Expert Systems with Applications: An International Journal 39.3 (2012): 3061-3070

**DOI:** http://dx.doi.org/10.1016/j.eswa.2011.08.168

# Adapting Searchy to Extract Data using Evolved Wrappers

David F. Barrero[a], María D. R-Moreno[a], David Camacho[b]

[a] *Universidad de Alcalá. Computer Engineering Department.*
*Escuela Politécnica. Ctra. Madrid-Barcelona km 31,600*
*28871 Alcal de Henares, Madrid, Spain*
*Phone: (+34) 91-885-69-20, fax: (+34) 885-66-41*
[b] *Universidad Autónma de Madrid. Escuela Politécnica Superior.*
*C/ Francisco Tomás y Valiente 11.*
*Ciudad Universitaria de Cantoblanco. 28049 Madrid, Spain*
*Phone: (+34) 91-497-22-88, fax: (+34) 91-497-22-35*

**Abstract**

Organisations need diverse information systems to deal with the increasing requirements in information storage and processing, yielding the creation of information islands and therefore an intrinsic difficulty to obtain a global view. Being able to provide such an unified view of the -likely heterogeneous- information available in an organisation is a goal that provides added-value to the information systems and has been subject of intense research. In this paper we present an extension of a solution named Searchy, an agent-based mediator system specialized in data extraction and integration. Through the use of a set of wrappers, it integrates information from arbitrary sources and semantically translates them according to a mediated scheme. Searchy is actually a domain-independent wrapper container that ease wrapper development, providing, for example, semantic mapping. The extension of Searchy proposed in this paper introduces an evolutionary wrapper that is able to evolve wrappers using regular expressions. To achieve this, a Genetic Algorithm (GA) is used to learn a regex able to extract a set of positive samples while rejects a set of negative samples.

*Keywords:* Wrappers, Genetic Algorithms, Information Extraction

## 1. Introduction

Organisations have to deal with increasing needs of process automation, yielding a grown of the number and size of software applications. As a result there is a fragmentation of the information: it is placed in different databases, documents of different formats or applications that hide valuable data. Thus, it originates the creation of information islands within the organisation. Then, it has a negative impact when users need a global view of the information, increasing the complexity and development costs of applications. Usually ad-hoc applications are developed despite its lack of generality and maintenance costs. Information Integration [1] is a research area that addresses the several problems that emerge when dealing with such scenario.

When a bunch of organizations are involved in an integration process, the problems associated in the integration are increased. Some traditional integration problems, such as information heterogeneity, are amplified and new problems such as the lack of centralized control over the information systems arises. One of the most interesting problems in such context is how to ensure administrative autonomy, i.e., limit as much as possible the constrains that the integration might impose to data sources. We have developed a data integration solution called Searchy with the intention of addressing those constrains.

Searchy [2] is a distributed mediator system that provides a virtual unified view of heterogeneous sources. It receives a query and maps it into one or more local queries, then translates the responses from the local schema to a mediated one defined by an ontology and integrates them. It separates the integration issues from the data extraction mechanism, and thus it can be seen as a wrapper container that eases wrapper development. It is based on Web Standards like RDF (Resource Description Framework) or OWL (Web Ontology Language). Then, Searchy can be easily integrated in other platforms and systems based on the Semantic Web or SOA (Service Oriented Architecture).

Experience using Searchy in production environments has shown issues to be enhanced. One of the most successful wrappers in Searchy was the regex wrapper, a wrapper that extracts data from unstructured documents using a regular expression (or simply *regex*). Regex is a powerful tool able to extract strings that match a given pattern. Two problems were found related to wrapper-based regex utilization: the need of an engineer (or a specialized user, which we usually denoted as wrapper engineer) with specific skills in regex programming, and the lack of automatic way to handle errors in the extraction process. These problems lead us to adapt the Searchy architecture to support evolved wrappers. That is, wrappers based on regex

*Email addresses:* david@aut.uah.es (David F. Barrero), mdolores@aut.uah.es (María D. R-Moreno), david.camacho@uam.es (David Camacho)

that have been previously generated using Genetic Algorithms (GAs). This wrapper uses supervised learning to generate a regex able to extract records automatically from a set of positive and negative samples.

Wrappers in Searchy may implement arbitrary complex extraction algorithms. The wrapper may, for instance, rely in a Multiagent System (MAS) to generate a composed regular expression able to extract records that match with a training set, as it is described in detail in [3].This paper describes a wrapper able to evolve a simple regex thought a VLGA with an alphabet automatically generated and extract records matching the regex. It also provides an empirical evaluation of these evolved wrappers.

A second contribution of this paper is the description of a complex wrapper able to extract data by means of evolutionary *regular expressions*. That is, regular expressions (or simply *regex*) that have been generated using Genetic Algorithms (GA). This wrapper uses supervised learning to generate a regex able to extract records automatically, without human supervision, from a set of positive and negative samples.

This article is structured as follows. Section 2 provides a general overview the system architecture. The information retrieval and information integration mechanism used in Searchy is briefly described in section 3. The evolved regex wrapper is presented in section 4 followed by a description of the alphabet construction algorithm. Some experiments carried out by the regex wrapper are shown in section 6. Section 7 describes related work. Finally, some future steps are outlined and conclusions are summarized.

## 2. Searchy Architecture

Many properties of Searchy are consequences of two design decisions: the MAS approach [4, 5] and the Web standards compliance. Using MAS gives Searchy a distributed and decentralized nature well suited for the integration scenario described in the introduction. Web Services are used by Searchy agents as an interface to access their functionalities meanwhile the Semantic Web standards are used to provide an information model for semantic and structural integration [6]. From an architectural point of view agents were designed to maximize modularity decoupling integration from extraction issues, easing the implementation of extraction algorithms.

In our architecture each agent is composed by four components, as can be seen in Figure 1. Some of the key properties of Searchy are directly derived from this architecture. These elements are the communication layer, the core, the wrappers and the information source. The next lines describe these components related to the FIPA Agent Management Reference Model.

**Communication layer** It provides features related to the communications such as SOAP message processing, access control and message transport. The Communication layer is equivalent to the Message Transport System (MTS) in the FIPA model.

**Core** It contains the basic skills used by all the agents, including configuration management, mapping facilities or agent identification. Any feature shared by all the agents is contained in the core. It presents some of the features defined by FIPA for the Agent Management System (AMS), however they are not equivalents. AMS are supposed to control the access of the agents to the Agent Platform (AP) and their life cycle. Meanwhile the agent core supports the operation of the wrappers.

