HAL
archives-ouvertes.fr

Intelligent Content Acquisition in Web Archiving

Muhammad Faheem

▶ **To cite this version:**

Muhammad Faheem. Intelligent Content Acquisition in Web Archiving . Computer Science [cs]. TELECOM ParisTech, 2014. English. <NNT : 2014-ENST-0084>. <tel-01177622>

HAL Id: tel-01177622

https://tel.archives-ouvertes.fr/tel-01177622

Submitted on 17 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0084

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Muhammad Faheem

le 17 décembre 2014

Acquisition des contenus intelligents dans l'archivage du Web

Directeur de thèse : **Pierre Senellart**

Jury

M. Dario Rossi, Prof. Télécom ParisTech		
M. Giovanni Grasso, Researcher Univ. Oxford		
M. Philippe Rigaux, Prof. CNAM		
M. Pierre Senellart, Prof. Télécom ParisTech		directeur
Mme. Sihem Amer-Yahia, Directrice de Recherche CNRS		rapporteur
M. Valter Crescenzi, Prof. Univ. Roma Tre		rapporteur

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

THÈSE

Acquisition des contenus intelligents dans l'archivage du Web

Intelligent Content Acquisition in Web Archiving

Muhammad Faheem

Résumé

Les sites Web sont par nature dynamiques, leur contenu et leur structure changeant au fil du temps ; de nombreuses pages sur le Web sont produites par des systèmes de gestion de contenu (CMS). Les outils actuellement utilisés par les archivistes du Web pour préserver le contenu du Web collectent et stockent de manière aveugle les pages Web, en ne tenant pas compte du CMS sur lequel le site est construit et du contenu structuré de ces pages Web. Nous présentons dans un premier temps un *application-aware helper* (AAH) qui s'intègre à une chaine d'archivage classique pour accomplir une collecte intelligente et adaptative des applications Web, étant donnée une base de connaissance de CMS courants. L'AAH a été intégrée à deux crawlers Web dans le cadre du projet ARCOMEM : le crawler propriétaire d'Internet Memory Foundation et une version personnalisée d'Heritrix. Nous proposons ensuite un système de crawl efficace et non supervisé, ACEBot (Adaptive Crawler Bot for data Extraction), guidé par la structure qui exploite la structure interne des pages et dirige le processus de crawl en fonction de l'importance du contenu. ACEBot fonctionne en deux phases : dans la phase hors-ligne, il construit un plan dynamique du site (en limitant le nombre d'URL récupérées), apprend une stratégie de parcours basée sur l'importance des motifs de navigation (sélectionnant ceux qui mènent à du contenu de valeur) ; dans la phase en-ligne, ACEBot accomplit un téléchargement massif en suivant les motifs de navigation choisis. L'AAH et ACEBot font 7 et 5 fois moins, respectivement, de requêtes HTTP qu'un crawler générique, sans compromis de qualité. Nous proposons enfin OWET (Open Web Extraction Toolkit), une plate-forme libre pour l'extraction de données semi-supervisée. OWET permet à un utilisateur d'extraire les données cachées derrière des formulaires Web.

Abstract

Web sites are dynamic by nature with content and structure changing overtime; many pages on the Web are produced by content management systems (CMSs). Tools currently used by Web archivists to preserve the content of the Web blindly crawl and store Web pages, disregarding the CMS the site is based on and whatever structured content is contained in Web pages. We first present an *application-aware helper* (AAH) that fits into an archiving crawl processing chain to perform intelligent and adaptive crawling of Web applications, given a knowledge base of common CMSs. The AAH has been integrated into two Web crawlers in the framework of the ARCOMEM project: the proprietary crawler of the Internet Memory Foundation and a customized version of Heritrix. Then we propose an efficient unsupervised Web crawling system ACEBot (Adaptive Crawler Bot for data Extraction), a structure-driven crawler that utilizes the inner structure of the pages and guides the crawling process based on the importance of their content. ACEBot works in two phases: in the offline phase, it constructs a dynamic site map (limiting the number of URLs retrieved), learns a traversal strategy based on the importance of navigation patterns (selecting those leading to valuable content); in the online phase, ACEBot performs massive downloading following the chosen navigation patterns. The AAH and ACEBot makes 7 and 5 times, respectively, fewer HTTP requests as compared to a generic crawler, without compromising on effectiveness. We finally propose OWET (Open Web Extraction Toolkit) as a free platform for semi-supervised data extraction. OWET allows a user to extract the data hidden behind Web forms.

Mots clefs : système de gestion de contenu, crawling, archivage du Web, XPath

Keywords: content management system, crawling, Web archiving, XPath

À l'exception de l'annexe A, qui propose une adaptation en français du chapitre 3, cette thèse est rédigée en anglais.

With the exception of Appendix A, an adaptation into French of the Chapter 3, this thesis is written in English.

Cette thèse est rédigée à l'aide du système de composition de documents LaTeX. Les bandes dessinées introduisant chaque chapitre ont été prises de `www.phdcomics.com`. Ils sont reproduits ici en vertu du droit de citation.

This thesis is written with the help of the typesetting system LaTeX. The comic strips that introduce each chapter are from `www.phdcomics.com`. They are reprinted here under fair use.

Contents

List of Algorithms

List of Figures

List of Tables

If all you can do is crawl, start crawling.

Rumi

Life can be like a roller coaster...
And just when you think you've had enough, and your ready to get off the ride and take the calm, easy merry-go round...
You change your mind, throw you hands in the air and ride the roller coaster all over again.
That's exhilaration... that's living a bit on the edge... that's being ALIVE.

Stacey Charter

Learn from yesterday,
live for today,
hope for tomorrow.
The important thing is not to stop questioning.

Albert Einstein

Acknowledgments

First and foremost, I would like to express the words of gratitude to my scientific advisor Pierre Senellart who taught me the new horizons of research. I have always admired his rigorous scientific approach, extensive knowledge, and problem solving skills. He was always available for advice and I am very thankful for his support and time he spent working with me on papers, presentations, and new ideas. I am also very thankful for his generosity, kindness, optimism, and endless patience, for which I remain indebted. Working with him has been both a pleasure and privilege!

I would like to thank the reviewers of my thesis, Valter Crescenzi and Sihem Amer-Yahia, who accepted to spend time reading this thesis and giving a feedback; and Giovanni Grasso, Philippe Rigaux and Dario Rossi who accepted to be the thesis committee members and evaluate this thesis. I also would like to thank Stéphane Gançarski for his feedback during mid-term evaluation.

I had the honor of spending two months in Oxford, visiting the DIADEM lab, and working with three extraordinary computer scientists Giovanni Grasso, Tim Furche, and Christian Schallhart. Oxford is a very nice city, and Giovanni's quick mind and skill at software engineering is equal to his impressive experience in developing large-scale applications. It was a pleasure working with him and I learned a lot about developing interesting prototypes.

This PhD would not have been possible without the scholarship funded by the European Union's Seventh Framework Program (FP7/2007–2013) under grant agreement 270239 (ARCOMEM). It has been an honor to be part of ARCOMEM project. My profound thanks to Wim Peters, Thomas Risse, Vassilis Plachouras, Julien Masanès, Yannis Stavrakas, Florent Carpentier, Dimitris Spiliotopoulos, Alex Jaimes, Robert Fischer, Gerhard Gossen, Nikolaos Papailiou, and Patrick Siehndel.

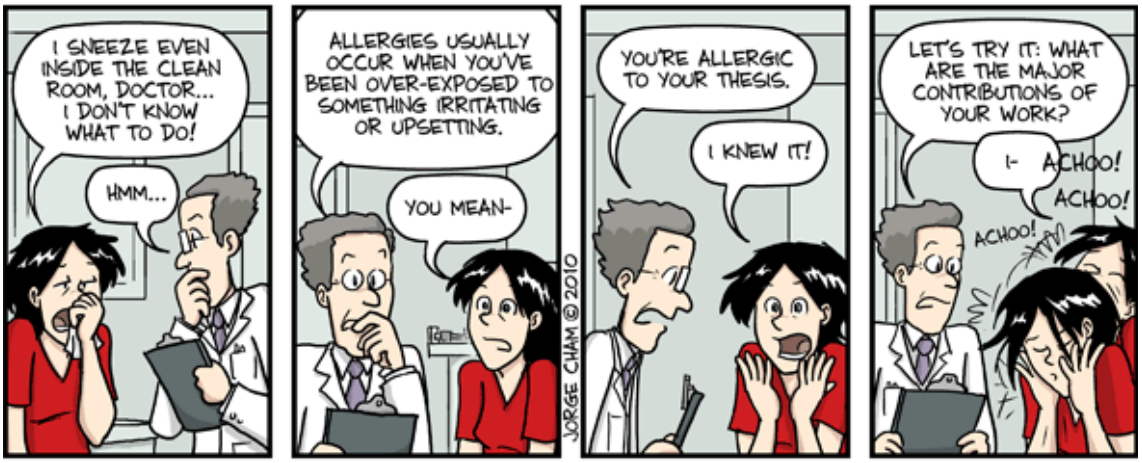
During the years spent at Télécom ParisTech, I had the chance of being surrounded by people who have inspired me through their genuine passion and hard work. I am

very grateful to the DBWeb Team, including Talel Abdessalem, Bogdan Cautis (former member), Fabian M. Suchanek, Antoine Amarilli, Mouhamadou Lamine Ba, Roxana Gabriela Horincar, Imen Ben Dhia (former member), Silviu Maniu (former member), Asma Souihli (former member), Evgeny Kharlamov (former member) and Georges Gouriten (former member). Please forgive in advance all omissions that I am bound to have made in such a list. The atmosphere all of you created in the group is very pleasant and inspiring.

Let me also thank my family and friends, that have supported me these last few years and who helped me overcome some of the most stressful moments. I would like to thank my dad Muhammad Sharif and mom Naseem Sharif, who have supported me these last few years and encouraged me when I was down. Wholehearted thanks to my brother Shaukat Waseem, and sisters who has always been there for me. My special thanks to my childhood friends Imran Cheema, and Seith Khalid for bearing with me!

Finally and on a more personal note, closest to my heart is my wife, who went with me through the hardest part of the PhD times, and who encouraged me in going forward. Her love, belief in my abilities helped me overcome all those many bitter moments. I am forever grateful to her.

Introduction



The World Wide Web has become an active publishing system and is a rich source of information, thanks to contributions of hundreds of millions of Web users, who use the social Web as a medium for broadcasting their emotions, publishing content, discussing political issues, sharing videos, posting comments, and also stating their personal opinion in ongoing discussions. Part of this public expression is carried out on social networking and social sharing sites (Twitter, Facebook, Youtube, etc.), part of it on independent Web sites powered by content management systems (CMSs, including blogs, wikis, news sites with comment systems, Web forums). Content published on this range of *Web applications* does not only include the ramblings of common Web users but also pieces of information that are newsworthy today or will be valuable to tomorrow's historians. Barack Obama thus first announced his 2012 reelection as US president on Twitter [Jup12]; blogs are more and more used by politicians both to advertise their political platform and to listen to citizen's feedback [Col08]; Web forums have become a common way for political dissidents to discuss their agenda [MC02]; initiatives like the Polymath Project* transform blogs into collaborative research whiteboards [Nie11]; user-contributed wikis such as Wikipedia contain quality information to the level of traditional reference materials [Gil05].

Because Web content is distributed, perpetually changing, often stored in proprietary platforms without any long-term access guarantee, it is critical to preserve this valuable material for historians, journalists, or social scientists of future generations. This is the objective of the field of *Web archiving* [Mas06], which deals with discovering, crawling, storing, indexing, and ensuring long-term access to Web data. Hence, many efforts have been made to utilize the Web as an information repository. However, the existing tools used by Web archivists blindly crawl and preserve the Web pages, disregarding the nature of Web sites currently accessed (which leads to suboptimal crawling strategies) and whatever

*`http://polymathprojects.org/`

structured information is contained in Web pages (which results in page-level archives whose content is hard to exploit). In this PhD work, we overcome these limitations and introduce flexible, adaptive, and intelligent content acquisition in Web archiving. We have developed intelligent systems (AAH, see Chapters 3, 4; and ACEBot, see Chapter 5) for crawling publicly accessible Web sites. Indeed these systems have reduced requirements in bandwidth, time, and storage (by using fewer HTTP requests, avoiding duplicates) using limited computational resources in the process. In this thesis, we also report on a collaboration with the DIADEM lab* at the University of Oxford, in which we have developed a prototype, OWET (see Chapter 6), to harvest Web pages hidden behind Web forms.

In Chapter 1, we first give some preliminary knowledge to help understand the research domain in general. Firstly, we introduce the Web technologies used for encoding and querying Web documents. Secondly, we introduce the Web browser and client-side scripting language (JavaScript) that one can use to develop dynamic Web sites. Finally, to understand how current archiving approaches are not fully up to the task of Web archiving, we present the architecture of a traditional Web crawler. We explain how a traditional Web crawler crawls the Web in a conceptually very simple way.

In Chapter 2, we discuss related work in our area of research. We briefly summarize the state of the art in Web archiving, Web crawling, structure-driven crawling, hidden-Web crawling, focused crawling, forum crawling, and Web data extraction.

In Chapter 3, we present the *application-aware helper* (AAH) that fits in an archiving crawl processing chain to perform intelligent and adaptive crawling of Web sites. Here, we claim that the best crawling strategy to crawl these Web sites is to make the crawler aware of the kind of Web sites currently processed. Because the AAH is aware of the Web site currently crawled, it is able to refine the list of URLs to process, and also annotate the archive with information about the structure of crawled Web content. The AAH is assisted by a knowledge base of Web application type to make the crawling process intelligent and adaptive to the nature of the Web application. This knowledge base specifies how to detect a specific Web application, and which crawling and content extraction methodology is appropriate. The knowledge base is described in a custom XML format, so as to be easily shared, updated, and (hopefully) managed by non-programmers.

The AAH has been integrated into two Web crawlers (see Chapter 4): the proprietary crawler of the Internet Memory Foundation [Intb], with whom we have closely collaborated, and into a customized version of Heritrix [Sig05], developed by the ATHENA research lab in the framework of the ARCOMEM project [ARC13]. The application-aware helper (AAH) has been introduced in the ARCOMEM crawling processing chain in place of the usual link extraction function, in order to identify the Web application that is currently being crawled, and decide on and categorize the best crawling actions.

The AAH has introduced a semi-automatic crawling approach that relies on hand-written description of known Web sites. Beyond the scope of the ARCOMEM project, our focus is on *fully automatic* system that does not require any human intervention to crawl publicly accessible Web sites. In Chapter 5, we propose ACEBot (Adaptive Crawler Bot for data Extraction), a structure-driven crawler (fully automatic) that utilizes the inner structure of the Web pages and guides the crawling process based on the importance of their content. ACEBot works in two phases: in the *offline* phase, it constructs a dynamic site map

*`http://diadem.cs.ox.ac.uk/`

(limiting the number of URLs retrieved), learns the best traversal strategy based on the importance of *navigation patterns* (selecting those leading to valuable content); in the *online phase*, ACEBot performs massive crawling following the chosen navigation patterns. ACEBot ignores duplicate and invalid pages and crawls only important content with high precision and recall.

A large part of the information on the Web is hidden behind Web forms (known as the deep Web, or invisible Web, or hidden Web). The fact that the important information lies in databases behind forms means that a deeper crawl is required than the usual surface crawl. In Chapter 6, we propose OWET (Open Web Extraction Toolkit) as such a platform, a free, publicly available data extraction framework. OWET encompasses three components: *OWETDesigner*, a Firefox plugin acting as visual front-end to design and validate wrappers; *OWETGenerator*, for robust wrapper induction from few examples; *OWETExecutor*, for large-scale wrapper scheduling and execution. We mostly worked on the development of OWETDesigner, and also collaborated on its integration with other components.

All ideas presented in this thesis have been implemented in prototypes to validate the approach and test the systems. The AAH and ACEBot are implemented in Java, whereas OWET is developed as a Mozilla Firefox plugin.

To summarize, we highlight the four main contributions of our thesis:

(1) We introduce an *application-aware helper* (AAH) that fits into an archiving crawl processing chain to perform intelligent and adaptive crawling of Web sites (see Chapter 3).

(2) The AAH is part of the ARCOMEM crawling architecture (see Chapter 4), where the purpose of this module is to make the large-scale crawler aware of the type of Web site currently processed, refine the list of URLs, and extract structured information (e.g., author, timestamp, comment) from crawled Web pages.

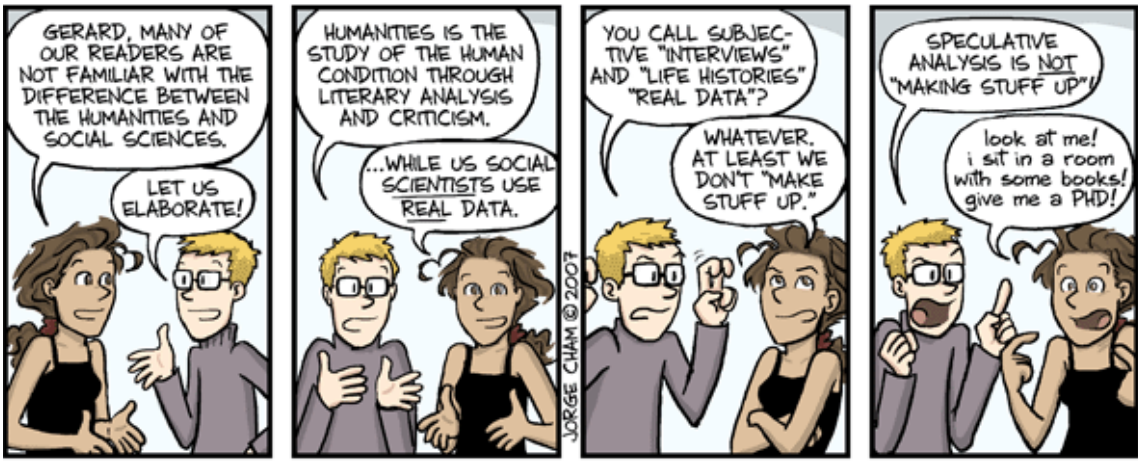
(3) We propose ACEBot (Adaptive Crawler Bot for data Extraction), that utilizes the inner structure of Web pages and guides the crawling process based on the importance of their content (See Chapter 5).

(4) We introduce OWET (Open Web Extraction Toolkit), implemented as Mozilla Firefox plugin that harvest the Web pages from deep Web (see Chapter 6).

Appendix A is a translation into French of Chapter 3.

Chapter 1.

Preliminaries



In order for this work to be self-contained to the greatest extent possible, we introduce the foundations of relevant Web technologies and research.

1.1. Markup Languages

We start with a brief discussion of common markup languages used for writing Web documents. Markup languages offer a means of annotating text with some specific metadata information, which varies depending on the markup language used. For instance, the Hypertext Markup Language (HTML) describes presentation semantics, and structured documents rendered in a Web browser are human readable. In contrast, the Extensible Markup Language (XML) describes the content and is aimed towards both human and machine readability. In the markup languages, the Web pages are annotated (hyperlinked) with *tags*. For example, consider the following encoding:

```
<a href="http://www.example.org">
Example
</a>
```

In the element above, the text "Example" is annotated as a hyperlink. The start tag *<a>* has an attribute (*@href*), which provides a Uniform Resource Locator (URL) for the hyperlink.

My first heading

My first paragraph

Figure 1.1.: Sample HTML code output

1.1.1. HTML

The Hypertext Markup Language (HTML) is a language for describing (rather than displaying) Web pages. An HTML document contains HTML *tags* and plain text. HTML provides a mean of rendering standardized structured documents on various platforms, with features such as forms, tables, images, as well as embedded objects. HTML allows the developer to use Cascading Style Sheet (CSS) to define a standardized layout and presentation of Web pages. Let us consider the following HTML code:

```
<html>
 <body>
    <h1>My first heading</h1>
    <p>My first paragraph</p>
 </body>
</html>
```

A Web browser has displayed the code above as shown in Figure 1.1.

HTML forms are used to select different kinds of user input and allow user to submit the inputs to a server to illicit some response. This response can vary depending on Web sites, but usually performs an update to the underlying database (e.g., placing an order on ebay.fr) or signals a query submission (e.g., determining the room availability and pricing on tripadvisor.com). An HTML form can contain input elements such as text fields, radio-buttons, checkboxes, submit buttons, selection lists, fieldsets, label elements, etc. Consider the following HTML code:

```
<form>
  <legend>Personal information:</legend>
  Name: <input type="text" size="30"><br>
  E-mail: <input type="text" size="30"><br>
  Date of birth: <input type="text" size="10"><br>
  <input type="radio" name="sex" value="male">Male<br>
  <input type="radio" name="sex" value="female">Female<br>
  <input type="submit" value="Submit">
</form>
```

The Web browser has displayed the above code as shown in Figure 1.2. The Web forms can be submitted as GET (input data is sent in the URL) or POST (input data is sent in the HTTP message body) requests. For a more detailed study of HTML, please see [RHJ99]. Further, for full specification of CSS, please visit [Ete08]

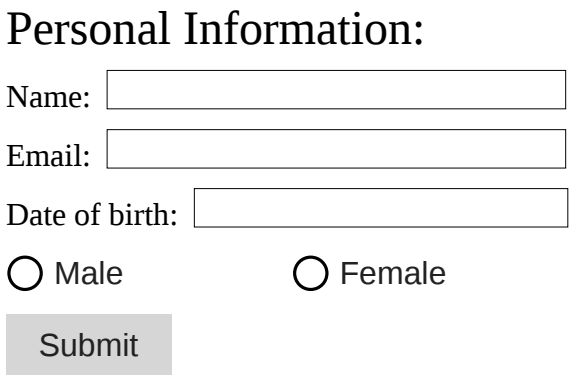


Figure 1.2.: Sample HTML form output

1.1.2. XML and XHTML

The extensible Markup Language (XML) is designed to describe data (not to display data). XML tags are not predefined and a set of rules are provided for adding markup to data. It is important to understand that XML is not a replacement for HTML, but rather a complement. When writing Web applications, XML is used to described the data, while HTML is used to format and display the data. Indeed, XML is a software and hardware independent tool. XML allows us to define the schema (e.g., DTD) for XML documents. An XML schema introduces a way for people to agree on a standard for interchanging data and provides greater flexibility and expressiveness for organizing the structured data.

A Web document written with XML is self-descriptive. Consider the following simplified example:

```
<?xml version="1.0"?>
2
<publications>
4
  <paper>
6   <title>Demonstrating Intelligent Crawling and Archiving of Web
        Applications</title>
    <author>Muhammad Faheem</author>
8   <author>Pierre Senellart</author>
    <booktitle>Proc. CIKM</booktitle>
10  <year>2013</year>
  </paper>
12
  <paper>
14  <title> Intelligent and Adaptive Crawling of Web Applications for Web
        Archiving</title>
    <author>Muhammad Faheem</author>
16  <author>Pierre Senellart</author>
    <booktitle>Proc. of ICWE</booktitle>
18  <year>2013</year>
```

```
    </paper>
20
</publications>
```

The example above specifies the set of publications, for instance one can store easily this structured informations in a database. The XML schema followed by a document provides a mean to query the underlying data for further analysis.

Interestingly, each Web page can be represented as a tree and each node (individual tag) can have namespace prefixes, delimited by an identifier followed by the ":" character preceding a node name (tag name). XML documents can have various source schemas, therefore, namespace identifier are very important and thus serve to uniquely identify node types during schema verification activities. The querying capability of XML has pushed introducing a cleaner and stricter version of HTML. Therefore, HTML has been extended to the Extensible Hypertext Markup Language (XHTML). XHTML is a proper subset of XML. XHTML is specified by schemas; these schemas can be found at [PtWHWG00]. Please note that, like HTML, XHTML provides a presentation semantics for creating structured documents viewable by human. For a full description of XML, please see [SMMB+08].

1.2. Querying Web Documents

The querying capability of XML has provided a way to find items of interest in an XML document. The query languages (e.g., XPath, XQuery) utilize the XML tree structure representation and apply more efficient strategies to execute queries on the document object or any of the nodes in the document. Before we discuss the query languages, let us briefly consider the model provided by the Document Object Model (DOM).

1.2.1. DOM

The Document Object Model (DOM) is a cross-platform and language-independent API that provides a means to interact with HTML and XML documents. It specifies the logical structure of documents and how programmers can navigate their structure, and add, delete, or modify elements and content. Please note that the DOM is not a data model itself. Let us consider the following table (example reprinted from [W3C98]):

```
<TABLE>
2     <TBODY>
      <TR>
4     <TD>Shady Grove</TD>
      <TD>Aeolian</TD>
6     </TR>
      <TR>
8     <TD>Over the River, Charlie</TD>
      <TD>Dorian</TD>
10    </TR>
      </TBODY>
12 </TABLE>
```

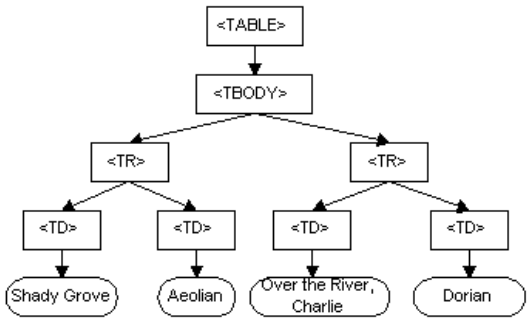


Figure 1.3.: DOM representation (reprinted from [W3C98])

The DOM representation of the table above is shown in Figure 1.3. This representation is very much like a tree; however in order to avoid ambiguities, the term *structure model* is preferred to describe this tree-like representation.

For a more detailed description, please see the DOM specification [W3C98].

1.2.2. XPath

The XML Path language (XPath) is a query language for fetching information in an XML document. XPath utilizes the same tree representation introduced in Section 1.2.1 to build the path expressions to select nodes or node-sets in an XML document. The path expressions very much resemble the traditional Unix-based file system navigating expressions. Consider the example XML document given in Section 1.1.2, then the XPath expression

```
//paper/ title
```

would return the following:

```
<title>Demonstrating Intelligent Crawling and Archiving of Web Applications</title>
<title> Intelligent and Adaptive Crawling of Web Applications for Web
    Archiving</title>
```

In XPath, there are seven kinds of nodes that include *element*, *attribute*, *text*, *document nodes*, *comment*, *processing-instruction*, and *namespace*. XPath navigational axes define a nodeset relative to the context nodes. The axes that are defined in specification are *ancestor*, *ancestor-or-self*, *attribute*, *child*, *following*, *following-sibling*, *descendant*, *descendant-or-self*, *namespace*,*parent*, *self*, *preceding*, and *preceding-sibling*. Figure 1.4, reprinted from [GVD04], depicts the XPath navigation axes within a DOM tree. For a more in-depth description, please see the XPath specification [W3C99].

1.2.3. Other Technologies for Querying XML

One of the great strength of XML is its flexibility in representing information from diverse sources and XML query language must provide features for manipulating information from these diverse sources. XQuery [W3C07a] is a superset of XPath 2.0 and is designed to meet the requirements [W3C07b] identified by W3C XML Query working group. Further,

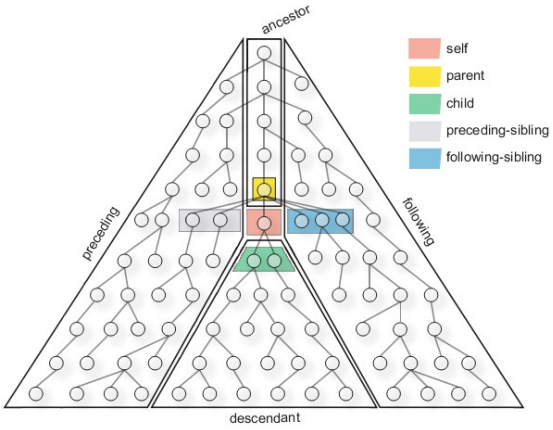


Figure 1.4.: XPath Axes (reprinted from [GVD04])

significant efforts has been conducted to couple the XML and Relational Database Management System (RDBMS). A survey [MB06] of these technologies has discussed in detail some special challenges and opportunities in this area of research.

1.3. Web Browsers

A Web browser is an application to access and view the information resources (i.e., resources described by a URL) on the World Wide Web. An information resource may be a Web page, image, audio, video or some other content. Hyperlinks allow users to navigate their browser to the related resources. The common Web browsers include Firefox, Microsoft Internet Explorer, Google Chrome, Opera, and Apple Safari. There are also many headless Web browsers such as HtmlUnit, ZombieJS, Ghost, etc., which do not have a Graphical User Interface, but still access the Web pages (without showing them to a human), for instance, to provide the content of Web pages to other systems. The headless browsers are very important and almost behave like a normal browser. For example, HtmlUnit, allows high level manipulation (e.g., filling and submitting forms, clicking hyperlinks) of Web sites from a Java application. More precisely, such browser can be used for test automation, Web scraping or for downloading the Web content.

1.4. Client-side Web Scripting

Client-side scripting languages are integrated into Web browsers and JavaScript is the most popular language in this context. JavaScript can manipulate the DOM to change the HTML content (e.g., style). This allows to develop dynamic Web pages with Web forms and objects. In an HTML document, JavaScripts are inserted between <script> and </script> tags, whereas JavaScript can also be placed in external files (e.g., publications.js). External JavaScripts do not contain <script> tags and reference to these scripts can be placed in <head> or <body> tags. Since the JavaScripts are executed on the client side, many Web form validation steps (e.g., validating a date, if it is in correct format) can be performed without sending the requests to the Web server. This indeed saves bandwidth

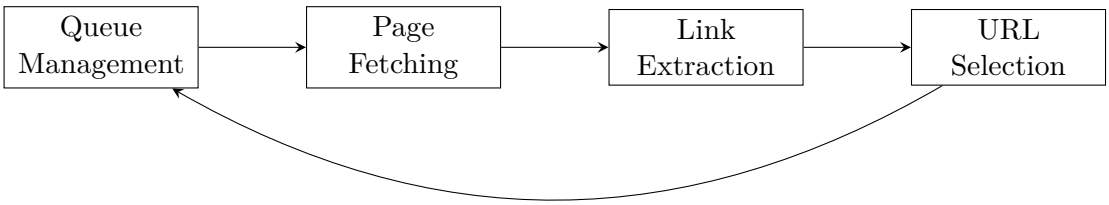


Figure 1.5.: Traditional processing chain of a Web crawler

and strain on Web server. For in-depth discussion, please visit Mozilla's official JavaScript pages at `http://developer.mozilla.org/en/docs/Web/JavaScript`.

1.5. Crawling the Web

Many public and private institutions are archiving large collections of public content. This is the case of *Internet Archive*[*], *Internet Memory*[†], or a variety of national libraries around the world. Robots of commercial search engine such as `Google` and `Bing` crawl information from user-created content (UCC) on the Web to improve the quality of search results. Though Web content exists in large amount, due to limited bandwidth, storage, or indexing capabilities, only a small fraction of the content can be harvested by Web crawlers. This is true for crawlers of institutions with limited resources (e.g., the national library of a small country). This is even true for a company such as `Google`, that has discovered more than a trillion unique URLs [AH08] in the frontier, but indexed around 40 billions Web pages (as of June, 2013) [dK13]. This suggests a need to develop a crawling strategy that not only effectively crawls Web content from template-based Web sites, but also efficiently minimizes the number of HTTP requests by avoiding non-interesting Web pages.

1.6. Web Crawler

A *Web crawler* (also known as a *robot* or a *Web spider*) is a computer program that browse the World Wide Web (WWW) in a methodical, automated and orderly fashion. Web crawlers have been used for several purposes. Web search engines (e.g., Google) have been using Web crawlers for the purpose of Web indexing or/and for updating Web content. They assemble a corpus of Web pages, index them, and allow users to issue queries against the Web index and find the Web pages that satisfy the queries. A related use is Web data mining, where Web pages are analyzed for statistical properties, or for performing Web data analytics. Web crawlers are also used in the chain of Web monitoring services, but one of the most prominent use is Web archiving (e.g., Internet Archive [Inta], Internet Memory Foundation [Intb]), where a large set of Web pages are harvested and archived in a repository for public use, for future research, and for historian.

Traditional Web crawlers crawl the Web in a conceptually very simple way. Consider the simplified architecture of a traditional Web crawler (such as Heritrix [Sig05]) depicted in Figure 1.5. They start from a *seed* list of the URLs to be stored in a queue (e.g., the

[*] `http://archive.org/`

[†] `http://internetmemory.org/`

starting URL may be the homepage of a Web site). Web pages are then fetched from this queue one after the other and links are extracted from these Web pages. If they are in the scope of the archiving task, the newly extracted URLs are added to the queue. This process ends after a specified time or when no new interesting URLs can be found.

A survey [ON10] has discussed the challenges faced by Web crawling:

Web scale The Web is a very large content repository and is continually growing. Nowadays search engines require thousand of computers and dozens of high-speed links to index Web pages. According to a study [GS05] large-scale search engines index no more than 40-70% of the indexable Web.

Social obligations Crawlers should be "good citizens" and must obey the social obligations of the Web. Twitter stores a vast amount of data but only allows a few number of requests in a 15-minute period on its API [Twi13]. Indeed, large scale crawlers should not impose too much request on the Web site they crawl, otherwise they may carry out a denial of service attack.

Content selection tradeoffs Web crawler should be selective in content retrieval. The goal should be to acquire the high-value content with high precision and recall; and avoid low quality, redundant, irrelevant, and malicious content.

Adversaries Crawlers should be intelligent to ignore the Web links that (mis)direct the traffic to commercial Web sites.

