An Architecture for Efficient Web Crawling

Inma Hernández, Carlos R. Rivero, David Ruiz, and Rafael Corchuelo

University of Sevilla, Spain
{inmahernandez,carlosrivero,druiz,corchu}@us.es

Abstract. Virtual Integration systems require a crawling tool able to navigate and reach relevant pages in the Deep Web in an efficient way. Existing proposals in the crawling area fulfill some of these requirements, but most of them need to download pages in order to classify them as relevant or not. We propose a crawler supported by a web page classifier that uses solely a page URL to determine page relevance. Such a crawler is able to choose in each step only the URLs that lead to relevant pages, and therefore reduces the number of unnecessary pages downloaded, minimising bandwidth and making it efficient and suitable for virtual integration systems.

Keywords: Web Crawling, Crawler Architecture, Virtual Integration.

1 Introduction

Virtual Integration aims at accessing web information in an automated manner, considering the Web as a source of information. The Virtual Integration process starts with a query, in which the user expresses her interests, and its goal is to obtain information relevant to that query from different sites of the Web (usually Deep Web sites), and present it uniformly to the user in a transparent way. From now on, we refer to relevant pages as web pages containing information related to the user interests. Note that these interests may change over time, so the same page can be either relevant or irrelevant.

This process is online, which means that while the system is looking for information, the user is waiting for a response. Therefore, bandwidth and efficiency are important issues for any Virtual Integration approach, and downloading only relevant pages is mandatory.

Virtual Integration processes require a tool able to navigate through web sites, looking for the information. A web crawler is an automated process that navigates the Web methodically, starting on a given set of seed pages and following a predefined order. Once the crawler has collected the relevant pages, they are passed on to an information extractor, which obtains the information that is contained in pages and gives it some structure, before returning it back to the

⋆ Supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, and TIN2010-09988-E).

user. In this paper, we focus on the crawling aspects of a Virtual Integration process.

As Edwards et al. [10] noted, it is imperative to improve crawlers efficiency in order to adjust to the available bandwidth, specially in Virtual Integration contexts, in which the goal is to retrieve only pages that are relevant to the user query. Consequently, the improvement of crawling efficiency is attracting the interest of many researchers.

The main requirements we consider in the design of a Virtual Integration crawler are efficiency, form filling and unlabelled training sets. The crawler must be efficient, that is, it should minimize bandwidth usage and download only relevant pages. Also, to access pages in the Web that are behind forms, the crawler must be able to fill in forms giving values to the different fields, and submit them. Finally, creating large labeled training sets is burdensome for the user. Instead, we focus on training the crawler using an unlabeled set obtained automatically, minimising user intervention.

Many crawling techniques have been proposed so far in the literature, such as traditional crawlers, focused crawlers or recorders. Traditional crawlers navigate sites by retrieving, analysing and classifying all pages that are found, including non-relevant pages [17]. Hence, they do not fulfill the efficiency requirement. Focused crawlers reduce the amount of irrelevant pages downloaded, usually by applying a content-based web page classifier [1, 7, 6, 9, 12, 14, 15, 16]. They usually deal with large collections of static pages, and therefore form filling is not a priority issue. Finally, recorders are crawlers in which each navigation step has been defined by the user [2, 3, 4, 5, 8, 13, 18]. Therefore, although they deal with form filling and efficiency requirements, the user has to label large training sets.

Our goal is to design a crawler supported by a URL-based classifier to determine page relevance. Thus, it does not need to download a page in order to classify it, reducing the bandwidth and making it efficient and suitable for virtual integration systems. Also, our crawler is able to fill in and submit forms. Furthermore, the crawler includes a setup phase in which a link training set for the classifier is automatically collected, not requiring intervention from the user.