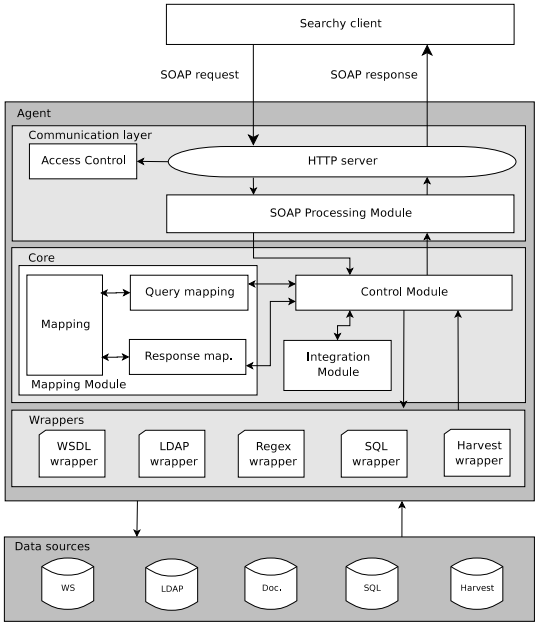


Figure 1: Searchy platform architecture.

**Wrapper** A wrapper is the interface between the core agent and a data source, extracting information from the mediated data source. Wrappers are a key point in order to achieve generality and extensibility. Agents in the FIPA model have some similarities with Searchy wrappers from an architectural point of view. An AP in the FIPA model may contain several agents meanwhile each agent in Searchy may contain several wrappers. Both of them are containers for some software asset, agents in case of FIPA or wrappers in case of Searchy.

**Data source** It is where information that is the object of the integration process is stored. Almost any digital information source might be used as data source. Due to the nature of Searchy, data sources are usually some kind of information system such as a web server or an index. However any source of digital information is a potential Searchy data source. There is no equivalent in the FIPA model to data sources.

Figure 1 shows the architecture of a Searchy agent with its four components. Agents interfaces are published thought the HTTP server, one of the subsystems of the communication layer. It receives the HTTP request that has been sent by the Searchy client and extracts the SOAP message. In order to provide a first layer of security, the HTTP subsystem filters the

request using the Access Control Module. This module is an IP based filter that enables basic access control. The HTTP server has responsibilities with the SOAP messages transport, but the processing of these messages is done by its own module, the SOAP Processing Module. It processes the SOAP messages and then it transfers the operation to the Control Module or returns an error message. Once the message has been successfully processed, the Control Module begins to operate.

The Control Module sets the flow of operations that the different elements involved in the integration must perform, including the wrappers, the Mapping Module, and the Integration Module. The Mapping Module is composed of three subsystems, with different responsibilities in the mapping process. The Query Mapping subsystem performs the query rewriting translating the query from the mediated schema into the local schema, for example, SQL. Meanwhile, the Response Mapping subsystem translates the response from a local schema like SQL, into RDF following a mediated schema defined by an ontology. Both, Query and Response Mapping subsystems use the Mapping subsystem, that provides common services related to mappings and rules management to Query and Response Mapping subsystems. The way in which the integration and mapping processes operate is described in the section 3. Responsibility for Information extraction as well as communication among the agents falls in the wrappers.

In our architecture the coordination among agents is based on an organizational structuring model with two different discovery mechanisms. In the first mechanism each agent has a static knowledge about which agents it must query, where it can find them and how access. The result is a static hierarchical structure. It is useful in order to adapt a Searchy deployment to the hierarchy of a organisation, however it cannot take full benefice of a MAS such as parallelism, the reliability of the whole system is reduced and it is difficult to integrate in dynamic environments.

To overcome some of these disadvantages a second coordination mechanism has been implemented. Using our previous organizational structuring model, relationships among the agents are not stored within the agents, but externally in a WSDL document that can be fetched by any agent from a HTTP or FTP server. This agent discovery mechanism is simpler than using an UDDI (Universal Description, Discovery, and Integration) directory or a Directory Facilitator (DF) in a FIPA platform. Agents are accessed as another data source, and thus it is done by a set of wrappers responsible of the discovery and communication between Searchy agents: the Searchy and WSDL wrappers. These wrappers implement the coordination mechanism in Searchy, however wrappers' main purpose is to extract data from data sources.

At present Searchy includes four ordinary wrappers: SQL, LDAP, Harvest and regex. By means of SQL and LDAP wrappers, structured data in databases and LDAP directories may be accessed. Using the Harvest wrapper Searchy can integrate resources available in an intranet like HTML, LaTeX, Word, PDF documents and other formats. The support of new data sources is done by the development of new wrappers. In section 4 we explain the regex wrapper. There is no restriction on the algorithm and data source that the wrapper might implement, it may be a direct access to a database, a data mining algorithm or data obtained from a sensor. Mapping and integration issues are managed by the agent's core, and thus the wrapper has not to be concerned by these issues. Next section describes how these tasks are performed.

## 3. Mapping and Integration in Searchy

Integrating information means dealing with heterogeneity in several dimensions [6]. Technical heterogeneity can be overcame by selecting the proper implementation technology. In our work it has been done using Web Services (WS) as an interface to access to the service. Addressing information heterogeneity requires the definition of a global information model, the mediated schema, among all the entities involved in the integration process, as well as a mapping mechanism to perform a mapping between the different local information models and the global information model. Defining this model is a critical subject in an information integration system.

Searchy uses semantic technologies standardized by the WWW -RDF, RDFS and OWL- to represent the integrated information. RDF is basically an abstract data model that can be represented using several syntaxes. Searchy uses RDF serialized with XML to represent the information. This combination of RDF and XML grants interoperability in a structural level. Semantic integration requires an agreement about the meaning of the information to deal with semantic heterogeneity. This agreement is performed by using shared ontologies expressed in RDFS or OWL. Then, there must be an explicit agreement among all the actors involved in a Searchy deployment to establish at least one global ontology. A set of mapping rules are needed in order to map entities according to a local schema into the global schema. Rules are used to map queries to a local schema and responses to the mediated schema.

Query format is a tuple <attribute, query> of strings, the Query Mapping subsystem rewrites the query to obtain a query valid for the local data source. The first element in the tuple is an URI that represents the concept to which the query is referred, meanwhile the query is a string with the content of the concept that is being queried. The query model is simple but enough to fulfill the requirements of the application. The translation of the query to the local schema is performed using the Mapping Module (see Figure 1). Mappings are done by means of a string substitution mechanism very similar to the traditional *printf()* function in C. This mechanism is enough to satisfy the needs in almost all cases. Once a query has been translated the response of the local information source must be extracted, mapped to a shared ontology and integrated, respectively, by the Response Mapping and Integration subsystems.