Web spam Web spam (a.k.a. search spam) describes Web pages that are designed to spam search results to improve the result page ranking. Web spam is a big headache for today's search engines. Indeed, spam Web sites do not contain any useful information. Crawling such Web sites is just a waste of time, effort and storage space. Several techniques have been employed by spam Web sites in order to get higher rankings than they deserve. These techniques can be categorized as hiding or boosting [CM07]. In April 2012, Google has announced its Penguin-codenamed "Webspam Algorithm Update" that better identifies Web sites using aggressive Web spam tactics.

In this PhD work, we aim at tackling these challenges and introducing more intelligent crawling techniques (see Chapters 3, 4, and 5).

1.6.1. GNU Wget

GNU wget* is a baseline crawler and has been used as a basis for the GUI-based GWget crawler. This crawler is very simple and supports bulk downloading via the HTTP, HTTPS, and FTP protocols. GNU wget has many features (e.g., recursive downloading, conversion of hyperlinks for offline viewing) to retrieve large files or even to mirror a whole Web site.

1.6.2. Heritrix

Heritrix is Internet Archive's [Inta] open-source, extensible, and Web scale crawler which is well spread and fairly well used among archiving institutions. It is a pure Java program. In different areas of the crawl (e.g., pre-fetch chain, queue distribution) several behaviours

*`http://www.gnu.org/software/wget/`

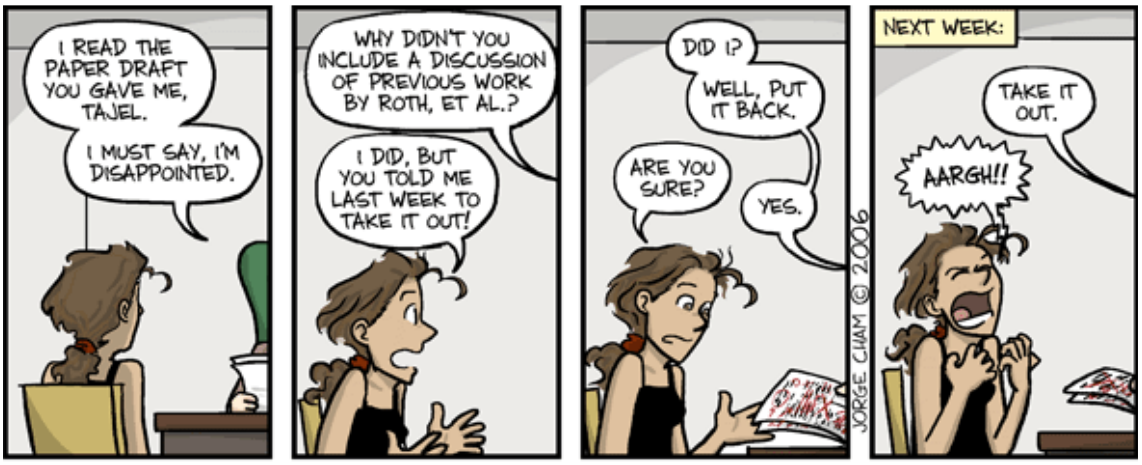
are implemented in specific classes. The use of these classes, their parameters when applicable, and their order of use can be configured. This is done in a job configuration: a fairly complex XML file that reflects the Java configuration objects. Specifying a complex behaviour demands a deep knowledge of Heritrix's internals. In addition, at runtime, the crawler can be controlled through a RESTful interface. Basic start/stop/pause commands can be easily achieved by simple HTML GETs and POSTs, however for more complex behaviour like pausing the fetching of a specific site or adjusting the scope of the crawl one needs to know the code of Heritrix in order to use the enhanced scripting functionality offered for this purpose.

1.6.3. Internet Memory Foundation Crawler

The Internet Memory Foundation (IMF, partner of the ARCOMEM project, formerly known as European Archive or EA) is currently developing its own crawler. It is designed from the start as a distributed crawler. This is proprietary software, entirely implemented in Erlang. Configuration and control of a crawl, however specific, is done through a user-friendly Web interface.

Chapter 2.

State of the art



2.1. Web Archiving

Web archiving refers to the collection and long-term preservation of data available on the Web [Mas06]. Since archiving the whole Web is a very challenging task due to its size and dynamics, there have been several national initiatives for preserving the Web of a country, based on full crawls in Sweden [APM00] and on a selective collection of Web pages in the United Kingdom [BT06] and Australia [CWW09]. The former approach aims at providing complete snapshots of a domain taken at regular intervals. A drawback of this approach is the lack of knowledge about changes of Web pages between crawls and the consistency of the collected data [SDM+09]. The latter approach results in higher quality collections restricted only to selected Web sites. Denev et al. [DMSW09] introduce a framework for assessing the quality of archives and tuning the crawling strategies to optimize quality with given resources. Gomes et al. [GMC11] provide a survey of Web archiving initiatives.

2.2. Web Crawling

Web crawling is a well-studied problem with still ongoing challenges. A survey of the field of Web archiving and archival Web crawling is available in [Mas06, ON10].

Content in Web applications or *content management systems* is arranged with respect to a template (template components may for example include left or right sidebar of the Web page, navigation bar, header and footer, main content, etc.). Among the various works on

template extraction, Gibson, Punera, and Tomkins [GPT05] have performed an analysis of the extent of template-based content on the Web. They have found that 40–50% of the Web content (in 2005) is template-based, i.e., part of some Web application. Their findings also suggested that template-based Web pages are growing at the rate of 6–8% per year. This research is a strong hint at the benefit of handling Web application crawling in a specific manner.

Descriptions of early versions of Google's and Internet Archive's large-scale crawler systems appeared in [BP98] and [Bur97], respectively. However, one of the first detailed descriptions of a scalable Web crawler is that of Mercator by Heydon and Najork [HN99], who provide information on the various modules of the crawler and the design options. Najork and Heydon also describe a distributed crawler based on Mercator in [NH02]. Shkapenyuk and Suel [SS02] introduce a distributed and robust crawler, managing the failure of individual servers. Heritrix [MKSR04] is an archival-quality and modular open source crawler, developed at the Internet Archive. Boldi et al. [BCSV04] describe UBI-Crawler, a distributed Web crawler, implemented in Java, which operates in a decentralized way and uses consistent hashing to partition the domains to crawl across the crawling servers. Lee et al. [LLWL09] describe the architecture and main data structures of IRL-Bot, a crawler which implements DRUM (Disk Repository with Update Management) for checking whether a URL has been seen previously. The use of DRUM allows IRLBot to maintain a high crawling rate, even after crawling billions of Web pages.

As the Web evolves, and Web pages are created, modified, or deleted [NCO04, FMNW03], effective crawling approaches are needed to handle these changes. Cho and Garcia Molina [CGM00] describe an incremental crawler for optimizing the average freshness of crawled Web data. Olston and Pandey [OP08] describe re-crawling strategies to optimize freshness based on the longevity of information on Web pages. Pandey and Olston [PDO04] also introduce a parameterized algorithm for monitoring Web resources for updates and optimizing timeliness or completeness depending on application-specific requirements.

While crawling appears to be a simple process, there are several associated challenges, especially when the aim is to crawl a large number of Web pages [ON10], in order to create the index of a Web search engine, or to archive them for future reference.

2.3. Focused Crawling

Focused crawling was first proposed in [CvdBD99] to attempt to only crawl Web pages that are relevant to a predefined topic or to a set of labeled examples. Focused or topical crawlers [GMS14] provide an effective way to balance the cost, coverage, and quality aspects of data collection from the Web [THCG05], by selectively crawling pages that are relevant to a set of topics, defined as a set of keywords [MPSR01], by example documents mapped to a taxonomy of topics [CvdBD99], or by ontologies [HNVV03, ISH$^+$96, EM03]. Recent approaches also address the crawling of information for specific geographical locations [AB09, GLM06].

The main challenges in focused crawling relate to the prioritization of URLs not yet visited, which may be based on similarity measures [MPSR01, HNVV03], hyperlink distance-based limits [DBP94, HJM$^+$98], or combinations of text and hyperlink analysis with Latent Semantic Indexing (LSI) [AKP07]. Machine learning approaches, including naïve Bayes classifiers [CvdBD99, DCL$^+$00], Hidden Markov Models [LJM06], reinforcement

learning [PPV08], genetic algorithms [JTG03], and neural networks [ZKK08], have also been applied to prioritize the unvisited URLs.

However, a *focused*, or *goal-directed*, crawler, crawls the Web according to a predefined set of topics [CvdBD99]. This approach is a different way of influencing the crawler behavior, not based on the structure of Web applications as is our aim, but on the content of Web pages. Our proposed approaches (*AAH* [4], see Chapter 3; and *ACEBot* [9], see Chapter 5) do not have the same purpose as focused crawling: they aim instead at a better archiving of the Web applications by exploiting their inner structure of Web pages. Both strategies for improving a conventional crawler are thus complementary.

Focused crawlers can be used for creating focused Web archives, by relying on a selective content acquisition approach. The crawling process in the ARCOMEM architecture [1] (see Chapter 4) changes the paradigm in how content is collected technically via Web crawling, by performing selective crawls and also leveraging information found in online social media.

2.4. Forum Crawling

Though application-aware crawling in general has not yet been addressed, there have been some efforts on content extraction from Web forums [GLZZ06, CYL+08].

The first approach [GLZZ06], dubbed Board Forum Crawling (BFC), leverages the organized structure of Web forums and simulates user behavior in the extraction process. BFC deals with the problem effectively, but is still confronted to limitations as it is based on simple rules and can only deal with forums with some specific organized structure.

The second approach [CYL+08], however, does not depend on the specific structure of the Web forum. The iRobot system assists the extraction process by providing the sitemap of the Web application being crawled. The sitemap is constructed by randomly crawling a few pages from the Web application. This process helps in identifying the rich repetitive regions that are then further clustered according to their layouts [ZSWW07]. After sitemap generation, iRobot obtains the structure of the Web forum in the form of a directed graph consisting of vertices (Web pages) and directed arcs (links between different Web pages). Furthermore a path analysis is performed to provide an optimal traversal path which leads the extraction process in order to avoid duplicate and invalid pages. A later effort [YT12] analyzed iRobot and identified a few drawbacks. [YT12] extends over the original system in a number of way: a better minimum spanning tree discovery technique [Edm67], a better measure of the cost of an edge in the crawling process as an estimation of its approximate depth in the site, and a refinement of the detection of duplicate pages.

iRobot [CYL+08, YT12] is probably the work closest to ours. In contrast with that system, the AAH we propose is applicable to any kind of Web application, as long as it is described in our knowledge base. Also differently from [CYL+08, YT12], where the analysis of the structure of a forum has to be done independently for each site, the AAH exploits the fact that several sites may share the same content management system. Our system also extracts structured and semantic information from the Web pages, where iRobot stores plain HTML files and leaves the extraction for future work. We have drawn a comparison of the performance of iRobot vs AAH in Section 3.6, and iRobot vs ACEBot in Section 5.3.5.

2.5. Structure-Driven Crawling

We have proposed ACEBot [9] (see Chapter 5) that utilizes the inner structure of Web pages to determine the best crawling strategy. Several approaches have been proposed in the literature to implement an efficient and structure-driven system for Web crawling, we review them here.

In [LNL04], Liu et al. proposed an algorithm, called *SEW*, that models a Web site as a hypertext structure. The skeleton is organized in a hierarchical manner, with nodes either navigation pages or content pages. The navigation pages contain links to the content pages, whereas the content pages provide the information content. SEW relies on a combination of several domain-independent heuristics to identify the most important links within a Web page and thus discover a hierarchical organization of navigation and content pages. Kao et al. [KLHC04] have addressed a similar problem, and they propose a technique to distinguish between pages containing *links* to news posts and the pages containing these news. Their proposed method eliminates the redundancy of hypertext structure using entropy-based analysis; similarly a companion algorithm is used to discard redundant information. Compared to our proposed approach in Chapter 5, both techniques above only cluster the Web pages into two predefined classes of pages: navigational and content pages. In addition, Kao et al. [KLHC04] focus on pages of a specific domain (news). In contrast, we have proposed a system that performs unsupervised crawling of Web sites (domain independent). Our proposed system, ACEBot, may have several classes and, indeed, the crawler will follow the best traversal path to crawl the content-rich area. In addition, our approach does not make any prior assumption about the number of classes.

In [CMM03, CMM05, BCM08], a similar version to our model (presented in Chapter 5, see section 5.1) has been adopted. [CMM03, CMM05] aim to cluster Web pages into different classes by exploiting their structural similarity at the DOM tree level, while [BCM08] introduces crawling programs: a tool (implemented as a Firefox plugin) that listen to the user interaction, registers steps, and infers the corresponding intentional navigation. A user is just required to browse the Web site towards a page of interest of a target class. This approach is semi-supervised as it requires human interaction to learn navigation patterns to reach the content-rich pages. A Web crawler is generally intended in a massive crawling scenario, and thus semi-automatic approaches would require a lot of human interaction for each seed Web site, not feasible in our setting. Therefore, in ACEBot, we have introduced the offline phase which learns navigation patterns (that leads to content-rich pages) in an unsupervised manner and hence will have better Web-scale performance.

Another structure-driven approach [VdSdMC06b, VdSdMC06a], has proposed a Web crawler, named *GoGetIt!* [VdSdMC06a] that requires minimum human effort. It takes a sample page (page of interest) and entry point as input and generates a set of navigation patterns (i.e., sequence of patterns) that guides a crawler to reach Web pages structurally similar to the sample page. As stated above, this approach is also focused on a specific type of Web page, whereas our approach performs massive crawling at Web scale for content-rich pages. Similarly, [LMWL06] presents a recipe crawler (domain dependent) that uses certain features to retrieve the recipe pages.

Several domain-dependent *Web forum* crawling techniques [JSYL13, CYL$^+$08, WYL$^+$08, LST13] have been proposed recently. In [JSYL13], the crawler first clusters the Web pages into two groups from a set of manual annotated pages using Support Vector Machines with

some predefined features, and then, within each cluster, URLs are clustered using partial tree alignment. Further a set of ITF (index-thread-page-flipping) *regexes* are generated to launch a bulk download of a target Web forum. The iRobot system [CYL+08] (already discussed above) obtains the structure of the Web forum. The skeleton is obtained in the form of a directed graph consisting of vertices (Web pages) and directed arcs (links between different Web pages). Furthermore, a path analysis is performed to learn an optimal traversal path which leads the extraction process in order to avoid duplicate and invalid page. The technique proposed in [WYL+08] explore an appropriate traversal strategy to direct the crawling of a given Web forum. The traversal strategy consists of the identification of the skeleton links and the detection of the page-flipping links. A lightweight algorithm [LST13] performs content based features to cluster the links for an automatic crawling of Web forums. This approach first identifies the signature and common keywords for the reference links on the forum posts. Then it finds XPath expressions for each content page and content regions within it. An extraction phase is introduced to fetch the actual content. This approach also does not perform on Web scale but is restricted to specific Web forums.

2.6. URL-Based Clustering for Web Crawling

A Web page URL tells a lot about the content of the Web page. The purpose of Web site clustering is to cluster the Web pages so that the pages generated by the same script are in the same cluster. The foundation of the URL-based clustering is to measure the similarity of any two URLs [RFC]. A Web scale approach [BDM11] has introduced an algorithm that performs URL-based clustering of Web pages using some content features. The iRobot system [CYL+08] also performs URL-based sub clustering of Web pages.

However in practice, URL-based clustering of Web pages is less reliable in the presence of dynamic nature of Web. It is even very hard for human being to write down appropriate patterns to cluster the Web pages with same functions. The automated URL pattern generation is very risky, for instance, a Web forum may use multifarious techniques to formulate URLs [BYKS07].

2.7. Web Wrappers

Web wrappers are programs that extract data from HTML pages and transfer them into structured (e.g., XML) format. Several efforts have been concentrated on the development of methods and techniques for the generation of Web wrappers (for a survey on this topic see [CKGS06]). More precisely, we categorize such programs based on semi-automatic and automatic approaches. The wrapper induction techniques extract only the data the user is interested in, and due to manual labeling the matching problem is simpler. The key advantage of automatic approaches is that they are applicable at large-scale and do not require wrapper maintenance.

2.7.1. Wrapper Induction

Wrapper induction has been the subject of several studies. In wrapper induction techniques, some positive pages are selected as examples and then wrappers are trained to extract the

data from Web pages. Manual labeling of data on exemplary Web pages is time consuming, error prone, and labor intensive. Furthermore, the labeling process may need to be repeated for different templates. Early approaches were based on semi-automatic techniques [Fre98, GBE96, Kus00a, KWD97, MMK99, Sod99], and a recent work [ZSWW07] further combines template detection technique with wrapper generation to improve the data extraction accuracy. However, generated wrappers may expire in the future [Kus00b], and thus the wrapper maintenance [MHL03a] problem arose. Wrapper maintenance techniques assumes that there are only minor changes in Web pages, but schema-based changes can cause the templates induced from Web pages to be invalid. Therefore, wrapper induction does not support long-time and continuous data extraction.

2.7.2. Automatic Extraction

Automatic extraction techniques can automatically extract structured data from Web pages. Since automatic approaches do not require wrapper maintenance, it has become more popular in recent years. Therefore, recent efforts have focused on automatic generation of Web wrappers [GM99, CMM01, CMM02, AGM03, WL02, WL03, ZL05].

[GM99] is the first reported work on automated data extraction. In [GM99], they assume that the existence of collections of data rich pages share the similar structure. In RoadRunner [CMM01, CMM02], an algorithm has been proposed to infer the union-free regular expressions to represent Web page templates. But their proposed algorithm has an exponential time complexity and indeed it is impractical for large-scale data extraction systems. An improved version with a polynomial time complexity (achieved by employing several heuristics) has been proposed in [AGM03]. [WL03] has tried to solve the automated data extraction problem by view the pages as long as string, through employing similar generalization mechanisms. A much more advanced technique based on partial tree alignment was proposed in [ZL05]. This technique does not require wildcards generalization to align the corresponding data fields in data records.

2.8. Wrapper Adaptation

Wrapper adaptation, the problem of adapting a Web information extractor to (minor) changes in the structure of considered Web pages or Web sites, has received quite some attention in the research community. An early work is that of Kushmerick [Kus99] who proposed an approach to analyze Web pages and already extracted information, so as to detect changes in structure. A "wrapper verification" method is introduced that checks whether a wrapper stops extracting data; if so, a human supervisor is notified so as to retrain the wrapper. Chidlovskii [Chi01] introduced some grammatical and logic-based rules to automate the maintenance of wrappers, assuming only slight changes in the structure of Web pages. Meng, Hu, and Li [MHL03b] suggested a schema-guided wrapper maintenance approach called SG-WRAM for wrapper adaptation, based on the observation that even when structure changes in Web pages, some features are preserved, such as syntactic patterns, hyperlinks, annotations, and the extracted information itself. Lerman, Minton, and Knoblock [LMK03] developed a machine learning system for repairing wrapper for small markup changes. Their proposed system first verifies the extraction from Web pages, and if the extraction fails then it relaunches the wrapper induction for data extraction. Raposo et al. [RPAH07] collect valid query results during the use of the wrapper; when

structural changes occur in Web sources then use those results as inputs to generate a new training set of labeled examples which can be used again for wrapper induction.

Our template adaptation technique (see Section 3.5; introduced for application-aware helper) is inspired by the previously cited works: we check whether patterns of our wrapper fail, and if so, we try fixing them assuming minor changes in Web pages, and possibly using previously crawled content on this site. One main difference with existing work is that our approach is also applicable to completely new Web sites, never crawled before, that just share the same content management system and a similar template.

2.9. Data Extraction from Blogs, Forums, etc

A number of works [XYG+11, FB10, LN01, AF08] aim at automatic wrapper extraction from CMS-generated Web pages, looking for repeated structure and typically using tree alignment or tree matching techniques. This is out of scope of application-aware helper, where we assume that we have a preexisting knowledge base of Web applications. Gulhane et al. [GMM+11] introduced the Vertex wrapper induction system. Vertex extracts structured information from template-based Web pages. The system cluster the pages using shingle vectors, learns extraction rules, detects site changes and relearn broken rules. Vertex detects site changes by monitoring a few sample pages per site. Any structural change can result in changes in page shingle vectors that may render learned rules inapplicable. In our AAH system, we do not monitor sampled Web pages but dynamically adapt to pages as we crawl them. Our proposed crawler also applies adaptation to different versions of a content management system found on different Web sites, rather than to just a specific Web page type.

2.10. Web Application Detection

As we shall explain, our approach relies on a generic mechanism for detecting the kind of Web application currently crawled. Again there has been some work in the particular cases of blogs or forums. In particular, [KFJ06] uses support vector machines (SVM) to detect whether a given page is a blog page. Support vector machines [BGV92, OFG97] are widely used for text classification problems. In [KFJ06], SVMs are trained using various traditional feature vectors formed of the content's bag of words or bag of n-grams, and some new features for blog detection are introduced such as the bag of linked URLs and the bag of anchors. Relative entropy is used for feature selection.

More generally, a few works [ACD+03, LL07, LL06] aim at identifying a general category (e.g., blog, academic, personal, etc.) for a given Web site, using classifiers based on structural features of Web pages that attempt to detect the *functionalities* of these Web sites. This is not directly applicable to our setting, first because the classification is very coarse, and second because these techniques function at the Web site level (e.g., based on the home page) and not at the level of individual Web pages.

2.11. Deep-Web Crawling

Focused crawlers and crawlers in general can harvest data from the publicly indexable Web by following hyperlinks between Web pages. However, there is a very large part of the Web

that is hidden behind HTML forms [Ber00]. Such forms are easy to complete by human users. Content of Deep Web (a.k.a, hidden Web or invisible Web) crawling [LdSGL04] is usually stored in databases and deep Web crawler focuses on how to prepare appropriate queries to retrieve invisible Web pages. Automatic deep-Web crawlers, however, need to complete HTML forms and retrieve results from the underlying databases. Barbosa and Freire [BF04, BF05, BF07] develop mechanisms for generating simple keyword queries that cover the underlying database through unstructured simple search forms. Madhavan et al. [MKK+08] handle structured forms by automatically completing subsets of fields, aiming to obtain small coverage over many hidden Web databases. Deep Web crawling can be useful for a variety of tasks including data integration [MJC+07], and Web indexing [MKK+08].

2.12. Web Extraction Languages

Extraction languages [Mih96, AM98, LL01, SA99, SMSK07, SDNR07] use a declarative approach, however, they do not adequately facilitate deep Web interaction such as Web form submissions and neither provide native constructs for page navigation, apart from retrieving a page given a URI.

XLog [SDNR07] uses Datalog as a programming language for extracting data from Web pages. This approach introduces the Xlog language as an application of Datalog, which includes predefined extraction predicates. This technique opens the window for the researcher to use Datalog as a base for the extraction process, but still requires some amount of effort to deal with the deep Web. [SMSK07] faces the same challenge. This approach also extracts the content from simple pages or from bibliographic pages. It does not simulate any action for form filling or page navigation. They introduced a wrapping language named as Wraplet, that extracts structured data from Web pages (written in HTML). This language is based on wraplet expressions written as a script (a data extraction expression), takes an HTML document as an input, and produces output in the form of XML. But unfortunately this approach can be applied only to singleton pages, which do not hold the dynamic nature of the Web.

The BODED extraction language [SSWC10] is able to deal with modern Web sites and facilitates functionalities such as form submission and Web page navigation. The paper [SSWC10] is written in a very simple tone, provides several examples to clarify the concepts. The authors have introduced a system named browser-oriented data extraction system (BODE) that extracts the information from Web pages and uses URL links to navigate to the next page for information extraction. BODE harvests Web applications even when they which are using scripting functions such as JavaScript or AJAX for accessing the next page. It also simulates user actions to deal with the deep Web. In despite of all these advantages, BODE does not consider memory management constraints, which makes it unsuitable for our system where we want to continuously crawl and archive large volumes of data: browser instances are replicated for multiple page.

OXPATH [FGG+11] aims to be the first declarative Web extraction formalism to fully integrate features such as interactivity that are necessary for extraction from modern Web applications. OXPath is an extension of XPath, with added facilities for interacting with Web applications and extracting relevant data. It allows the simulation of user actions to interact with scripted multipage interfaces of the Web application (the evaluator relies either on a Mozilla-based or Webkit-based browser). It inherits from XPath as well as

allows using CSS-based selectors. It makes possible to navigate through different pages by using clicks and even allows to extracting information from previous pages. An open-source implementation is available.

Let us take the example of the vBulletin Web forum content management system, where pages at the level of "list of forums" are identified* when they match the

```
//a[@class="forum"]/@href
```

XPath expressions. A simple OXPath action that can be performed on the vBulletin example is

```
//a.forum/@href/{click/}
```

that "clicks" on every link to an individual Web forum. We refer to [FGG+11] for more elaborate examples of OXPath expressions.

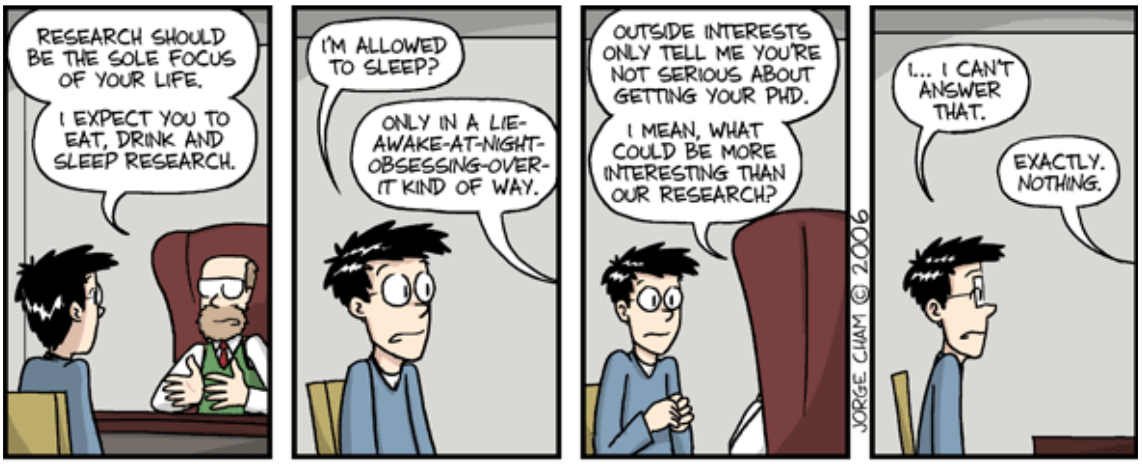
We have developed the OWET tool [8] (see Chapter 6) based on this language for complicated interaction with Web server to get data hidden behind a Web form or an AJAX application; and for learning the navigation patterns for Web application crawling.

*The example is simplified for the sake of presentation; in reality we have to deal with several different layouts that vBulletin can produce.

Chapter 3.

Intelligent and Adaptive Crawling of Web Applications for Web Archiving

Initial ideas leading to this work were presented as a PhD workshop article in [5]. This chapter presents the detailed description of the work published in [4, 2, 7]. The AAH system has been demonstrated in [3, 6].



The steady growth of the World Wide Web raises challenges regarding the preservation of meaningful Web data. Tools used currently by Web archivists blindly crawl and store Web pages found while crawling, disregarding the kind of Web site currently accessed (which leads to suboptimal crawling strategies) and whatever structured content is contained in Web pages (which results in page-level archives whose content is hard to exploit). A generic Web crawler [BP98] performs inefficient crawling of the Web sites. It crawls the Web with no guarantee of content quality. A naive *breadth-first* [BYCMR05] (shallow) strategy cannot ensure access to all content-rich pages, while a simple *depth-first* (deep) crawl may fetch too many redundant and invalid pages (e.g., may login failure pages), in addition to the possibility of falling into *robot traps*. Thus, a generic methodology crawls a set of Web pages that may lack real interest for a given user or application and may also ignore the content relationships among Web pages [BP98]. An ideal crawling approach should achieve a tradeoff between *performance* and *cost* to retrieve high-quality content with minimum bandwidth and storage cost. Web crawlers should solve the two following problems: first what kind of Web pages are *important* to crawl (to avoid redundant and

invalid pages); and second which *important* links should be followed and what navigation patterns are required on the Web site? An intelligent crawler should be aware of what kind of pages should be crawled and how to skip the invalid, redundant pages to save network bandwidth. We focus in this PhD work on the crawling and archiving of publicly accessible Web applications, especially those of the social Web. We add adaptive characteristics to a generic Web crawler: being able to identify when a Web page belongs to a given Web application and applying the appropriate crawling and content extraction methodology.

Application-Aware Archiving Current archival crawlers, such as Internet Archive's Heritrix [Sig05], work in a conceptually simple manner. They start from a *seed* list of the URLs to be stored in a queue (e.g., the starting URL may be the homepage of a Web site). Web pages are then fetched from this queue one after the other (respecting crawling ethics, limiting the number of requests per server), stored as is, and links are extracted from them. If these links are in the scope of the archiving task, the newly extracted URLs are added to the queue. This process ends after a specified time or when no new interesting URL can be found.

This approach does not confront the challenges of modern Web application crawling: the nature of the Web application crawled is not taken into account to decide the crawling strategy or the content to be stored; Web applications with dynamic content (e.g., Web forums, blogs, etc.) may be crawled inefficiently, in terms of the number of HTTP requests required to archive a given site; content stored in the archive may be redundant, and typically does not have any structure (it consists of flat HTML files), which makes access to the archive cumbersome.

The aim of this work is to address this challenge by introducing a new *application-aware* approach to archival Web crawling. Our system, the *application-aware helper* (AAH for short) relies on a knowledge base of known Web applications. A *Web application* is any HTTP-based application that utilizes the Web and Web browser technologies to publish information using a specific template. We focus in particular on social aspects of the Web, which are heavily based on user-generated content, social interaction, and networking, as can be found for instance in Web forums, blogs, or on social networking sites. Our proposed AAH only harvests the important content of a Web application (i.e., the content that will be valuable in a Web archive) and avoids duplicates, uninteresting URLs and templates that just serve a presentational purpose. In addition the application-aware helper extracts from Web pages individual items of information (such as blog post content, author, timestamp), etc., that can be stored using semantic Web technologies for richer and semantic access to the Web archive.

To illustrate, consider the example of a Web forum, say, powered by a content management system such as vBulletin. On the server side, forum threads and posts are stored in a database; when a user requests a given Web page, the response page is automatically generated from this database content, using a predefined template. Frequently, access to two different URLs will end up presenting the same or overlapping content. For instance, a given user's posts can be accessed both through the classical threaded view of forum posts or through the list of all his or her post displayed on the user profile. This redundancy means that an archive built by a classical Web crawler will contain duplicated information, and that many requests to the server do not result in novel pieces of content. In extreme cases, the crawler can fall into a *spider trap* because it has infinitely many links to crawl.

There are also several noisy links such as to a print-friendly page or advertisement, etc., which would be better to avoid during the constitution of the archive. On the contrary, a Web crawler that is aware of the information to be crawled can determine an optimal path to crawl all posts of a forum, without any useless requests, and can store individual posts, together with their authors and timestamps, in a structured form that archivists and archive users can benefit of.

Template Change Web applications are dynamic in nature; not only their content changes over time, but their structure and template does as well. Content management systems provide several templates that one can use for generating wiki articles, blog posts, forum messages, etc. These systems usually provide a way for changing the template (which ultimately changes the structure of Web pages) without altering the informational content, to adapt to the requirements of a specific site. The layout may also change as a new version of the CMS is installed.

All these layout changes result in possible changes in the DOM tree of the Web page, usually minor. This makes it more challenging to recognize and process in an intelligent manner all instances of a given content management systems, as it is hopeless to hope to manually describe all possible variations of the template in a Web application knowledge base. Another goal of the work is to produce an intelligent crawling approach that is resilient to minor template changes, and, especially, automatically adapts its behavior to these changes, updating its knowledge of CMSs in the process. Our adaptation technique relies on both relaxing the crawling and extraction patterns present in the knowledge base, and on comparing successive versions of the same Web page.

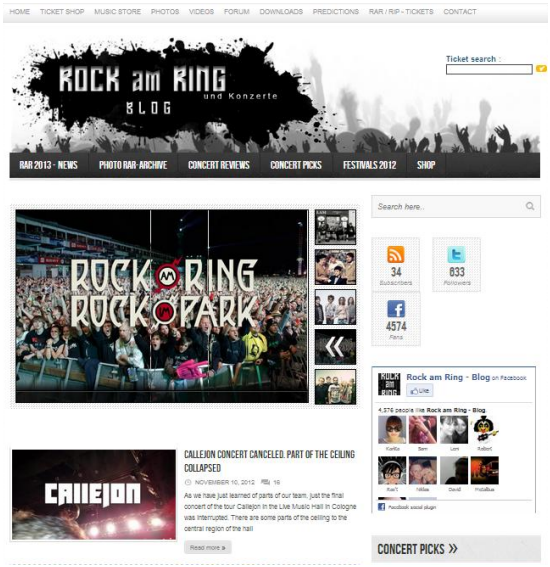
Outline We next present some preliminary definitions in Section 3.1, we describe our knowledge base of Web applications in Section 3.2. The methodology that our application-aware helper implements is then presented in Section 3.4. We discuss the specific problem of adaptation to template changes and related algorithms in Section 3.5 before covering implementation issues and explaining how the AAH fits into a crawl processing chain in Section 3.3. We finally compare the efficiency and effectiveness of our AAH with respect to classical crawling approach in crawling blogs and Web forums in Section 3.6.

3.1. Preliminaries

This section introduce some terminology that we will use throughout this chapter.