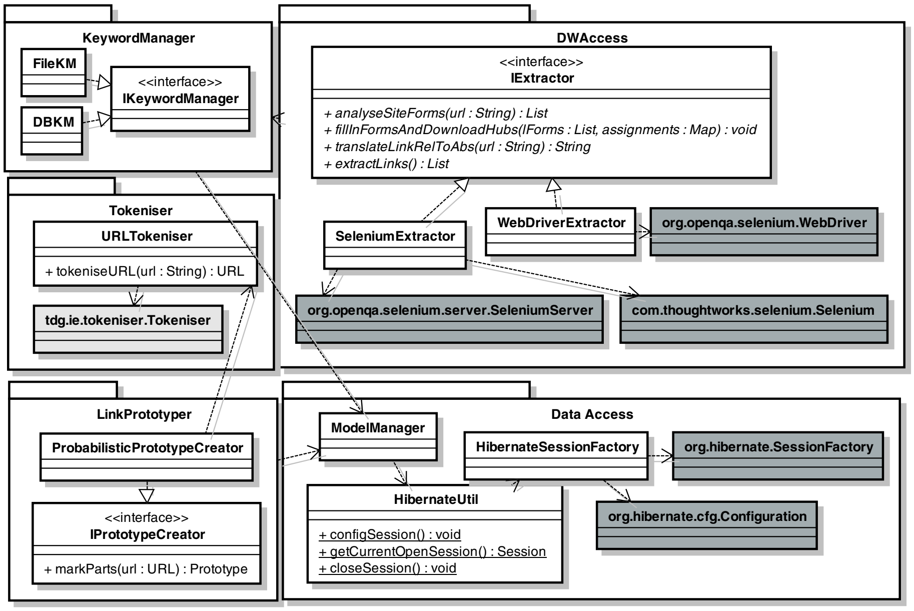
The rest of the article is structured as follows. Section 2 presents the architecture proposed to solve the aforementioned problem; and Section 3 lists some of the conclusions drawn from the research and concludes the article.

2 Architectural Proposal

First, we describe the workflow of the system; then, we present the architectural design, describing for each module a definition of its responsibilities, an example of a typical use case, and a list of the possible issues that should be considered during the design.

2.1 Workflow

Figure 2 presents the system workflow. A Virtual Integration process starts with an enquirer, which translates the user interests into queries that are issued to



KeywordManager
FileKM
DBKM
<<interface>>
IKeywordManager
DWAccess
<<interface>>
IExtractor
+ analyseSiteForms(url : String) : List
+ fillInFormsAndDownloadHubs(IForms : List, assignments : Map) : void
+ translateLinkRelToAbs(url : String) : String
+ extractLinks() : List
SeleniumExtractor
WebDriverExtractor
org.openqa.selenium.WebDriver
org.openqa.selenium.server.SeleniumServer
com.thoughtworks.selenium.Selenium
Tokeniser
URLTokeniser
+ tokeniseURL(url : String) : URL
tdg.ie.tokeniser.Tokeniser
LinkPrototyper
ProbabilisticPrototypeCreator
<<interface>>
IPrototypeCreator
+ markParts(url : URL) : Prototype
Data Access
ModelManager
HibernateSessionFactory
org.hibernate.SessionFactory
HibernateUtil
+ configSession() : void
+ getCurrentOpenSession() : Session
+ closeSession() : void
org.hibernate.cfg.Configuration

Fig. 1. Class diagram

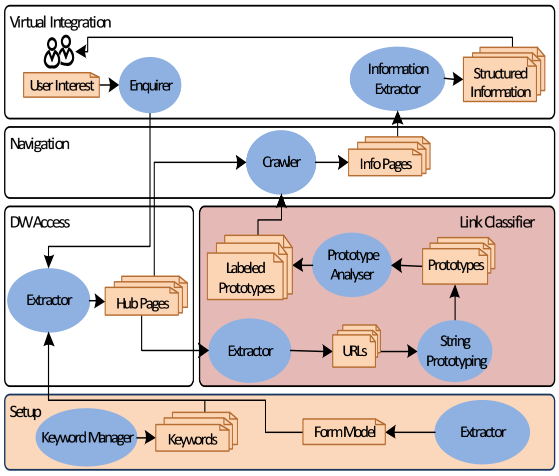


Fig. 2. Workflow diagram

forms. Usually, response pages are hubs, lists of results ordered and indexed, each of them showing a link to another page with detailed information. Relevant pages, when found, are passed on to the information extractor, which obtains and structures the information, that is returned to the user.

We distinguish two phases: the setup and the normal execution phase. In the former, the keyword manager and form filler are focused on obtaining automatically a set of links from the site under analysis, which is later used to train the classifier. The only requirement for the set is to be representative of the site, hence it is extracted from hubs, which are the pages that contain a higher number of links in any given site. In the latter phase, the form filler is used to reach pages behind the forms, and then the crawler makes use of the trained classifier to select which links to follow, avoiding those not leading to relevant pages. In this paper we focus on the setup phase of the architecture, namely in the setup and classifier modules.

2.2 Architectural Design

We present the architecture of our crawler in Figure 1, which includes the main modules in our design: Keyword Manager, Deep Web Access, Tokeniser, String Prototyper and Data Access. Classes of external libraries are highlighted in grey.