Response mappings are done in two stages:

1. The response is mapped semantically conforming to a shared ontology. It is done using the same mechanism than the Query Mapping subsystem. A critical aspect is to provide a URI identifier for each resource, just like RDF requires to identify any resource. There is no unified way

to do this task: each type of wrapper and user policy define a way to name resources.

2. Each response of each wrapper is integrated in the Integration Module. Integration is based on the URI of the resource returned by the wrappers. When two wrappers return two resources identified by the same URI, the agent interprets that they are referred to the same object and thus they are merged.

Figure 2 shows a simple example of an integration process within Searchy. There are two data sources: a relational database and a LDAP directory service. In a first stage the wrappers retrieve the information from the local data source and this is mapped into a RDF model. The mapping is done by using the terms defined by an ontology and according to some rules given by the system administrator. The ontologies used within the integration process must be shared among all the agents. In general, a one to one correspondence between a data field and an ontology term will be defined. Several local fields or fixed texts may compose one value in RDF, this feature aids the administrator to define more accurate mappings. The mapping rules defined in the example shown in Figure 2 for the database wrapper are depicted in Example 1.

**Example 1** Query mapping rules example

```
rdf:about IS "http://www.example.org/" + name
dc:title IS name + " " + surname
foaf:family_name IS surname
```

The first rule defines that the RDF attribute *rdf:about* is built with the concatenation of the string "http://www.example.org/" and the attribute Person as it is defined in the local schema. The rest of rules are defined in a similar way. Meanwhile the mapping rules for the directory wrapper can be seen in Example 2.

**Example 2** Response mapping rules example

```
rdf:about IS "http://www.example.org/" + uid
rdf:type IS foaf:Person
foaf:mbox IS email
foaf:homepage IS web
```

The wrappers in the example use two vocabularies: Dublin Core and FOAF. Each object retrieved from the data source must be identified by an URI, that in this case is built using local data with a fixed text. The second stage integrates the entities returned by the wrappers. The agent core identifies the two objects as the same object by comparing their URI and merges the attributes, providing a RDF object with attributes retrieved from two different sources.

Mapping and Integration Modules decouple data integration and mapping from the extraction, and thus it is possible to develop wrappers in Searchy without any concern about these issues. Next section shows an example of how a complex wrapper may be developed using the infrastructure provided by Searchy.

The original architecture of Searchy [3] provided an easy to use extraction and integration platform. However, it required human supervision in some parts of the process. One of the most useful wrappers supported by Searchy is the regex wrapper, which is able to extract data from unstructured documents. One problem associated with this wrapper is the need of a wrapper engineer skilled in regex programming. Another problem is the error detection, that is, detect when the wrapper is not extracting data correctly and correct it.

It lead us to extend the original Searchy regex wrapper able to extract data using a regex created by the wrapper engineer with an evolved regex agent able to generate a regex from a set of positive and negative examples using a GA. Figure 3 depicts the extended architecture, where the original architecture is extended with control and evolutive agents. The MAS contains three kind of agents: control, extractor and evolutive agents. The three types of agents share the same agent architecture depicted in Figure 1, they differ from an architectural point of view in the wrappers they use. Figure 3 uses solid lines to represent the iteration among the agents and resources with the exception of iterations that involve regex, which is represented with dotted lines.

There must be one control agent that receives queries from the user and forwards it to the extractors, which are agents with a regex wrapper. Regex wrappers in the original Searchy architecture obtained the regex from the wrapper engineer, who generated manually the regex. When the wrapper detected a fail in the data extraction, i.e., when it was unable to extract data from a source, the wrapper notified it to the wrapper engineer who had to identify the problem and in case the regex was incorrectly constructed, he generates a new one. When the wrapper detected a fail in the data extraction, i.e., it is unable to extract data from a source, the wrapper notifies the wrapper engineer and he had to identify the problem and in case the regex was incorrectly constructed he had to generate a new one.

The new architecture aims to automate this approach, using an evolutive agent that fulfills some roles of the wrapper engineer. Extraction agents obtain the regex from the evolutive agents on start-up time, but also when they identify an extraction error. In this case, instead of requesting a new regex to the wrapper engineer, it would request it to the evolutive agent. When an evolutive agent is required to generate a new regex, it executes a GA that is described in the next section.

## 4. Wrapper based on evolved regular expressions

The implementation of the evolved regex was done as a Searchy wrapper using the Searchy wrapper API. When an agent with the evolved regex wrapper is run, the wrapper generates a valid regex executing the described VLGA with a given training set. Once a suitable regex is generated, the wrapper can begin to extract records from any text file accessible thought HTTP or FTP. It does not have to manage any issue related to mapping since the Mapping Module performs this task.

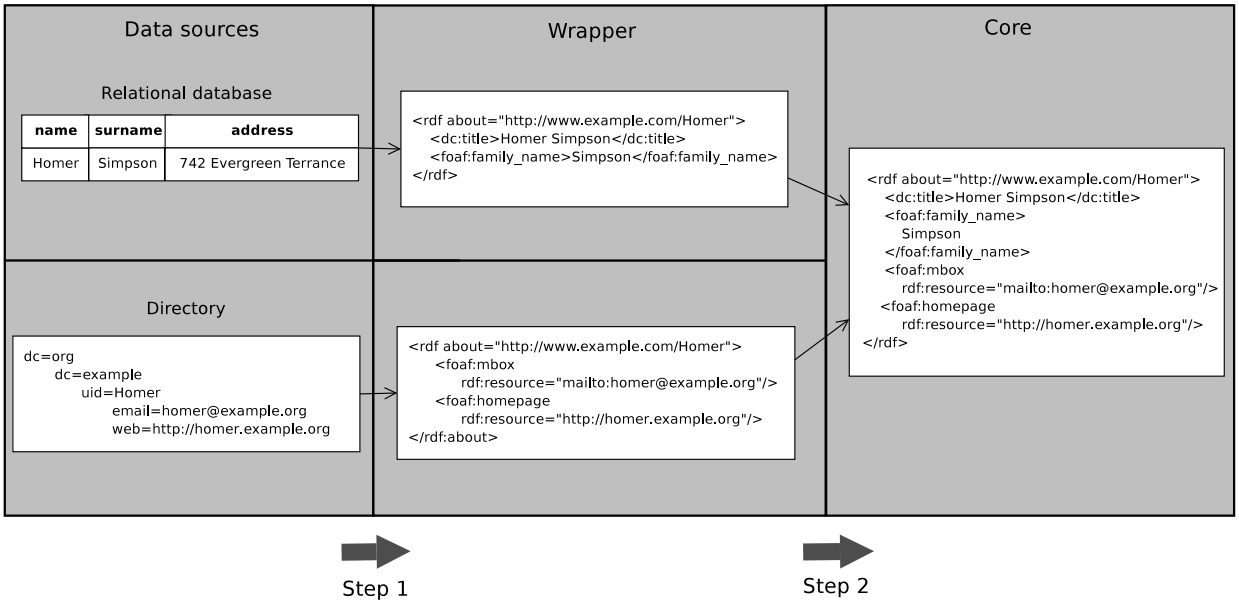


Figure 2: Example of the integration process in Searchy, with two data sources, one relational database and a directory.