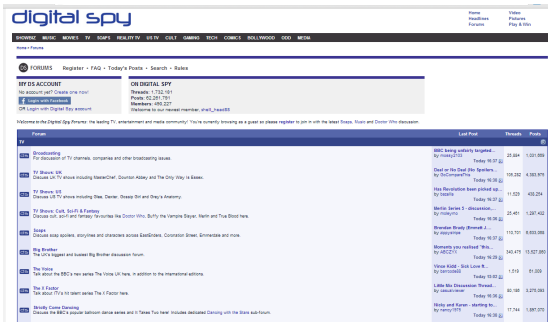
A *Web application* is any application or Web site that uses Web standards such as HTML and HTTP to publish information on the Web in a specific template, in a way that is accessible by Web browsers. Examples include Web forums, social networking sites, geolocation services, etc.

A *Web application type* is the content-management system or server-side technology stack (e.g., vBulletin, WordPress, the proprietary CMS of Flickr, etc.) that powers this Web application and provides interaction with it. Several different Web applications can share the same Web application type (all vBulletin forums use vBulletin), but some Web application types can be specific to a given Web application (e.g., the CMS powering Twitter is specific to that site). There can be significant differences in the appearance

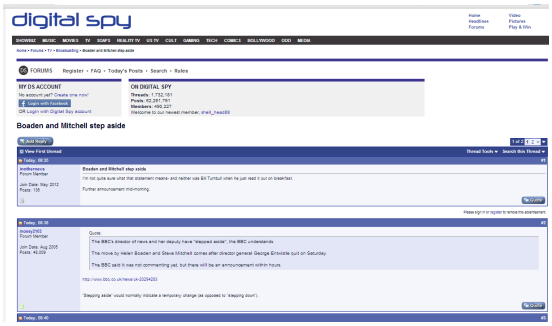


(a) Rock am Ring blog

(b) Carleton Newsroom



(c) Intermediate level



(d) Terminal level

Figure 3.1.: Example Web pages of some Web applications

of Web applications of the same type; compare for instance Figures 3.1a and 3.1b, two WordPress-powered sites.

The content presented by Web applications is typically stored in databases and predefined templates are used to generate data-rich Web pages. For intelligent crawling, our AAH needs not only to distinguish among Web application types, but among the different kinds of Web pages that can be produced by a given Web application type. For instance, a Web forum software can generate content-pages of various kinds, such as *list of forums*, *list of threads*, *list of posts*, *individual post*, *user profile*. We call such a specific template kind the *level* of the Web application.

We use a simple subset of the XPath expression language [W3C99] to describe *patterns* in the DOM of Web pages that serve either to identify a Web application type or level, or to determine navigation or extraction *actions* to apply to that Web page. A grammar for the subset we consider is given in Figure 3.2. Basically, we only allow downwards axes and very simple predicates that perform string comparisons. The semantics of these expressions is the standard one [W3C99]. Unless otherwise specified, a leading "//" is implicit: an expression can match any node at any depth in the DOM tree. In the following, an *XPath*

```
⟨expr⟩       ::= ⟨step⟩ | ⟨step⟩ "/" ⟨expr⟩
                 ⟨step⟩ "//" ⟨expr⟩
⟨step⟩       ::= ⟨nodetest⟩ | ⟨step⟩ "[" ⟨predicate⟩ "]"
⟨nodetest⟩   ::= tag | "@" tag | "*" | "@*" | "text()"
⟨predicate⟩  ::= "contains(" ⟨value⟩ "," string ")" |
                 ⟨value⟩ "=" string | integer | "last()"
⟨value⟩      ::= tag | "@" tag
```

Figure 3.2.: BNF syntax of the XPath fragment used. The following tokens are used: *tag* is a valid XML identifier (or NMTOKEN [W3C08]); *string* is a single- or double-quote encoded XPath character string [W3C99]; *integer* is any positive integer.

expression is always one of this sublanguage.

A *detection pattern* is a rule for detecting Web application types and Web application levels, based on the content of a Web page, HTTP metadata, URL components. It is implemented as an XPath expression over a virtual document that contains the HTML Web page as well as all other HTTP metadata.

A *crawling action* is an XPath expression over an HTML document that indicates which action to perform on a given Web page. Crawling actions can be of two kinds: *navigation actions* point to URLs to be added to the crawling queue; *extraction actions* point to individual semantic objects to be extracted from the Web page (e.g., timestamp, blog post, comment). For instance, `div[contains(@class,'post')]//h2[@class='post-message']//a/@href` is a navigation action to follow certain types of links.

The application-aware helper distinguishes two main kinds of Web application levels: *intermediate* pages, such as lists of forums, lists of threads, can only be associated with navigation actions; *terminal pages*, such as the list of posts in a forum thread, can be associated with both navigation and extraction actions. The idea is that the crawler will navigate intermediate pages until a terminal page is found, and only content from this terminal page is extracted; the terminal page may also be navigated, e.g., in the presence of paging. To illustrate, Figure 3.1c and 3.1d show, respectively, intermediate and terminal pages in a Web forum application.

Given an XPath expression e, a *relaxed expression* for e is one where one or several of the following transformations has been performed:

- a predicate has been removed;
- a *tag* token has been replaced with another *tag* token.

A *best-case relaxed expression* for e is one where at most one of these transformations has been performed for every step of e. A *worst-case relaxed expression* for e is one where potentially multiple transformations have been performed on any given step of e.

To illustrate, consider $e =$ `div[contains(@class,'post')]//h2[@class='post-message']`. Examples of best-case relaxed expressions are `div[contains(@class,'post')]//h2` or `div[contains(@class,'post')]//h2[@id='post-content']`; on the other hand, `div[contains(@class,'post')]//div[@id='post-content']` is an example worst-case relaxed expression.

3.2. Knowledge Base

The AAH is assisted by a *knowledge base* of Web application types that describes how to crawl a Web site in an intelligent manner. This knowledge specifies how to detect specific Web applications and which crawling actions should be executed. The types are arranged in a hierarchical manner, from general categorizations to specific instances (Web sites) of this Web application. For example the social media Web sites can be categorized into blogs, Web forums, microblogs, video networks, etc. Then we can further categorize these specific types of Web applications on the basis of the content management system they are based on. For instance, WordPress, Movable Type, etc., are examples of blog content management system, whereas phpBB and vBulletin, etc., are examples of Web forum content management systems. The knowledge base also describes the different levels under a Web application type and then, based on this, we can define different crawling actions that should be executed against this specific page level.

The knowledge base is specified in a declarative language, so as to be easily shared and updated, hopefully maintained by non-programmers, and also possibly automatically learned from examples. The W3C has normalized a Web Application Description Language (WADL) [W3C09] that allows describing resources of HTTP-based application in a machine processable format. WADL is used for describing the set of resources, their relationship with each other, the method that can be applied on each resource, resource representation formats, etc. WADL may be a candidate component and export format of our knowledge base, but does not satisfy all our needs: in particular, there is no place for the description of Web application recognition patterns. Consequently, our knowledge base is described in custom XML format, well-adapted to the tree structure of the hierarchy of Web applications and page levels.

Web application type and level For each Web application type, and for each Web application level, the knowledge base contains a set of detection patterns that allows to recognize whether a given page is of that type or that level.

Let us take the example of the vBulletin Web forum content management system, that can for instance be identified by searching for a reference to a `vbulletin_global.js` JavaScript script with the detection pattern: `script[contains(@src,'vbulletin_global.js')]`. Pages of the "list of forums" type are identified* when they match the pattern `a[@class="forum"]/@href`.

Crawling actions After detecting the application to whom the current Web page belongs, the next stage is to determine the corresponding crawling actions. Crawling action scopes go beyond just a list of URLs to add to the queue. It can be any action that involves using APIs to extract relevant data from the social network sites, such as Twitter, or performing complicated interactions with AJAX-based applications, or identifying Web objects in particular Web application. More specifically, crawling actions are of two kinds:

Navigation actions: to navigate to another Web page or Web resources (typically in an attribute `a/@href`).

*The example is simplified for the sake of presentation; in reality we have to deal with several different layouts that vBulletin can produce.

Extraction actions: to extract individual semantic objects from Web pages (e.g., timestamp, the blog post, the comments).

For example, the following lines describes how to detect the vBulletin type of Web application and which crawling actions must be executed to crawl such Web sites.

```
<knowledgebase>
  <cms  name="vBulletin" type="webforum">
   <detection-rules>
     <xpath-expression>
      //script/@src[contains(.,'vbulletin_global.js')]
     </xpath-expression>
   </detection-rules>
   <page-level-cat>
     <list-of-forum>
        <detection-rules>
           <xpath-expression type="1">
            //a[@class="forum"]/@href
           </xpath-expression>
           <xpath-expression type="2">
             //h2[@class="forumtitle"]/a/@href
           </xpath-expression>
        </detection-rules>
        <crawling-action>
             <action id="1">
                //a.forum/@href
              </action>
              <action id="2">
                //td.forumtitle/div/a/@href
              </action>
           </crawling-action>
       </list-of-forum>
       <list-of-thread>
             .
       </list-of-thread>
       <thread>
             .
       </thread>
    </page-level-cat>
  </cms>
</knowledgebase>
```

3.3. System

We now present the main components of the AAH for intelligent and adaptive crawling of known and adaptable Web applications. We refer to Figure 3.3 as an illustration of

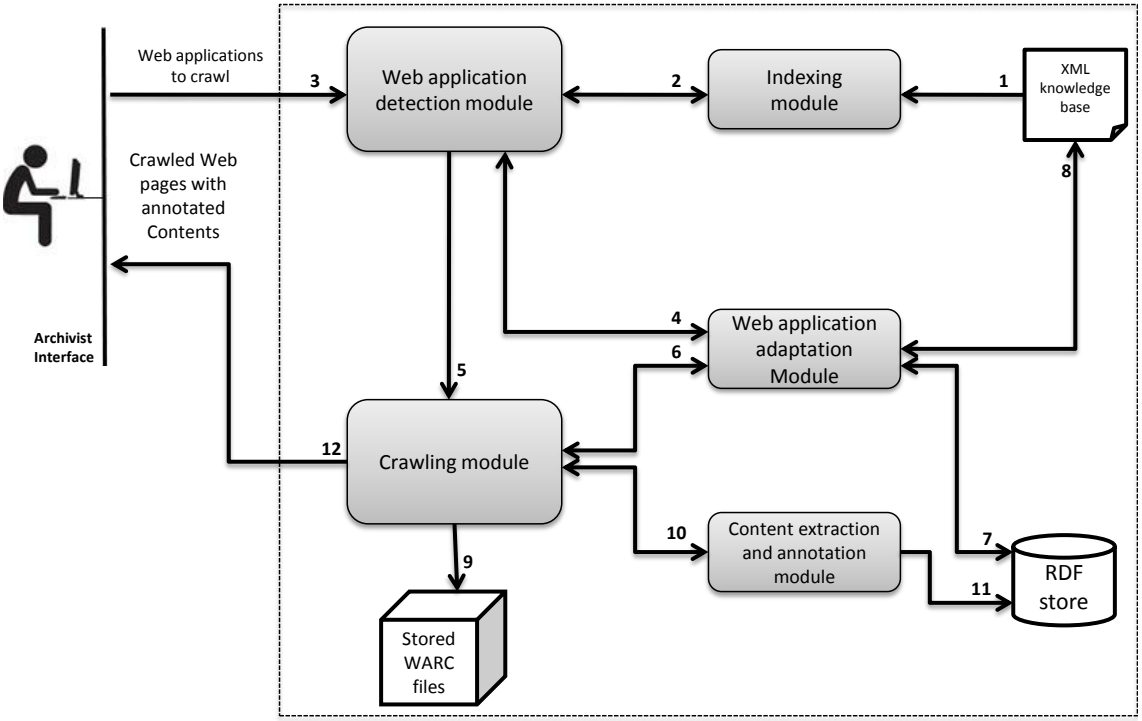


Figure 3.3.: Architecture of the AAH

the architecture of the system, with items of the following enumeration referring to the numbers in the figure.

(1) The AAH relies on a knowledge base of Web application types which describes how to crawl a Web site in an intelligent manner. The knowledge specifies precisely how to detect a specific Web application type (CMS) and which crawling actions should be executed to crawl it. The knowledge base also describes the different levels under a Web application type and then, based on this, different crawling actions that should be executed against this specific page level. The knowledge base is described in a custom XML format, well-adapted to the tree structure of a hierarchy of Web applications and page levels. The Web application detection patterns and crawling actions are written in an XPath fragment language.

(2) The system loads the Web application type detection patterns from the knowledge base and executes them against a given Web application. If the Web application type is detected, the system runs all possible Web application level detection patterns until a match is found. The number of detection patterns for detecting Web application type and level will grow with the addition of knowledge about new Web applications. To optimize this detection, we maintain an index of these patterns, that uses a version of the YFilter [DAF+03] NFA-based filtering system for XPath expressions adapted to our purposes.

(3) Once the system receives a crawling request, it first makes a lookup on the YFilter index to detect the Web application type and level.

(4) If the Web application type is not detected, the AAH applies an adaptation strategy to find a relaxed match: we look for approximate matches of detection patterns,

thereby handling different versions of a CMS.

(5) When the Web application is successfully detected, the AAH loads the corresponding crawling strategy from the knowledge base and crawls the Web application accordingly. This module also implements the basic queue management and resource fetching components.

(6) If the system fails to crawl the Web application because of structural changes with respect to the knowledge base, it tries determining the changes and then adapt failed crawling actions. The AAH deals with two different cases of adaptation: first, when (part of) a Web application has been crawled before the template change and a recrawl is carried out after that (a common situation in real-world crawl campaigns); second, when crawling a new Web application that matches the Web application type detection patterns but for which (some of) the actions are inapplicable.

(7) A Web application that has been crawled before but cannot be recrawled with the same crawling actions is relearned for adaptation by searching for already crawled content (URLs corresponding to navigation actions, Web objects, etc., stored in an RDF store).

(8) In the process of adaptation, the system also automatically maintains the knowledge base with the newly discovered patterns and actions.

(9) The crawled Web pages are stored in the form of WARC [ISO] files, the standard preservation format for Web archiving. For scalability and archive availability purposes, WARC files are stored in a Hadoop* cluster on HDFS, and the RDF store we use, H2RDF [PKTK12], is implemented on top of HBase [The12]

(10) Structured content (individual Web objects with their semantic metadata) is extracted from each crawled page, as described in the knowledge base.

(11) Structure content is stored in an RDF store, that can be queried by an archive user through SPARQL queries.

(12) Crawled Web pages with annotated content are displayed to the archivist through a GUI that we describe in more detail in following demonstration scenario.

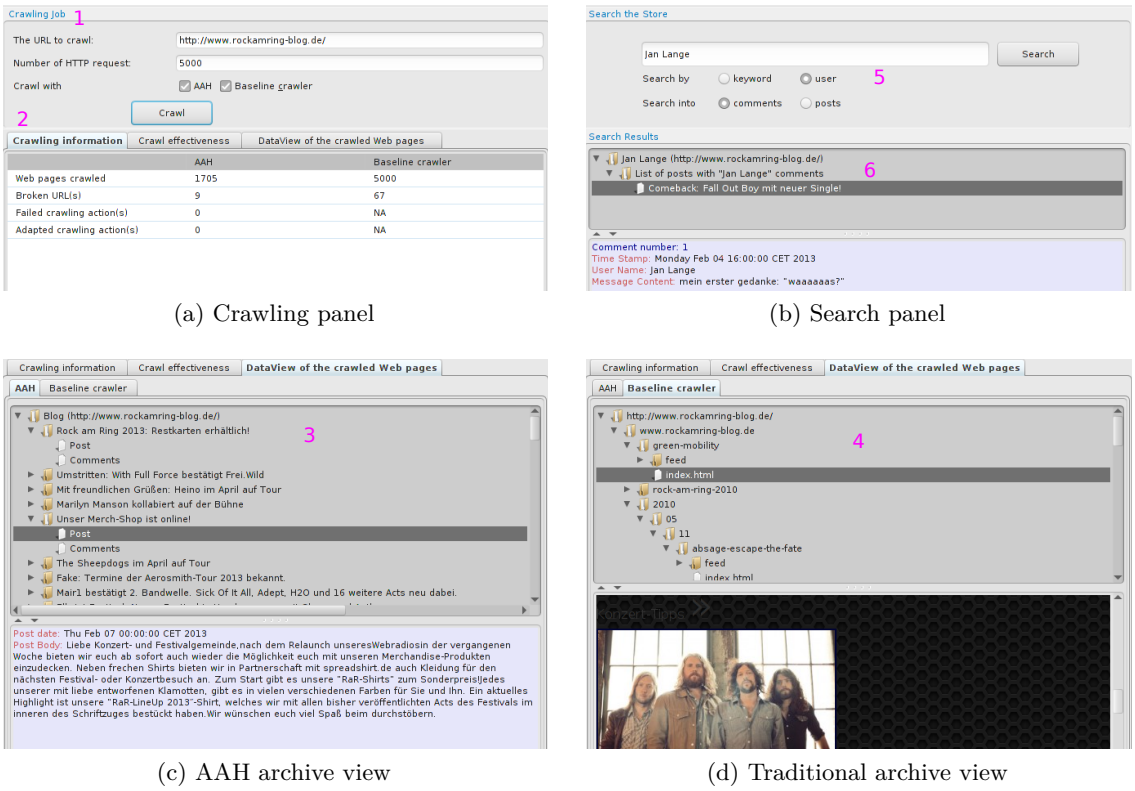
Demonstration Scenario We now describe a specific use case where the AAH helps building richer archives with less resources wasted. This is a real-world case where the AAH has been used.

The German broadcasting company SWR is a partner each year of the *Rock am Ring* music festival. SWR archivists are interested in building archives of both official information and public perception of this music festival as displayed on the Web. Many Web sites are in the scope of this archival campaign, however, crawling, storage, analysis resources are limited, and timely archival matters, which requires to limit crawling requests to those that will effectively bring meaningful content to the archive. The AAH has been designed to meet these challenges and ensure that duplicates and templates are avoided, and all useful information has crawled.

We now present in detail our demonstration scenario (See Figure 3.4).

(1) The user enters in the *crawling* panel the URL of a Web site to crawl with the possibility to limit the number of HTTP requests. The user will be offered to choose any URL she wants; obviously, interesting results will be produced only for Web application types supported by the AAH (currently, various versions of WordPress, phpBB, and

*`http://hadoop.apache.org/`



(a) Crawling panel

(b) Search panel

(c) AAH archive view

(d) Traditional archive view

Figure 3.4.: Graphical user interface of the AAH

vBulletin, with a robustness to template change thanks to the adaptation module). A few Web sites will be pre-crawled locally to simulate their crawl without having to rely on network delays (especially an issue since the crawler respects per-server crawling delays as per crawling ethics). The user is also given the choice of the specific crawler to run the job: either the AAH crawler, or a baseline non-intelligent crawler, or both. Selecting both allows comparing their performance.

(2) Once the crawl is completed (a few seconds when run from a local cache of the Web site, arbitrarily longer for a remote crawl), the system shows the number of HTTP requests made by both crawlers, as well as the number of broken links found. For instance, on one specific crawl of the blog `http://www.rockamring-blog.de/`, the AAH has made only 1,705 HTTP requests to crawl the whole given Web application, compared to 5,000 for the baseline crawler with no certainty that all useful content has been retrieved. Comparatively, the AAH also encounters fewer broken links than the baseline crawler. We additionally provide the number of failed and adapted crawling actions. The *crawl effectiveness* panel compares crawl performance on a plot of Web site coverage (measured as a number of distinct k-grams [4]) vs number of requests.

(3) The system provides the data view of the crawled Web pages with the AAH. The AAH crawls the Web pages in an intelligent manner and extracts useful information. For instance, depending on the Web application, we may show a list of blog posts with crawled content and Web objects, identifying semantic components such as post body, publication date, author, and post comments with their publication date and author. Similar views

exist for Web forum content.

(4) The data view of the crawled Web pages with the baseline crawler is rather very simple (direct HTML display), as Web pages are crawled blindly, without avoiding spider traps and noisy links.

(5) The AAH does not only crawl the Web application in an intelligent manner but also extracts Web objects (e.g., timestamp, comments, author). Crawled Web pages and objects are stored in a large-scale RDF store [PKTK12] in the form of RDF triples. The user can run semantic queries on the triple store. For instance, she can look for the posts or comments posted by specific users by specifying their names. The interface also allows searching post bodies or comments with respect to a given keyword.

(6) The result of the query are shown in the form of a data view, where, for instance, the list of posts or comments with the given user name are shown.

The Java implementation of the AAH is available in open source from `http://perso.telecom-paristech.fr/~faheem/aah.html`. In addition to being usable in a standalone mode for testing and demonstration purposes, the AAH has also been integrated, in the framework of the ARCOMEM project*, in the crawl processing chain of both Internet Archive's Heritrix crawler [Sig05] (modified for our purposes) and Internet Memory's† proprietary crawler.

3.4. Application-aware Helper (AAH)

Our main claim is that different crawling techniques should be applied to different types of Web applications. This means having different crawling strategies for different forms of social Web sites (blogs, wikis, social networks, social bookmarks, microblogs, music networks, Web forums, photo networks, video networks, etc.), for specific content management systems (e.g., WordPress, phpBB), and for specific sites (e.g., Twitter, Facebook). Figures 3.1a and 3.1b are example Web applications that are based on the WordPress CMS. Our proposed approach will detect the type of Web application (general type, content management system, or site) currently processed by the crawler, and the kind of Web pages inside this Web application (e.g., a user profile on a social network) and decide on further crawling actions (following a link, using an API, submitting a form, extracting structured content) accordingly.

To adapt the behavior of traditional crawlers according to our requirements, we have chosen to extend the traditional architecture of a Web crawler in the way depicted in Figure 3.5. Here the page fetching module (see Figure 1.5) is replaced by some more elaborate resource fetching component that is able to retrieve resources that are not just accessible by a simple HTTP GET request (but by a succession of such requests, or by a POST request, or by the use of an API), or that are individual Web objects inside a Web page (e.g., a blog post, a comment, a poster's name). An application-aware helper module is then introduced in place of the usual link extraction function, in order to identify the Web application that is currently being crawled, and decide and categorize crawling actions that can be performed on this particular Web application.

These modifications are implemented in two Web crawlers: the proprietary crawler of the Internet Memory Foundation, with whom we are closely collaborating, and into a

*`http://www.arcomem.eu/`

†`http://internetmemory.org/`

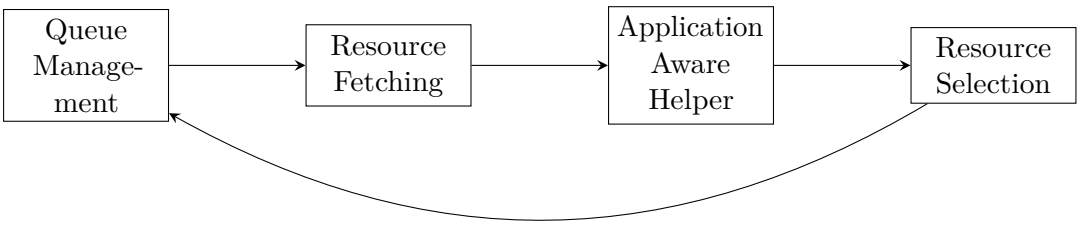


Figure 3.5.: Extended architecture of the Web crawler

customized version of Heritrix [Sig05], developed by the ATHENA research lab in the framework of the ARCOMEM project [ARC13].

The AAH detects the Web application and Web page type before deciding which crawling strategy is appropriate for the given Web application. More precisely, AAH works in the following order:

(1) it detects the Web application type;
(2) it detects the Web application level;
(3) it executes the relevant crawling actions: extract the outcome of extraction actions, and add the outcome of navigation actions to the URL queue.

Detection of Web application type and level The AAH loads the Web application type detection patterns from the knowledge base and executes them against the given Web application. If the Web application type is detected, the system executes all the possible Web application level detection patterns until it gets a match. The number of detection patterns for detecting Web application type and level will grow with the addition of knowledge about the new Web applications. To optimize this detection, the system needs to maintain an index of these patterns.

To this purpose, we have integrated the YFilter system [DAF+03] (an NFA-based filtering system for XPath expressions) with slight changes according to our requirements, for efficient indexing of detection patterns, in order to quickly find the relevant Web application types and levels. YFilter is developed as part of a publish–subscribe system that allows users to submit a set of queries that are to be executed against streaming XML pages. By compiling the queries into an automaton to index all provided patterns, the system is able to efficiently find the list of all users who submitted a query that matches the current document. In our integrated version of YFilter, the detection patterns (either for Web application type or level) will be submitted as queries; when a document satisfy a query, the system will stop processing the document against all remaining queries (in contrast to the standard behavior of YFilter), as we do not need more than one match.

3.5. Adaptation to Template Change

Two types of changes can occur in a Web page: Web content changes, and Web structure changes. Change in the rating of a movie, a new comment under a post, the deletion of a comment, etc., are examples of Web content changes. These kind of changes are easy to identify by simply comparing the current Web page with a recently crawled version. An example of Web structure change is a change to the presentational template of a Web

site template change, which is more complicated to identify and adapt. Our adaptation approach deals with this challenge.

Structural changes with respect to the knowledge base may come from varying versions of the content management system, or from alternative templates proposed by the CMS or developed for specific Web application, If the template of a Web page is not the one given in the Web application knowledge base, Web application detection patterns and crawling actions may fail on a Web page. The AAH aims at determining when a change has occurred and adapting patterns and actions. In the process, it also automatically maintains the knowledge base with the newly discovered patterns and actions.

Most of the time, detection patterns for determining the Web application type looks for a reference to script files, stylesheets, HTTP metadata (such as cookie names) or even textual content (e.g., a "Powered by" text fragment), which are quite robust to template changes. Accordingly, we will assume in the following that our Web application type detection patterns never fail. We did not see any instance where this would happen in our experiments. On the other hand, we consider that Web application level detection patterns and crawling actions may become inapplicable after structural changes.

We deal with two different cases of adaptation: first, when (part of) a Web application has been crawled before the template change and a recrawl is carried out after that (a common situation in real-world crawl campaigns); second, when crawling a new Web application that matches the Web application type detection patterns but for which (some of) the actions are inapplicable.

3.5.1. Recrawl of a Web Application

We first consider the case when part of a Web application has been crawled successfully using the patterns and actions of the knowledge base. The template of this Web application then changes (because of an update of the content management system, or a redesign of the site) and it is recrawled. Our core adaptation technique relearns appropriate crawling actions for each crawlable object; the knowledge base is then updated by adding newly relearned actions to it (since the existing actions may work on an other instance of this Web application type, we do not replace existing ones).

As later described in Section 3.3, crawled Web pages with their Web objects and metadata are stored in the form of RDF triples into an RDF store. Our proposed system detects structural changes for already crawled Web applications by looking for the content (stored in RDF store) in the Web pages with the crawling actions used during the previous crawl. If the system fails to extract the content with the exact same actions, then it means the structure of the Web site has changed.

Input: a URL u, sets of Web application detection patterns D and crawling actions A
if *alreadyCrawled*(u) **then**
  if *hasChanged*(u) **then**
    markedActions $\leftarrow$ *detectAndMarkStructuralChanges*(u, A);
    newActions $\leftarrow$ *alignCrawlingActions*$(u, D,$ *markedActions*$)$;
    addToKnowledgeBase(*newActions*);

Algorithm 3.1: Adaptation to template change (recrawl of a Web application)

Algorithm 3.1 gives a high-level view of the template adaptation mechanism in the case

of a recrawl. The system processes each relevant Web page of a given Web application. Any Web page that has been crawled before but now fails to crawl will be passed to Algorithm 3.1 for structural adaptation. This algorithm only fixes failed crawling actions for already crawled Web pages; however, since newly learned crawling actions are added to the knowledge base, the system will still be able to process pages from the same Web application that were not crawled before. Note that in any case the system does not blindly try to relearn the failed crawling actions for each Web page, but rather first checks whether already fixed crawling actions for the same Web application level still works.

Algorithm 3.1 first checks whether a given URL has already been crawled by calling the *alreadyCrawled* Boolean function, which just looks for the existence of the URL in the RDF store. An already crawled Web page will then be checked for structured changes with the *hasChanged* Boolean function.

Structural changes are detected by searching for already crawled content (URLs corresponding to navigation actions, Web objects, etc.) in a Web page by using the existing and already learned crawling actions (if any) for the corresponding Web application level. The *hasChanged* function takes care of the fact that failure to extract deleted information should not be consider as a structural change. For instance, a Web object such as a Web forum's *comment* that was crawled before may not exist anymore. This scenario is very likely to occur as an administrator or even a comment's author himself may remove it from the Web forum. For that reason, the system will always check the existence of a Web object with all of its metadata in the current version of a Web page; if the Web object has disappeared, it is not considered for detecting structural changes.

In the presence of structural changes, the system calls the *detectAndMarkStructuralChanges* function which detects inapplicable crawling actions and mark them as "failed". The *detectAndMarkStructuralChanges* returns a set of marked crawling actions. All crawling actions which are marked as failed will be aligned according to structural changes. The *alignCrawlingActions* function will relearn the failed crawling actions.

If a Web application level detection pattern also fails then the system will apply an approach similar to that later described in Section 3.5.2 for adapting the level detection pattern.

3.5.2. Crawl of a New Web Application

We are now in the case where we crawl a completely new Web applications whose template is (slightly) different from that present in the knowledge base. We assume that the Web application type detection patterns fired, but either the application level detection patterns or the crawling actions do not work on this specific Web application. In this situation, we cannot rely on previously crawled content from the knowledge base.

Let us first consider the case where the Web application level detection pattern works.

Web application level detected Recall that there are two classes of Web application levels: intermediate and terminal. We make the assumption that on intermediate levels, crawling actions (that are solely navigation actions) do not fail – on that level, navigations actions are usually fairly simple (they typically are simple extensions of the application level detection patterns, e.g., `//div[contains(@class,'post')]` for the detection pattern and `//div[contains(@class,'post')]//a/@href` for the navigation action). In our experiments we never needed to adapt them. We leave the case where they might fail to future work.

On the other hand, we consider that both navigation actions and extraction actions from terminal pages may need to be adapted.

```
Input: a URL u and a sets of crawling actions A
if not alreadyCrawled(u) then
    for a ∈ A do
        if hasExtractionFailed(u, a) then
            relaxedExpressions ← getRelaxedExpressions(a);
            for candidate ∈ relaxedExpressions do
                if not hasExtractionFailed(u, candidate) then
                    addToKnowledgeBase(candidate);
                    break ;
```

Algorithm 3.2: Adaptation to template change (new Web application)

The main steps of the adaptation algorithm are described in Algorithm 3.2. The system first checks the applicability of existing crawling actions and then fix the failed ones. The *getRelaxedExpressions* creates two set of relaxed expression (for best-case and worst-case). For each set, different variations of crawling action will be generated by relaxing predicates and tag names, enumerated by the number of relaxation needed (simple relaxations come first). Tag names are replaced with existing tag names of the DOM tree so that the relaxed expression matches. When relaxing an attribute name inside a predicate, the AAH only suggests candidates that would make the predicate true; to do that, the AAH first collects all possible attributes and their values from the page.

When the crawling action has for prefix a detection pattern, we do not touch this prefix but only relax the latter part. As an example, if an expression like `div[contains(@class,'post')]//h2[@class='post-title']` fails to extract the post title, and `div[contains(@class,'post')]` is the detection pattern that fired, then we will try several relaxations of the second half of the expression, e.g., replacing `@class` with `@id`, `'post-title'` with `'post-head'`, `h2` with `div`, etc. We favor relaxations that use parts from crawling actions in the knowledge base for other Web application types of the same general category (e.g., Web forum).

Once the system has generated all the possible relaxed expressions, it first tries with best-case ones and if they do not work with worst-case ones. More generally, expressions are ordered by the number of required relaxations. Any expression which succeeds in the extraction will be still tested with a few more pages of the same Web application level before being added to the knowledge base for future crawling.

Web page level not detected If the system does not detect the Web application level, then the crawling strategy cannot be initiated. First, the system tries adapting the detection pattern before fixing crawling actions. The idea is here the same as in the previous part: the system collects all candidate attributes, values, tag names from the knowledge base for the detected Web application type (e.g., WordPress) and then creates all possible combinations of relaxed expressions, ordered by the amount of relaxation, and test them one by one until one that works is found.

To illustrate, assume that the candidate set of attributes and values are: `@class='post'`, `@id=forum`, `@class='blog'` with candidate set of tag names `article` ,`div`, etc. The set of relaxed expressions will be generated by trying out each possible combination:

```
// article [contains(@class,'post')]
// article [contains(@id,'forum')]
// article [contains(@class,'blog')]
```

and similarly for other tag names.

After the collection of the set of relaxed expressions, the system will attempt to detect the Web application level by testing each relaxed expression and, if the system finally detects the Web application level, then the system will apply the crawling action adaptation as described above. If the system does not detect the Web page level then the adaptation fails.

3.6. Experiments

We present in this section experimental performance of our proposed system on its own and with respect to a baseline crawler, GNU wget* (since the scope of the crawl is quite simple – complete crawling of specific domain names – wget is as good as Heritrix here).

3.6.1. Experiment Setup

To evaluate the performance of AAH, we have crawled 100 Web applications (totaling nearly 3.3 millions Web pages) of two forms of social Web sites (Web forum and blog), for three specific content management system (vBulletin, phpBB, and WordPress). The Web applications of type WordPress (33 Web applications, 1.1 million of Web pages), vBulletin (33 Web applications, 1.2 million of Web pages) and phpBB (34 Web applications, 1 million Web pages) were randomly selected from three different sources:

(1) `http://rankings.big-boards.com/`, a database of popular Web forums.
(2) A dataset related to European financial crisis.
(3) A dataset related to the *Rock am Ring* music festival in Germany.

