[\{firstName,\ lastName\},\ fullName] = \text{"}fullName \leftarrow firstName\ //\ lastName\text{"}$$

As an abuse of notation, when $\boldsymbol{A}$ is a singleton $\{A_1\}$, we simply write $\mu[A_1,B_i]$, rather than $\mu[\{A_1\},B_i]$. Finally, a match $(\boldsymbol{A},\boldsymbol{B})$ is *simple* iff it is *1:1*, that is, of the form $(\{A_1\},\{B_1\})$, and the mapping $\mu[A_1,B_1]$ is the *identity transformation rule*, defined as "$(s,\ B_1\ ,\ l) \leftarrow (s,\ A_1,\ l)$"; otherwise, the match is *complex*.

2.2 An Instance-Based Process for Vocabulary Matching

In this section, we very briefly summarize the instance-based process to create vocabulary matchings introduced in [9].

In outline, the process goes as follows:

(1) Generate a preliminary property matching using similarity functions.
(2) Use the property matching obtained in Step (1) to generate a class matching.
(3) Use the property matching obtained in Step (1) to generate an instance matching.
(4) Use the class matching obtained in Step (2) and the instance matching obtained in Step (3) to generate a refined property matching.

The final vocabulary matching is the result of the union of the class matching obtained in Step (2) and the refined property matching obtained in Step (4).

The intuition used in all steps of that process is that "*two schema elements match iff they have many values in common and few values not in common*", i.e. iff they are similar above a given similarity threshold.

Step (1) generates preliminary 1:1 property matchings based on the intuition that two properties match iff their instances share similar sets of values. In case of string properties, their values are replaced by the tokens extracted from their values. Step (1) provides evidences on class and instance matchings, explored in the next two steps.

Step (2) generates class matchings based on the intuition that two classes match iff their sets of properties are similar. This step uses the property matchings generated in Step (1).

Step (3) generates instance matchings based on the intuition that two instances match iff the values of their properties are similar. However, equivalent instances from different classes may be described by very different sets of properties.

Therefore, extracting values from all of their properties may lead to the wrong conclusion that the instances are not equivalent. Therefore, Leme et al. [9] propose to extract values only from the matching properties of the instances.

3 The Two-Phase Property Matching Technique

In this section, we introduce a technique to partly implement and extend Step 4 of the ontology matching process of Section 2.2 to compute complex *n:1* datatype property matches (the technique does not cover *n:m* matches). The technique comprises two phases: Phase 1 uses Estimated Mutual Information matrices, defined in Section 3.1, to compute *1:1* simple matches, while Phase 2 uses genetic programming to compute complex *n:1* matches, based on the information returned by Phase 1.

3.1 Phase 1: Computing Simple Datatype Property Matches with Estimated Mutual Information

Let $\boldsymbol{p}=(\boldsymbol{p}_1,\ldots,\boldsymbol{p}_u)$ and $\boldsymbol{q}=(\boldsymbol{q}_1,\ldots,\boldsymbol{q}_v)$ be two lists of sets. The *co-occurrence matrix* of $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the matrix $[m_{ij}]$ such that $m_{ij} = |\ \boldsymbol{p}_i \cap \boldsymbol{q}_j\ |$, for $i \in [1,u]$ and $j \in [1,v]$. The *Estimated Mutual Information matrix (EMI)* [8,9] of $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the matrix $[EMI_{pq}]$ such that

$$EMI_{pq} = \frac{m_{pq}}{M}\log\left(M \frac{m_{pq}}{\sum_{j=1}^{v} m_{pj} * \sum_{i=1}^{u} m_{iq}} \right), \text{ where } M = \sum_{i=1}^{u}\sum_{j=1}^{v} m_{ij} \qquad (1)$$

We now adapt these concepts to define Phase 1 of the datatype property matching process. Let S and T be two classes with sets of datatype properties $\boldsymbol{A}=\{A_1,\ldots,A_u\}$ and $\boldsymbol{B}=\{B_1,\ldots,B_v\}$, respectively. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be sets of instances of the properties in $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively ($\boldsymbol{s}$ and $\boldsymbol{t}$ therefore are sets of RDF triples).

Rather than simply using the cardinality of set intersections to define the co-occurrence matrix $[m_{ij}]$, Phase 1 computes $[m_{ij}]$ using *set comparison functions* that take two sets and return a non-negative integer. Such functions play the role of *flexibilization points* of Phase 1, as illustrated in Section 4.1.