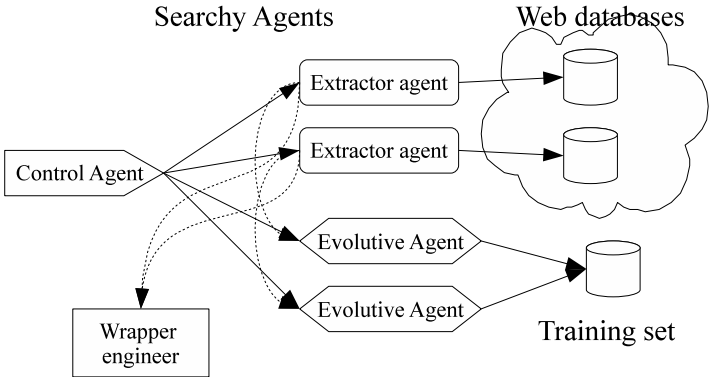


Figure 3: Searchy evolutive agents.

### *4.1. Codification*

Any GA has to set a way to codify the solution into a chromosome. The VLGA implemented in the wrapper uses a binary genome divided in several genes of fixed length. Each gene codes a symbol $\sigma$ from an alphabet $\Sigma$ composed by a set of valid regular expressions constructions, as described in section 5.

Some words should be dedicated to how genes codes regex. The alphabet is not composed by single characters, but by any valid regex, in this way the search space is restricted leading to a easier search. These simple regular expressions are the building blocks of all the evolved regex and cannot be divided, thus, we will call them atomic regex. The position (or *locus*) of a gene determines the position of the atomic regex. Gen in position *i* is mapped in the chromosome to regex transformation as an atomic regex in the position *i*. Figure 4 represents a simple example of how the regex *ca[tr]* could be coded in the GA.

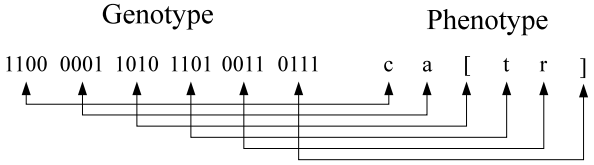


Figure 4: Example of chromosome encoding.

### 4.2. Evolution strategy

Genetic operators used in the evolution of regular expressions are the mutation and crossover. Since the codifications rely in a binary representation, the mutation operator is the common inverse operation meanwhile the recombination is performed with a cut and splice crossover. Given two chromosomes, this operator selects a random point in each chromosome and use it to divide it in two parts, then the parts are interchanged. Obviously, the resulting chromosomes will likely be of different lengths. Selective pressure is introduced by a tournament selection where *n* individuals are randomly taken from the population and the one that scores a higher fitness is selected for reproduction. An elitist strategy has also be used, where some of the best individuals in the population are transferred without any modification to the new generation. In this way it is assured that the best genetic information is not lost.

### 4.3. Fitness

How goodness of any solution is measured is a key subject in the construction of a GA. In our case, for each positive example, the proportion of extracted characters is calculated. Then the fitness is calculated subtracting the average proportion of false positives in the negative example set to the average of characters correctly extracted. In this way, the maximum fitness that a chromosome can achieve is one. This happens when the evolved regex has correctly extracted all the elements of positive examples while any element of the negative examples has been matched. An individual with a fitness value of one is called *ideal individual*.

From a formal point of view, the fitness function that has been adopted in the wrapper uses a training set composed by a positive and a negative subset of examples. Let $\mathcal{P}$ be the set of positive samples and $\mathcal{Q}$ the set of negative samples, such as $\mathcal{P} = \{p_1, p_2, ..., p_M\}$ and $\mathcal{Q} = \{q_1, q_2, ..., q_N\}$. Both, $\mathcal{P}$ and $\mathcal{Q}$ are subsets of the set of all strings $\mathcal{G}$ and they have no common elements, so $\mathcal{P} \cap \mathcal{Q} = \varnothing$.

Chromosomes are evaluated as follows. Given a chromosome it is transformed into the corresponding regex $r \in \mathcal{R}$, then tries to match against the elements of $\mathcal{P}$ and $\mathcal{Q}$. The set of strings that $r$ extracts from a string $p$ is given by the function $\varphi(p, r) : (\mathcal{S} \times \mathcal{R}) \longrightarrow \mathbb{R}$ while the number of characters retrieved is represented by $|\varphi(p, r)|$. The percentage of extracted characters of $p_i$ such as $i = 0, ..., M$ is averaged and finally the fitness is calculated subtracting the average proportion of false positives in the negative example set to the average of characters correctly extracted, as is expressed by equation (1).

$$\mathfrak{F}(r) = \frac{1}{|P|} \sum_{p_i \in \mathcal{P}} \frac{|\varphi(p_i, r)|}{|p_i|} - \frac{1}{|Q|} \sum_{q_i \in Q} M_r(q_i) \tag{1}$$

where $|p_i|$ is the number of characters of $p_i$, $|P|$ the number of elements of $\mathcal{P}$, $|Q|$ the number of elements of $\mathcal{Q}$ and $M_r(q_i)$ is defined as

$$M_r(q_i) = \begin{cases} 1 & if \quad |\varphi(q_i, r)| > 0 \\ 0 & if \quad |\varphi(q_i, r)| = 0 \end{cases} \tag{2}$$

## 5. Zipf's law based alphabet construction

### 5.1. Preliminary considerations

Section 4.1 has shown how a classical binary codification is used to select one symbol $\sigma$ from a predefined set $\Sigma$ of symbols or atomic regex. The construction of $\Sigma$ is a critical task since it determines the search space, its size and its capacity to express a correct solution. Of course, the simplest approach is to manually select the alphabet, however this approach may devaluate the added value of evolved regex: the automatic generation of regex.