The second and third dataset were collected in the framework of ARCOMEM project [ARC13]. In these real-world datasets corresponding to specific archival tasks, 68% of the seed URLs of Web forum type belongs to either vBulletin or phpBB, which explains while we target these two CMSs. WordPress is also a prevalent CMS: the Web as a whole has over 61 million Wordpress sites [Wor12] out of a number of blogs indexed by Technorati [The09] of around 133 million. Moreover, Wordpress has a 48% market share of the top 100 blogs [Roy12]. All 100 Web applications were both crawled using wget and the AAH. Both crawlers are configured to retrieve only HTML documents, disregarding scripts, stylesheets, media files, etc.

The knowledge base is populated with detection patterns and crawling actions for one specific version of the three considered CMSs (other versions will be handled by the adaptation module). Adding a new Web application type to the knowledge base takes a crawl engineer about 30 minutes.

3.6.2. Performance Metrics

The performance of the AAH will be mainly measured by evaluating the number of HTTP requests made by both systems vs the amount of *useful* content retrieved. Note that since

*`http://www.gnu.org/software/wget/`

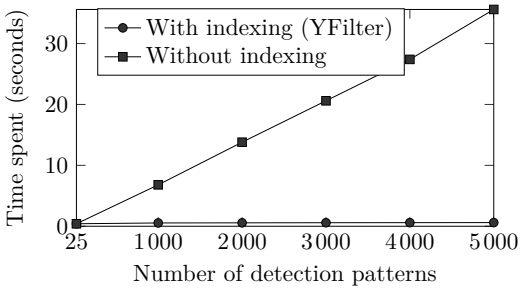


Figure 3.6.: Performance of the detection module

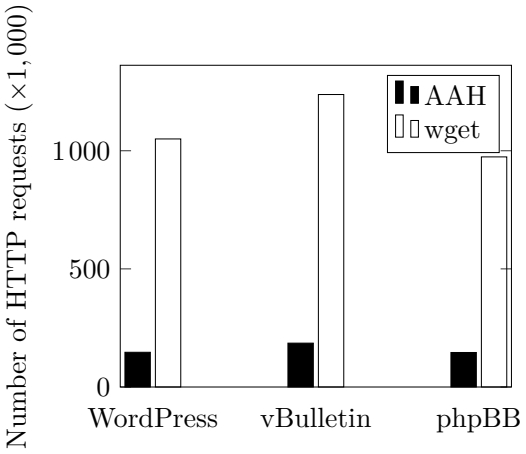


Figure 3.7.: Total number of HTTP requests used to crawl the dataset

wget is performing a complete, blind retrieval of the Web site, all URLs crawled by the AAH will also be crawled by wget.

Evaluating the number of HTTP requests is easy to perform by simply counting requests made by both crawlers. Coverage of useful content is more subjective; since it is impossible to ask for user evaluation given the volumes we consider, we use the following proxies:

(1) Counting the amount of textual content that has been retrieved. For that, we compare the proportion of 2-grams (sequences of two consecutive words) in the crawl result of both systems, for every Web application.

(2) Counting the number of external links (i.e., hyperlinks to another domain) found in the two crawls. The idea is that external links are of particularly important part of the content of a Web site.

3.6.3. Efficiency of Detection Patterns

We first briefly discuss the use of YFilter to speed up the indexing of detection patterns. In Figure 3.6 we show the time required to determine the Web application type in a synthetically generated knowledge base as the number of Web application types grows up to 5,000, with or without using YFilter indexing. The system takes a time linear in the number of detection patterns when indexing is turned off, taking up to several dozens of seconds. On the other hand, detection time is essentially constant with YFilter activated.

3.6.4. Crawl Efficiency

We compare the number of HTTP requests required by both crawlers to crawl each set of Web applications of the same type in Figure 3.7. Notice how the application-aware helper makes much fewer requests (on average 7 times fewer) than a regular blind crawl. Indeed, for blog-like Web sites, a regular crawler makes redundant HTTP requests for the same Web content, accessing to a post by tag, author, year, chronological order, etc. In a Web forum, many requests end up being search boxes, edit areas, print view of a post, areas protected by authentication, etc.

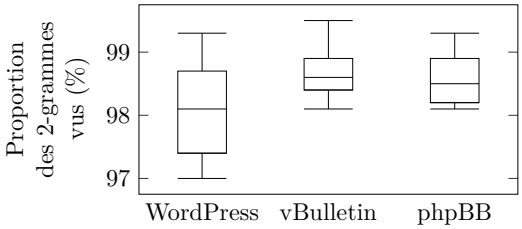


Figure 3.8.: Box chart of the proportion of seen n-grams for the three considered CMSs. We show in each case the minimum and maximum values (whiskers), first and third quartiles (box) and median (horizontal rule).

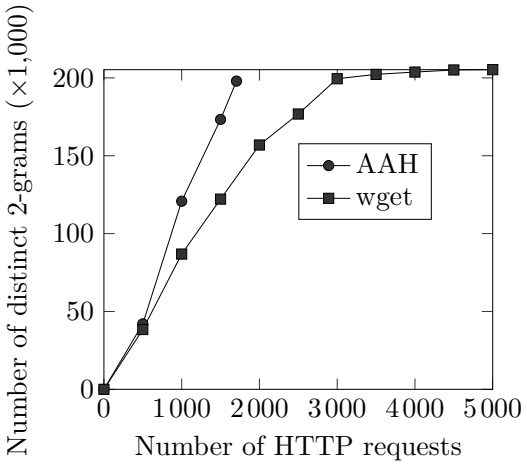


Figure 3.9.: Crawling `http://www.rockamring-blog.de/`

Table 3.1.: Coverage of external links in the dataset crawled by the AAH

CMS	External links	External links (w/o boilerplate)
WordPress	92.7%	99.8%
vBulletin	90.5%	99.5%
phpBB	92.1%	99.6%

3.6.5. Crawl Effectiveness

The crawling results, in terms of coverage of useful content, are summarized in Figure 3.8 and in Table 3.1. Figure 3.8 presents the distribution of the proportion of n-grams crawled by the AAH with respect to those of the full crawl. Not only are the numbers generally very high (for the three types, the median is greater than 98%), but the results are also very stable, with a very low variance: the worst coverage score on our whole dataset is greater than 97% (typically, lower scores are achieved for small Web sites where the amount of boilerplate text such as menus or terms of use remains non negligible). This hints at the statistical significance of the results.

The proportion of external links covered by the AAH is given in Table 3.1. The application-aware helper has ignored nearly 10 percent of external links since every page may use widgets, such as those of Facebook, Amazon, etc., with URLs varying from one page to another. Once we have excluded boilerplate with defined set of patterns, we see that more than 99.5% of the external links are present in the content crawled by the AAH.

To reach a better understanding of how an application-aware crawl enfolds, we plot in Figure 3.9 the number of distinct 2-grams discovered by the AAH and wget during one crawl (in this particular case, of a given WordPress blog), as the number of requests increase. We see that the AAH directly targets the interesting part of the Web application, with a number of newly discovered 2-grams that grows linearly with the number of requests made, to reach a final level of 98% 2-gram coverage after 1,705 requests. On the other

Table 3.2.: Examples of structural pattern changes: desktop vs mobile version of `http://www.androidpolice.com/`

Desktop version	Mobile version
div[@class='post_title']/**h3**/a	div[@class='post_title']/**h2**/a
div[@class='post_info']	div[@class='post_author']
div[@class='post_content']	div[@class='content']

hand, wget discovers new content with a lower rate, and, especially, spends the last 2/5 of its requests discovering very few new 2-grams.

3.6.6. Comparison to iRobot

The iRobot system [CYL+08] that we discussed in Chapter 2 is not available for testing because of intellectual property reasons. The experiments of [CYL+08] are somewhat limited in scope, since only 50,000 Web pages are considered, over 10 different forum Web sites (to compare with our evaluation, on 3.3 million Web pages, over 100 different forum or blog Web sites). To compare the AAH to iRobot, we have crawled one of the same Web forum used in [CYL+08]: `http://forums.asp.net/` (over 50,000 Web pages). The completeness of content of the AAH (in terms of both 2-grams and external links, boilerplate excluded) is over 99 percent; iRobot has a coverage of *valuable pages* (as evaluated by a human being) of 93 percent on the same Web application. The number of HTTP requests for iRobot is claimed in [CYL+08] to be 1.73 times less than a regular Web crawler; on the `http://forums.asp.net/` Web application, the AAH makes 7 times fewer requests than wget.

3.6.7. Adaptation when Recrawling a Web Application

To test our adaptation technique in the case of a recrawl of a Web application in a realistic environment (without having to wait for Web sites actually to change), we have considered sites that have both a desktop and mobile version with different HTML content. These sites use two different templates to present what is essentially the same content. We simulated a recrawl by first crawling the Web site with a `User-Agent:` HTTP header indicating a regular Web spider (the desktop version is then served) and then recrawling the mobile version using a mobile browser `User-Agent:`.

Our system was not only able to detect the structural changes from one version to another, but also, using already crawled content, to fix the failed crawling actions. Table 3.2 presents one exemplary Web application that has both a desktop and mobile versions, with a partial list of the structural changes in the patterns across the two versions. Our system was able to automatically correct these structure changes in both navigation and extraction, reaching a perfect agreement between the content extracted by the two crawls.

3.6.8. Adaptation for a New Web Application

As stated earlier, we have experimented our system with 100 Web applications, starting from a straightforward knowledge base containing information about one specific version of the three considered content management systems. Among the 100 applications, 77 did

not require any adaptation, which illustrates that many Web applications share common templates. The 23 remaining ones had a structure that did not match the crawling actions in the knowledge base; the AAH has applied adaptation successfully to these 23 cases. Most of the adaptation consisted in relaxing the class or id attribute rather than replacing the tag name of an element. When there was a tag name change, it was most often from span to div to article or vice versa, which is fairly straightforward to adapt. There was no case in the dataset when more than one relaxation for a given step of an XPath expression was needed; in other words, only best-case relaxed expressions were used. In 2 cases, the AAH was unable to adapt all extraction actions, but navigation actions still worked or could be adapted, which means the Web site could still be crawled, but some structured content was missing.

As an example of relaxation, from the Web application `http://talesfromanopenbook.wordpress.com/`, the extraction of the post title by the action `div[@class='post']/h2[@class='post-title']` failed and system tried with many candidate attributes, but the relaxation `div[@class='post']/h2[@class='storytitle']` was found.

3.7. Conclusions

In Web archiving, scarce resources are bandwidth, crawling time, and storage space rather than computation time [Mas06]. We have shown how application-aware crawling can help reduce bandwidth, time, and storage (by requiring less HTTP requests to crawl an entire Web application, avoiding duplicates) using limited computational resources in the process (to apply crawling actions on Web pages). Application-aware crawling also helps adding semantics to Web archives, increasing their value to users. The AAH has been integrated in two Web crawlers (see Chapter 4): the proprietary crawler of the Internet Memory Foundation [Intb], with whom we have closely collaborated, and into a customized version of Heritrix [Sig05], developed by the ATHENA research lab in the framework of the ARCOMEM project [ARC13].

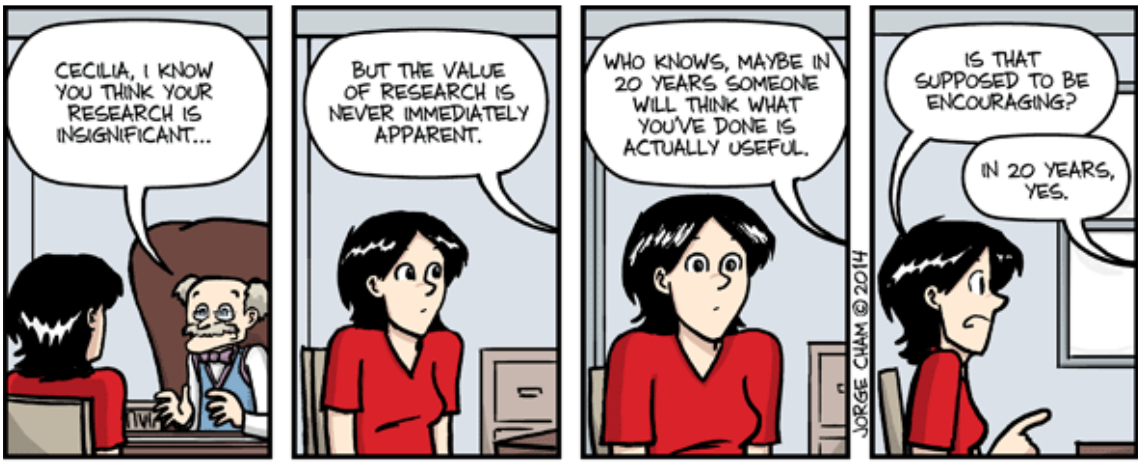
The application-aware helper is based on a hand-written knowledge base of given Web applications. The engineer may miss specifying navigation patterns to crawl other important content (e.g., interview pages in the case at hand). Therefore, in Chapter 5 we present the ACEBot (Adaptive Crawler Bot for data Extraction) system that learns the set of navigation patterns in a fully unsupervised manner. Indeed, it illustrates that an automatic approach not only crawls important pages, but also ensures that all portions of Web site have been considered.

The AAH does not crawl deep Web applications. In Chapter 6, we propose OWET (Open Web Extraction Toolkit) as such a platform, a free, publicly available data extraction framework that crawl complex Web applications making use of AJAX or Web forms.

Chapter 4.

ARCOMEM: Archiving Community Memories

This chapter describes the architecture of the Web crawling process inside the ARCOMEM project (`http://www.arcomem.eu/`). The content of this chapter is published in [1]. The AAH (see Chapter 3) has been integrated in the crawl processing chain of the ARCOMEM crawler. We are fully responsible for the AAH and its integration with other modules inside the ARCOMEM project.



The World Wide Web is the largest information repository. But this information is very *volatile*: the typical half-life of content referenced by URLs is of a few years [Koe03]; this trend is even aggravated in social media, where social networking APIs sometimes only extend to a week's worth of content [Twi11]. Web archiving [Mas06] deals with the collection, enrichment, curation, and preservation of today's volatile Web content in an archive that remains accessible to tomorrow's historians. Different strategies for Web archiving exist: bulk harvesting, selective harvesting and combinations of both. Bulk harvesting aims at capturing snapshots of entire domains. In contrast selective harvesting is much more focused, e.g., on an event or a person. Combined strategies include less frequent domain snapshots complemented with regular selective crawls. In the following we will focus on the technical aspects of selective crawls.

Selective crawls require a lot of manual work for the crawl preparation, crawler control, and quality assurance. On the technical level, current-day archiving crawlers, such as

Internet Archive's Heritrix [Sig05], crawl the Web in a conceptually simple manner (See Figure 1.5). They start from a *seed* list of URLs (typically provided by a Web archivist) to be stored in a queue. Web pages are then fetched from this queue one after the other, stored as is in the archive, and further links are extracted from them. If newly extracted links point to URLs that are in the scope of archiving tasks (usually given by a list or regular expressions of URLs to consider), they are added to the queue. This process ends after a specified time or when there is no interesting URL left to crawl. Due to this simple way of crawling, bulk domain crawls are well supported while selective crawls necessitate additional manual work for the preparation and quality assurance. It is the aim of the ARCOMEM* project to support the selective crawling on the technical level by leveraging social media and semantics to build meaningful Web archives [RDP+12]. This requires, in particular, a change of paradigm in how content is collected technically via Web crawling, which is the topic of the present chapter.

This traditional processing chain of a Web crawler like Heritrix [Sig05] has several major limitations:

- Only regular Web pages, accessible through hyperlinks and downloadable with an HTTP GET request, are ever candidates for inclusion in the archive; this excludes other forms of valuable Web information, such as that accessible through Web forms, social networking RESTful APIs, or AJAX applications.

- Web pages are stored as is in the archive, and the granularity of the archive is that of a Web page. Modern Web applications, however, often present individual blocks of information on different parts of a Web page: think of the messages on a Web forum, or the different news items on a news site. These individual *Web objects* can be of independent interest to archive users.

- The crawling process does not vary from one site to another. The crawler is blind to the kind of Web application hosted by this Web site, or to the software (typically, a *content management system*) that powers this Web application. This behavior might lead to resource loss in crawling irrelevant information (e.g., login page, edition page in a wiki system) and prevents any optimization of the crawling strategy within a Web site based on how the Web site is structured.

- The scope of a selective crawl is defined by a crude whitelist and blacklist of URL patterns; there is no way to specify that relevant pages are those that are related to a given semantic entity (say, a person) or that are heavily referenced from influential users in social networks.

- The notion of scope is binary: either a Web page is in the scope or it is not – on the other hand, it is very natural for a Web archivist to consider various degrees of relevance for different pieces of Web content; and ideally content should be crawled by decreasing degree of relevance.

The crawling architecture of ARCOMEM aims at solving these different issues by providing flexible, adaptive, intelligent content acquisition. This is achieved by interfacing traditional Web crawlers such as Heritrix with additional modules (complex resource

*`http://www.arcomem.eu/`

fetching, Web-application-aware extraction and crawling, online and offline analysis of content, prioritization), as well as by adapting the internals of the crawlers when needed (typically for managing priorities of content relevance). The objective of this chapter is to present an overview of this crawling architecture. The work presented in this chapter extends [PCM+13].

4.1. Architecture of the ARCOMEM Crawling System

The goal for the development of the ARCOMEM crawler architecture was to implement a socially aware and semantic-driven preservation model [RDP+12]. This requires thorough analysis of the crawled Web page and its components. These components of a Web page are called *Web objects* and can be the title, a paragraph, an image, or a video. Since a thorough analysis of all Web objects is time-consuming, the traditional way of Web crawling and archiving is no longer functioning. Therefore the ARCOMEM crawl principle is to start with a *semantically enhanced crawl specification* that extends traditional URL-based seed lists with semantic information about entities, topics or events. This crawl specification is complemented by a small reference crawl to learn more about the crawl topic and intention of the archivist. The combination of the original crawl specification with the extracted information from the reference crawl is called the *Intelligent Crawl Definition* (ICD). This specification, together with relatively simple semantic and social signals, is used to guide a broad crawl that is followed by a thorough analysis of the crawled content. Based on this analysis a semi-automatic selection of the content for the final archive is carried out.

The translation of these steps into the ARCOMEM crawling architecture foresees the following processing levels: the *crawler level*, the *online processing level*, the *offline processing level*, and the *cross-crawl analysis* that revolve around the ARCOMEM database as depicted in Figure 4.1 (See [1] for detailed description). Since the focus of this chapter is the crawling and the online analysis we will focus on these levels in the rest of the chapter and give only a brief overview on the other levels. More details about the other processing levels and the whole architecture can be found in [RDP+12].

4.2. AAH Integration in the ARCOMEM Project

The *application-aware helper* is fully integrated within the ARCOMEM system. It has been integrated in both *online* and *offline* modules. The goal of the application-aware helper (AAH) is to make the ARCOMEM crawler aware of the particular kind of Web application it is crawling, in order to adapt the crawling strategy accordingly. The presence of the AAH in the crawling processing chain ensures that the Web content is crawled in an intelligent and adaptive manner. A detailed description of the functions provided by the AAH is given in Chapter 3.

4.2.1. Online Analysis Modules

The online processing is tightly connected with the crawling level. At this level a number of semantic and social signals such as information about persons, locations, or social structure taken from the intelligent crawl specification are used to prioritize the crawler processing queue. Due to the near-real-time requirements, only time-efficient analysis can

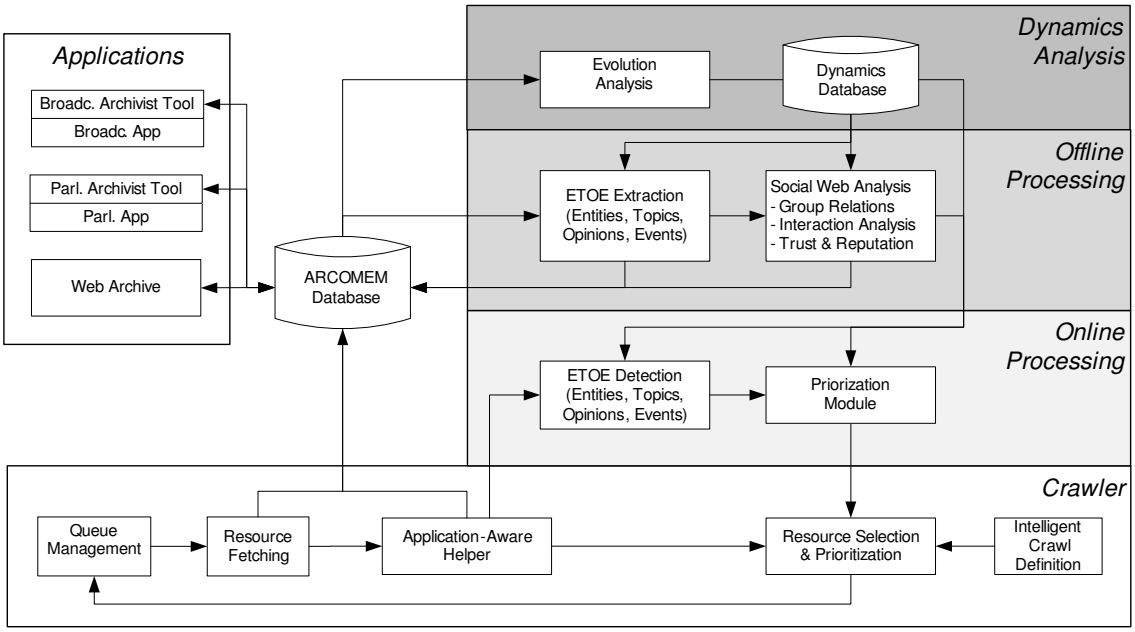


Figure 4.1.: Overall Architecture

be performed, while complex analysis tasks are moved to the offline phase. The logical separation between the online processing level and the crawler level will allow the extension of existing crawlers at least with some functionalities of the ARCOMEM technology.

Within the online analysis, several modules analyze crawled Web objects in order to guide the crawler. The purpose of this process is to provide scores for detected URLs. These scores are used for guiding the crawler in order to obtain a focused crawl with respect to the intelligent crawl definition (ICD). The main modules used within the online analysis are the AAH, the GATE platform for text analysis [CMB+11], and a prioritization module. The actual online processing consists of three phases which are displayed in Figure 4.2; within Figure 4.1 these phases are related to the connections between the online processing and the crawler.

(1) The AAH performs preprocessing steps on the crawled Web pages;

(2) Online analysis modules run on relevant document parts;

(3) The output of online analysis modules is aggregated and a score for each URL is provided.

A more detailed description of the these steps is provided in the remainder of this section.

First, we run the AAH on the Web page to detect regions of interest in the document and discard irrelevant parts. The input document is split into one or more document parts, and each document part is processed separately from now on. The AAH first identify a given Web page, and then a set of URLs are extracted and scores are assigned to help the prioritization module. The AAH also extract the Web objects (e.g., comments) for diversification module.

The online analysis also involves the textual analysis module (GATE), and priority aggregation module. We are not responsible of these modules, but rather are implemented

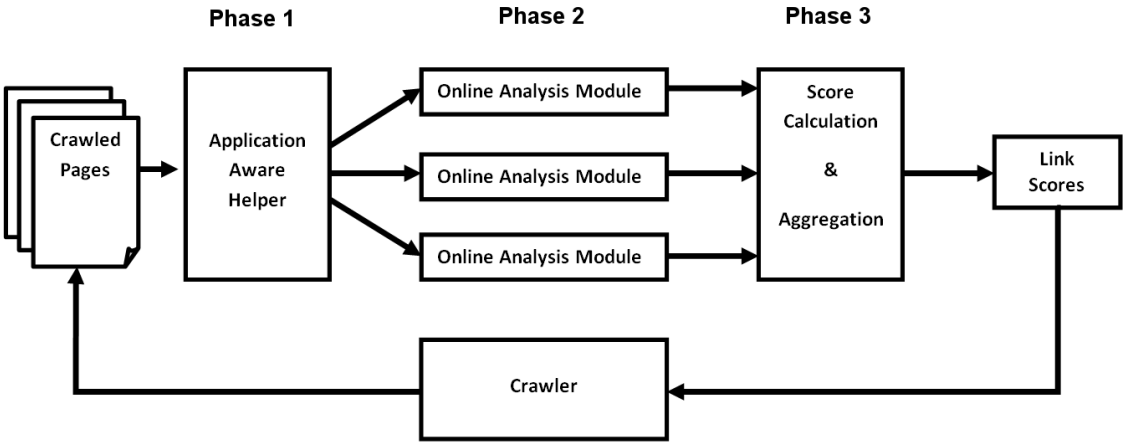


Figure 4.2.: Interaction of online analysis modules

in the framework of ARCOMEM project by our other partners. The detail of these modules is presented in [1].

4.2.2. Offline Processing Level

At this level, the AAH knowledge base is updated with newly learned knowledge base. The extracted Web objects from given documents are stored in the ARCOMEM database in the form of RDF triples and are available for further processing and information mining. After all the relevant processing has taken place, the Web pages to be archived and preserved are selected in a semi-automatic way and transferred to the Web archive (in the form of WARC files).

The offline, fully-featured, versions of the entity, topics, opinions, and events analysis (ETOE analysis) and the analysis of the social contents operate over the cleansed data from the crawl that are stored in the ARCOMEM database. These processing tools perform linguistic, machine learning and NLP methods in order to provide a rich set of metadata annotations that are interlinked with the original data. In addition to processing of textual content, multimedia content can also be analyzed and enriched with meta-information.

4.2.3. Conclusions

The scale of the Web, the volatility of information found in it, as well as the emergence of social media, require a shift in the way Web archiving is performed. Towards this goal, the ARCOMEM project has developed a scalable and effective framework that allows archivists to leverage social media and guide crawlers to collect both relevant and important information for preservation and future reference. Overall, the proposed crawling architecture is both extensible, by adding new modules to expand the analysis, and scalable, offering a new approach to crawling content for Web archiving.

The *application-aware helper* with other main modules play an important role in ARCOMEM's crawling architecture. The AAH extracts structured information from Web applications and assigns scores to the extracted URLs. A detailed description of ARCOMEM's crawling architecture, involved modules and experimental results can be found

in [1].

Chapter 5.

Adaptive Crawling Driven by Structure-Based Link Classification

The content of this chapter is also presented in [9].

The Web and User-Created Content The incredible growth of the World Wide Web has revolutionized the information age. Now the Web has become the largest data repository available to humankind, and several efforts have been made to utilize this resource. The Web is widespread (but ephemeral) and many social activities are happening there everyday. User-created content (UCC) has been growing at a very fast pace [OD08]. The cultural effects of this social phenomenon are also significant. Indeed, the Web has become a vast digital cultural artifact that needs to be preserved. But, unfortunately, important parts of our cultural heritage have already disappeared; as an example, the first ever Web page (its first version) that Tim Berners-Lee wrote back in 1990 has reportedly been lost [Bru13]. Certainly, Web preservation is a cultural and historical necessity for preserving valuable information for historians, journalists, or social scientists. Indeed, this is an objective of *Web archiving* [Mas06], which deals with selecting, crawling, preserving, and ensuring long-term access to historical Web content.

A large part of Web content (especially user-created content) belongs to Web sites powered by content management systems (CMSs) such as vBulletin, phpBB, or WordPress [GPT05]. The presentation layer of these CMSs use predefined templates (which may include left or

right sidebar of the Web document, header and footer, navigation bar, main content, etc.) for populating the content of the requested Web document from an underlying database.

These templates control the layout and appearance of the Web pages. A study [GPT05] has found that 40-50% content on the Web (in 2005) was template-based, growing at the rate of 6-8% per year. Depending on the request, the CMSs may use different templates for presenting information; e.g., for blogs, the *list of posts* type of page may use a different template than the *single post* Web page that also include comments. These template-based Web pages form a meaningful structure that mirror the implicit logical relationship between Web content across different pages within a Web site. Many templates are used by CMSs for generating different type of Web pages. Each template generates a set of Web pages (e.g., list of blog posts) that share the common structure, but differ in terms of content. These templates are consistently used across different regions of Web site. More importantly, in a given template (say, list of posts), links leading to a specific kind of content (say, individual posts) usually share a common layout and presentation properties.

Towards an Intelligent Crawler We introduce in this chapter an intelligent crawling technique that meet the above-stated criteria. We propose a *structure-driven* approach that is more precise, effective, and achieve higher quality level, without loss of information. It guides the crawler towards content-rich areas: this is achieved by learning the best traversal strategy (a collection of important navigation patterns) during an offline phase that ultimately guides the crawler to crawl only content-rich Web pages during online phase.

Our structure-driven crawler, ACEBot, first establishes connections among Web pages based on their root-to-link paths, then rank paths according their importance (i.e., root-to-links paths that lead to content-rich Web pages), and further learns a traversal strategy for bulk-downloading of the Web site. Most existing efforts on Web crawling only utilize hyperlink-related information such as URL pattern and anchor text [CvdBD99, MPSR01] to design a crawling strategy. Such approaches ignore the relationships among various Web pages and may judge the same kind of hyperlink appearing on different pages independently. Our main claim is that structure-based crawling strategy not only cluster Web pages which require similar crawling actions, but also helps to identify duplicates, redundancy, boilerplate, and plays as well an important role in prioritizing the frontier. Web pages that requires similar navigation patterns are believed to have similar types of content and to share the same structure.

We first present our model in Section 5.1. The algorithm that ACEBot follows is then presented in detail in Section 5.2. We discuss experiments in Section 5.3. Finally, we present our concluding remarks in Section 5.4.

5.1. Model

In this section, we formalize the model of our proposed approach: we see the Web site to crawl as an abstract directed graph, that is rooted (typically at the homepage of a site), and where edges are labeled (by structural properties of the corresponding hyperlink). We first consider the abstract problem, before explaining how we turn a Web site to crawl into an instance of this problem.

5.1.1. Formal Definitions

We fix countable sets of *labels* $\mathcal{L}$ and *items* $\mathcal{I}$. Our main object of study is the graph to crawl:

Definition 1. A *rooted graph* is a 5-tuple $G = (V, E, r, \iota, l)$ where V is a finite set of vertices, $E \subseteq V^2$ is a set of directed edges (potentially including loops), $r \in V$ is the *root* of the graph, $\iota : V \to 2^{\mathcal{I}}$ assigns a set of items to every vertex, and $l : E \to \mathcal{L}$ assigns a *label* to every edge.

Here, items serve to abstractly model the interesting content of Web pages. We naturally extend the function ι to a set of nodes X from G by posing: $\iota(X) = \bigcup_{u \in X} \iota(u)$.

We introduce the standard notion of paths within the graph:

Definition 2. Given a rooted graph $G = (V, E, r, \iota, l)$ and vertices $u, v \in V$, a *path* from u to v is a finite sequence of edges $e_1 \dots e_n$ from E such that there exists a set of nodes $u_1 \dots u_{n-1}$ in V with:

- $e_1 = (u, u_1)$;
- $\forall 1 < k < n, e_k = (u_{k-1}, u_k)$;
- $e_n = (u_{n-1}, v)$.

The *label* of the path $e_1 \dots e_n$ is the word over $\mathcal{L}$ $l(e_1) \dots l(e_n)$.

Critical to our approach is the notion of *navigation pattern* that uses edge labels to describe which paths to follow in a graph:

Definition 3. A *navigation pattern* p is a regular expression over $\mathcal{L}$. Given a graph $G = (V, E, r, \iota, l)$, the result of applying p onto G, denoted $p(G)$, is the set of nodes u such that there exists a path from r to u with a label *a prefix of a word* in the language defined by p.

We extend this notion to a finite set of navigation patterns P by letting $P(G) := \bigcup_{p \in P} p(G)$.

Note that we require only a *prefix* of a word to match: a navigation pattern does not only return the set of pages whose path from the root matches the regular expression, but also pages on those paths. For instance, consider a *path* $e_1 \dots e_n$ from r to a node u, such that the navigation pattern p is the regular expression $l(e_1) \dots l(e_n)$. Then, the result of executing navigation pattern p contains u, but also all pages on the path; more generally, p returns all pages whose path from the root matches a prefix of the expression $l(e_1) \dots l(e_n)$.

Navigation patterns are assigned a score:

Definition 4. Let $G = (V, E, r, \iota, l)$ be a rooted graph. The *score* of a finite set of navigation patterns P over G, denoted $\omega(P, G)$ is the average number of distinct items per node in $P(G)$, i.e.:

$$\omega(P, G) = \frac{|\iota(P(G))|}{|P(G)|}.$$

We can now formalize our problem of interest: given a rooted graph G and a collection of navigation patterns $\mathcal{P}$ (that may be all regular expressions over $\mathcal{L}$ or a subclass of regular expressions over $\mathcal{L}$), determine the set of navigation patterns $P \subseteq \mathcal{P}$ of maximal score over G: $\arg\max_{P \subseteq \mathcal{P}} \omega(P, G)$.. Unfortunately, we will show this problem is NP-hard. First, determining if the score is at least a given number is intractable:

Proposition 5.1. *Given a graph G, a collection of navigation patterns $\mathcal{P}$, and a constant K, determining if there exists a finite subset $P \subseteq \mathcal{P}$ with score over G at least K is an NP-complete problem.*

Proof. First, we argue that our problem is in NP, since given a graph G, a collection of navigation patterns $\mathcal{P}$ and a constant K, we can non-deterministically guess a subset $P \in \mathcal{P}$, compute its score over G, and check whether this score is at least K. The running time is polynomial.