The set comparison functions depends on the types of the values of the datatype properties as well as on whether the functions take advantage of instance matches. For example, given a pair of datatype properties A_i and B_j, m_{ij} may be defined as the number of pairs of triples (a,A_i,b) in $\boldsymbol{s}$ and (c,B_j,d) in $\boldsymbol{t}$ such that instances a and c match (or are identical) and the literals b and d are equal (or are considered equal, under a *literal comparison function* defined for the specific datatype of b and d).

Phase 1 proceeds by computing the EMI matrix based on the co-occurrence matrix, as in Eq. (1). Next, it computes a *1:1* matching, μ_{EMI}, between the properties in $\boldsymbol{A}=\{A_1,\ldots,A_u\}$ and those in $\boldsymbol{B}=\{B_1,\ldots,B_v\}$ such that, for any pair of properties A_p and B_q,

$(A_p,B_q) \in \mu_{EMI}$ iff $EMI_{pq} > 0$ and $EMI_{pj} \leq 0$, for all $j \in [1,v]$, with $j \neq q$, and $EMI_{iq} \leq 0$, for all $i \in [1,u]$, with $i \neq p$. Furthermore, Phase 1 assumes that the property mappings, $\mu_{EMI}[A_r, B_s]$, are always the identity function.

Finally, Phase 1 also outputs a list of datatype properties to be considered for complex matching in Phase 2. For the k^{th} column of the EMI matrix, it outputs the pair $(\boldsymbol{A}^k,B_k)$ as a candidate *n:1* complex match, where B_k is the property of T that corresponds to the k^{th} column and $\boldsymbol{A}^k$ is the set of properties A_i of S such that $EMI_{ik} > 0$. Indeed, if $EMI_{ik} \leq 0$, then A_i and B_k have no information in common. However, note that this heuristics does not indicate what is a candidate property mapping $\mu[\boldsymbol{A}^k,B_k]$. This problem is faced in Phase 2.

3.2 Phase 2: Computing Complex Property Matches with Genetic Programming

The second phase of the technique uses genetic programming to create mappings between the properties that have some degree of correlation, as identified in the first phase. Briefly, the process goes as follows.

Recall that *genetic programming* refers to an automated method to create and evolve programs to solve a problem [7]. A *program*, also called an *individual* or a *solution*, is represented by a tree, whose nodes are labeled with functions (concatenate, split, sum, etc) or with values (strings, numbers, etc). New individuals are generated by applying *genetic operations* to the current population of individuals. Note that genetic programming does not enumerate all possible individuals, but it selects individuals that should be bred by an evolutionary process. The *fitness function* assigns a *fitness value* to each individual, which represents how good the individual is compared to others, i.e., the survival probability of the individual in the genetic process.

The process requires two configuration steps, carried out just once. First, certain parameters of the process must be properly calibrated to prevent overfitting problems, to avoid unnecessary runtime overhead, and to help finding good solutions (see also Section 4). Once the parameters are calibrated, the second configuration step is to determine the stop criterion. We opted to stop after a predetermined maximum number of generations and return the best-so-far individual to limit the cost of searching for individuals.

We now show how to use genetic programming to compute complex datatype property matches. As in the previous section, let S and T be two classes with sets of datatype properties $\boldsymbol{A}=\{A_1,\ldots,A_u\}$ and $\boldsymbol{B}=\{B_1,\ldots,B_v\}$, respectively. Let $\boldsymbol{s}$ and $\boldsymbol{t}$ be lists of sets of instances of the properties in $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively.

The genetic programming phase receives as input the candidate matches that Phase 1 outputs and the sets $\boldsymbol{s}$ and $\boldsymbol{t}$. For each input candidate match, it outputs a property mapping $\mu[\boldsymbol{A}^k,B_k]$, if one exists; otherwise it discards the candidate match.

Let $(\boldsymbol{A}^k,B_k)$ be a candidate match output by the first phase, where $\boldsymbol{A}^k$ is a set of properties in $\boldsymbol{A}$ and B_k is a property in $\boldsymbol{B}$. The genetic programming phase first generates a random initial population of candidate property mappings. In each iteration step, it creates new candidate property mappings using genetic operations. It keeps the best-so-far individual, and returns it when the stop criterion is reached.

The process depends on the following specifications (see also Table 2 in Section 4.2 for a concrete example), which should be regarded as flexibilization points.