We can state that construction of $\Sigma$ must satisfy three constrains.

1. $\Sigma$ must be *sufficient*, i.e., it must exist at least an element $r \in \Sigma^*$ such as $r$ is an ideal individual. In other words, it must be possible to construct at least one valid solution using the elements of $\Sigma$.
2. $|\Sigma|$ must contain the minimum number of elements able to satisfy the sufficiency constrain. Of course, being able to satisfy this condition is a challenging task with deep theoretical implications. From a practical point of view, this constrain can be reformuled to keep $|\Sigma|$ as low as possible.
3. Symbol selection must be automatic, with minimal human interaction and number of parameters.

### 5.2. Alphabet construction algorithm

To reduce the number of elements of $\Sigma$, and keep the search space as small as possible, we aim to identify patterns in the positive samples and use them as building blocks. In order to satisfy the previous constrains we propose the following algorithm. $\Sigma$ is built as the union of $\mathcal{F}$, $\mathcal{D}$ and $\mathcal{T}$, where $\mathcal{F}$, is the set of fixed symbols, $\mathcal{D}$ the set of delimiters and $\mathcal{T}$ the set of tokens.

$$\Sigma = \{\sigma_i\} \backslash \sigma_i \in \mathcal{F} \cup \mathcal{D} \cup \mathcal{T} \tag{3}$$

$\mathcal{F}$ contains hand-created reusable symbols that are meant to be common cross-domain regex, and thus, once they have been defined they can be used to evolve different regex. It should be noticed that $\mathcal{F}$ may contain any valid regex, nevertheless it is supposed to contain generic use regex such as *\d+* or *[1-9]+*. Since $\mathcal{F}$ is supposed to include common used complex regex, it contributes to reduce the search space and increase individual fitness by introducing high fitness building blocks.

The sets $\mathcal{D}$ and $\mathcal{T}$ are constructed using a more complex mechanism based on Zipf's Law [7]. It states that occurrences of words in a text are not uniformally distributed, rather only a very limited number of words concentrates a high number of occurrences. This fact can be used to identify patterns in $\mathcal{P}$ and use them to construct a part of $\Sigma$.

Since the tokens do not contain delimiters, the sufficiency constrain cannot be satisfied, so, each delimiter that appear in the examples is included in the set $\mathcal{D}$. The overall process is described in Algorithm 1. Of course, $|\Sigma|$ must be equal to the number of elements of the union of $\mathcal{F}$, $\mathcal{D}$ and $\mathcal{T}$, as is expressed in equation (4).

**Algorithm 1** Selection of alphabet tokens.

```
1 .- P := Set of positive examples
2 .- S := Set of candidate delimiters
3 .- D := T := { }
4 .-
5 .- for each p in P
6 .-     for each s in S
7 .-         tokens := split p using s
8 .-         numberTokens := number of tokens
9 .-
10.-         for each token in tokens
11.-             occurrence(token) := occurrence(token) + 1
12.-         endfor
13.-
14.-         if (numberTokens > 0) add s to D
15.-     endfor
16.- endfor
17.-
18.- sort occurrence
19.- add n first elements of occurrence to T
```

$$|\Sigma| = |\mathcal{F} \cup \mathcal{D} \cup \mathcal{T}| \tag{4}$$

given that

$$|\mathcal{F} \cap \mathcal{D} \cap \mathcal{T}| = |\mathcal{F} \cap \mathcal{D}| = |\mathcal{F} \cap \mathcal{T}| = |\mathcal{D} \cap \mathcal{T}| = \oslash \tag{5}$$

### *5.3. Complexity analysis*

A better understanding of the algorithm can be achieved by a time complexity analysis. As can be seen in Algorithm 1, there are two main loops (see Algorithm 1, lines 5 and 6) that depends on the number of examples $|P|$ and the number of potential delimiters $|S|$. The complexity of the algorithm is given by these loops and the operations that are performed inside.

Splitting a string $p_i \in \mathcal{P}$ (line 7) is proportional to the length of the string $|p_i|$, so the mean time required to perform this operation is proportional to the mean string length $|\overline{p}|$. Lines 19 to 21 include a loop that is repeated as many times as the tokens are in the string. A hash table is accessed inside the loop (line 20), so it makes sense to suppose that its complexity is given by the calculus of the key, a string, therefore its time complexity is $n|\overline{p}|$, where $n$ is the number of tokens. Finally sorting *occurrence* can be performed in $n_{tot}\log(n_{tot})$ where $n_{tot}$ is the number of tokens stored in *occurence*. The rest of the operations in the algorithm can be performed in a negligible time. We can express these considerations in equation (6).

$$t \propto |P| \cdot |S| \cdot [|\overline{p}| + n|\overline{p}|] + n_{tot}\log(n_{tot}) \tag{6}$$

Both $n$ and $n_{tot}$ are unknown and we have to estimate it for the average case. A string $p \in P$ of length $|p|$ can contain approximately a maximum of $\frac{|\overline{p}|}{2}$ tokens. We have supposed there is one delimiter for each token. The maximum number of tokens that can be stored in *occurrences* are $\frac{|P|\cdot|S|\cdot|\overline{p}|}{2}$. Then

$$n = \frac{|\overline{p}|}{2} \tag{7}$$

$$n_{tot} = \frac{|P| \cdot |S| \cdot |\overline{p}|}{2} \tag{8}$$

and 6 can be expressed as

$$t \propto |P| \cdot |S| \cdot |\overline{p}|[1 + \frac{|\overline{p}|}{2}] + \frac{|P| \cdot |S| \cdot |\overline{p}|}{2}\log(\frac{|P| \cdot |S| \cdot |\overline{p}|}{2}) \tag{9}$$

Some terms can be removed

$$t \propto \frac{|P||S||\overline{p}|}{2}\log(\frac{|P| \cdot |S| \cdot |\overline{p}|}{2}) \tag{10}$$

Using Big Oh notation it yields that the time complexity is given by

$$O(k\log(k)) \tag{11}$$

where $k = |P||S||\overline{p}|$ and therefore we can conclude that the time complexity is linearithmic.

## 6. Evaluation

Two phases have been used in the evaluation, a first phase where the basic behaviour of the GA is analyzed, and a second phase that uses the knowledge acquired along the first phase to measure extraction capabilities of the evolved regex wrapper. Measures that have been used are the well known precision, recall and F-measure. The sets of experiments described in this section are focused in the extraction of three types of data: URLs, phone numbers and email addresses.