For hardness, we apply a reduction from Set Cover. Given an instance of Set Cover (i.e., given a finite set U, a collection of sets $(S_i)_{1 \leqslant i \leqslant m}$ of elements of U, and an integer k, determine whether there exists a set $C \subseteq \{1,2 \dots m\}$ such that $|C| \leq k$ and $\bigcup\limits_{i \in C} S_i = U$), we construct the instance of our problem.

The set of items $\mathcal{I}$ is a superset of U; the set of labels has $m+1$ distinct labels $l_0, l_1 \dots l_m$. We first construct $G = (V, E, r, \iota, l)$ as follows: $V = \{x_0, x_1, \dots, x_N, s_1, \dots, s_m\}$ is a set of $N+1+m$ nodes (where N is an integer that we will define later on). E consists of the following edges:

- For all $1 \leqslant i \leqslant N-1$, an edge (x_i, x_{i+1}) with label $l(x_i, x_{i+1}) = l_0$.
- For all $1 \leqslant i \leqslant m$, an edge (x_N, s_i) with label $l(x_N, s_i) = l_i$.

The root r is x_0. The mapping ι is defined by $\iota(x_i) := \varnothing$ for $0 \leqslant i \leqslant N$ and $\iota(s_i) := S_i$ for $1 \leqslant i \leqslant m$. For $\mathcal{P}$ we take the set $\{\, l_0^* l_i \mid 1 \leqslant i \leqslant m \,\}$. The score of a pattern $l_0^* l_i \in \mathcal{P}$ is $\omega(l_0^* l_i, G) = \frac{|S_i|}{N+1}$. For $P_C = \{\, l_0^* l_i \mid i \in C \,\}$ with $C \subseteq \{1,2 \dots m\}$, $\omega(P_C, G) = \frac{\left|\bigcup\limits_{i \in C} S_i\right|}{|C|+N}$. See Figure 5.1 for an illustration of the construction.

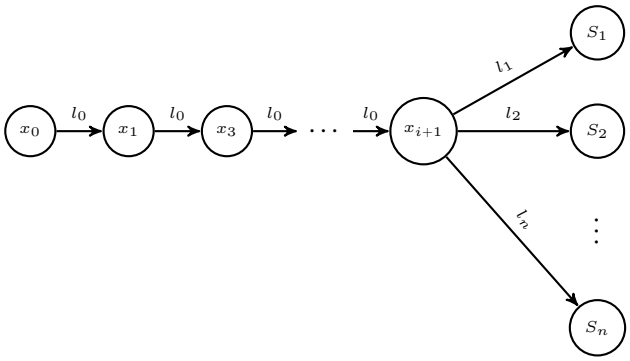


Figure 5.1.: Reduction from Set Cover

We now show that there exists a subset $P \subseteq \mathcal{P}$ with score over G at least $\frac{|U|}{k+N}$ if and only if the set cover instance has a solution of size at most k.

$\Leftarrow$ Assume the set cover instance has a solution of size at most k. Then there is a set $C \subseteq \{1,2 \dots m\}$ such that $|C| \leqslant k$ and $\bigcup_{i \in C} S_i = U$. The score of the set of patterns P_C is: $\omega(P_C, G) = \frac{\left|\bigcup\limits_{i \in C} S_i\right|}{|C|+N} \geqslant \frac{|U|}{k+N}$.

$\Rightarrow$ Let P_C such that $\omega(P_C, G) \geqslant \frac{|U|}{k+N}$. We distinguish two cases. First, assume $\bigcup\limits_{i \in C} S_i = U$. Then $\frac{\left|\bigcup\limits_{i \in C} S_i\right|}{|C|+N} \geqslant \frac{|U|}{k+N}$ implies that $|C| \leqslant k$ and $(S_i)_{i \in C}$ is a solution to the set cover problem.

Otherwise, we have $\left|\bigcup_{i\in C} S_i\right| \leqslant |U|-1$. We will show that this leads to a contradiction for a well-chosen value of N (observe that the choice of N has been left fully open until now). We take $N = m\times(|U|-1)+1$ (as U is an input to the set cover problem, having $N+1+m$ nodes in G still results in a polynomial-time construction). We have:

$$\begin{aligned}\omega(P_C,G) = \frac{\left|\bigcup_{i\in C} S_i\right|}{|C|+N} &\leqslant \frac{|U|-1}{|C|+m\times(|U|-1)+1}\\ &\leqslant \frac{|U|-1}{m\times(|U|-1)+1} = \frac{1}{m+\frac{1}{|U|-1}}\end{aligned}$$

whereas:

$$\begin{aligned}\omega(P_C,G) \geqslant \frac{|U|}{k+N} &= \frac{|U|}{k+m\times(|U|-1)+1}\\ &\geqslant \frac{|U|}{m+m\times(|U|-1)+1}\\ &= \frac{|U|}{m|U|+1} = \frac{1}{m+\frac{1}{|U|}}\\ &> \frac{1}{m+\frac{1}{|U|-1}}.\end{aligned}$$

We reach a contradiction that shows that, necessarily, $\bigcup_{i\in C} S_i = U$.

This concludes the reduction. The construction we used is polynomial time: the graph G is at most quadratic in the size of the original set cover instance (the quadratic explosion comes from the choice of N). □

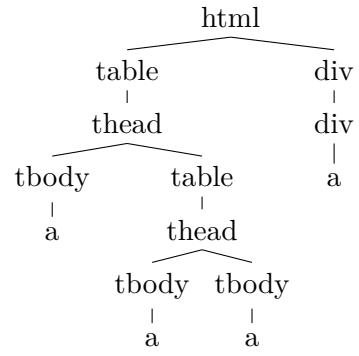
A simple corollary of this proposition shows the hardness of determining if a set of navigation patterns has optimal score:

Corollary 1. Given a graph G and a collection of navigation patterns $\mathcal{P}$, determining if one finite subset $P\subseteq\mathcal{P}$ has maximal score over G is a coNP-complete problem.

Proof. First, We argue that determining whether one subset $P\subseteq\mathcal{P}$ has maximal score over G is in coNP: to determine whether P is *not* maximal, guess another subset of navigation patterns P', compute its score in polynomial time, and check whether its less than the score of P.

Let's turn to the hardness. We reduce from the previous proposition. Let $G=(V,E,r,\iota,l)$ be a graph, $\mathcal{P}$ be a collection of navigation patterns, K a constant. Let $n=|V|$. Without loss of generality, we assume that $\iota(r)=\varnothing$ (otherwise, just add another dummy root, and update other parameters accordingly). Let $q = \arg\min_{1\leqslant i\leqslant n, \lfloor Ki\rfloor\neq Ki} K - \frac{\lfloor Ki\rfloor}{i}$.

We construct a graph $G'=(V',E',r,\iota',l')$ as an extension of G: $V' = V\cup\{y_0,\dots,y_q\}$, $E' = E\cup\{(r,y_0),(y_0,y_1),\dots,(y_{q-1},y_q)\}$, r is the same, ι' is the same on nodes of V, $\iota(y_i)=\varnothing$ for $0\leqslant i<n$, and $\iota(y_n)$ is a set of $\lfloor K\times q\rfloor$ fresh items. The label of the fresh edges is set to a fresh label λ. For the collection of navigation patterns, we take $\mathcal{P}' = \mathcal{P}\cup\{\lambda^*\}$.



(a) DOM tree representation for selected Web links

l_1 html-table-thead-table-thead-tbody-a
l_2 html-div-div-a
l_3 html-table-thead-tbody-a

(b) root-to-link paths

Figure 5.2.: A sampled Web page represented by a subset of the root-to-link paths in a corresponding DOM tree representation.

Now, we claim that there exists a subset $P \subseteq \mathcal{P}$ with score over G at least K if and only if $\{\lambda^*\}$ has maximal score over G'. Note that $\omega(\{\lambda^*\}, G') = \frac{\lfloor Kq \rfloor}{q} < K$. First assume that such a subset with score over G at least K exists. But $\omega(G', P) = \omega(G, P) \geqslant K$. Then $\{\lambda^*\}$, with score $< K$, cannot have maximal score. Conversely, assume such a subset does not exist. Let P be any subset of $\mathcal{P}$. By definition of q, $\omega(G', P) = \omega(G, P) \leqslant \frac{\lfloor Kq \rfloor}{q} = \omega(G', \{\lambda^*\})$ and thus $\{\lambda^*\}$ has maximal score. □

Thus, there is no hope of efficiently obtaining an optimal set of navigation patterns. In Section 5.2, we will develop a greedy approach to the selection of navigation patterns.

5.1.2. Model Generation

We now explain how we consider crawling of a Web site in the previously introduced abstract model.

A Web site is any HTTP-based application, formed with a set of interlinked Web pages that can be traversed from some base URL, such as `http://www.wsdm-conference.org/`. The base URL of a Web site is called the entry point of the site. For our purpose, we model a given Web site as a directed graph (see Definition 1), where the base URL becomes the root of the graph. Each vertex of the graph represents a distinct Web page and, following Definition 1, a set of items is assigned to every vertex. In our model, the items are all *distinct 2-grams* seen for a Web page. A 2-gram for a given Web page is a contiguous sequence of 2 words within its HTML representation. The set of 2-grams has been used as a summary of the content of a Web page [4]; the richer a content area is, the more distinct 2-grams. The set of items associated to each vertex plays an important role in the scoring function (see Definition 4), which eventually leads to selecting a set of Web pages for crawling.

A Web page is a well-formed HTML document and its Document Object Model (DOM [W3C98]) specifies how objects (i.e., texts, links, images, etc.) in a Web page are accessed. Hence, a *root-to-link* path is a location of the link (i.e., <a> HTML tag) in the corresponding DOM tree [W3C98]. Figure 5.2a shows a DOM tree representation and Figure 5.2b

illustrates its *root-to-link* path for a sample Web page.

Following the Definition 1, each edge of the graph is labeled with a *root-to-link* path. Assume there is an edge $e(u,v)$ from vertex u to v, then a label $l(e)$ for edge e is the *root-to-link* path of the hyperlink pointing to v in vertex (i.e., Web page) u. Navigation patterns will thus be (see Definition 3) regular expressions over root-to-link paths.

Two Web pages reachable from the root of a Web site with paths p_1 and p_2 whose label is the same are said to be *similar*.

Consider the scoring of a navigation pattern (see Definition 4). We can note the following:

- the higher the number of requests needed to download pages comprised by a navigation pattern, the lower the score;
- the higher the number of distinct n-grams in pages comprised by a navigation pattern, the higher the score.

5.2. Deriving the Crawling Strategy

In this section, we detail the crawling strategy for any given Web site. We begin by illustrating our approach with a simple example and then formally describe our unsupervised crawling technique.

5.2.1. Simple Example

Consider the homepage of a typical Web forum, say `http://forums.digitalspy.co.uk/`, as the entry point of the Web site to crawl. This Web page may be seen as a two different regions. There is a region with headers, menus and templates, that are presented across several Web pages, and is considered as a non-interesting region from archiving perspective. The other region at the center of the Web page is a content-rich area and required to be be archived. Since these pages are generated by a CMS (vBulletin in this particular case), the underlying templates have a coherent structure across similar Web pages. Therefore links contained in those pages obey regular formating rules. In our example Web site, the links leading to blog posts and the messages within an individual post should have some layout and presentational similarities.

Figure 5.2a presents a subset of the DOM tree for the example entry point Web page and its *root-to-link* paths are shown in Figure 5.2b. Figure 5.4a shows a truncated version of the generated graph (see Section 5.1.2) for the corresponding site. Each vertex represents a unique Web page in the graph. These vertices are connected through directed edges, labeled with *root-to-link* paths. Each vertex of the graph is assigned a number of distinct 2-gram seen for the linked Web page (e.g., 3,227 distinct 2-grams seen for p_3). Furthermore, the set of Web pages (i.e., vertices) that share the same path (i.e., edge label) are clustered together (see Figure 5.4b). The newly clustered vertices are assigned a collective 2-gram set seen for all clustered Web pages. For instance, the clustered vertex $\{p_3, p_4\}$ has now 5,107 distinct 2-gram items. After clustering, all possible navigation patterns are generated for the graph. This process is performed by traversing the directed graph. Table 5.1 exhibits all possible navigation patterns. Afterwards, each navigation pattern (possibly combination of *root-to-link* path) is assigned a score. The system does not compute the score for any navigation pattern that does not lead the crawler from the entry point of the Web site. Therefore, the system has ignored the navigation pattern l_4 (shown underlined). Here the score $\underline{2600}$ for navigation pattern l_4 is computed just for the sake of understanding, in

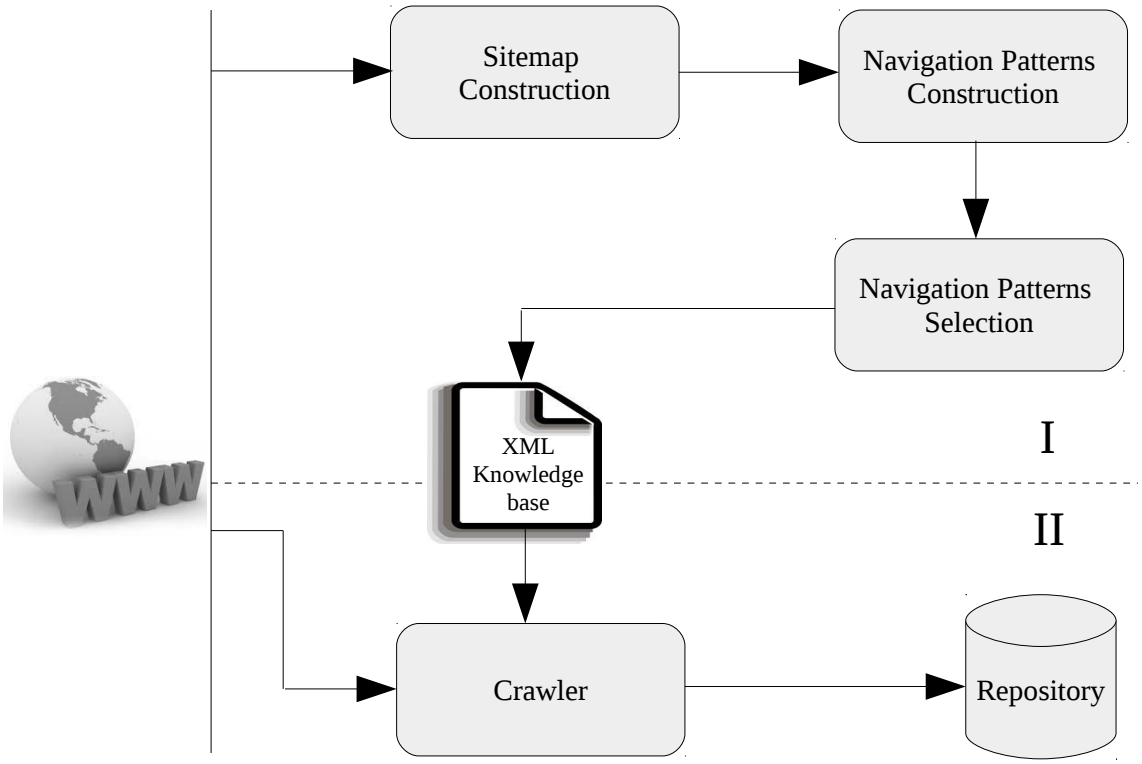


Figure 5.3.: Architecture of ACEBot, which consists of two phases: (I) offline sitemap construction and navigation patterns selection; and (II) online crawling.

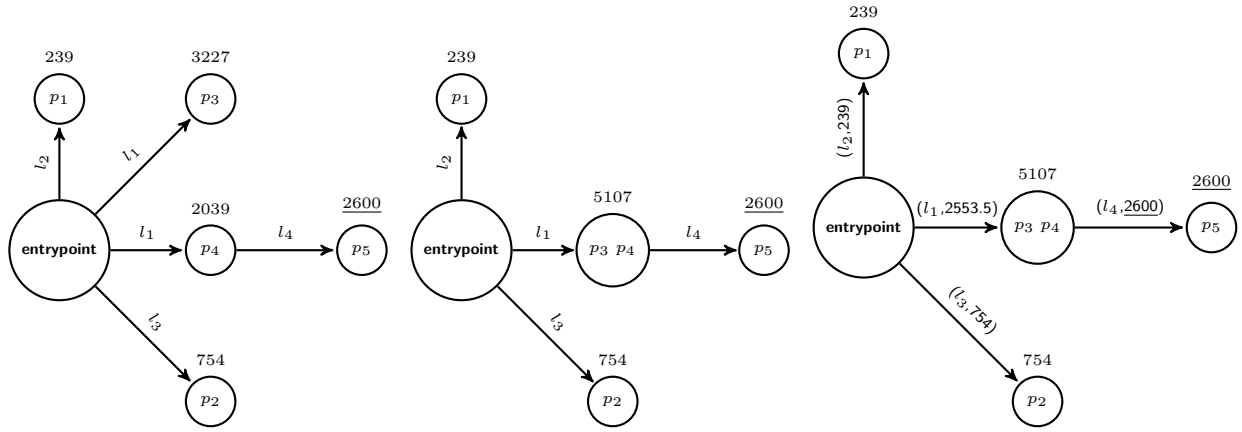
Table 5.1.: Navigation patterns with score for the example of Figure 5.4

NP	total 2-grams	distinct 2-grams	score
l_4	2600	2600	2600
l_1	5266	5107	2553.5
l_1, l_4	7866	7214	2404.7
l_3	754	754	754
l_2	239	239	239

practice, the system will not compute it. Once all possible navigation patterns are scored then the navigation pattern with highest score is selected (since highest score ensures the archiving of core-contents). Here, the navigation pattern l_1 is selected (not l_4).

The process of assigning the score to the navigation patterns keeps going after each selection for navigation patterns not selected so far. Importantly, 2-gram items for already selected vertices are not considered again for non-selected navigation patterns. Therefore, in the next iteration, the navigation pattern $l_1 l_4$ does not consider items from Web pages of the l_1 navigation pattern. The process of scoring and selecting ends when no interesting navigation pattern is left to follow.

Since we believe Web sites that belong to the same CMSs may enjoy the common templates, learned set of navigation patterns may work for several such Web sites.



(a) Before clustering (b) After clustering (c) After scoring

Figure 5.4.: Truncated Site Model with scores for the exemplary Web site (see Figure 5.2b for full labels)

5.2.2. Detailed Description

In this section, we detail the process of the unsupervised structure-driven crawler (ACEBot) proposed in our approach. First, we discuss the sitemap construction module. Then we present the navigation pattern construction and selection modules.

ACEBot (see Figure 5.3) mainly consists of two phases: offline and online phase. The aim of the offline phase is to first construct the sitemap and cluster the vertices that share

Input: entry point r, dynamic sitemap d, navigation pattern continuity limit ac, navigation-step k, a set of attributes a, completion ratio cr
Output: a set of selected navigation patterns SNP
$siteMap \leftarrow generateSiteMap(r, d)$;
$clusteredGraph \leftarrow performClustering(siteMap)$;
for $r \in R$ **do**
  $navigationPatterns \leftarrow getNavigationPatterns(r, clusteredGraph, k, ac, a)$;
  $NP \leftarrow updateNavigationPatterns(navigationPatterns)$;
while *not* cr **do**
  $topNP \leftarrow getTopNavigationPattern(NP, SNP)$;
  $SNP \leftarrow addToSelectedNP(topNP)$;
  $NP \leftarrow removeSubNavigationPatterns(topNP)$;

Algorithm 5.1: Selection of the navigation patterns

the similar edge label. Then a set of crawling actions (i.e., best navigation patterns) are learned to guide the massive crawling in online phase.

Algorithm 5.1 gives a high-level view of the navigation pattern selection mechanism for a given entry point (i.e., home page). Algorithm 5.1 has six parameters. The entry point r is the home page of a given Web site. The Boolean value of the parameter d specifies whether the sitemap of the Web site should be constructed dynamically. The argument k defines the depth (i.e., level or step) of navigation patterns to explore. The Boolean ac specifies whether to limit the continuity of navigation patterns to a fixed value of 3. For instance, for page-flipping navigation pattern /(/html/body/div[contains(@id,"navigation")]/a/@href{click/})+, the + indicate that the action will be executed at least once (i.e., $\{1, *\}$, where 1 is a lower limit). When Boolean value true is passed to ac, then the $\{1,3\}$ bound will be used. We have restricted the upper bound to 3; this will ensure that the navigation pattern that form the continuity of similar pages (e.g., page flipping navigation pattern) is selected. If we do not consider the upper bound 3 but just 1, then there are chances that page-flipping navigation patterns may be neglected because of lower score. Such navigation patterns may consist of a single page, and score based on one page may not lead to the right selection. This may also select other non important navigation patterns (that are not page-flipping actions). The experiments have shown that there were cases when weak navigation patterns were selected, but were avoided when limiting the continuity constraint to 3. The argument a passes the set of attributes (e.g., id, and class) that should be considered when constructing navigation patterns. cr sets the completion ratio, and the process of selecting navigation patterns ends when this criteria met.

The goal of the offline phase is to obtain useful knowledge for a given Web site based on a few sample pages. The sitemap construction is the foundation of the whole crawling process. The quality of sampled pages is important to decide whether learned navigation patterns target the content-rich part of a Web site. We have implemented a double-ended queue (similar to the one used in [CYL+08]), and then fetched the Web pages randomly from the front or end. We have limited the number of sampled pages to 3000, and detailed experiments (See Section 5.3) show that the sample restriction was enough to construct the sitemap of any considered Web site. The *generateSiteMap* procedure takes a given entry point as parameter and returns a sitemap (i.e., site model) as described in Definition 1. The

graph vertices are Web pages and edges between these vertices are *root-to-link* (location of the Web link) paths. For each Web page, the procedure analyze the links (i.e., <a> HTML tags) and their location within the DOM tree. It also computes distinct 2-grams (contiguous sequence of 2 consecutive words) seen for each Web page. A unique vertex is formed for each Web page and it is connected with its links via directed outgoing edges (See Figure 5.4a). Further, each Vertex of the graph is labeled with a number of distinct 2-grams seen for its Web page, and each arc leading to link is labeled with *root-to-link* path(see Figure 5.2b, 5.4a).

Intuitively, we assume that few Web links (i.e., <a> HTML tags) share the *root-to-link* paths within a Web page. Therefore, a vertex may have several destination vertices who share the same edge label. The importance of a specific navigation pattern (i.e., *root-to-link* path) ensures that the destination nodes hold the content-rich area of a Web site. This eventually helps to estimate the importance of the crawled Web pages. We cluster Web pages which share the similar navigation patterns, and thus it is a first approximation to approach the problem of discovering similar Web pages.

The procedure *performClustering* in algorithm 5.1 clusters the vertices with similar edge labels. It performs breadth-first traversal over the graph, starting from each root till the last destination vertex. For instance, Figure 5.4b, vertex p_3 and p_4 share the label l_1 and thus are clustered together. The 2-gram measure is also computed for each clustered vertex. More precisely, the clustering of the similar nodes is performed in two way:

- Clustering the destination vertices that share the edge label. For instance, list of blog posts where label l_2 is shared among several vertices.
- Let vertex v' has an incoming edge from vertex v with label l_1, and also vertex v' has an outgoing edge to vertex v'' with similar label l_1. Since v' and v'' share edge label, therefore these vertices will be clustered. For instance, page-flipping links (For instance, post messages that may exist across several pages) usually has the same *root-to-link* path. These type of navigation patterns end with + (e.g., /html/body/div[contains(@class,"navigation")]+), that indicate that crawling action should be performed more than once on similar Web pages during online phase. These type of actions has two advantages: learn a navigation pattern that hold the portion of Web site with many pages, and give enough evidence of it selection; inform the online phase to execute the action several times.

Moreover, the clustering phase also ensures that redundant Web pages do not play a role in selection of the best navigation patterns. Since we are computing the number of distinct 2-gram items for each Web page, the clustering phase does not group any new Web page which has similar 2-gram items to the existing (already clustered) Web page. For instance, the login page might always have the same 2-grams items.

Once the graph is clustered, the *getNPWithScore* extracts all possible navigation patterns for each root vertex $r \in R$. The procedure takes three parameters *clusteredGraph*, r, and k as input. The k parameter limits the number of navigation steps for a specific root vertex r. The procedure generates the navigation patterns using depth-first traversal approach where depth is limited to k (i.e., number of navigation-steps). Since the aim of our approach is to find a set of navigation patterns that lead the crawler from the entry point of the Web site to the interesting pages, here we only consider the navigation patterns that has root vertex as a start node. Hence, a set of navigation patterns are generated, starting from root vertex and counting the k number of navigation-steps. This step will be performed for each root vertex and *updateNavigationPatterns* will update the set of navigation patterns

NP accordingly.

The *getTopNavigationPattern* procedure returns a top navigation pattern on each iteration. The procedure takes two parameters NP (a set of navigation patterns), and SNP (a set of selected navigation patterns) as input. This procedure applies the subset scoring function (see Definition 4) and computes the score for each navigation pattern according to the rewritten scoring function as defined in section 5.1.2. The $items(NP)$ is computed by counting the total number of distinct 2-grams words seen for all vertices that share the navigation pattern NP. The size of the navigation pattern NP (i.e., $size(NP)$) is the total number of vertices that shares the NP. The SNP parameter is passed to the procedure to ensure that only new data rich areas are identified. Therefore, the scoring function does not consider the Web pages (i.e., clustered vertices), which are already discovered and hence does not take into account the score associated with them to evaluate new vertices. More precisely, assume the l_1l_2 navigation pattern is already selected. Now the scoring function for navigation pattern $l_1l_2l_3$ does not take into account the score for navigation pattern l_1l_2, but only l_3 score will play a role in its selection. Eventually, it guarantees that the system always selects the navigation patterns with newly discovered Web pages with valuable content. The *removeSubNavigationPatterns* procedure removes all the sub navigation patterns. For instance, if navigation pattern $l_1l_4l_5$ is selected first, then there is no need to evaluate the score for the navigation pattern l_1l_4 (if not already selected), since it will not guide to the new Web pages.

The redundant cluster (i.e., any two navigation patterns that has similar 2-gram items) will also not be selected by the *getTopNavigationPatterns* procedure. For instance, for blog-like Web sites, the same blog post may be accessible by tag, year, etc. Consider navigation patterns l_1l_5 and l_1l_6 that lead the crawler to redundant Web pages, if the l_1l_5 is already selected, then the l_1l_6 will not be selected. The l_1l_6 will get low score since it does not learn the new 2-grams items. Similarly, any navigation pattern that has redundant pages inside will have *few* distinct 2-grams items, and therefore will get a low score from the scoring function and thus will be less likely selected.

The selection of navigation patterns ends when all the navigation patterns from the set NP are selected or when content-rich Web pages met with the criteria (i.e., *completion ratio cr* condition satisfied). In our implementation, we have set the criteria to the 95% coverage of distinct 2-gram items seen for given entry point. The example of such a navigation pattern is:

```
doc("www.rockamring-blog.de/index.html")
/(/html/body/div[contains(@id,"wrapper")]/div[contains(@id,"navigation")]/a//@href{click/})+
/(/html/body/div[contains(@id,"wrapper")]/div[contains(@id,"post-")]/a/@href{click/})
```

We use here a restriction of the OXPath [SFG+11] language (see Figure 5.5) for the syntax navigation patterns.

When selected navigation patterns reaches the *cr* coverage of the total number of distinct 2-grams seen on the sitemap, the system will call the online phase and feed the selected navigation patterns to the crawler for massive online crawling.

5.3. Experiments

In this section, we present the experimental results of our proposed system. We compare the performance of ACEBot with AAH [4] (our previous work, that relies on a hand-written

```
⟨expr⟩      ::= "doc" "(" ⟨url⟩ ")" (⟨estep⟩)+
⟨estep⟩     ::= ⟨step⟩ | ⟨kleene⟩
⟨step⟩      ::= "/" "("⟨action⟩ ")"
⟨kleene⟩    ::= "/" "(" ⟨action⟩ ")" ( "*" | ⟨number⟩ )
⟨action⟩    ::= ("/" ⟨nodetest⟩)+ "{click/}"
⟨nodetest⟩  ::= tag | "@" tag
```

Figure 5.5.: BNF syntax of the OXPath [SFG+11] fragment used for navigation patterns. The following tokens are used: *tag* is a valid XML identifier; <url> follows the w3c BNF grammar for URLs is located at `http://www.w3.org/Addressing/URL/5_BNF.html#z18`.

description of given Web sites), iRobot [CYL+08] (a system of the literature dedicated to the efficient crawling of Web forums), and GNU wget* (a traditional Web crawler), in terms of efficiency and effectiveness.

The open-source ACEBot code and the experimental dataset (a list of sites) are freely available at `http://perso.telecom-paristech.fr/~faheem/acebot.html`.

5.3.1. Experiment Setup

To evaluate the performance of ACEBot at a Web-scale, we have carried out the evaluation of our system in various settings. Here, first we describe the dataset and performance metrics and different settings of our proposed algorithm.

Dataset We have selected 50 Web sites (totaling nearly 2 million Web pages) with diverse characteristics, to analyze the behavior of our system for small Web sites as well as for Web-scale extraction. The evaluation of ACEBot for different content domains is performed to examine its behavior in various situations. We consider Web sites of type blogs, news, music, travel, and books. We crawled 50 Web sites with both wget (for a full, exhaustive crawl), and our proposed system. To compare the performance of ACEBot with AAH, 10 Web sites (nearly 0.5 million Web pages) were crawled with both ACEBot and AAH.

Site map In offline phase, the site map of a given Web site is constructed either from the whole mirrored Web site or from a smaller collection of randomly selected sample pages. The mechanism for random (i.e., dynamic) selection of sample pages for a given entry point is described in Section 5.2.2. We found that ACEBot requires a sample of 3,000 pages to achieve optimal crawling quality, comparable to what was done for iRobot [CYL+08] (1,000 pages) and a supervised structure driven crawler [VdSdMC06b] (2,000 pages). We present in Figure 5.6 the variation of performance of ACEBot (in term of number of crawled distinct n-grams) as the number of sampled pages per site increases, shown averaged over 10 Web sites having each 50,000 total number of pages. ACEBot achieves nearly 80% of the proportion of seen *n-grams* with 2,000 sample pages, though stable performance for different Web sites is achieved over 2,500 sample pages. Since the sample pages are chosen

*http://www/gnu.org/software/wget/

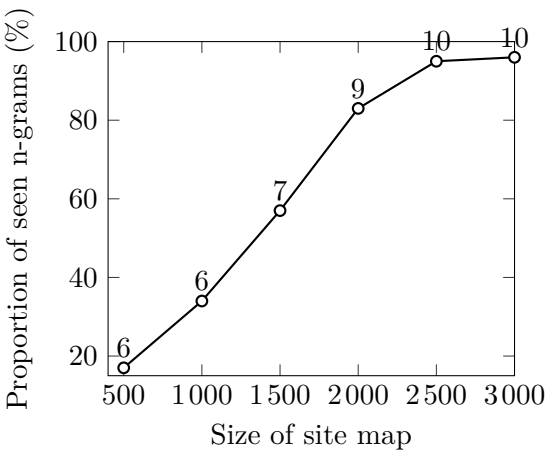


Figure 5.6.: Proportion of seen *n-grams* for 10 Web sites as the number of sample pages varies; plot labels indicate the number of times, out of 10 runs on the same Web sites, the maximum number of distinct n-grams was reached

randomly, and a set of navigation patterns learned for several runs may vary, we have run ACEBot for each sample size (e.g., 500) 10 times, to evaluate the performance over several runs. For sample size 500, the system selects the more inconsistent navigation patterns. ACEBot has seen 6 times (consider the plot point label for size 500) the same amount of distinct *n-grams* with size 500, whereas with size 3,000, every time the same navigation patterns were selected. Clearly, site map construction with size 3,000 yields the best result and performs consistently.

Algorithm We will consider several settings for our proposed algorithm 5.1. The additional parameters d, cr, k, ac, and a form several variants of our technique: The sitemap d may be dynamic (limiting to 3,000 Web pages; default if not otherwise specified) or complete (whole Web site mirror). The completion ration cr may take values 85%, 90%, 95% (default). The level depth k is set to either 2, 3 (default), or 4. The continuity limit ac, that specifies whether to limit repetition of navigation patterns to a fixed number (3, default) or consider arbitrary number of repetitions, is a Boolean. Finally, the attributes used, a, may be set to *id* (default), *class*, or both. Our system was tested for all these variants to evaluate the performance for different settings. We describe next the results, that we found highly stable and consistent. Occasionally, we present experiments for values of the parameters beyond the ones above-mentioned.

Performance Metrics We have compared the performance of ACEBot with AAH and GNU wget, by evaluating the number of HTTP requests made by these crawlers vs the number of useful content retrieved. We have considered the same performance metrics used by AAH [4], where the evaluation of number of HTTP requests is performed by simply counting the requests. Coverage of useful content is evaluated by comparing the proportion of 2-grams (sequences of two consecutive words) in the crawl result of three systems, for every Web site, and by counting the number of external links (i.e., hyperlinks to another domain) found in the three crawls. External links are considered an important part of the

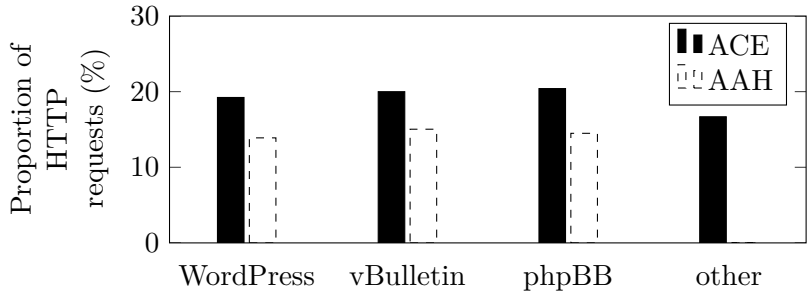


Figure 5.7.: Total number of HTTP requests used to crawl the dataset, in proportion to the total size of the dataset, by type of Web site

content of a Web site.