A candidate property mapping $\mu[\mathbf{A}^k, B_k]$ (the individual in this case) is represented as a tree whose leaves are labeled with the properties in $\mathbf{A}^k$ and whose internal nodes are labeled with *primitive mapping functions*.

The maximum population size, $\sigma_{population}$, is a parameter of the process. The initial population consists of n randomically generated trees, where $n = \sigma_{population}$. Each tree has a maximum height, defined by the parameter σ_{height}, each leaf is labeled with a property from $\mathbf{A}^k$ and each internal node is labeled with a primitive mapping function.

The reproduction operation simply preserves a percentage of the property mappings from one generation to the next, defined by the parameter $\sigma_{reproduction}$.

The crossover operation exchanges subtrees of two candidate property mappings to create new candidate mappings. For example, suppose that $\mathbf{A}^k$={*firstName, middleName, lastName*} and B_k=*fullName* and consider the following two candidate property mappings (which use the concatenation operation, "//", and are represented using the notation adopted in Section 2.1):

$\mu_1[\mathbf{A}^k, B_k]$ = "*fullName* ← *(lastName* // **(*firstName* //*middleName*)**)"
$\mu_2[\mathbf{A}^k, B_k]$ ="*fullName* ← **((*middleName* //*firstName*) //** *lastName)*"

The crossover operation might generate the following two new candidate property mappings (by swapping the sub-expressions in boldface)

$\mu_3[\mathbf{A}^k, B_k]$ = "*fullName* ← *(lastName* // **(*middleName* // *firstName*)**)"
$\mu_4[\mathbf{A}^k, B_k]$ = "*fullName* ← **((*firstName* // *middleName*) //** *lastName)*"

The mutation operation randomly alters a node (labeled with a property or with a primitive mapping function) of a candidate property mapping. For example, the node labeled with "*middleName*" of $\mu_4[\mathbf{A}^k, B_k]$ can be mutated to "*firstName*", resulting in a new candidate property mapping (which is acceptable, but not quite reasonable, since it repeats *firstName*):

$\mu_5[\mathbf{A}^k, B_k]$ = "*fullName* ← *((firstName* // *firstName)* // *lastName)*"

Finally, recall that $\boldsymbol{s}$ and $\boldsymbol{t}$ are lists of sets of instances of the properties in $\mathbf{A}$ and $\mathbf{B}$, respectively. The fitness value of $\mu[\mathbf{A}^k, B_k]$ is computed by applying $\mu[\mathbf{A}^k, B_k]$ to the instances of the properties in $\mathbf{A}^k$ occurring in $\boldsymbol{s}$, creating a new set of instances for B_k, which is then compared with the set of instances of B_k occurring in $\boldsymbol{t}$. As in Section 3.1, the exact nature of fitness function depends on the types of the values of the datatype properties as well as on whether the function takes advantage of instance matches or not (which is possible when implementing Step (4)).

4 An Example

4.1 Phase 1: Computing Simple Property Matches with Estimated Mutual Information

With the help of an example, we illustrate how to implement the two-phase technique. We assume that the implementation is in the context of Step (1) of the process described in Section 2.2, that is, we will not use instance matches. We start with Phase 1, described in Section 3.1.

Table 1. Example Schemas.

P		*Q*	
A_1	*FirstName*	B_1	*FullName* (*FirstName // LastName*)
A_2	*LastName*		
A_3	*eMail*	B_2	*eMail*
A_4	*MaritalStatus*	B_3	*MaritalStatus*
A_5	*Address*	B_4	*FullAddress* (*Address // AddressNumber // AddressComplement // Neighborhood*)
A_6	*AddressNumber*		
A_7	*AddressComplement*		
A_8	*Neighborhood*		
A_9	*City*	B_5	*Place* (*City // State // Country*)
A_{10}	*State*		
A_{11}	*Country*		
A_{12}	*ZIPCode*	B_6	*ZIPCode*
…	…	…	…
A_{24}	*PhoneNumber*	B_{17}	*FullPhoneNumber* (*AreaCode // PhoneNumber*)
A_{25}	*AreaCode*		

The example contains 6,000 real-world records with personal information, modeled by class *P*, with 25 properties. We artificially modified the original dataset, creating a new class *Q*, with 17 properties. Table 1 shows classes *P* and *Q* and which properties or sets of properties match. For example, $\{A_1, A_2\}$ matches B_1.