### *6.1. Parameter tuning*

Some initial experiments were carried out to acquire knowledge about the behaviour of the regex evolution and select the GA parameters to use within the wrapper. Experiments showed that despite the differences between phone, URL and emails all the case studies have similar behaviors. In this way it is possible to extrapolate the experimental results and thus to use the same GA parameters. Setup experiments showed that best performance is achieved with a mutation probability of 0.003 and a tournament size of 2 individuals. A population composed by 50 individuals is a good trade-off between computational resources and convergence speed. Initial population has been randomly generated with chromosome lengths that range from 4 to 40 bits and elitism of size one has been applied. Table 1 summarizes the parameter values used in the experiments.

### *6.2. Regex evolution*

Once the main GA parameters have been set, the wrapper can evolve the regex. Experiments have used three datasets to evolve regex able to extract records in the three case studies under study. Figure 5 depicts the mean best fitness (MBF) and mean average fitness (MAF) of 100 runs. The fitness evolution of the case studies follows a similar path. The best MBF and

Table 1: GA parameters summary.

| Parameter | Value |
|---|---|
| Population | 50 |
| Mutation probability | 0.003 |
| Crossover probability | 1 |
| Tournament size | 2 |
| Elitism | 1 |
| Initial chromosome length | 4 - 40 |

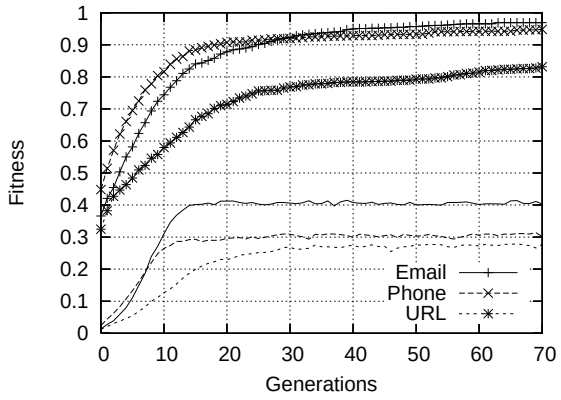


Figure 5: Best and average fitness of phone, URL and email regex.

MAF are achieved by the email regex, while the poorest performance is given by the URL regex, with lower fitness values.

The dynamics of the chromosome length can be observed in the Figure 6. It is clear that there is a convergence of the chromosome length and thus chromosome bloating does not appear. It can be explained by the lack of non-coding and overlapping regions in the chromosome, i.e, if the chromosome has achieved a maximum it hardly can increase its size without a penalty in its fitness. The longer is the chromosome, the more restrictive is the phenotype and it is closely related to the associated fitness. URL regex has a stronger tendency to local maximum, this fact is reflected in Figure 6, where a lower MBF and MAF are achieved. This fact also explains why URL chromosome length depicted in Figure 6 is shorter than phone regex: the local maximum of URL regex tends to generate populations with an insufficient chromosome length. Those results are not surprising since URLs follow a far more complex pattern than phone numbers or emails. The same can be affirmed about emails in comparison to phone numbers.

Figure 6 shows another interesting behaviour. As the GA begins to run, the average chromosome length is reduced until a point where it begins to increase, then the chromosome length converges into a fixed value. In early generations individuals have not suffered evolution and thus its genetic code has a strong random nature. Individuals with longer genotype have longer phenotypes and thus more restrictive regex that will likely have smaller fitness values. So long chromosomes are discarded in early stages of the evolutive process until the population is composed by individuals representing basic phenotypes, then recombination leads to increase complexity of individuals until they reach a length associated with a local or global maximum.

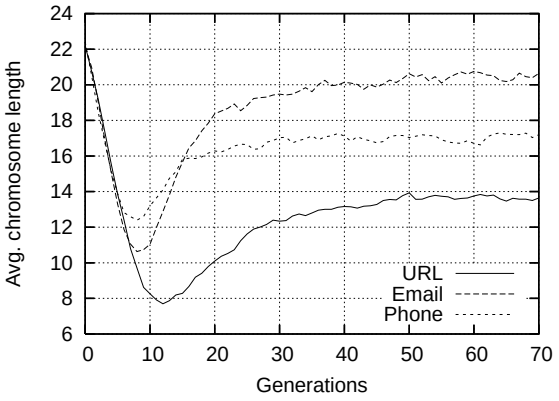


Figure 6: Evolution of regular expressions.

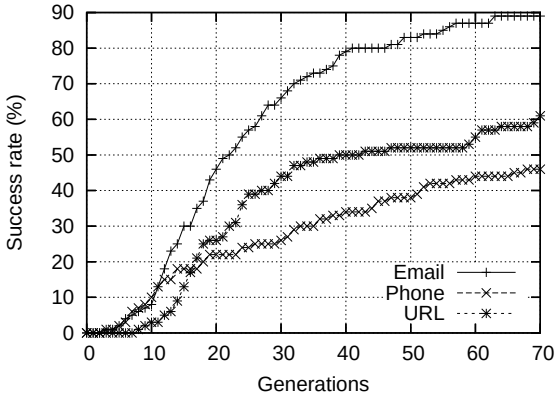


Figure 7: Probability of finding an ideal regex able to accept all the positive examples while rejecting the negative ones.

Some of the facts found previously are confirmed by Figure 7, where the success rate (SR) [8] is depicted versus the generation. SR is defined as the probability of finding an ideal individual in a given generation. It should be noted that Figure 7 depicts the average success rate of 100 runs of the experiment. It can be seen that email achieves a SR of 91%, phone numbers 54% and URLs 63% in generation 70. These results are consistent with those in Figure 6 and show that the hardest study cases are URLs, phone numbers, and emails, in that order. Here the term ”hard” should not be understood in a strict absolute way since the hardness of the search space is influenced from several factors, such as the training set, the selection of negative samples or the alphabet chosen.