5.3.2. Crawl Efficiency

We have computed the number of pages crawled with ACEBot, AAH, and GNU wget, to compare the crawl efficiency of the three systems (see Figure 5.7). Here, wget obviously crawls 100% of the dataset. ACEBot makes 5 times fewer requests than blind crawl, and slightly more than AAH, the latter being only usable for the three CMS it handles. The performance of ACEBot remains stable when we experimented with WordPress, vBulletin, phpBB, and with other template-based Web sites. ACEBot exploits the link structure of a given Web site, and selects a set of navigation patterns with high score (see Definition 4). Therefore, our approach is not much affected by noisy links, invalid, and duplicates pages. Indeed, our approach avoids redundant requests for the same Web content (e.g., access a blog post by different services: tag, year, author, print view). Since all redundant pages have different navigation patterns, if one navigation pattern is selected then the other navigation patterns (leading to the redundant pages) will have lower score and hence will be automatically discarded.

The results shown in Figure 5.8 plot the number of seen 2-grams, and the number of HTTP requests made for a selected number of navigation patterns. The number of HTTP requests and discovered *n-grams* for a navigation pattern decide of its selection (a navigation pattern with best score is selected). Therefore a navigation patterns with one single page, but with many new *n-grams* may be selected ahead of a navigation pattern with many HTTP requests. Indeed, the Figure 5.8 elaborates that prospect, where the 10th selected navigation pattern crawl a large number of pages but this navigation pattern was selected only because it reached a higher completion ratio. The higher completion ratio ensure that the optimal number of navigation patterns are selected, including a navigation pattern with most (but important) number of HTTP requests.

5.3.3. Crawl Effectiveness

ACEBot crawling results in terms of coverage of useful content are summarized in Figures 5.9, 5.10, 5.11, and in Table 5.2.

Figure 5.9 illustrates the proportion of crawled *n-grams* by ACEBot for three possible combination of attributes *id*, and *class* with both complete (whole mirror of Web site) and dynamic (randomly selected 3000 pages) site maps. The experiments depicts the importance

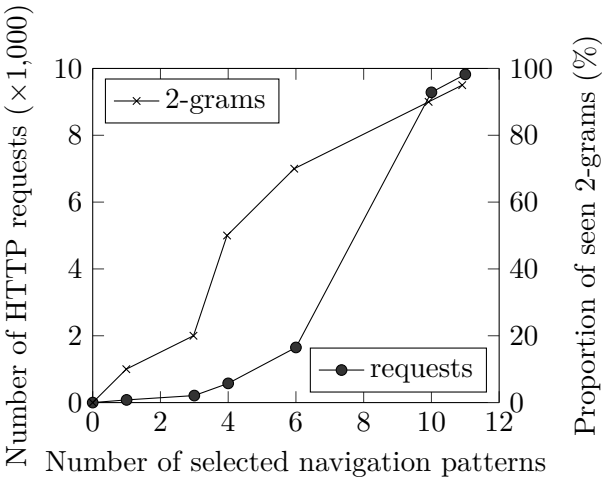
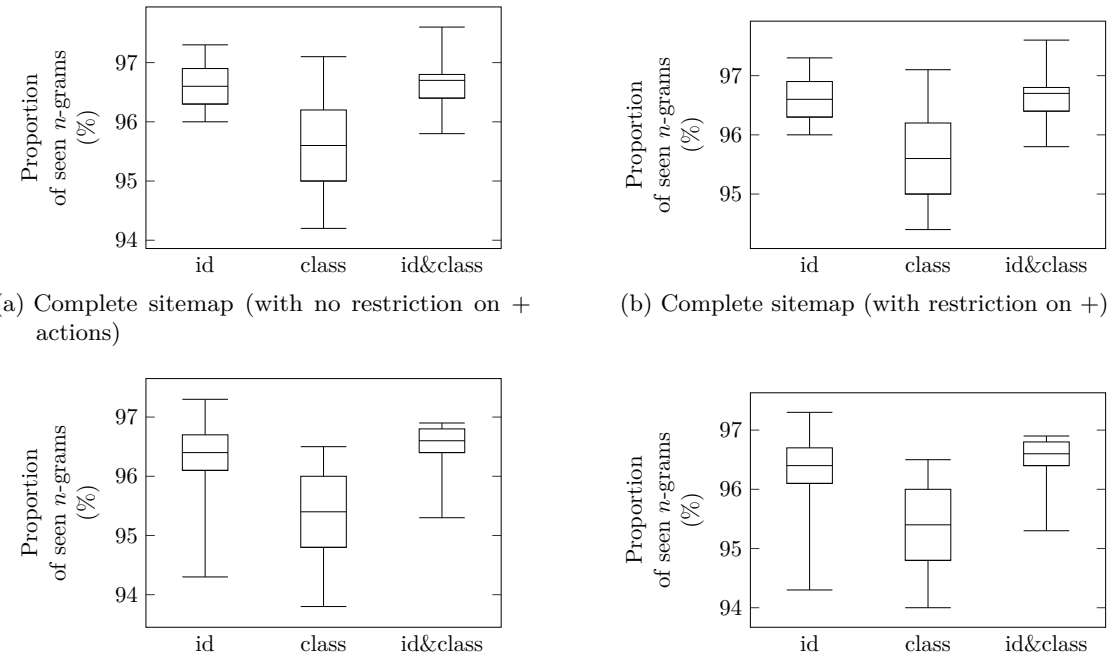


Figure 5.8.: Number of HTTP requests and proportion of seen n-grams for 10 Web sites as the number of selected navigation patterns increases

of a specific attribute by comparing the coverage of useful content. Figures 5.9a, 5.9b exhibit the results for a complete site map, whereas Figures 5.9c, 5.9d show the effectiveness for a dynamic sitemap. Further, a set of navigation patterns are learned by limiting (or not) the continuity constraints (for the action of + type). The experiments have shown that the navigation patterns with attribute *class* have less significance than *id* or when we consider both the attributes (i.e., *id* and *class*). The *class* attribute may have multiple values, and several *css* styles may be given to a node. Therefore, one has to consider that we ignore the multiple values for class attribute during the navigation pattern learning phase and just add a constraint `[@class]` for the node (e.g., `/div[@class]`). When we consider both attributes for learning the navigation patterns, the performance is more stable, but importantly, the learned navigation may work for the specific Web site and may lack the coverage of useful content when executed for similar kinds of Web sites. The *id* attribute has achieved a similar result, except for few cases, when higher coverage was achieved with both attributes. Also, it is to be expected that selected navigation patterns comprising just *id* attributes may show better performance for reuse in similar Web sites. ACEBot has achieved over 96% (median) effectiveness with both complete and dynamic sitemaps and whether (or not) the continuity action is restricted. The results are also very stable with very low variance: the worst coverage score for our whole dataset is 96% for complete site map and over 95% for dynamic site map with both attributes. Moreover, the restriction on action continuity does not effect the performances. This indeed shows the statistical significance of our results, and the appropriateness of limiting the continuity action to 3 repetitions.

The proportion of coverage of useful content and external links for different navigation steps (level) is shown in Table 5.2. Limiting navigation patterns to level 2 or 3 results in less HTTP requests, and a performance of 96% content with 95% completion ratio. But the level 3 performs better across many Web sites in terms of effectiveness, as important content exist till link depth 3. Once the learned navigation patterns achieve the 95% coverage of *n-grams* vs whole blind crawl, they will be stored in a knowledge base for future re-crawling.



(a) Complete sitemap (with no restriction on + actions)

(b) Complete sitemap (with restriction on +)

(c) Dynamic sitemap (with no restriction on +)

(d) Dynamic sitemap (with restriction on +)

Figure 5.9.: Box chart of the proportion of seen n-grams in the whole dataset for different settings: minimum and maximum values (whiskers), first and third quartiles (box), median (horizontal rule)

Figure 5.10 evaluate the performance of ACEBot for different completion ratio now 10 selected Web site, each with 50,000 Web pages. The selection of navigation patterns ends when the completion ratio has been achieved. The experiments have shown that the higher (and stable) proportion of *2-grams* are seen with completion ratio over 80%.

The proportion of external links coverage by ACEBot is given in Table 5.2. Since ACEBot selects the best navigation patterns and achieves higher content coverage, over 99% external links are present in the content crawled by ACEBot for the whole dataset.

5.3.4. Comparison to AAH

To reach a better understanding of the performance of ACEBot, we plot in Figure 5.11, the number of distinct *2-grams* seen by ACEBot, AAH, and wget during one crawl, as the number of requests increase. Clearly, AAH [4], and ACEBot directly crawl the interesting content of the Web site and newly discovered *2-grams* grow linearly with number of requests made. The results show the performance of automatic structure-based crawler (ACEBot) as close to the semi-automatic crawler (AAH). ACEBot makes 2,448 requests to reach 97% *2-grams* coverage, as compared to 92% content coverage with 2,200 requests by AAH. Since AAH is based on hand-crafted knowledge base, the engineer may miss to specify the navigation patterns to crawl other important content (e.g., interview pages in the case at hand). This indeed illustrates that an automatic approach not only crawls important pages, but also ensure that all portion of a Web site has considered.

The experiments of AAH [4] are performed for 100 Web sites (nearly 3.3 million Web

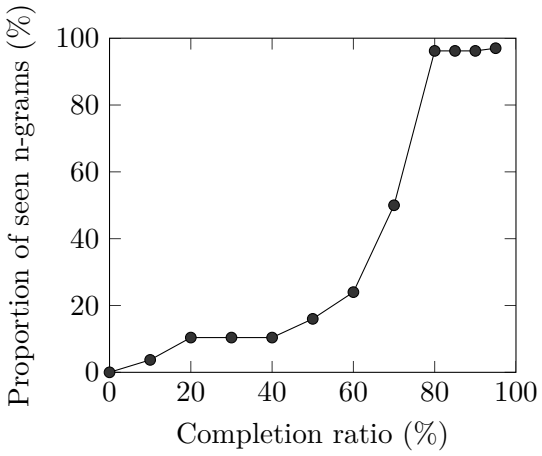


Figure 5.10.: Proportion of seen *n-grams* for different completion ratios for 10 Web sites

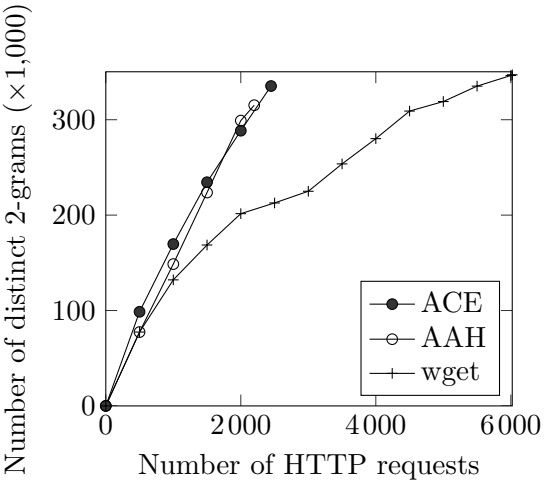


Figure 5.11.: Crawling `http://www.rockamring-blog.de/`

Table 5.2.: Performance of ACEBot for different levels with dynamic sitemap (restriction on +) for the whole data set (2 million pages)

Level	Requests	Content (%)	External Links (%)	Completion ratio (%)
2	376632	95.7	98.6	85
	377147	95.8	98.7	90
	394235	96.0	99.1	95
3	418654	96.3	99.2	85
	431572	96.6	99.3	90
	458547	96.8	99.3	95
4	491568	96.9	99.4	85
	532358	97.1	99.4	90
	588512	97.2	99.4	95

pages). To compare ACEBot to AAH more globally, we have crawled 10 of the same Web sites (nearly 0.5 million Web pages) used in AAH [4]. ACEBot is fully automatic, whereas the AAH is a semi-automatic approach (still domain dependent) and thus requires a hand-crafted knowledge base to initiate a bulk downloading of known Web applications. Over 96 percent crawl effectiveness in terms of *2-grams*, and over 99 percent in terms of external links is achieved for ACEBot, as compared to over 99 percent content completeness (in terms of both *2-grams* and external links) for AAH. The lower content retrieval for ACEBot than for AAH is naturally explained by the 95% target completion ration considered for ACEBot. However it is worth noticing that the performance of AAH relies on the hand written crawling strategy described in knowledge base by a crawl engineer. The crawl engineer must be fully aware of the Web pages structure (as well as the links organization) for the crawled Web site, to effectively download the important portion, as contrasted to our fully automatic approach, where one does not need to know such information for effective downloading and automatically learn the important portion of Web site. The current approach makes 5 times fewer HTTP requests as compared to 7 times for AAH (See Figure 5.7). Indeed the current effort crawls more pages than the AAH, but is fully unsupervised.

5.3.5. Comparison to iRobot

We have performed the comparison of our approach with the iRobot system [CYL+08]. iRobot is not available for testing because of intellectual property reasons. The experiments of [CYL+08] are performed just for 50,000 Web pages, over 10 different forum Web sites (to compare with our evaluation, on 2.0 million Web pages, over 50 different Web sites). To compare ACEBot to iRobot, we have crawled one of the same Web forum used in [CYL+08]: `http://www.tripadvisor.com/ForumHome` (over 50,000 Web pages). The completeness of content of the our system is nearly 97 percent in terms of *2-grams*, and 100 percent in terms of external links coverage; iRobot has a coverage of *valuable content* (as evaluated by a human being) of 93 percent on the same Web site. The crawl efficiency (in terms of number of HTTP requests) for iRobot is claimed in [CYL+08] to be 1.73 times less than a

regular Web crawler; on the `http://www.tripadvisor.com/ForumHome` Web application, ACEBot makes 5 times fewer requests than wget.

5.4. Conclusions

We have introduced an Adaptive Crawler Bot for data Extraction (ACEBot), that utilizes the inner structure of Web pages, rather than their content or URL-based clustering techniques, to determine which pages are important to crawl. Extensive experiments over a large dataset has shown that our proposed system performs well for Web sites that are data-intensive and, at the same time, present regular structure. We have compared the performance of ACEBot with generic crawler GNU wget, AAH, and iRobot. Our system significantly reduces duplicate, noisy links, and invalid pages without compromising on coverage of useful content, achieving high crawl quality as well. ACEBot only require 3,000 sample pages to construct a site map. Compared with a generic crawler, ACEBot makes 5 times fewer HTTP requests, slightly more than AAH (but AAH only handle three CMSs and also require hand-written knowledge base). ACEBot has also outperformed iRobot, that achieves 93% crawl effectiveness, compared to 97% in terms of useful content and 100% in terms of external links with ACEBot.

Chapter 6.

OWET: A Comprehensive Toolkit for Wrapper Induction and Scalable Data Extraction

This work has been carried out while visiting the DIADEM lab at University of Oxford. It is also presented in [8]. OWET encompasses three components: OWETDesigner, OWETGenerator, and OWETExecutor. We have contributed to OWETDesigner, and its integration to OWETGenerator and OWETExecutor.

Our proposed systems AAH (see Chapter 3) and ACEBot (see Chapter 5) cannot crawl the deep Web, i.e., pages behind the Web forms. Since a large part of information on the Web is hidden behind the Web forms, therefore a deeper crawl than a usual surface crawl is often required. In this chapter, we Introduce OWET (Open Web Extraction Toolkit) which performs the complex actions (such as form submission) to harvest the data from deep Web.

A Web of easily accessible data has often been dreamed, yet remains far from ubiquitous. There are plenty of data providers and efforts in creating and maintaining accurate knowledge-bases and APIs, yet most data remains hidden behind complex forms and only available in HTML. With the surge of big data analytics companies and citizens alike need this data more than ever, e.g., to fuel trading algorithms or pick a new camera, yet Web data extraction remains beyond the skills of most users.

Most data extraction systems tackle the three subsequent tasks site analysis, wrapper induction, and wrapper execution as a monolithic whole. For the lack of a reusable and publicly available platform, each system is burdened to implement all other components again and again. On the other hand, each of these three tasks has a clear interface, and a central goal to achieve, and is thus well-implementable as a reusable component: **(1)** *Site analysis.* The system must explore the given site and identify the relevant data to be extracted. This yields a sequence of navigation actions, leading to data rich listing pages, and a set of listing pages with annotations marking the contained records and their attributes. Critical to the site analysis is *accuracy,* as its results delimit the accuracy of the overall system. **(2)** *Wrapper induction.* Given the navigation actions and the annotated listing pages, the extraction system must derive a small program, called wrapper, which extracts the data from the site without relying on a costly analysis. The induced wrappers should be *robust* against minor site changes, i.e., the wrappers should tolerate minor variances, such as advertisements, and work accurately as long as possible. **(3)** *Wrapper execution.* Finally, the induced wrappers must be executed to extract the data. Scripting on modern sites requires the employment of standard browsers for loading pages and accessing their elements. For *scalability,* the wrapper execution framework should run large sets of wrappers concurrently efficiently, i.e., it should distribute and schedule multiple executions over several machines, recovering from unavoidable errors occurring during Web access.

Each of these components may be implemented very differently, e.g., **(1)** may work in a supervised fashion, asking the user for annotated records and attributes, or in a fully automated manner, relying on entity recognizers and template discovery. However, many systems produce a set of annotated example pages (or can be easily tweaked to do so), acting as interface between **(1)** and **(2)**. Likewise, many wrapper induction systems induce wrappers as a set of XPath expressions identifying these records and attributes, constituting the interface between **(2)** and **(3)**.

In this demonstration, we present Owet *(Open Web Extraction Toolkit),* a free, publicly available framework of easily reusable components implemented as REST services. Owet encompasses **(1)** OwetDesigner, a Firefox plugin acting as visual front-end to design and validate wrappers, **(2)** OwetGenerator for robust wrapper induction from few examples, and **(3)** OwetExecutor for large scale wrapper scheduling and execution.

OwetDesigner is one typical scenario for using a data extraction system, where the user supervises the wrapper induction. A typical OwetDesigner session consists of recording the navigation to one of the sought for listing pages, annotating some records and attributes thereupon, and marking the next link. While doing so, the OwetGenerator is invoked repeatedly to induce XPath expressions selecting navigation elements, records, attributes, and next links. OwetDesigner assembles from the induced expressions a wrapper to perform the same navigation sequence as the user and to extract all records on the reached pages. After validating the wrapper, it is handed to the OwetExecutor, possibly configured for repeated execution. OwetExecutor distributes the desired extractions over a cluster of extraction nodes and allows the user to view and control the progress of each extraction in real time. OwetGenerator and OwetExecutor are implemented as RESTful services, to be broadly reusable.

Now, we walk through our system, showing how to (i) annotate examples for the induction with OwetDesigner, (ii) generate expressions for these examples, (iii) integrate these expressions into a wrapper, (iv) test this wrapper, and (v) deploy it for execution.

However, this is just one use of Owet's components. In the diadem [FGG+12] project

we develop fully automated data extraction approaches, where OWETDESIGNER is replaced by sophisticated algorithms that generate examples automatically. Yet, it can use OWETGENERATOR and OWETEXECUTOR without modifications and as if those components would receive input from OWETDESIGNER. We will demonstrate this scenario with a set of extraction scenarios where the examples are generated by DIADEM. These scenarios are selected from over 10.000 UK real estate and used car sites included in a recent evaluation (see `http://diadem.cs.ox.ac.uk/evaluation/14/02`) where DIADEM induces wrappers with $> 97\%$ average accuracy for over 90% of sites.

6.1. OWET Highlights

We start our discussion of OWET with its wrapper language OXPath, since OXPath forms the interface between wrapper induction and execution. As such, OXPath must strike a balance between *high expressiveness* for accurate and robust wrappers and *low complexity* for their scalable execution. Furthermore, since most approaches leave some fringe cases to be dealt with by its users, OXPath must be also suitable for manual editing. Thus, OXPath should be *simple* enough to be edited by non-programmers. OXPath has been developed to accommodate these requirements in extending XPath slightly to interact with Web pages loaded into a life browser.

OXPath in a Nutshell OXPath is a superset of XPath, which is extended for declarative specification of interactions with Web applications for data extraction. To this end, OXPath introduces **(1)** the *action location step* for simulating user interaction such as mouse events, form filling; **(2)** the *style axis* for selecting nodes and fields based on actual visual attributes as rendered by the browser; **(3)** the *extraction marker predicate*, for marking data to be extracted, and **(4)** the *Kleene star operator*, for iterating expressions.

Actions, such as clicks or mouse-overs, can be executed in OXPath on any set of selected DOM nodes. For example, the following two lines enter "OXPath" into Google Scholar's search form and click its search button:

```
doc("scholar.google.com")/descendant::field()[1]/{"OXPath"}
  /following::field()[1]/{click/}
```

OXPath allows two types of action steps, namely *contextual actions steps*, such as {click} or {'OXPath'}, and *absolute actions steps* with a trailing slash, as in {click /}. An absolute action step returns the DOM root of the page after action execution, while contextual actions continue from the same context as the action, if possible.

The style axis allows selecting elements with their CSS properties, e.g., all visible fields (denoted as field()). The style axis relies on computed CSS properties and cannot be expressed in XPath. For example, we can select all paper links on Google Scholar:

```
//a[style::color="blue"][style::font-size="16px"]
```

Extraction markers allow as side effect the extraction of many data items while evaluating an OXPath expression (XPath, in contrast, returns a single node set). The following example extracts from Google Scholar each paper with its title and authors:

```
..//div[@class='gs_r']:<publication>[.//h3:<title=string(.)>]
  [.//span[@class='gs_a']:<authors=string(.)>]
```

OXPath supports many output formats, as XML this produces:

```
<publication><title>OXPath: A Language for ...</title>
    <authors>Tim Furche, ...</authors> ... </publication>
```

Kleene stars introduce iterated expressions. For instance, by Kleene-iterating an action following a next link, OXPath can reach all pages in a pagination sequence. The line below traverses all result pages on Google scholar and extracts the publication titles:

```
.../(// a[contains( string (.) , 'Next')]/{click/})*//h3:<title>
```

OWETDesigner OwetDesigner is a Mozilla Firefox plugin for building OXPath wrappers visually, by recording user actions and marking records and attributes to be extracted. OwetDesigner replaces VisualOXPath [KSG+12a], which was based on the Eclipse SWT Mozilla browser, which is no longer supported for Java. OwetDesigner improves over VisualOXPath by employing a real browser and integrating RESTful services to enhance induction robustness and extraction scalability.

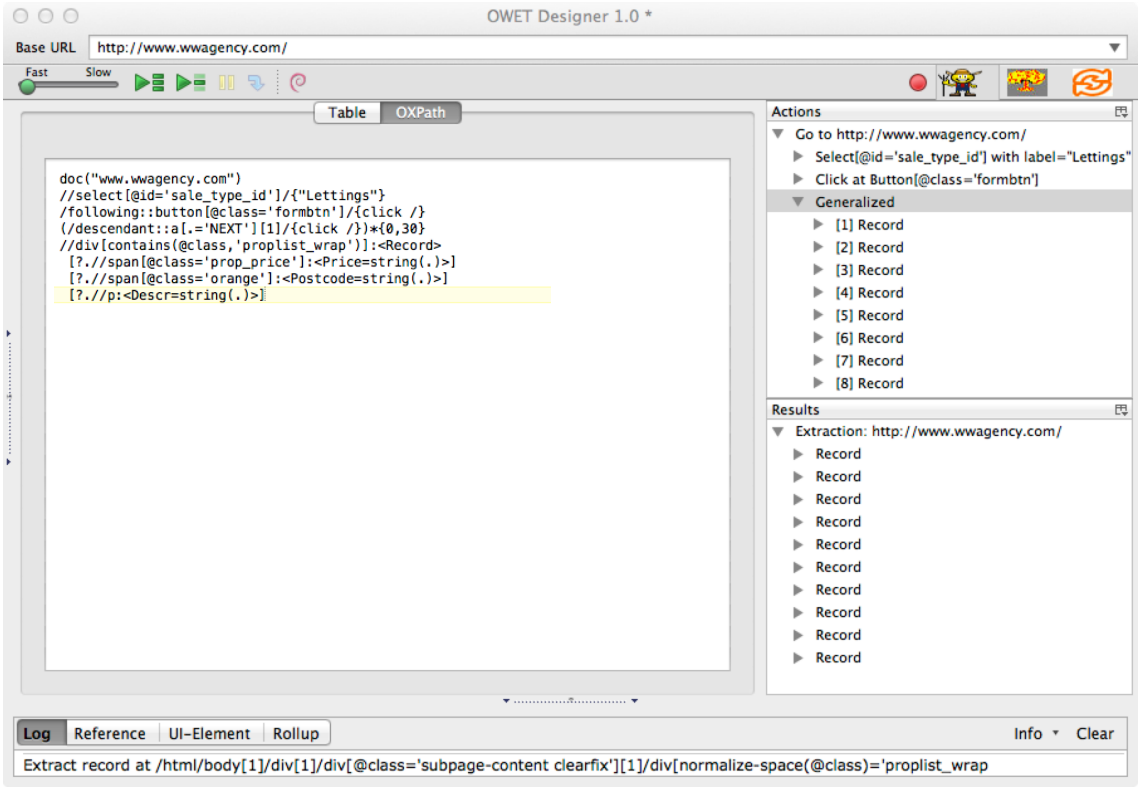
The top of Figure 6.1 shows OwetDesigner in our running example (detailed in Section 6.2). It consists mainly of three panels, the *Action View*, *OXPATH View,* and *Result View,* plus another view for logging information. The *Action View* (top-right) shows the editable hierarchy of recorded actions. The *Result View* (bottom-right) displays the data the current wrapper would extract on the current page, with records and respective attributes arranged as tree. Finally, the *OXPath View* (center) contains the wrapper source, as automatically derived from the recorded actions.

OwetDesigner has two modes, the *recording mode* activated with the red circle button above the action view, and the *replay mode* controlled with the buttons above the OXPath view. In recording mode, the system records the human interactions with the visited pages and generates OXPath code for re-enacting these actions. In replay mode, the tool replays the whole sequence of actions, highlighting the data marked for extraction. This is particularly useful to validate the wrapper not only on the current page, but beyond on further pages built from the same template.

Once the wrapper is defined and validated, OwetDesigner facilitates submission of the wrapper to the extraction server, configuring options such as scheduling frequency, output format, and browser to use.

OWETGenerator Relying on OXPath, we employ – like many other wrapper frameworks – XPath expressions to select relevant data corresponding to the loaded Web pages. While XPath is a powerful language for querying DOMs, most approaches only employ a small and often not well-specified fragment of XPath. In contrast, our approach generates expressions within a clearly defined XPath fragment, such that we can ask for the optimal XPath expression within this fragment to match a given node or node set. Herein, we rank XPath expressions according to the F-score for accuracy and a compostionally computed robustness score.

In this setting, inducing optimal XPath expressions leading from a single node to another single node is in polynomial time, while inducing optimal expressions for more complicated cases proves to be NP-hard. Our wrapper induction produces optimal expressions when feasible and otherwise falls back to heuristics which performed well in practical cases. In



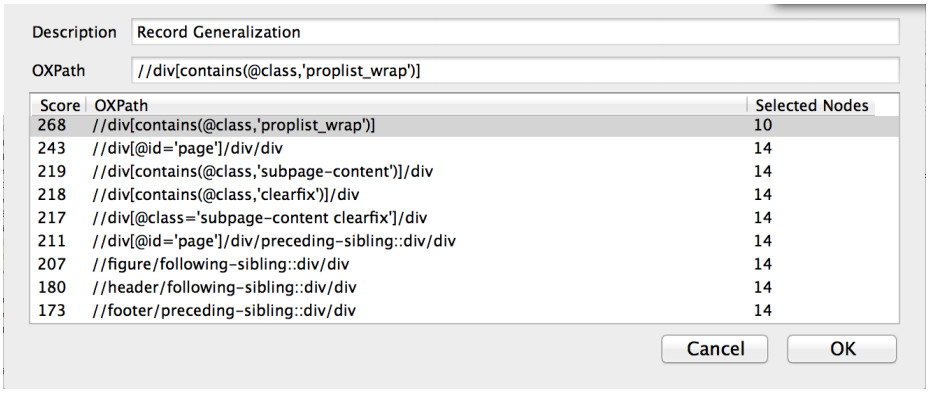


Figure 6.1.: OwetDesigner and Generalization Dialog

any case, the induced expressions can involve well-identifiable anchor nodes which are not necessarily on the direct path to the target nodes. This feature improves the quality of the generated expressions significantly.

Moreover, the wrapper induction generalizes the given examples whenever possible, i.e., we favor recall over precision under the assumption that manually provided examples are precise but incomplete. This differs from DIADEM's configuration where the automatic analysis delivered mostly complete but sometimes imprecise examples. Thus, for example, in this demo, our ranking relies on the F_1-score instead of the $F_{0.5}$-score in DIADEM.

OWETExecutor Extracting data from modern Web sites means going through scripted Web forms and result pages full of Ajax and frameworks like JQuery. To this end, OWET's extraction server executes OXPath wrappers in a browser, using a specifically-build DOM API on top of WebDriver to allow OXPath to gradually support all the major browsers (currently only Firefox is supported).

OWETEXECUTOR offers a rich REST API for submitting, deleting and checking the state of a task. In addition, it features: **(1)** *distribution* of the workload on several machines, **(2)** *scheduling* of tasks, **(3)** *error recovery and reporting*, and **(4)** automatic collection of *metadata on the runtime* execution.

To distribute tasks on several machines, we use Gearman (`gearman.org`) a popular message queue system that enables us to scale and naturally load balance by adding more queues (workers) processors when necessary. Scheduling of periodic e.g., daily or hourly, extractions it is crucial. Also, scheduling allows to implement strategies to avoid overtaxing the target Web site during multiple extraction, e.g. introducing delays among them. For task scheduling we use Quartz (`quartz-scheduler.org`). To deal with common errors e.g., network issues, unreachable sites, our server allows to specify whether, when and how many times the task should be restarted, and after how many failings to abort the task definitively. For each failed task a detailed report is created. Finally, OWETEXECUTOR collects automatically metadata on each wrapper execution, about e.g., runtime like memory usage and time, and on the extracted data e.g, number of records/attributes and visited pages. This information is very useful to automatically find out possible wrappers that need maintenance, as do not behave as in the past maybe due to changes in the target site.

In DIADEM we have employed OWETEXECUTOR for running a large scale evaluation on more than 10k Web sites in three domains, available at available at `http://diadem.cs.ox.ac.uk/evaluation/14/02`.

6.2. Demo Description

OWET can be used to turn any Web site into structured data to support your application. Say you are launching a new property rentals aggregator Web site, which receives the property listings directly from the individual agents. Because your policy dictates that agents cannot advertise the same property on another Web site at a lower price, an important issue to solve is how to automatically verify that the agents do not break this agreement. Such an application needs to get the properties' data as advertised on each agent Web site, and compare it with those in your current database. For each agent's Web site you create a wrapper scheduled for daily execution, which will feed your comparison algorithm. With many wrappers running daily, you may need error reporting to identify

Figure 6.2.: Selection data for extraction

failings wrappers due to site changes, as well as scheduling of extractions in order to optimize resource usage.

In this demonstration we showcase each of OWET's components, in particular, how a non-expert user can leverage OWETDESIGNER to select example data turned into a wrapper through OWETGENERATOR and executed and monitored by OWETEXECUTOR. In the following we use the `wwagency.com` Web site, a real-estate agent in Oxford. In addition to this live demonstration, we have prepared a set of examples taken from DIADEM' large scale evaluation, to show in more details other interesting aspects of the OWETGENERATOR, e.g., handling of navigation paths, induction of attributes as sub-text ranges. Similarly, to highlight the features of OWETEXECUTOR, we have prepared a set of wrappers to executes. For these, we exploit OWETGENERATOR and OWETEXECUTOR REST APIs.

The process begins with navigating to the Web site in Firefox and activating the OWETDESIGNER that by default starts recording any user interaction. OWETDESIGNER records the actions the user has to perform for the form filling. These actions are visualized in the "Action View" as shown in Figure 6.1, and can be possibly refined or deleted.

On the result page, partially showed in Figure 6.2, the user needs only to mark some examples, in terms of records and respective attributes. To help picking the right elements, OWETDESIGNER highlights the hovered node on the page. Typically, the user selects the first record of the listings, by choosing "Extract Record" from the contextual menu and assigns a name to this type of record (used as the table name for the extracted records). The chosen element is permanently highlighted by changing its background color on the page. The user then repeats this operation with one or two other similar records on the page. The "Action View" and the "Result View" in OWETDESIGNER are updated accordingly.

Once few examples have been marked, the user can invoke OWETGENERATOR for record generalization, which given the example expressions as input, tries to find a robust XPath expression that selects all the similar elements, in this case all the remaining records on the page. Specifically, OWETGENERATOR returns a set of expressions, ranked by a score of robustness, from which the user can pick the one that indeed selects all the wanted records. The bottom part of Figure 6.1 shows the dialog with the ranked expressions in case of records generalization. To ease choosing the right expression, OWETDESIGNER

allows to "test" these expressions by highlighting the elements they select on the page. In our example, the best expression it the first, //div[contains(@class,'proplist_wrap')], which indeed selects all the 10 records on the page. Note that this expression uses the class attribute value as condition, which is far more robust to possible changes than an expression based on less local criteria let alone canonical paths.