Recall from Section 3.1 that an implementation of Phase 1 requires defining set comparison functions used to compute the co-occurrence matrix $[m_{ij}]$. We discuss this point in what follows, with the help of the running example.

We assume that all property values are string literals and that we are given two samples, $\boldsymbol{p}$ and $\boldsymbol{q}$, of instances of properties of classes *P* and *Q*, respectively (each with 500 instances).

Leme et al. [9] adopt the cosine similarity function to compare two strings:

$$CosSim(s,t) = \frac{s \bullet t}{\|s\| . \|t\|} \tag{2}$$

where *s* and *t* are the vectors of tokens obtained from the strings; m_{ij} is then computed as the number of (string) values of triples for property A_i in $\boldsymbol{p}$ whose cosine distance to values of instances for property B_j in $\boldsymbol{q}$ is above a given threshold ($\alpha = 0.8$ in [9]).

Figure 1 shows the co-occurrence matrix computed as just described. Note that m_{43}=164,826, which is high because the values of A_4 and B_3 come from a controlled vocabulary with a small number of terms (not indicated in Table 1). By contrast, m_{32}=500, which is low because A_3 and B_2 are keys (also not indicated in Table 1).

To compute simple matches, the cosine similarity function proved to be appropriate, especially if the strings to be compared have approximately the same number of tokens. However, the cosine similarity function turned out not to be appropriate when using the co-occurrence matrix to suggest complex matches to Phase 2 of the technique. We therefore adopted a different similarity function, *BagSim*, to compute the co-occurrence matrix, defined as

$$BagSim(x,y) = \left| Bag(x) \cap Bag(y) \right| \tag{3}$$

which counts the number of tokens that strings *x* and *y* have in common.

Given two properties A_i and B_j, m_{ij} is computed as the sum of $BagSim(x,y)$, for all pairs of strings x and y such that there are triples of the form (a,A_i,x) in $\boldsymbol{p}$ and (b,B_j,y) in $\boldsymbol{q}$ (see Fig. 2). Once the co-occurrence matrix $[m_{ij}]$ is obtained, we compute the EMI matrix $[EMI_{ij}]$, as in Section 3.1 (see Fig. 3).

The result of Phase 1 therefore is the matching μ_{EMI} between the sets of properties $\{A_1,\ldots,A_u\}$ and $\{B_1,\ldots,B_v\}$, computed as in Section 3.1 (which we recall is *1:1*), assuming that, for each $(A_i,B_j)\in\mu_{EMI}$, the property mappings $\mu[A_i,B_j]$ is always the identity function.

	B_1	B_2	B_3	B_4	B_5	B_6	…	B_{14}	B_{15}	B_{16}	B_{17}
A_1	4	1	0	0	0	0		0	0	0	0
A_2	0	0	0	0	0	0		0	0	0	0
A_3	1	500	0	0	0	0	…	0	0	0	0
A_4	0	0	164,826	0	0	0		0	0	0	0
A_5	0	0	0	0	0	0		0	0	0	0
…	…										
A_{24}	0						…	0	0	0	0
A_{25}								0	0	0	0

Fig. 1. Co-occurrence matrix using the cosine similarity function for 500 instances.

	B_1	B_2	B_3	B_4	B_5	…
…	…					
A_5	5500	0	0	55723	3044	
A_6	0	0	0	726	0	
A_7	797	0	0	8527	1363	…
A_8	750	0	0	9576	1015	
A_9	1083	0	0	2690	14167	
…	…					

Fig. 2. Co-occurrence matrix using BagSim.

	B_1	B_2	B_3	B_4	B_5	B_6	…	B_{14}	B_{15}	B_{16}	B_{17}
A_1	0,0055	0,0	0,0	0,0004	-0,0004	0,0		0,0026	0,0067	0,0	0,0
A_2	0,0138	0,0	0,0	0,0020	-0,0009	0,0		0,0110	0,0135	0,0	0,0
A_3	0,0	0,0020	0,0	0,0	0,0	0,0		0,0	0,0	0,0	0,0
A_4	0,0	0,0	0,1493	0,0	0,0	0,0		0,0	0,0	0,0	0,0
A_5	0,0024	0,0	0,0	0,0677	0,0003	0,0	…	0,0028	0,0022	0,0	-0,0002
A_6	0,0	0,0	0,0	0,0009	0,0	0,0		0,0	0,0	0,0	0,0001
A_7	0,0002	0,0	0,0	0,0094	-0,0008	0,0		0,0001	0,0002	0,0	-0,0004
A_8	0,0002	0,0	0,0	0,0114	-0,0007	0,0		0,0003	0,0002	0,0	-0,0008
A_9	0,0002	0,0	0,0	0,0008	0,0040	0,0		0,0002	0,0004	0,0	-0,0001
…	…										
A_{24}	0,0	0,0	0,0	-0,0001	-0,0001	0,0	…	0,0	0,0	0,0	0,0406
A_{25}	0,0	0,0	0,0	0,0001	-0,0007	0,0		0,0	0,0	0,0	0,0007