### *6.3. Data extraction*

Three regex with an ideal fitness of one have been selected by the wrapper and its extraction capabilities have been evaluated by means of the precision, recall and F-measure. The experiments used a dataset composed by eight sets of documents from different origins containing URLs, emails and/or phone

Table 2: Extraction capacity of the evolved regex. The table shows the F-measure (F), precision (P) and recall (R) achieved in the three datasets (phone, URL and email addresses).

| | | | | **Phone regex** | | | **URL regex** | | | **Email regex** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Phone | URL | Email | **F** | **P** | **R** | **F** | **P** | **R** | **F** | **P** | **R** |
| Set 1 | 99 | 0 | 0 | 1 | 1 | 1 | - | - | - | - | - | - |
| Set 2 | 0 | 51 | 0 | - | - | - | 0.24 | 0.14 | 0.84 | - | - | - |
| Set 3 | 0 | 0 | 862 | - | - | - | - | - | - | 0.79 | 0.51 | 0.62 |
| Set 4 | 20 | 77 | 0 | 1 | 1 | 1 | 0.27 | 0.16 | 1 | - | - | - |
| Set 5 | 37 | 686 | 0 | 1 | 1 | 1 | 0.20 | 0.11 | 0.97 | - | - | - |
| Set 6 | 24 | 241 | 0 | 1 | 1 | 1 | 0.02 | 0.01 | 0.37 | - | - | - |
| Set 7 | 83 | 0 | 88 | 0.92 | 1 | 0.96 | - | - | - | 0.92 | 1 | 0.96 |
| Set 8 | 0 | 51 | 0 | - | - | - | 0.63 | 0.47 | 0.96 | - | - | - |
| **Avg.** | - | - | - | 0.98 | 1 | 0.99 | 0.27 | 0.18 | 0.83 | 0.85 | 0.79 | 0.79 |

numbers. Table 2 shows basic information about the datasets and their average records and Table 3 contains some evolved regex with their fitness value. Sets one, two and three are composed by examples extracted from the training set. The rest of the sets are web pages retrieved from the Web classified by their contents. An extracted string has been evaluated as correctly extracted if and only if it matches exactly the records, otherwise it has been computed as a false positive.

The results, as can be seen in Table 2, are quite satisfactory for phone numbers and testing sets, but measures get worse for real raw documents, specially the ones containing URL records. Phone regex has a perfect extraction with a F-measure value close to 1. The training set used to evolve regex contains phone numbers in a simple format (000)000 – 0000, the same that can be found in the testing set, the reduction of recall in set 7 is due to the presence of phone extensions that are not extracted.

On the contrary, measures achieved for URL extraction from raw documents are much lower. It can be explained looking at the regex used in the extraction, *http://\w+\.\w+\.com*. Documents used in the test contain many URLs with paths, so the regex is able to partially extract them, increasing the count of false positives. The result is a poor precision. An explanation of the poor recall measures in URLs extraction is found in the fact that the evolved regex only is able to extract URL whose first level domain is *.com*, so its recall in documents with a high presence of first level domains in other forms is worse.

Finally, email regex achieves an average F-measure of 0.85. Some of the factors that limits the URL regex extraction capabilities are also limiting email regex. However in this case the effects are not so severe for some reasons, for instance the lower percentage of addresses with more than two levels.

Table 3: Some examples of evolved regular expressions with their fitness values.

| Evolved regex (Phone) | Fitness |
|---|---|
| \w+ | 0 |
| \(\d+\) | 0.33 |
| \(\d+\)\d+ | 0.58 |
| \(\d+\)\d+-\d+ | 1 |
| **Evolved regex (URL)** | **Fitness** |
| http://-http://http:// | 0 |
| /\w+\. | 0.55 |
| http://\w+\.\w+\ | 0.8 |
| http://\w+\.\w+\.com | 1 |
| **Evolved regex (Email)** | **Fitness** |
| \w+\. | 0.31 |
| \w+\.\w+ | 0.49 |
| \w+@\w+\.\com | 1 |

## 7. Related Work

The use of ontologies [9] has attracted the attention of data integration community in the last years. It has provided a tool to define mediated schemas focused on knowledge sharing and interoperability, in contrast with traditional database centric schemas, whose goal is to query single databases [10]. The adoption of ontologies has lead to reuse results achieved by two communities such as the database and the AI communities to solve similar problems like schema mapping or entity resolution. A deep discussion about the role of the ontologies in the data integration can be found in [11].

We can define a collection of semantic solutions based on ontology technologies prior to the development of the SW. A introduction to this group of solutions can be found in [6]. We can remark classical literature examples such as InfoSleuth [12] or SIMS [13]. From these systems, we have to stress InfoSleuth, a solution that uses a MAS.

Semantic integration tools in the last years have adopted WS standards and technologies. One of the first ones can be found in [14]. Vdovjak proposes a semantic mediator for querying heterogeneous information sources, but limited to XML documents, furthermore, this solution relies on a wrapper layer that translates the local entities into XML and then the RDF is generated. A step forward is done by Michalowski with Building Finder [15], a domain specific mediator system aimed to retrieve and integrate information about streets and buildings from heterogeneous sources, presented to the user within satellite images. [16] describes an information integration tool that covers all the phases of integration, such as assisted mapping

definition and query rewrite.

Another newcomer into the IT toolbox is the Web Services technology. WS provide a means to access services in a loose coupling way. Despite WS and the SW face different problems -one models and represents knowledge meanwhile the another one is concerned with service provision-, they are related by means of semantic descriptions of WS thought Semantic Web Services. In this way WS are enhanced with semantic descriptions, enabling dynamic service composition and data integration [17].

A semantic integration solution based on SOA is SODIA (Service-Oriented Data Integration Architecture) [18]. It supports some integration approaches such as federated searches and datawarehouse. By using a SOA approach SODIA has many of the benefits of using an agent technology. However, this is a process centric solution and has limited semantic support. The most aligned solution to the one described in this paper is Knowledge Sifter [19]. It is an agent based approach that uses OWL to describe ontologies and WS as interface to the agents' services. Despite the lack of semantic support, WS integration or distributed nature, we have to mention the system proposed by [20], a system able to automate the full integration process by creating the mediated schema and schema mapping on-the-fly. Another interesting integration suite related to bioinformatics domain that that could be mentioned is INDUS [21].

Table 4 compares some representative federated ontology-driven search solutions. The scope of table 4 is limited, however some relevant facts are show. It depict if the integration system is supported by agents, it uses any WS or SW technology as well as the degree of specialization of the tool.

## 8. Future work and conclusions

Some issues are still open. One of them is the limited number of information sources that Searchy is able to integrate. WebMantic [22] is a wrapper web-centric information extraction tool that once integrated in Searchy will enhance its Web information extraction and integration capabilities.

One method to extract data from unstructured documents is the use of evolved regex. Genetic Algorithms provide a stochastic search method well suited for the complex search space conformed by regular expressions. Despite the fact that VLGAs are much simpler than the fixed-length approach described in [3], it still presents some important drawbacks, such as the intrinsic difficulty to evaluate parts of the evolved regex or the linear nature of the codification. In order to avoid this type of problems it is expected to use Genetic Programming and Grammatical Evaluation to generate regex. In particular, Genetic Programming has shown well performance in related areas [23] and is a promising approach.