On Accepting the chosen expression, OwetDesigner will automatically update the "Result View" with 10 record nodes, and the "Action View" with a Generalization action. With records at hand, the next step is the identification of the relevant attributes. As for the records, the user visually picks, e.g. the price of the first record on the page and chooses "Extract Attribute" from the contextual menu. Here, a name for the extraction marker can be specified, e.g., Postcode. Also, the record this attribute belongs to can be chosen from a drop-down list showing 'option from 'Record[1]" to "Record[10]". However, OwetDesigner tries to guess default values for these, correctly proposing "Record[1]" in this case. After marking the postcode for another record, and similarly for other attribute types e.g., price, description, the user can invoke again OwetGenerator, this time to find a robust expressions for each attribute type such that it identifies the respective attributes in all the 10 records. E.g., for price attribute, it returns .//span[@class='prop_price'], which identifies correctly the price in each record.

As shown in Figure 6.2, the selected attributes are highlighted on the page, and the "Result View" now presents 10 records, each with the proper attribute values. In fact, the current wrapper is ready to extract the data from current page. In order to extend the extraction to the other result pages of the Web site, it is necessary to inform the wrapper with the action to perform to reach each next result page in sequence. In our example, the user selects and specify the actual "next link" to click and possibly the maximum number of iterations before stopping (default 30). Note, that OwetDesigner and OXPath are flexible enough to describe page iteration patterns on Web site that use different pagination techniques, e.g., infinite scrolling.

Providing a next link does not automatically ensure that the wrapper as specified on the first page works perfectly on the following pages. It may be the case that some pages of the Web site slightly differ, e.g., due to random ads of featured properties. For this reason, OwetDesigner allows the user to replay the full sequence of actions that compose the wrapper and validate its behavior on each of the following pages, visually verifying the highlighted records and attributes, as well as the values in the "Result View". The user can validate as many pages as necessary to be confident that the wrapper is consistent for the whole Web site. Considering that the generated expression are robust to small changes, usually is enough to validate the wrapper on 2-3 pages. If on some page the wrapper does not match, the user can refine it on this page and repeat the validation process if necessary.

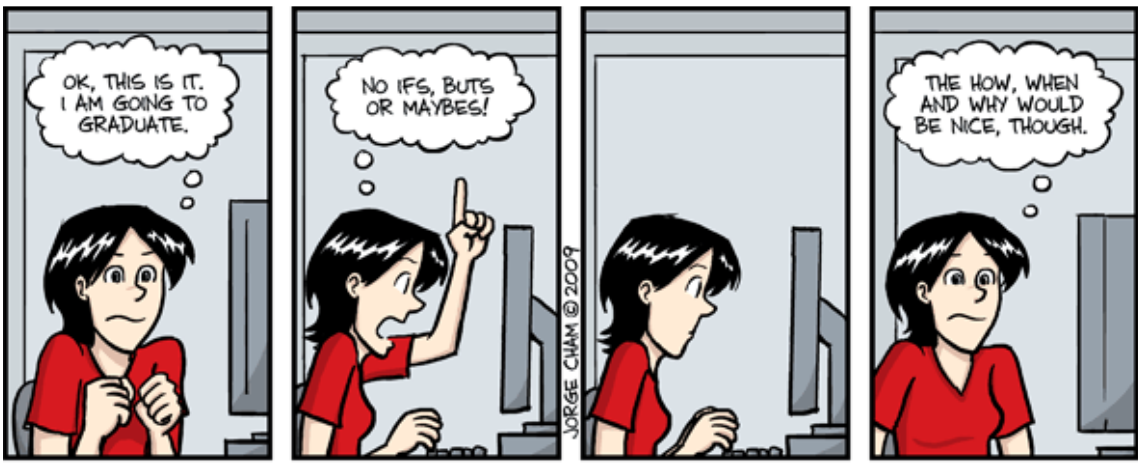
The following is the final wrapper obtained for our example, where only three attributes are considered for brevity.

```
doc("wwagency.com")//select[@id='sale_type_id']/{"Lettings"}
/following::button[@class='formbtn']/{click /}
(/descendant::a[.='NEXT'][1]/{click /})*{0,30}
//div[contains(@class,'proplist_wrap')]:<Property>
  [?.//span[@class='prop_price']:<Price=string(.)>]
  [?.//span[@class='orange']:<Postcode=string(.)>]
  [?.//p:<Descr=string(.)>]
```

In Line 1 the wrapper navigates to the target address, while Lines 2 and 3 deal with the form filling and submission. In Line 4 the iteration on the result pages is realized by clicking the next link until any is matched on the page, or for a maximum of 30 times. From Line 5 the wrapper specifies the data to extract by marking nodes as Property and extracting the text value of the nodes representing Postcode, Price and Descr attributes. Note that each attribute has been specified as optional (i.e., may be missing for some record) by prefixing the predicate expressions with a question mark.

At this point, the wrapper is ready to be executed. OwetDesigner allows the user to configure the extraction options and to submit the wrapper to OwetExecutor exploiting its REST API. In our example, we want to repeat the extraction with daily frequency. Also we specify XML as the output format (database or RDF other possible options), and Firefox as as browser to be used for running OXPath. Once the wrapper has been submitted, OwetExecutor returns a URL where the user can monitor the state of each daily extraction, collect the XML output files, view possible errors and even delete definitively the whole extraction task. To show the full set of features of OwetExecutor, we have prepared an additional set of wrappers to be executed with different output and configuration options, and an interface to monitor the progress of the parallel extractions.

Conclusion



In Web archiving, scarce resources are bandwidth, crawling time, and storage space rather than computation time [Mas06]. Existing approaches used by Web archivists to preserve the content of the Web blindly crawl and store Web pages, disregarding the CMS the site is based on (leading to suboptimal crawling strategies) and whatever structured content is contained in Web pages (resulting in page-level archives whose content is hard to exploit). We first propose the *application-aware helper* (AAH) (see Chapter 3) that extends the traditional architecture of a Web crawler (see the traditional architecture in Figure 1.5, and a modified architecture in Figure 3.5) to perform intelligent content acquisition for Web archiving, and show how application-aware crawling can help reduce bandwidth, time, and storage (by requiring less HTTP requests to crawl an entire Web application, avoiding duplicates) using limited computational resources in the process (to apply crawling actions on Web pages). The AAH makes 7 times fewer HTTP requests (see Figure 3.7) as compared to regular blind crawl without compromising on crawl effectiveness (see Figure 3.8 and Table 3.1). Application-aware crawling also helps adding semantics to Web archives, increasing their value to users. The AAH has been integrated in Internet Memory Foundation and Heritrix crawlers (see Chapter 4).

The AAH has introduced a semi-automatic crawling approach that relies on a hand-written description of known Web sites. In Chapter 5, we propose ACEBot (Adaptive Crawler Bot for data Extraction), a structure-driven crawler (fully automatic) that utilizes the inner structure of the Web pages and guides the crawling process based on the importance of their content. ACEBot has two phases, namely offline and online phase. The *offline* phase first constructs a dynamic site map (limiting the number of URLs retrieved), and then learns the best traversal strategy based on the importance of *navigation patterns* (selecting those leading to valuable content). The *online phase* performs massive crawling following the chosen navigation patterns. ACEBot crawls only important content with high precision and recall. Extensive experiments over a large dataset have shown that our

proposed system performs well for Web sites that are data-intensive and, at the same time, present regular structure. We have compared the performance of ACEBot with generic crawler GNU wget, AAH, and iRobot. Our system significantly reduces duplicate, noisy links, and invalid pages without compromising on coverage of useful content, achieving high crawl quality as well. ACEBot only requires 3,000 sample pages to construct a site map. Compared with a generic crawler, ACEBot makes 5 times fewer HTTP requests, slightly more than AAH (but AAH only handles three CMSs and also requires a hand-written knowledge base). ACEBot has also outperformed iRobot, that achieves 93% crawl effectiveness, compared to 97% in terms of useful content and 100% in terms of external links with ACEBot.

A large part of the information on the Web is hidden behind Web forms (known as deep Web, invisible Web, or hidden Web). Because of that, a deeper crawl than a usual surface crawl is often required. In Chapter 6, we propose OWET (Open Web Extraction Toolkit) as such a platform, a free, publicly available data extraction framework. OWET encompasses three components: *OWETDesigner*, a Firefox plugin acting as visual front-end to design and validate wrappers; *OWETGenerator*, for robust wrapper induction from few examples; *OWETExecutor*, for large-scale wrapper scheduling and execution.

Perspective and Future Work

There are different ways in which our proposed crawling models can be improved, in order to optimize or generalize current solutions. To conclude this thesis, we would like to highlight some future research directions stemming from the research conducted on intelligent Web crawling.

AAH interfacing with the API Blender

The application-aware helper has been integrated into the Internet Memory Foundation crawler in the framework of the ARCOMEM project and fits in an archiving processing chain to perform intelligent and adaptive crawling of Web applications. The AAH is able to refine the list of URLs to process. Another component developed within the ARCOMEM project is API Blender [GS12], that facilitates crawling of social networking sites through APIs. It would be useful to integrate the AAH with API Blender, to detect the set of URLs pointing to some sites accessible through APIs (e.g., Facebook, Twitter) and then pass detected URLs to API Blender for processing in a special way. Indeed, this will establish a direct pipe between both components. These changes can be implemented both in settings when AAH is standalone or when integrated into the IMF crawler.

Improving the AAH Adaptation Module

We feel the adaptation module introduced in AAH is at the moment too underspecified (especially for new Web applications), partly naive, and has not been tested enough. Since the AAH best performs for known Web applications and may adapt the wrapper to new Web applications, the adaptation module is the most interesting part of the system. We would like to conduct more comprehensive experiments to further verify our adaptation technique and improve upon it. Especially, the XPath expressions adapted in a worst-case scenario, and that totally depend on the already learned candidate attributes to relax

expression with, may end up trying many possible combinations and still fail to properly extract the Web objects. One possible direction could be to assist the adaptation module with a GUI. The GUI may let the user specify (or select) a set of Web objects she wants to extract from the similar type of Web pages. A set of extraction actions may be learned from the marked record and the local knowledge base may be updated accordingly for future crawling.

Introducing Web data Extraction within ACEBot

ACEBot guides the crawling process by learning a set of best navigation patterns. ACEBot does not extract the semantic objects (e.g., posts, comments) from the crawled Web pages. In the future, we aim to extend the proposed model (see Section 5.1), so that a set of wrappers are learned for each selected navigation pattern. Since we believe that a given navigation pattern leads to the same type of Web pages that require the same type of crawling actions. Therefore, a set of wrappers may be learned for each navigation pattern either by invoking a GUI (e.g., OWET) or by using a classifier (e.g., SVM) with some predefined features. The learned extraction actions will be stored in the local knowledge base.

Deep Web Crawling with AAH and ACEBot

Our proposed systems AAH and ACEBot cannot crawl the hidden Web, i.e., pages behind the Web forms. However, we have proposed OWET as such a platform, a free, and publicly available data extraction framework. In future, we aim to integrate both AAH and ACEBot with OWET. This indeed will allow both systems to also extract the data from important hidden sources. However, a challenge remains on how to locate the entry points to the hidden Web [BF07]. We will first investigate this direction to support the AAH and ACEBot in locating the hidden Web pages and then pass the set of URLs to OWET for Web crawling. We will also aim to explore query generation techniques [HXG+13] to fill in Web forms to make the crawling process fully unsupervised.

OWET Extensions

OWET has been implemented for the end users, who uses our system without deep understanding of the internal processes and are more interested in a simple tool, e.g., librarian, media group. In the future, we aim to introduce a advanced mode in OWET for advanced users. The advanced users (e.g., crawl engineer) are those who are familiar with Web technologies such as DOM, XML, XPath, OXPath, etc., and control the system and extraction process accordingly. We would also like to add support for some more complex actions, such as infinite scroll.

Appendix A.

Résumé en français

This appendix is an adaptation, in French, of the content of the Chapter 3; it does not contain any additional content, and may safely be skipped.

Introduction

Le World Wide Web est devenu un système de publication actif et une source riche en informations, grâce aux contributions de centaines de millions d'individus, qui utilisent le Web social comme médium pour diffuser leurs émotions, pour publier du contenu, pour discuter de sujets politiques, pour partager des vidéos, pour poster des commentaires et aussi pour donner leur opinion personnelle dans des discussions en cours. Une partie de cette expression publique s'accomplit sur les sites de réseaux sociaux et de partage social (Twitter, Facebook, Youtube, etc.), une partie sur des sites indépendants reposant sur des systèmes de gestion de contenu (*content management systems* ou CMS, incluant les blogs, les wikis, les sites d'actualités avec systèmes de commentaires, les forums Web). Le contenu publié sur cette gamme d'*applications Web* n'inclut pas seulement les divagations d'utilisateurs lambda du Web, mais également des informations qui sont d'ores-et-déjà remarquables ou qui seront précieuses aux historiens de demain. Barack Obama a ainsi d'abord annoncé sa réélection de 2012 comme président des États-Unis sur Twitter [Jup12] ; les blogs sont de plus en plus utilisés par les hommes politiques pour diffuser leur programme et pour être à l'écoute des citoyens [Col08] ; les forums Web sont devenue un mode commun, pour les dissidents politiques, de discussion de leurs opinions [MC02] ; des initiatives comme le projet Polymath* transforment les blogs en des tableaux interactifs collaboratifs pour la recherche [Nie11] ; les wikis construits par les utilisateurs, comme Wikipedia, contiennent des informations dont la qualité est au niveau des ouvrages de référence traditionnels [Gil05].

Parce que le contenu du Web est distribué, change pérpetuellement et est souvent socké dans des plates-formes propriétaires sans garantie d'accès à long terme, il est critique de préserver ces contenus précieux pour les historiens, journalistes, ou sociologues des générations futures. Ceci est l'objectif du domaine de l'*archivage du Web* [Mas06], qui traite de la découverte, du crawl, du stockage, de l'indexation, et de la mise à disposition à long terme des données du Web.

*`http://polymathprojects.org/`

Archivage dépendant de l'application

Les crawlers d'archivage actuels, tels que Heritrix d'Internet Archive [Sig05], fonctionnent de manière conceptuellement simple. Ils partent d'une liste *graine* d'URL à stocker dans une file (p. ex., l'URL de départ peut être la page principale d'un site). Les pages Web sont ensuite récupérées de cette pile l'une après l'autre (en respectant l'éthique du crawl, en limitant le nombre de requêtes par serveur), stockées telles qu'elles, et des hyperliens en sont extraits. Si ces liens sont dans la portée de la tâche d'archivage, les URL nouvellement extraites sont ajoutées à la file. Ce processus s'arrête après une durée déterminée ou quand aucune nouvelle URL intéressante ne peut être trouvée.

Cette approche ne répond pas aux défis du crawl d'applications Web modernes : la nature de l'application Web crawlée n'est pas prise en compte pour décider de la stratégie de crawl ou du contenu à être stocké ; les applications Web avec du contenu dynamiques (p. ex., les Web forums, les blogs) peuvent être crawlées de manière inefficace, en terme de nombre de requêtes HTTP requises pour archiver un site donné ; le contenu stocké dans l'archive peut être redondant, et n'a typiquement aucune structure (il consiste en des fichiers HTML plats), ce qui rend l'accès à l'archive pénible.

Le but de ce travail est d'adresser ces défis en introduisant une nouvelle approche *dépendante de l'application* pour le crawl Web d'archivage. Notre système, l'*application-aware helper* (ou AAH en bref) se repose sur une base de connaissances d'applications Web connues. Une *application Web* est n'importe quelle application basée sur HTTP qui utilise le Web et les technologies de navigateur Web pour publier de l'information en utilisant un modèle spécifique. Nous nous focalisons en particulier sur les aspects sociaux du Web, qui dépendent largement des contenus engendrés par les utilisateurs, de l'interaction sociale, et de la mise en réseaux, ainsi qu'on le trouve sur les forums Web, les blogs, ou sur les sites de réseaux sociaux. Notre AAH proposé ne récupère que la partie importante du contenu d'une application Web (c.à.d., le contenu qui sera précieux dans une archive Web) et évite les doublons, les URL inintéressantes et les parties de la page qui ont juste des but de présentation. De plus l'AAH extrait des pages du Web des contenus individuels (tels que le contenu d'un message de blog, son auteur, son estampille temporelle) qui peuvent être stockés en utilisant des technologies du Web sémantiques pour un accès plus riche et plus sémantique à l'archive.

Pour illustrer, considérons l'exemple d'un forum Web, disons s'appuyant sur un système de gestion de contenu comme vBulletin. Côté serveur, les fils de discussion et les messages du forum sont stockés dans une base de données. Fréquemment, l'accès à deux URL différentes va résulter en le même contenu ou en du contenu qui se recouvre partiellement. Par exemple, on peut accéder à un message d'une utilisatrice donnée à la fois à travers de la vue classique des messages d'un forum sous la forme d'un fil de discussion, ou par la liste de toutes ses contributions telles affichées sur son profil utilisateur. Cette redondance signifie qu'une archive construit par un crawler Web classique contiendra de l'information dupliquée, et que de nombreuses requêtes au serveur ne produiront aucune information nouvelle. Dans des cas extrêmes, le crawler peut tomber dans un *piège à robots* car il a un nombre virtuellement infini de liens à crawler. Il y a également plusieurs liens inutiles comme ceux vers une version imprimable d'une page, ou vers une publicité, liens qu'il faudrait mieux éviter durant la création d'une archive. Au contraire, le client Web qui est conscient de l'information à crawler peut déterminer un chemin optimal pour crawler l'ensemble des messages d'un forum, en évitant toute requête inutile, et peut stocker les

messages individuels, avec leurs auteur et estampille, dans une forme structurée que les archivistes et utilisateurs de l'archives pourront bénéficier.

Changement de structure

Les applications Web sont par nature dynamiques ; non seulement leur contenu change avec le temps, mais leur structure et apparence change également. Les systèmes de gestion de contenu fournissent différent thèmes qui peuvent être utilisés pour engendrer des articles de wiki, des messages de blog ou de forum, etc. Ces systèmes fournissent habituellement une manière de changer ce modèle (qui conduit ultimement à un changement de structure des pages Web) sans altérer le contenu en information, pour s'adapter aux besoins d'un site spécifique. Le rendu peut également changer quand une nouvelle version du CMS est installée.

Tous ces changements de rendu aboutissent à des changements possibles dans l'arbre DOM de la page Web, souvent mineurs. Cela rend plus difficile de reconnaître et de traiter de manière intelligente toutes les instances d'un système de gestion de contenu donné, car il est sans espoir de décrire manuellement toutes les variations possibles d'un thème dans une base de connaissances des applications Web. Un autre but de notre travail est de produire une approche de collecte intelligente qui est résistante aux changements mineurs de modèles et, en particulier, qui adapte automatiquement son comportement à ces changements, mettant à jour sa connaissance des CMS à cette occasion. Notre technique d'adaptation s'appuie à la fois sur une relaxation des motifs de crawl et d'extraction présents dans la base de connaissances et sur la comparaison des versions successives de la même page Web.

Plan

Nous présentons à la suite de cette introduction un état de l'art du crawl d'applications Web et de l'adaptation au changement de structure. Après avoir donné quelques définitions préliminaires en section A.0.5, nous décrivons notre base de connaissances d'applications Web en section A.0.5. La méthodologie que notre application-aware helper implémente est ensuite présentée en section A.0.5. Nous discutons du problème spécifique de l'adaptation au changement de structure et les algorithmes liés en section A.0.5, avant de couvrir les questions d'implémentation et d'expliquer comment l'AAH s'utilise conjointement à un crawler en section A.0.7. Nous comparons enfin l'efficacité (en terme de temps et de résultats) de notre AAH par rapport à l'approche classique de crawl pour crawler des blogs et forums Web en section A.0.7.

Cet article est une version étendue (et traduite) d'un travail publié à la conférence ICWE 2013 [4]. Des idées initiales ayant conduit à ce travail ont également été d'abord présentées comme un article d'un séminaire de doctorants en [5].

État de l'art

A.0.1. Crawl du Web

Le crawl du Web est un problème bien étudié qui comporte encore des défis. Un état de l'art du domaine de l'archivage du Web et du crawl pour l'archivage est disponible dans [Mas06].

Un crawler *dirigé* (*focused*) parcourt le Web en suivant un ensemble prédéfini de thèmes [CvdBD99]. Cette approche est une façon différente de la nôtre d'influencer le comportement d'un crawler, qui n'est pas basée sur la structure des applications Web comme nous le visons, mais sur le contenu des pages Web. Notre approche n'a pas le même but que le crawl dirigé : elle a plutôt vocation à mieux archiver des applications Web connues. Les deux stratégies d'amélioration d'un crawler conventionnel sont donc complémentaires.

Le contenu des applications Web ou des *systèmes de gestion de contenu* est organisé en fonction d'un modèle de rendu (les composants du modèle incluent par exemple les barres latérales d'une page Web, la barre de navigation, un en-tête ou pied de page, la zone de contenu principal, etc.). Parmi les nombreux travaux sur l'extraction de ce modèle, [GPT05] ont réalisé une analyse de l'étendue du contenu engendré par de tels modèles sur le Web. Ils ont découvert que 40 à 50 % du contenu du Web (en 2005) est basé sur des modèles, c'est-à-dire, fait partie d'une application Web. Leurs résultats suggéraient également que cette proportion augmente à un taux de 6 à 8 % par an. Ces travaux sont une forte indication du bénéfice à gérer le crawl d'applications Web de manière spécifique.

A.0.2. Crawl de forum

Même si le crawl basé sur l'application Web en toute généralité n'a pas encore été traité, il y a eu des efforts portant sur l'extraction de contenu depuis des forums Web [GLZZ06, CYL+08].

La première approche, appelée *Board Forum Crawling (BFC)* exploite la structure organisée des forums Web et simule le comportement de l'utilisateur dans le processus d'extraction. BFC résout le problème efficacement, mais est confronté à des limitations car il repose sur des règles simples et peut uniquement traiter des forums qui ont une structure organisée spécifique.

La deuxième approche [CYL+08], cependant, ne dépend pas de la structure spécifique d'un Web forum donné. Le système iRobot assiste le processus d'extraction en extrayant la carte de l'application Web crawlée. Cette carte est construite en crawlant de manière aléatoire quelques pages de l'application. Ce processus aide à identifier les régions répétitives riches qui sont par la suite regroupées suivant leurs modèles [ZSWW07]. Après la création de la carte, iRobot obtient la structure du forum Web sous la forme d'un graphe orienté consistant en des nœuds (les pages Web) et des arêtes orientées (les liens entre pages). Par la suite, une analyse de chemin est réaliséréalisée pour fournir un chemin de traversée optimale conduisant au processus d'extraction, cela afin d'éviter les pages doublons et invalides. Un travail ultérieur [YT12] a analysé iRobot et a identifié un certain nombre de limitations. [YT12] étend le système original par un ensemble de fonctionnalités : une meilleure découverte de l'arbre couvrant minimal [Edm67], une meilleure mesure du coût d'une arête dans le processus de crawl comme estimation de sa profondeur approchée dans le site et un raffinement de la détection des doublons.

iRobot [CYL+08, YT12] est probablement le travail le plus proche du nôtre. À l'inverse de ce système, l'AAH que nous proposons s'applique à n'importe quelle application Web, tant qu'elle est décrite dans notre base de connaissances. À la différence aussi par rapport à [CYL+08, YT12], où l'analyse de la structure d'un forum doit être faite indépendamment pour chaque site, l'AAH exploite le fait que plusieurs sites partagent le même système de gestion de contenu. Notre système extrait également de l'information structurée et sémantique des pages Web, tandis que iRobot produit des fichiers HTML bruts et laisse l'extraction à des travaux futurs. Nous donnons finalement en section A.0.7 une comparaison des performances de iRobot par rapport à AAH pour mettre en valeur la meilleure efficacité de notre approche. D'un autre côté, iRobot a pour but d'être une manière pleinement automatique de crawler un forum Web, tandis que l'AAH s'appuie sur une base de connaissances (construite manuellement mais maintenue automatiquement) d'applications Web ou de CMS.

A.0.3. Détection d'applications Web

Comme nous l'expliquerons, notre approche repose sur un mécanisme générique de détection du type d'applications Web couramment crawlé. Il y a à nouveau eu de la recherche sur ce sujet dans le cas particulier des blogs ou forums. En particulier, [KFJ06] utilise des *support vector machines (SVM)* pour détecter si une page donnée est une page de blog. Les SVM [BGV92, OFG97] sont largement utilisées pour les tâches de classification de texte. Dans [KFJ06], les SVM sont entraînées en utilisant des vecteurs de caractéristiques formés du multi-ensemble des mots du contenu ou des n-grammes, et de quelques autres caractéristiques spécifiques à la détection de blog, tels que le multi-ensemble des URL liées et des ancres. L'entropie relative est utilisée pour la sélection de caractéristiques.

Plus généralement, quelques travaux [ACD+03, LL07, LL06] visent à identifier une catégorie générale (p. ex., blog, académique, personnel) pour un site Web donné, en utilisant des systèmes de catégorisation basés sur des caractéristiques structurelles des pages Web qui visent à détecter les *fonctionnalités* de ces sites Web. Ce n'est pas directement applicable à notre cadre, tout d'abord parce que la catégorisation est très grossière, et ensuite parce que ces techniques fonctionnent au niveau du site Web (p. ex., en s'appuyant sur la page principale) et non au niveau des pages individuelles.

A.0.4. Adaptation d'extracteur

L'adaptation d'extracteur, le problème d'adapter un extracteur d'informations Web à des changements (mineurs) dans la structure des pages ou sites Web considérés a reçu une certaine attention dans la communauté académique. Un premier travail est celui de [Kus99] qui a proposé une approche d'analyse des pages Web et de l'information déjà extraite, afin de détecter des changements de structure. Une méthode de « vérification d'extracteur » est introduite, cette méthode vérifiant si un extracteur arrête d'extraire des données ; si oui, un superviseur humain est informé afin d'entraîner à nouveau l'extracteur. [Chi01] a introduit des règles grammaticales et logiques pour automatiser la maintenance d'extracteurs, en supposant qu'il n'y a que de petits changements dans la structure des pages Web. [MHL03b] ont suggéré une maintenance d'extracteurs guidé par le schéma, appelée SG-WRAM, qui repose sur l'observation que même quand la structure change dans des pages Web, certaines caractéristiques sont conservées, comme les motifs syntaxiques,

les hyperliens, les annotations, et l'information extraite elle-même. [LMK03] ont développé un système d'apprentissage pour réparer des extracteurs vis-à-vis de petits changements de balisage. Le système qu'ils ont proposé vérifie d'abord l'extraction des pages Web, et si celle-ci échoue, relance l'apprentissage d'extracteur. [RPAH07] collectent des résultats de requête valides durant l'utilisation de l'extracteur ; quand des changements structurels ont lieu dans des sources Web, ils utilisent ces résultats comme entrées pour générer un nouvel ensemble d'entraînement étiqueté qui peut être à nouveau utilisé pour l'apprentissage d'extracteur.

Notre technique d'adaptation de modèle est inspirée par les travaux précédemment cités : nous vérifions si notre extracteur échoue, et si c'est le cas, nous essayons de le corriger en faisant l'hypothèse de changements mineurs sur le site, et parfois en utilisant le contenu déjà crawlé. Une différence principale avec les travaux existants est que notre approche est également applicable à des sites Web complètement nouveau, qui n'ont jamais été crawlés, mais qui partagent juste le même système de gestion de contenu et un modèle similaire.

A.0.5. Extraction de données depuis des blogs, forums, etc.

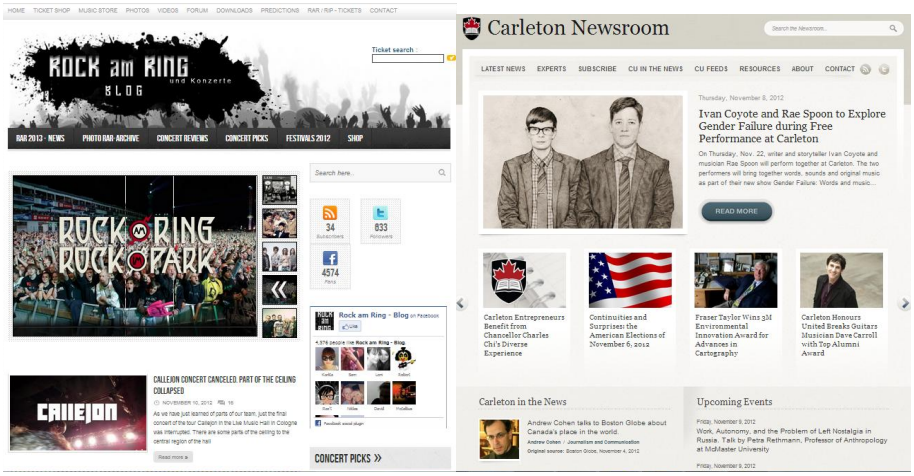
Un certain nombre de travaux [XYG+11, FB10, LN01, AF08] ont pour but l'extraction automatique depuis des pages Web engendrées par un CMS, en recherchant des structures répétées et, habituellement, en utilisant des techniques alignement ou d'appariement d'arbres. Ceci est hors de la portée de notre approche, où nous supposons que nous disposons d'une base pré-existante d'applications Web. [GMM+11] ont introduit le système d'apprentissage d'extracteur Vertex. Vertex extrait de l'information structurée depuis des pages Web basées sur un modèle. Le système classifie les pages en utilisant des vecteurs de n-grammes, apprend des règles d'extraction, détecter des changements du site et réapprend les règles qui ne fonctionnent plus. Vertex détecte les changements du site en surveillant quelques pages d'exemple du site. N'importe quel changement structurel peut résulter en des changements dans les vecteurs de n-grammes et peut rendre les règles apprises inapplicables. Dans notre système, nous ne surveillons pas des pages Web sélectionnées mais adaptons dynamiquement le comportement aux pages au fur et à mesure que nous les crawlons. Notre crawler s'adapte également à différentes versions d'un système de gestion de contenu, trouvé sur différents sites Web, plutôt que simplement à un type de page Web spécifique.

Préliminaires

Cette section introduit de la terminologie que nous utiliserons tout au long de l'article.

Une *application Web* est n'importe quelle application ou site qui utilise les standards du Web tels que HTML et HTTP pour publier de l'information sur le Web via un modèle donné, d'une manière accessible aux navigateurs Web. Des exemples sont les forums Web, les sites de réseaux sociaux, les services de géolocalisation, etc.

Un *type d'application Web* est le système de gestion de contenu ou la pile de technologies côté serveur (p. ex., vBulletin, WordPress, le CMS propriétaire de Flickr) qui fait tourner cette application Web et fournit une interaction avec elle. Plusieurs applications Web différentes peuvent partager le même type d'application Web (tous les forums vBulletin utilisent vBulletin), mais certains types d'application Web peuvent être spécifiques à une application Web donnée (p. ex., le CMS de Twitter est spécifique à ce site). Il peut y



(a) Rock am Ring blog (b) Carleton Newsroom

(c) Niveau intermédiaire (d) Niveau terminal

Figure A.1. – Exemples de pages Web d'applications Web

avoir des différences significatives dans l'apparence des applications Web du même type ; comparer par exemple les figures A.1a et A.1b, deux sites utilisant WordPress.

Le contenu présenté par les applications Web est habituellement stocké dans les bases de données et des modèles prédéfinis sont utilisés pour engendrer des pages Web riches en données. Pour le crawl intelligent, notre AAH doit savoir non seulement distinguer les types d'applications Web, mais aussi les différents types de pages Web qui peuvent être produites par un type d'application Web donné. Par exemple, un logiciel de forum Web peut engendrer des pages de contenu de diverses sortes, comme *liste de forums*, *liste de fils*, *liste de messages*, *message individuel*, *profil utilisateur*. Nous appelons une telle sorte de modèle spécifique le *niveau* de l'application Web.

Nous utilisons un sous-ensemble simple du langage XPath [W3C99] pour décirre des *motifs* dans le DOM des pages Web qui servent soit à identifier le type ou niveau d'application Web, soit à déterminer les *actions* de navigation ou d'extraction à appliquer à cette page Web. Une grammaire pour le sous-ensemble que nous considérons est donnée en figure A.2. Essentiellement, nous autorisons uniquement les axes de navigation vers le bas et des prédicats très simples qui comparent des chaînes de caractères. La sémantique de ces expressions est la sémantique standard [W3C99]. Sauf si précisé différemment, un « // » est implicite : une expression peut s'appliquer à n'importe quel nœud quel que soit sa profondeur dans l'arbre DOM. Dans ce qui suit, une *expression XPath* désigne toujours une expression de ce sous-langage.

```
⟨expr⟩       ::= ⟨étape⟩ | ⟨étape⟩ "/" ⟨expr⟩
                 ⟨étape⟩ "//" ⟨expr⟩
⟨étape⟩      ::= ⟨testNœud⟩ | ⟨étape⟩ "[" ⟨prédicat⟩ "]"
⟨testNœud⟩   ::= élément | "@" élément | "*" | "@*" | "text()"
⟨prédicat⟩   ::= "contains(" ⟨valeur⟩ "," chaîne ")" |
                 ⟨valeur⟩ "=" chaîne | entier | "last()"
⟨valeur⟩     ::= élément | "@" élément
```

Figure A.2. – Syntaxe BNF du fragment XPath utilisé. Les lexèmes suivants sont utilisés : élément est un identifiant XML valide (ou NMTOKEN [W3C08]) ; chaîne est une chaîne de caractères XPath entourée par des guillemets simples ou doubles [W3C99] ; entier est un entier positif.