The Keyword Manager. The keyword manager is responsible for finding a list of keywords that allow to obtain a representative collection of links when performing the corresponding searches in a given web site. As an example, consider a domain English-spoken web site like Amazon.com. The keyword manager chooses the words that are more likely to appear in the Amazon product database. Given that Amazon store offers a wide product range, a list of the most common English words would suffice to obtain a representative collection of links. Instead, another site like Microsoft Academic Search belongs to a more specific domain, so a list of the most cited authors, for example, would be more useful for this purpose.

The main concerns in this module are related to the language and type of words that are accepted by each site, specially stop words. Stop words tend to have a higher frequency, and they usually yield a higher number of links, but not every site allows searching using stop words. Consider Wordpress.com, a popular blog hosting site, which is unable to find any result related to the keywords 'a' or 'the', while the same words in Youtube.com yield respectively 32,800,000 and 40,000,000 results. Furthermore, stop words may deviate the search and deteriorate results. The lexical type of word is also to be considered, given that verbs are not as frequent as nouns, for example, so they may yield a smaller number of results. Other important factor is the domain to which the site belongs, that defines a specific vocabulary; for example, general, academic, technological or economical, amongst others.

Deep Web Access. The Deep Web Access module is responsible for the interaction with the Deep Web sites, including analysing and filling forms, retrieving response pages, and extracting links from those pages. This module is supported by Selenium Java Library, and it relies on the Keyword Manager that supplies the keywords for form filling. The main interface in this module is IExtractor, which includes all the former methods, and which can be implemented by any suitable library (e.g., Selenium or WebDriver).

For example, to extract information from Amazon.com, the extractor finds the following form in Amazon.com home page:

```
<form action="searchAction" name="site-search">
 <select name="url" id="searchDropdownBox" >
  <option value="aps">...</option>
 </select>
 <input type="text" id="twotabsearchtextbox" name="field-keywords" />
 <input type="image" src="http://images-amazon.com/images/..."/>
</form>
```

The extractor obtains a model that includes the form, with a name attribute with value site-search, and no id attribute, the three fields included in the form, and the submission method, which consists on clicking over the image.

One of the main issues to be solved by the extractor is the lack of standardisation in forms and fields identification. A well defined HTML page should include, at least, an identification attribute for each element, either an id, a name, or both of them, but actually in some web sites we find elements with none of them. In the latter case, the extractor has to deal with the problem of referencing that HTML element for later processing: in the case of a form, to submit it, or to fill it in if the element is a field. We can cite Youtube.com, which is the most visited video sharing web site, as an example of this lack of uniformity. In its home page we can find, amongst others, the following forms:

1. A form with id and no name: `<form id="masthead-search" action="/results" onsubmit="...">`
2. A form with name and no id `<form name="logoutForm" action="/">`
3. A form without name or id `<form action="/addtoajax">`

As for the link extraction, the main issue is that a single URL may be written in different formats, either in relative or absolute form. For example, in Amazon.com, link 1 can be written http://www.amazon.com/ref=logo, /ref=logo or ./ref=logo.

Tokeniser. The Tokeniser module is responsible for processing URLs extracted from the sites pages, and splitting them into tokens, using some configuration. Our implementation of the tokeniser is based on RFC 3986 recommendation for URIs, although not every site conforms to it. Sometimes URLs include special characters, spaces and other symbols that difficult URL parsing. Also, URL query strings are composed of parameters, which may be optional or mandatory, and which may be arranged in different orders.

The String Prototyper. The String Prototyper module is responsible for building a collection of string prototypes, using links extracted from a site, where each prototype is a regular expression.

The main IPrototypeCreator contains method markParts, which discerns variable parts of strings that must be abstracted to create prototypes. An example of implementation for this interface is ProbabilisticPrototypeCreator, which uses a probability-based technique to make the former distinction.

Once the prototypes are built, they are analysed and improved so that their classification results are more accurate. Finally, the prototyper helps the user to assign a label to each prototype, defining the semantic concept contained in the links of the cluster that the prototype represents, and to select relevant concepts. In the Amazon.com example, after processing the prototypes, we obtain the labeled prototypes included in Table 1. More details about the implementation of the prototyper are published at [11].

Table 1. Labeled prototypes obtained from Amazon.com

Label	Prototype (Regular Expression)	Coverage
Reviews	http://www.amazon.com/⋆/product-reviews/⋆?ie=UTF8	30%
Products	http://www.amazon.com/⋆/dp/⋆?ie=UTF8&sr=⋆	30%
Buy New	http://www.amazon.com/gp/offer-listing/ref=olp?ie=UTF8&sr=⋆&condition=new	8%
Buy Used	http://www.amazon.com/gp/offer-listing/ref=olp?ie=UTF8&sr=⋆&condition=used	6%

Data Access. Data access module is responsible for persisting all data in our system. It is based on Hibernate, which manages all objects persistence in a given database (in our case, Oracle). The main classes in this module are HibernateUtil, which contains all methods needed to create and manage a Hibernate session, and ModelManager, which includes methods to manage all objects, including creating, updating and deleting them, and also performing queries to retrieve those objects, using Hibernate support.