From the point of view of the Searchy platform, The creation of a wide network of agents implies the management of huge amount of information. Then, the physical scalability of the system should be done in parallel with the intelligence system improvement. Some techniques such as ranking or information filtering with a case based reasoning or collaborative filtering are considered to provide some intelligence to the system that will produce better user satisfaction.

Along this paper we have briefly presented the problem of information extraction and integration and we have proposed an extension of a partial solution called Searchy. This extension aims to automatice some of the tasks asigned originally to the wrapper engineer thought some agents able to use Machine Learning to generate regex. When an extractor agent requires a regex (it can be in initialization time or because it cannot extract data with a given regex) it request one to a evolutive agent that using a set of positive and negative examples and a Genetic Algorithm is able to generate the regex.

## 9. Acknowledgements

The authors gratefully acknowledge Martín Knoblauch for his useful suggestions and valuable comments. This work has been partially supported by the Spanish Ministry of Science and Innovation under the projects ABANT (TIN 2010-19872), COMPUBIODIVE (TIN2007-65989) and by Castilla-La Mancha project PEII09-0266-6640.

## References

[1] L. Haas, Beauty and the beast: The theory and practice of information integration, ICDT 2007 (2007) 28–43.

[2] D. F. Barrero, M. D. R-Moreno, D. R. López, Information integration in searchy: an ontology and web services approach, International Journal of Computer Science and Applications (IJCSA) 7 (2010) 14–29.

[3] D. F. Barrero, D. Camacho, M. D. R-Moreno, In Data Mining and Multiagent Integration. Chapter 9: Automatic Web Data Extraction Based on Genetic Algorithms and Regular Expressions (2009), Springer, pp. 143–154.

[4] F. Wan, M. P. Singh, Commitments and causality for multiagent design, in: A. Pres (Ed.), 2nd International Joint Conference on Autonomous Agents and multiagent Systems (AAMAS), Melbourne, Australia.

[5] R. Aler, J. M. Valls, D. Camacho, A. Lopez, Programming robosoccer agents by modeling human behavior, Expert Systems with Applications 36 (2009) 1850–1859.

[6] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, S. Hübner, Ontology-based integration of information — a survey of existing approaches, in: IJCAI–01 Workshop: Ontologies and Information Sharing, Seattle, Washington, USA, pp. 108–117.

[7] G. Zipf, The Psycho-Biology of Language, Houghton Mifflin, Boston, MA, 1935.

[8] D. F. Barrero, D. Camacho, M. D. R-Moreno, Confidence intervals of success rates in evolutionary computation, in: Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2010)., ACM, Portland, Oregon,USA, 2010, pp. 975–976.

[9] T. Grubber, A translation approach to portable ontology specifications, Knowledge Acquisition 5 (1993) 199–220.

[10] M. Uschold, M. Gruninger, Ontologies and semantics for seamless connectivity, SIGMOD Rec. 33 (2004) 58–64.

[11] N. F. Noy, Semantic integration: a survey of ontology-based approaches, SIGMOD Rec. 33 (2004) 65–70.

[12] M. H. Nodine, J. Fowler, T. Ksiezyk, T. Perry, M. Taylor, A. Unruh, Active information gathering in infosleuth, International Journal of Cooperative Information Systems 9 (2000) 3–28.

[13] C. A. Knoblock, J.-L. Ambite, Agents for information gathering, in: J. M. Bradshaw (Ed.), Software Agents, AAAI Press / The MIT Press, 1997, pp. 347–374.

[14] R. Vdovjak, G. Houben, Rdf based architecture for semantic integration of heterogeneous information sources, in: E. Simon, A. Tanaka (Eds.), Proceedings of the International Workshop on Information Integration on the Web, Rio de Janeiro, Brazil, pp. 51–57.

Table 4: Semantic information integration tools comparison.

| Platform | Agent support | Semantic Web | Web Services | Interdomain support |
|---|---|---|---|---|
| InfoSleuth | Yes | No | No | Yes |
| SIMS | Yes | No | No | Yes |
| Building Finder | No | Yes | No | No |
| SODIA | No | Yes | Yes | Yes |
| Knowledge Sifter | Yes | Yes | Yes | Limited |
| Searchy | Yes | Yes | Yes | Yes |


[15] M. Michalowski, J. Ambite, S. Thakkar, R. Tuchinda, C. Knoblock, S. Minton, Retrieving and semantically integrating heterogeneous data from the web, IEEE Intelligent Systems 19 (2004) 72–79.

[16] J. Yuan, A. Bahrami, C. Wang, M. Murray, A. Hunt, A semantic information integration tool suite, in: VLDB '06: Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment, 2006, pp. 1171–1174.

[17] P. Deepti, B. Majumdar, Semantic web services in action - enterprise information integration., in: B. J. Kramer, K.-J. Lin, P. Narasimhan (Eds.), ICSOC, volume 4749 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 485–496.

[18] F. Zhu, M. Turner, I. A. Kotsiopoulos, K. H. Bennett, M. Russell, D. Budgen, P. Brereton, J. Keane, M. Rigby, J. Xu, Dynamic data integration using web services., in: IEEE International Conference on Web Services (ICWS'04), San Diego, California, USA, pp. 262–269.

[19] L. Kerschberg, M. Chowdhury, A. Damiano, H. Jeong, S. Mitchell, J. Si, S. Smith, Knowledge sifter: Ontology-driven search over heterogeneous databases., in: 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), IEEE Computer Society, Santorini Island, Greece, 2004, pp. 431–432.

[20] A. D. Sarma, X. Dong, A. Halevy, Bootstrapping pay-as-you-go data integration systems, in: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, New York, NY, USA, 2008, pp. 861–874.

[21] D. Caragea, J. Pathak, J. Bao, A. Silvescu, C. Andorf, D. Dobbs, V. Honavar, Information integration and knowledge acquisition from semantically heterogeneous biological data sources, in: Proceedings of the 16th International Workshop on Database and Expert Systems Applications, Springer-Verlag, 2005, pp. 175–190.

[22] D. Camacho, M. D. R-Moreno, D. F. Barrero, R. Akerkar, Semantic wrappers for semi-structured data extraction, Computing Letters (COLE) 4 (2008) 1–14.

[23] You-Heng, L. Ge, Learning ranking functions for geographic information retrieval using genetic programming, Journal of Research and Practice in Information Technology 41 (2009) 39–52.