Fig. 3. EMI matrix: dark grey cells represent simple matches and light grey cells (with positive values) represent possible complex matches for the property in the column.

4.2 Phase 2: Computing Complex Property Matches with Genetic Programming

The second phase of the technique was implemented using a genetic programming toolkit [12], with the parameters shown in Table 2 (the discussion on calibration is omitted for brevity).

The fitness function used is based on the Levenshtein similarity function, normalized to fall into the interval [0,1], where 1 indicates that a string is exactly equal to the other and 0 that the two strings have nothing in common.

Recall that we are given two samples, $\boldsymbol{p}$ and $\boldsymbol{q}$, of instances of properties of classes P and Q, respectively. Construct the set X of strings that occur as literals of instances of B_k obtained by applying $\mu[\boldsymbol{A}^k, B_k]$ to $\boldsymbol{p}$, and the set Y of strings that occur as literals of instances of B_k in $\boldsymbol{q}$. The fitness score for a candidate property mapping is:

$$fitnessScore\ (\mu[\boldsymbol{A}^k, B_k]) = \frac{1}{n} \sum_{x \in X, y \in Y} \text{Levenshtein}(x, y) \qquad (4)$$

where n is the number of pairs in $X \times Y$.

For example, recall that the first phase of the technique outputs a candidate match between properties A_5, A_9, A_{10} and A_{11} (*Address*, *City*, *State* and *Country*, respectively) and property B_5 (*Place*). Table 3 summarizes the search for a property mapping. It indicates that the process stops with an expression that represents a property mapping that maps the concatenation of the properties A_9, A_{10} and A_{11} (that is, the expression ((*City // State*) // *Country*)) into property B_5 (that is, *Place*).

Table 2. Adjusted genetic parameters.

Parameter	Adjusted Values
Population Size ($\sigma_{population}$)	40
Maximum height (σ_{height})	3
Number of Generations ($\sigma_{generations}$)	50
Mutation Rate ($\sigma_{mutation}$)	2%
Crossover Proportion ($\sigma_{crossover}$)	60%
Reproduction Proportion ($\sigma_{reproduction}$)	40%

Table 3. Representation of the possible mappings to the property B_5 (Place).

Suggested Properties	Possible Expressions	Expected Mapping
Address, *City*, *State*, *Country*	*{(Address), (City), (State), (Country)}*	*Place* ← *((City // State) // Country)*
	{(Address // City), (Address // State), (Address // Country), (City // State), …, (State // Country)}	
	{((Address // City) // (State)), ((Address // City) // (Country)), …, ((City // State) // (Country))}	
	{((Address // State) // (City // Country)), …, ((Country // State) // (City // Address))}	

4.3 Results

The first result in this paper is the comparison of the two approaches, Estimated Mutual Information and genetic programming, when separately evaluated.

Column EMI of Table 4 indicates that, using only the Estimated Mutual Information approach, we obtained a precision of 1.0, which indicates that none of the matches were mistakenly found; the rate of recall was low, 0.35, indicating a high rate of missed property matches; and the F-Measure was 0.52, hinting that this approach is insufficient to find simple and complex matches. Indeed, out of the 12 simple matches expected, this approach correctly obtained 6 matches only.

However, according to the discussion at the end of Section 3.1, as well as by observing Fig. , there are 11 candidate complex matches that were suggested to the genetic programming phase. Note that among those are the exact remaining matches not found by the EMI technique. This is an indication that, although not sufficient in itself, the EMI approach doubles as a very effective preprocessing stage to the genetic programming approach, by reducing the complexity of the search space while providing a high quality list of candidate complex matches.

Column GP of Table 4 indicates that, using genetic programming alone, the F-Measure obtained was higher, and that all simple mappings were found. However, precision was 0.81, which indicates that some matches were mistakenly suggested.