3 Conclusions

In this paper, we present an architecture for an efficient crawler that fills in and submits forms, accessing pages in the Deep Web. With respect to the requirements we mentioned in section 1, our proposal classifies web pages depending on the link URL format, so it is applicable to any web site. Hence, our proposal is not only efficient, but also generic and applicable in different domains. Also, our proposal is able to integrate existing form modeling proposals into our crawler, which makes it able to fill in and submit forms, hence dealing with the Deep Web. Finally, our classifier is trained using a set of links collected automatically. The system analyses them and gives the user a list of prototypes representing concepts, while the user is only responsible for defining his or her interest, by labeling and picking one or more prototypes. Not that users intervention is unavoidable, given that the relevancy criteria depends solely on them.

As a result, we designed an efficient crawler, able to access web pages automatically, while requiring as little intervention as possible from the users. A demo of our implementation of the link classifier is available in the author web site [1].

[1] http://www.tdg-seville.info/inmahernandez/CALA+Demo

References

1. Aggarwal, C.C., Al-Garawi, F., Yu, P.S.: On the design of a learning crawler for topical resource discovery. ACM Trans. Inf. Syst. 19(3), 286–309 (2001)
2. Anupam, V., Freire, J., Kumar, B., Lieuwen, D.F.: Automating web navigation with the webvcr. Computer Networks 33(1-6), 503–517 (2000)
3. Baumgartner, R., Ceresna, M., Ledermuller, G.: DeepWeb navigation in web data extraction. In: CIMCA/IAWTIC, pp. 698–703 (2005)
4. Bertoli, C., Crescenzi, V., Merialdo, P.: Crawling programs for wrapper-based applications. In: IRI, pp. 160–165 (2008)
5. Blythe, J., Kapoor, D., Knoblock, C.A., Lerman, K., Minton, S.: Information integration for the masses. J. UCS 14(11), 1811–1837 (2008)
6. Chakrabarti, S., Dom, B., Raghavan, P., Rajagopalan, S., Gibson, D., Kleinberg, J.M.: Automatic resource compilation by analyzing hyperlink structure and associated text. Computer Networks 30(1-7), 65–74 (1998)
7. Chakrabarti, S., van den Berg, M., Dom, B.: Focused crawling: A new approach to topic-specific web resource discovery. Computer Networks 31(11-16), 1623–1640 (1999)
8. Davulcu, H., Freire, J., Kifer, M., Ramakrishnan, I.V.: A layered architecture for querying dynamic web content. In: SIGMOD, pp. 491–502 (1999)
9. de Assis, G.T., Laender, A.H.F., Gonçalves, M.A., da Silva, A.S.: Exploiting Genre in Focused Crawling. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 62–73. Springer, Heidelberg (2007)
10. Edwards, J., McCurley, K.S., Tomlin, J.A.: An adaptive model for optimizing performance of an incremental web crawler. In: WWW, pp. 106–113 (2001)
11. Hernández, I., Rivero, C.R., Ruiz, D., Corchuelo, R.: A Tool for Link-Based Web Page Classification. In: Lozano, J.A., Gámez, J.A., Moreno, J.A. (eds.) CAEPIA 2011. LNCS, vol. 7023, pp. 443–452. Springer, Heidelberg (2011)
12. Mukherjea, S.: Discovering and analyzing world wide web collections. Knowl. Inf. Syst. 6(2), 230–241 (2004)
13. Pan, A., Raposo, J., Álvarez, M., Hidalgo, J., Viña, Á.: Semi-automatic wrapper generation for commercial web sources. In: Engineering Information Systems in the Internet Context, pp. 265–283 (2002)
14. Pant, G., Srinivasan, P.: Learning to crawl: Comparing classification schemes. ACM Trans. Inf. Syst. 23(4), 430–462 (2005)
15. Pant, G., Srinivasan, P.: Link contexts in classifier-guided topical crawlers. IEEE Trans. Knowl. Data Eng. 18(1), 107–122 (2006)
16. Partalas, I., Paliouras, G., Vlahavas, I.P.: Reinforcement learning with classifier selection for focused crawling. In: ECAI, pp. 759–760 (2008)
17. Raghavan, S., Garcia-Molina, H.: Crawling the hidden web. In: WWW (2001)
18. Wang, Y., Hornung, T.: Deep web navigation by example. In: BIS (Workshops), pp. 131–140 (2008)