Table 4 shows that our two-phase technique resulted in a considerable improvement over the independent use of the EMI and genetic programming approaches when used independently. This improvement is related to the fact that the first phase, using the EMI matrix, correctly found all simple matches and suggested correct complex matches to the second phase.

The fact that the EMI matrix suggests correlated properties helps reduce the solution space considered by the genetic programming algorithm, thus improving its overall performance. In our tests, the run time of the combined approach showed an improvement of approximately 36% when compared with the run time of the genetic programming approach alone.

The only mapping not found by our technique was that from properties *Address*, *AddressNumber*, *AddressComplement* and *Neighborhood* of class *P* to property *FullAddress* of class *Q* (see Table 1). In fact, property *AddressNumber* was the only one not included in any mapping generated by the genetic programming phase, possibly because most values were mistakenly found in the property *Address*, leaving empty many values of the property *AddressNumber*.

Table 4. Mapping results from schema P to schema Q.

Measures	EMI	GP	Two-Phase Approach	Correct Matches
1:1	6	12	12	12
n:1	11*	1	4	5
F-Measure	0.52	0.78	0.96	
Recall	0.35	0.76	0.94	
Precision	1.0	0.81	1.0	

(*) Complex matches suggested by EMI.

5 Conclusion and Future Work

In this paper, we described an instance-based, property matching technique that follows a two-phase strategy. The first phase constructs the Estimated Mutual Information matrix of the property values to identify simple property matches and to suggest complex matches, while the second phase uses a genetic programming approach to detect complex property matches and to generate their property mappings. Our early experiments suggest that the technique is a promising approach to construct complex property matches, a problem rarely addressed in the literature.

Acknowledgements

This work was partly supported by CNPq, under grants 473110/2008-3 and 557128/2009-9, by FAPERJ under grant E-26/170028/2008, and by CAPES under grant CAPES/PROCAD NF 21/2009.

References

[1] Blanco, L., Bronzi, M., Crescenzi, V., Merialdo, P., Papotti, P: Redundancy-driven web data extraction and integration. WebDB 2010.

[2] Carvalho, M. G., Laender, A. H., Gonçalves, M. A., da Silva, A. S. Replica identification using genetic programming. In Proc. 2008 ACM SAC, 2008, pp. 1801-1806.

[3] Dhamankar, R., Lee, Y., Doan, A., Halevy, A., and Domingos, P.: iMAP: discovering complex semantic matches between database schemas. SIGMOD 2004.

[4] Doan, A. H., Domingos, P., Halevy, A. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. SIGMOD 2001.

[5] Doan, A. H., Madhavan, J., Domingos, P., Halevy, A. Learning to Map between Ontologies on the Semantic Web. WWW 2002.

[6] Euzenat, J., Shvaiko, P. Ontology matching. Springer-Verlag (2007).

[7] Koza, J. Genetic Programming. The MIT press, 1998.

[8] Leme, L. A. P. P., Brauner, D. F., Breitman, K. K., Casanova, M. A., Gazola, A. Matching Object Catalogues, Innov. in Sys. and Soft. Eng. Springer, 4(4), 2008, pp. 315-328.

[9] Leme, L. A. P. P., Casanova, M. A., Breitman, K. K., Furtado, A. L. Instance-Based OWL Schema Matching, Lectures Notes in Business Info. Proc., vol. 24, 2009, pp.14-25.

[10] Li, W. S., Clifton, C. SemInt: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. DKE 33(1), 2000, pp. 49-84.

[11] Li, W. S., Clifton, C., Liu, S. Y. Database Integration Using Neural Networks: Implementation and Experiences. Knowledge and Information Systems 2(1), 2000.

[12] Meffert, K. et al.: JGAP - Java Genetic Algorithms and Genetic Programming Package. URL: http://jgap.sf.net, accessed on 09/2010.

[13] Nguyen, T.H., Nguyen, H., Freire, J. PruSM: a prudent schema matching approach for web forms. CIKM 2010. pp. 1385-1388.

[14] Wang, J., Wen, J., Lochovsky, F., Ma, W. Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. VLDB 2004. pp. 408-419.

[15] Wang, Z., Zhang, X., Hou, L., Li, J. RiMOM2: A Flexible Ontology Matching Framework. In: Proc. ACM WebSci'11, Koblenz, Germany, pp. 1-2. (2011).