Un *motif de détection* est une règle pour détecter les types et niveaux d'applications Web, basée sur le contenu des pages Web, des méta-données HTTP, des composants de l'URL. Il est implémenté comme une expression XPath sur un document virtuel qui contient la page Web HTML ainsi que les autres méta-données HTTP.

Une *action de crawl* est une expression XPath sur un document HTML qui indique quelle action accomplir sur page Web donnée. Les actions de crawl peuvent être de deux sortes : les *actions de navigation* pointent vers des URL à ajouter dans la file de crawl ; les *actions d'extraction* pointent vers des objets sémantiques individuels à extraire de la page Web (p. ex., estampille temporelle, message de blog, commentaire). Un exemple d'action de navigation pour suivre certains types de liens est :
`div[contains(@class,'post')]//h2[@class='post-message']//a/@href`.

L'AAH distringue deux degrés principaux de niveaux d'applications Web : les pages *intermédiaires*, comme les listes de forums, les listes de fils de discussion, ne peuvent être associées qu'à des actions de navigation ; les *pages terminales*, comme la liste des messages d'un fil de discussion, peuvent être associées à la fois à des actions de navigation et d'extraction. L'idée est que le crawleur naviguera parmi les pages intermédiaires jusqu'à trouver une page terminale, et seul le contenu de cette page terminale sera extraite ; la page terminale peut également être naviguée, p. ex., quand il y a un découpage d'un fil de discussion par pages numérotées. Pour illustrer, les figure A.1c et A.1d montrent, respectivement, des pages intermédiaires et terminales dans une application de forum Web.

Étant donnée une expression XPath e, une *expression affaiblie* pour e est une expression où une ou plusieurs des transformations suivantes ont été accomplies :

- un prédicat a été supprimée ;
- un lexème *élément* a été remplacé par un autre lexème *élément*.

Une *expression affaiblie meilleur-cas* pour e est une expression où au plus l'une des transformations a été effectuée pour chaque étape de e. Une *expression affaiblie pire-cas* pour e est une expression où un nombre arbitraire de transformations ont été effectuées pour chaque étape de e.

Pour illustrer, posons $e =$ `div[contains(@class,'post')]//h2[@class='post-message']`. Des exemples d'expressions affaiblies meilleurs-cas sont `div[contains(@class,'post')]//h2` ou bien `div[contains(@class,'post')]//h2[@id='post-content']` ; en revanche, on remarque que `div[contains(@class,'post')]//div[@id='post-content']` est un exemple d'expression affaiblie pire-cas.

Base de Connaissances

L'AAH est assisté d'une *base de connaissances* des types d'application Web qui décrit comment crawler un site Web intelligemment. Ces connaissances spécifient comment détecter des applications Web spécifiques et quelles actions de crawl doivent être exécutées. Les types sont organisés de manière hiérarchique, des catégorisations générales aux instances spécifiques (sites Web) de cette application Web. Par exemple, les sites Web de médias sociaux peuvent être classés en blogs, forums Web, microblogs, réseaux de partage de vidéos, etc. Ensuite, nous pouvons décomposer encore ces types d'applications Web sur la base des systèmes de gestion de contenu sur lequel ils sont basés. Ainsi, WordPress, Movable Type, sont des exemples de système de gestion de contenu de blog, tandis que phpBB et vBulletin sont des exemples de systèmes de gestion de contenu de forum Web. Cette base de connaissances décrit également les différents niveaux sous un type d'application Web ; en fonction de ceux-ci, nous pouvons définir différentes actions de crawl qui doivent être exécutées pour chaque niveau.

La base de connaissances est spécifiée dans un langage déclaratif, afin d'être aisément partagée et mise à jour, si possible maintenue par des non-programmeurs et aussi peut-être automatiquement apprise à partir d'exemples. Le W3C a standardisé un langage de description d'application Web (*Web Application Description Language* ou *WADL*) [W3C09] qui permet de décrire les ressources des applications basées sur HTTP dans un format interprétable par un programme. WADL est utilisé pour décrire l'ensemble des ressources, leurs relations les unes avec les autres, les méthodes qui peuvent être appliquées sur chaque ressource, les formats de représentation de ressources. WADL peut être un candidat de composant et de format d'export pour notre base de connaissances, mais ne remplit pas tous nos besoins : en particulier, il n'y a aucune place pour la description des motifs de reconnaissance d'applications Web. Par conséquent, notre base de connaissances est décrite dans un format XML approprié, de manière bien adaptée à la structure arborescente de la hiérarchie d'applications Web et de niveaux de pages.

Type et niveau d'application Web

Pour chaque type d'application Web et pour chaque niveau, la base de connaissances contient un ensemble de motifs de détection qui permet de reconnaître si une page donnée est de ce type ou de ce niveau.

Prenons l'exemple du système de gestion de contenu de forum Web vBulletin, qui peut ainsi être identifié en cherchant une référence à un script JavaScript `vbulletin_global.js` avec le motif de détection :

```
script [contains(@src,'vbulletin_global.js')]
```

Les pages du type « liste de forums » sont identifiées* quand elles correspondent au motif `a[@class="forum"]/@href`.

Actions de crawl

De manière similaire, pour chaque type et niveau d'application Web, un ensemble d'actions de navigation et d'extraction (dans ce dernier cas, seulement dans le cas des niveaux

*L'exemple est simplifié pour les besoins de la présentation ; en réalité il faut gérer les différents rendus que vBulletin peut produire.

terminaux) est fourni. Les actions de navigation pointent vers des liens (typiquement dans un attribut `a/@href`) à suivre ; les extractions d'action, également associés à des types sémantiques, pointent vers du contenu à extraire.

Application-aware Helper (AAH)

Notre position principale dans cet article est que des techniques de crawl différentes doivent être appliquées à différents types d'applications Web. Cela veut dire des stratégies de crawl différentes pour différentes formes de sites Web sociaux (blogs, wikis, réseaux sociaux, marque-pages sociaux, microblogs, partage de musique, forums Web, partage de photos, partage de vidéos, etc.), pour des systèmes de gestion de contenu spécifiques (p. ex., WordPress, phpBB) et pour des sites spécifiques (p. ex., Twitter, Facebook). Les figures A.1a et A.1b sont des exemples d'applications Web qui sont basées sur le CMS WordPress. Notre approche sera de détecter le type d'application Web (type général, système de gestion de contenu ou site) en train d'être traité par le crawler et le type de pages Web à l'intérieur de cette application Web (p. ex., un profil utilisateur sur un réseau social) et de décider d'actions de crawl (suivre un lien, extraire du contenu structuré) en conséquence. Le crawler proposé est suffisamment intelligent pour crawler et stocker tous les commentaires liés à un message de blog en un endroit, même si ces commentaires sont distribués sur plusieurs pages Web.

L'AAH détecte l'application Web et le type de page Web avant de décider quelle stratégie de crawl est appropriée pour l'application Web donnée. Plus précisément, l'AAH fonctionne dans l'ordre suivant :

(1) il détecte le type d'application Web ;
(2) il détecte le niveau d'application Web ;
(3) il exécute les actions de crawl pertinentes : extraire le résultat des actions d'extraction et ajouter le résultat des actions de navigation à la file d'URL.

Détection du type et niveau d'application Web

L'AAH charge les motifs de détection de type d'application Web depuis la base de connaissances et les exécute sur l'application Web. Si le type est détecté, le système exécute tous les motifs de détection de niveau de l'application Web jusqu'à trouver un résultat. Le nombre de motifs de détection pour détecter un type d'application Web va croître avec l'ajout de connaissances sur de nouvelles applications Web. Pour optimiser cette détection, le système a besoin de maintenir un index de ces motifs.

Dans ce but, nous avons intégré le système YFilter [DAF+03] (un système de filtrage pour expressions XPath basé sur un automate non-déterministe) avec des petits changement pour nos besoins, pour l'indexation efficace des motifs de détection, afin de trouver rapidement les types et niveaux d'application Web pertinents. YFilter est développé comme partie d'un système de publication–souscription qui permet aux utilisateurs de soumettre un ensemble de requêtes qui doivent être exécutées sur des pages XML en flux. En compilant les requêtes dans un automate pour indexer tous les motifs fournis, le système est capable de trouver efficacement la liste de tous les utilisateurs ayant soumis une requête qui est valide sur le document courant. Dans notre version intégrée de YFilter, les motifs de détection (que ce soit pour le type ou le niveau d'application Web) sont soumis comme des requêtes ; quand un document satisfait à une requête, le système arrête de tester les autres requêtes sur ce

document (contrairement au comportement standard de YFilter), car nous n'avons pas besoin de plus d'une correspondance.

Adaptation au changement de modèle

Deux types de changements peuvent se produire dans une page Web : le contenu peut changer et la structure peut changer. Des changements dans la note d'un film, un nouveau commentaire sous un message de blog, la suppression d'un commentaire, etc., sont des exemples de changement de contenu Web. Ces types de changements sont faciles à identifier en comparant simplement la page Web courante avec une version récemment crawlée. Un exemple de changement de modèle ou de structure est un changement dans le modèle de présentation d'un site Web, qui est plus complexe à identifier et à adapter. Notre approche de l'adaptation apporte une solution à ce défi.

Les changements de structure par rapport à ce qui est dans la base de connaissances peut venir de versions différentes d'un système de gestion de contenu, ou de thèmes alternatifs proposés par ce CMS ou développés pour des applications Web spécifiques. Si le modèle d'une page Web n'est pas celui donné dans la base de connaissances des applications Web, les motifs de détection des applications Web et les actions de crawl peuvent échouer sur une page Web. L'AAH a pour but de déterminer quand un changement a eu lieu et d'adapter les motifs et actions en conséquence. Au cours de ce processus, il maintient également automatiquement la base de connaissances avec les nouveaux motifs et actions découverts.

Les motifs de détection pour déterminer le type d'une application Web sont la plupart du temps détectés en recherchant une référence à un fichier de script, une feuille de style, des méta-données HTTP (tels que des noms de cookies) ou même un contenu textuel (p. ex., un fragment textuel « Powered by »), qui sont en général robustes aux changements de structure. Nous allons faire l'hypothèse dans la suite que les motifs de détection de type d'application Web n'échouent jamais. Nous n'avons constaté aucun cas où cela se produisait dans nos expériences. En revanche, nous considérons que les motifs de détection du niveau de l'application Web et les actions de crawl peuvent devenir inapplicable après un changement de modèle.

Nous traitons de deux cas différents d'adaptation : tout d'abord, quand (une partie d')une application Web a déjà été crawlée avant le changement et qu'un nouveau crawl est fait après le changement (une situation courante dans les campagnes de crawl réelles) ; deuxièmement, quand une nouvelle application Web est crawlée, qu'elle correspond aux motifs de détection des types d'application Web, mais pour laquelle certaines actions sont inapplicables.

A.0.6. Crawl d'une application Web déjà crawlée

Nous considérons d'abord le cas où une partie d'une application Web a été crawlée avec succès en utilisant les motifs et les actions de la base de connaissances. Le modèle de cette application Web change ensuite (en raison d'une mise à jour du système de gestion de contenu, ou d'un changement de conception du site) et l'application est crawlée à nouveau. La base de notre technique d'adaptation est d'apprendre à nouveau les actions de crawl appropriées pour chaque objet qui peut être crawlé ; la base de connaissances est ensuite mise à jour en y ajoutant les actions nouvellement apprises (comme les actions existantes

peuvent toujours fonctionner sur une autre instance de ce type d'application Web, nous les gardons dans la base).

Comme nous décrivons plus tard en section A.0.7, les pages Web crawlées sont stockées, avec leurs objets Web et méta-données, sous la forme de triplets RDF dans une base RDF. Notre système détecte les changements de structure pour les applications Web déjà crawlées en recherchant le contenu (stocké dans la base RDF) dans les pages Web avec les actions utilisées durant le crawl précédent. Si le système ne parvient pas à extraire le contenu avec les mêmes actions, cela veut dire que la structure du site Web a changé.

Entrées: URL u, ensembles de motifs de détection d'applications Web D et d'actions de crawl A
si *déjàCrawlé*(u) **alors**
  si *aChangé*(u) **alors**
    actionsMarquées ← *détecteChangementsStructurels*(u, A);
    nouvellesActions ← *aligneActionsCrawl*(u, D, *actionsMarquées*);
    ajouteBaseConnaissance(*nouvellesActions*);

Algorithm A.1: Adaptation au changement de modèle (crawl d'une Web application déjà crawlée)

L'algorithme A.1 donne une vue de haut niveau du mécanisme d'adaptation à un changement de modèle dans ce cas. Le système traite chaque page Web pertinente d'une application Web. Toute page Web qui a déjà été crawlée mais ne peut maintenant être crawlée sera envoyée à l'algorithme A.1 pour adaptation de structure. Cet algorithme ne répare que les actions de crawl qui ont échoué pour des pages déjà crawlées ; cependant, comme les actions de crawl nouvellement crawlées sont ajoutées à la base de connaissances, le système sera capable de traiter des pages de la même application Web qui n'ont pas encore été crawlées. Il faut remarquer que dans tous les cas l'algorithme n'essaie pas d'apprendre à nouveau les actions de crawl ayant échoué pour chaque page Web, mais commence plutôt par vérifier si les actions de crawl déjà réparées pour le même niveau d'application Web fonctionnent toujours.

L'algorithme A.1 commence par vérifier si une URL donnée a déjà été crawlée en appelant la fonction booléenne *déjàCrawlé* qui vérifie simplement si l'URL existe dans la base RDF. Une page Web déjà crawlée sera ensuite analysée pour détecter des changements de structure avec la fonction booléenne *aChangé*.

Les changements de structure sont détecté en cherchant du contenu déjà crawlé (URL correspondant à des actions de navigation, objets Web, etc.) dans une page Web en utilisant les actions de crawl existantes et déjà apprises (le cas échéant) pour le niveau d'application Web. La fonction *aChangé* prend en compte le fait qu'échouer à extraire de l'information supprimée ne doit pas être considéré comme un changement de modèle. Par exemple, un objet Web tel qu'un commentaire dans un forum Web qui a été crawlé auparavant peut ne plus exister. Ce scénario peut se produire car un administrateur, voire l'auteur du commentaire lui même, a pu le supprimer du forum. Pour cette raison, le système vérifiera toujours l'existence d'un objet Web, avec toutes ses méta-données, dans la version courante d'une page Web ; si l'objet Web a disparu, il n'est pas pris en compte pour la détection de changements de modèle.

En présence de changements de structure, le système appelle la fonction *détecteChangementsStructurels* qui détecte les actions inapplicables et les marque comme « échouées ». La

fonction *détecteChangementsStructurels* retourne un ensemble d'actions de crawl marquées. Toutes les actions de crawl qui sont marquées comme échouées vont être adaptées aux changements de structure. La fonction *aligneActionsCrawl* apprend à nouveau les actions de crawl ayant échoué.

Si un motif de détection de niveau d'application Web échoue également, alors le système applique une approche similaire à celle décrite en section A.0.7 pour adapter le motif de détection de niveau.

A.0.7. Crawl d'une nouvelle application Web

Nous sommes maintenant dans le cas où nous crawlons une application Web complètement nouvelle dont le modèle a (légèrement) changé par rapport à celui contenu dans la base de connaissances. Nous supposons que les motifs de détection de type d'application ont fonctionné, mais que soit les motifs de détection de niveau, soit les actions de crawl ne fonctionnent pas sur cette application Web précise. Dans cette situation, nous ne pouvons nous appuyer sur du contenu précédemment crawlé.

Considérons d'abord le cas où le motif de détection de niveau d'application Web fonctionne.

Le niveau d'application Web est détecté

Rappelons qu'il y a deux types de niveaux d'application Web : intermédiaire et terminal. Nous supposons qu'aux niveaux intermédiaires, les actions de crawl (qui sont uniquement des actions de navigation) n'échouent pas – à ce niveau, les actions de navigation sont habituellement assez simples (elles sont typiquement de simples extensions des motifs de détection de niveau, p. ex., `//div[contains(@class,'post')]` pour le motif de détection et `//div[contains(@class,'post')]//a/@href` pour l'action de navigation). Dans nos expériences nous n'avons jamais eu besoin de les adapter. Nous laissons le cas où elles peuvent échouer à des travaux ultérieurs. En revanche, nous considérons qu'à la fois les actions de navigation et les actions d'extraction des pages terminales peuvent nécessiter une adaptation.

Entrées: URL u et ensemble d'actions de crawl A
si *non déjàCrawlé*(u) **alors**
  pour $a \in A$ **faire**
    si *extractionEchouée*(u, a) **alors**
      expressionsAffaiblies $\leftarrow$ *affaiblit*(a);
      pour *candidat* $\in$ *expressionsAffaiblies* **faire**
        si *non extractionEchouée*(u, *candidat*) **alors**
          ajouteBaseConnaissance(*candidat*);
          quitteBoucle ;

Algorithm A.2: Adaptation au changement de modèle (nouvelle application Web)

Les étapes principales de l'algorithme d'adaptation sont décrites dans l'algorithme A.2. Le système vérifie d'abord l'applicabilité des actions de crawl existantes et corrige ensuite celles ayant échoué. La fonction *affaiblit* crée deux ensembles d'expressions affaiblies (meilleur-cas et pire-cas). Pour chaque ensemble, les différentes variations des actions de crawl vont être

engendrées en affaiblissant les prédicats et noms d'éléments, et énumérées en fonction du nombre d'affaiblissement requis (les affaiblissements simples venant d'abord). Les noms d'élément sont remplacés avec les noms d'élément existant dans l'arbre DOM afin que l'expression DOM ait une correspondance. Quand un nom d'attribut est affaibli à l'intérieur d'un prédicat, l'AAH suggère uniquement les candidats qui sont susceptibles de rendre ce prédicat vrai ; à cette fin, l'AAH collecte d'abord tous les attributs possibles et leurs valeurs depuis la page.

Quand l'action de crawl a pour préfixe un motif de détection, nous ne touchons pas à ce préfix mais affaiblissons uniquement la deuxième partie. Par exemple, si une expression comme `div[contains(@class,'post')]//h2[@class='post-title']` échoue à extraire le titre d'un message et que `div[contains(@class,'post')]` est le motif de détection qui a été déclenché, nous essayons des affaiblissements de la deuxème partie de l'expression, p. ex., remplacer `@class` avec `@id`, `'post-title'` avec `'post-head'`, `h2` avec `div`, etc. Nous favorisons les relaxations qui utilisent des parties d'actions de crawl de la base de connaissances pour d'autres types d'applications Web de la même catégorie générale (p. ex., forum Web).

Une fois que le système a engendré tous les affaiblissements possibles, il essaie d'abord les motifs meilleur-cas et, s'ils ne fonctionnent pas, les motifs pire-cas. Plus généralement, les expressions sont ordonnées par le nombre d'affaiblissements requis. Toute expression qui réussit dans l'extraction sera testée sur quelques autres pages du même niveau d'application Web avant d'être ajoutée dans la base de connaissances pour crawl futur.

Le niveau d'application Web n'a pas été détecté

Si le système ne détecte pas le niveau d'application Web, la stratégie de crawl ne peut être lancée. Le système essaie d'abord d'adapter les motifs de détection avant de corriger les actions de crawl. L'idée est ici la même que dans la partie précédente : le système collecte tous les attributs, valeurs, noms d'éléments candidats depuis la base de connaissances du type d'application Web détecté (p. ex., WordPress) et crée ensuite toutes les combinaisons possibles d'expressions affaiblies, ordonnées par le nombre d'affaiblissements, et les teste une par une jusqu'à en trouver une qui fonctionne.

Pour illustrer, supposons que l'ensemble d'attributs et valeurs candidats sont : `@class='post'`, `@id=forum`, `@class='blog'` avec les ensembles candidats de noms d'éléments `article` , `div`, etc. L'ensemble des expressions affaiblies va être engendré en essayant chaque combinaison possible : `//article[contains(@class,'post')]`
`//article[contains(@id,'forum')]`
`//article[contains(@class,'blog')]`
et similairement pour les autres noms d'éléments.

Après la collection de l'ensemble des expressions relâchées, le système va essayer de détecter le niveau d'application Web en testant chaque expression relâchée et, si le système détecte finalement le niveau, il applique finalement la technique d'adaptation décrite précédemment. Si le système ne détecte pas le niveau d'application Web, l'adaptation échoue.

Système

L'application-aware helper est implanté en Java. Au démarrage, le système charge la base de connaissances et indexe les motifs de détection en utilisant une implantation de YFilter

[DAF+03] adaptée de celle disponible à `http://yfilter.cs.umass.edu/`. Une fois que le système reçoit une requête de crawl, il consulte d'abord l'index YFilter pour détecter le type et niveau d'application Web. Si le type d'application Web n'est pas détecté, l'AAH applique la stratégie d'adaptation pour trouver une correspondance affaiblie comme précédemment décrit. Si aucune correspondance n'est trouvé (c.-à-d., si l'application Web est inconnue), une extraction générique des liens est réalisée.

Quand l'application Web est détectée avec succès, l'AAH charge la stratégie de crawl correspondante de la base de connaissances et crawle l'application Web en fonction de celle-ci, éventuellement en utilisant la stratégie d'adaptation. Les pages Web crawlées sont stockées sous la forme de fichiers WARC [ISO] – le format standard de préservation pour l'archivage Web – tandis que le contenu structuré (les objets Web individuels avec leur méta-données sémantiques) est stocké dans une base RDF. Pour le passage à l'échelle et la disponibilité de l'archive, les fichiers WARC sont mis sur un cluser Hadoop* sous HDFS, et la base RDF que nous utilisons, H2RDF [PKTK12], est implantée au-dessus de HBase [The12]. La base de connaissance est potentiellement mise à jour avec de nouveaux motifs de détection ou des actions de crawl.

L'AAH est intégré au sein d'Heritrix [Sig05], le crawler libre† developpé par Internet Archive. Dans la chaîne de traitement du crawl, l'AAH remplace le module d'extraction de liens conventionnel. Les actions de crawl déterminées par l'AAH sont renvoyées dans la file des URL à crawler de Heritrix.

Le code de l'AAH (ainsi que la liste de tous les sites de notre jeu de données expérimental) est disponible sur `http://perso.telecom-paristech.fr/~faheem/aah.html` sous licence GPL 3.0.

Expériences

Nous présentons dans cette partie la performance expérimentale du système que nous proposons, en elle-même et par comparaison à un crawler traditionnel, GNU wget‡ (comme la portée du crawl est ici très simple – un crawl complet d'un ensemble de noms de domaines – wget est ici aussi bien que Heritrix).

Protocole expérimental

Pour évaluer la performance de notre système, nous avons crawlé 100 applications Web (avec un total de près de 3,3 millions de pages Web) de deux formes de sites Web sociaux (forum Web et blog), pour trois systèmes de gestion de contenu (vBulletin, phpBB et WordPress). Les applications Web de type WordPress (33 applications Web, 1,1 millions de pages Web), vBulletin (33 applications Web, 1,2 millions de pages Web) et phpBB (34 applications Web, 1 million de pages Web) ont été sélectionnées aléatoirement de trois sources différentes :

(1) `http://rankings.big-boards.com/`, une base de données de forums Web populaires.

(2) Un jeu de données sur le sujet de la crise financière en europe.

*`http://hadoop.apache.org/`

†`http://crawler.archive.org/`

‡`http://www.gnu.org/software/wget/`

(3) Un jeu de données lié au festival de musique *Rock am Ring* en Allemagne.
Les deuxième et troisième jeux de données ont été collectés dans le cadre du projet ARCOMEM [ARC13]. Dans ces jeux de données du monde réel correspondant à des tâches d'archivage spécifiques, 68% des URL initiales de type forum sont des sites que font tourner soit vBulletin soit phpBB, ce qui explique que nous nous intéressons à ces deux CMS. WordPress est également un CMS prévalent : le Web dans son ensemble a plus de 61 millions de sites Wordpress [Wor12] sur un total de blogs indexés par Technorati [The09] d'à peu près 133 millions. De plus, Wordpress a une part de marché de 48% des 100 blogs les plus visités [Roy12]. Chacune des 100 applications Web a été crawlée en utilisant wget et l'AAH. Les deux crawlers sont configurés pour récupérer uniquement les documents HTML, sans tenir compte des scripts, des feuilles de style, des fichiers média, etc.

La base de connaissances est peuplée de motifs de détection et d'actions de crawl pour une version spécifique des trois systèmes de gestion de contenu considérés (les autres versions seront traitées par le module d'adaptation). Ajouter un nouveau type d'application à la base de connaissances prend à un ingénieur de crawl de l'ordre de 30 minutes.

Métriques de performance

La performance de l'AAH sera principalement mesurée en évaluant le nombre de requêtes HTTP réalisées par les deux systèmes par rapport au volume de contenu *utile* récupéré. Notons que comme wget récupère aveuglément l'intégralité du site Web, toutes les URL crawlées par l'AAH seront aussi crawlées par wget.

Évaluer le nombre de requêtes HTTP est facile, il suffit de compter les requêtes réalisées par les deux crawlers. La couverture du contenu utile est une notion plus subjective ; comme il est impossible de demander une évaluation utilisateur étant donnés les volumes que nous considérons, nous utilisons les alternatives suivantes :

(1) Compter le volume de contenu textuel récupéré. À cette fin, nous comparons la proportion des bigrammes (séquences de deux mots consécutifs) dans le résultat du crawl des deux systèmes, pour chaque application Web.

(2) Compter le nombre de liens externes (c.-à-d., d'hyperliens vers un autre domaine) trouvés dans les deux crawls. L'idée est que les liens externes sont une partie importante du contenu du site Web.

Passage à l'échelle de la détection du type

Nous discutons d'abord brièvement de l'utilisation de YFilter pour améliorer l'indexation des motifs de détection. Dans la figure A.3 nous montrons le temps requis pour déterminer le type d'application Web dans une base de connaissances engendrée de manière synthétique quand le nombre de types d'application croît jusque 5 000, avec ou sans l'indexation par YFilter. Le système prend un temps linéaire dans le nombre de motifs de détection quand l'indexation est désactivée, prenant jusqu'à plusieurs dizaines de seconde. En revanche, le temps de détection est quasi-constant quand YFilter est activé.

Efficacité du crawl en nombre de requêtes

Nous comparons le nombre de requêtes HTTP requises par les deux crawlers pour crawler chaque ensemble d'applications Web de même type en figure A.4. Noter que l'application-aware helper fait beaucoup moins de requêtes (en moyenne 7 fois moins) qu'un crawl

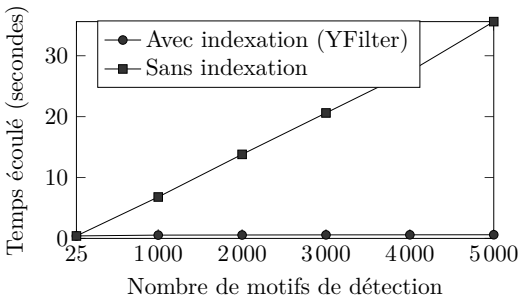


Figure A.3. – Performance du module de détection

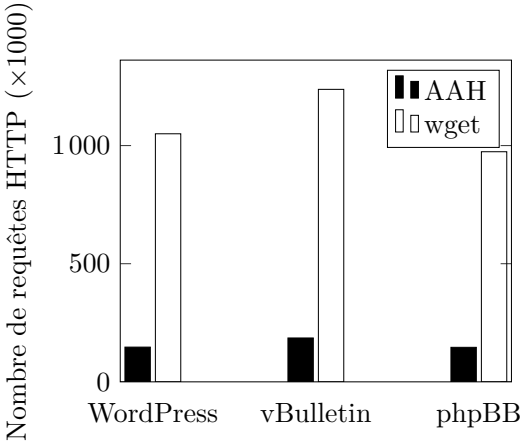


Figure A.4. – Nombre total de requêtes HTTP utilisées pour crawler le jeu de données

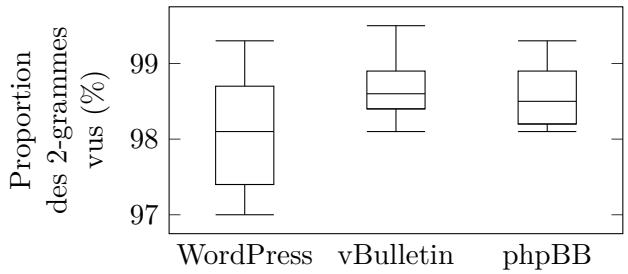


Figure A.5. – Boîte à moustaches des bigrammes vus pour les trois CMS considérés. Nous montrons dans chaque cas les valeurs minimum et maximum (moustaches), le premier et troisième quartiles (boîte) et la médiane (ligne horizontale).

aveugle classique. En effet, pour les sites Web de type blog, un crawler classique réalise un nombre abondant de requêtes HTTP redondantes pour le même contenu Web, accédant à un message par tag, auteur, année, ordre chronologique, etc. Dans un forum Web, de nombreuses requêtes se trouvent pointer vers des fonctionnalités de recherche, des zones d'édition, une vue pour impression d'un message, des zones protégées par authentification, etc.

Qualité du crawl

Les résultats de crawl, en terme de couverture du contenu utile, sont résumés dans la figure A.5 et dans le tableau A.1. La figure A.5 présente la distribution de la proportion de bigrammes crawlés par l'AAH en fonction de ceux du crawl complet. Non seulement les chiffres sont en général très haut (pour les trois types, la médiane est supérieure à 98 %) mais les résultats sont également très stables, avec une très faible variance ; le pire score de couverture sur notre jeu de données complet est supérieur à 97 % (habituellement, des scores plus faibles viennent de petits sites Web où la proportion de texte faisant partie du modèle comme des noms de menu ou les conditions d'utilisation du site restent non

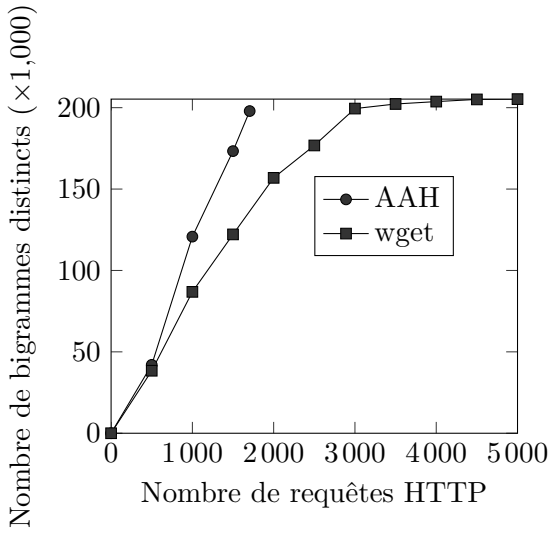


Figure A.6. – Crawl de `http://www.rockamring-blog.de/`

Table A.1. – Couverture des liens externes dans le jeu de données crawlé par l'AAH

CMS	**Liens externes**	**Liens externes (sans modèle)**
WordPress	92,7 %	99,8 %
vBulletin	90,5 %	99,5 %
phpBB	92,1 %	99,6 %

négligeables). Cela indique que les résultats sont significatifs.

La proportion de liens externes couverte par l'AAH est donnée dans le tableau A.1. L'application-aware helper a ignoré près de 10 % des liens externes car chaque page peut utiliser des composants, comme ceux de Facebook, d'Amazon, etc., avec des URL variant d'une page à l'autre ; une fois ces composants faisant réellement partie du modèle exclus avec un ensemble de motifs fixé, nous voyons que plus de 99,5 % des liens externes sont présents dans le contenu crawlé par l'AAH.

Pour mieux comprendre comment un crawl de l'AAH se déroule, nous montrons en figure A.6 le nombre de bigrammes distincts découverts par l'AAH et wget durant un crawl (dans ce cas particulier, un blog WordPress), alors que le nombre de requêtes augmente. Nous voyons que l'AAH cible directement la partie intéressante d'une application Web, avec un nombre de bigrammes nouvellement découverts qui croît linéairement avec le nombre de requêtes effectuées, pour atteindre un niveau final de 98 % de couverture de bigrammes après 1 705 requêtes. D'autre part, wget découvre du contenu nouveau à un rythme plus lent et, en particulier, passe les dernières 2/5 de ses requêtes à découvrir très peu de nouveau bigrammes.

Comparaison à iRobot

Le système iRobot [CYL+08] dont nous avons parlé en section A n'est pas disponible à des fins de test pour des raisons de propriété intellectuelle. Les expériences de [CYL+08] sont assez limitée en porté, avec seulement 50 000 pages Web considérées, sur 10 sites de forum Web différents (à comparer à notre évaluation, sur 3,3 millions de pages Web, et 100 sites de forum et blog différents). Pour comparer AAH et iRobot, nous avons crawlé l'un des mêmes forums Web utilisé dans [CYL+08] : `http://forums.asp.net/` (plus de 50,000 pages Web). La couverture du contenu par l'AAH (en terme de bigrammes et de liens externes, modèle exclu) est de plus de 99 % ; iRobot a une couverture des *pages de valeur* (estimées par un humain) de 93 % sur la même application Web. Le nombre de requêtes HTTP pour iRobot est dite être dans [CYL+08] 1,73 fois moins qu'un crawler Web classique ; sur l'application Web `http://forums.asp.net/`, l'AAH fait 10 fois moins de requêtes que wget.

Adaptation pour un crawl d'une application déjà crawlée

Pour tester notre technique d'adaptation dans le cas d'un crawl d'une application Web déjà crawlée dans un cadre réaliste (sans avoir à attendre que les sites Web changent effectivement), nous avons considéré des sites ayant à la fois une version « ordinateur de bureau » et une version mobile, avec du contenu HTML différent. Ces sites utilisent deux modèles différents pour présenter ce qui est essentiellement le même contenu. Nous avons simulé le double crawl en crawlant d'abord le premier site avec un en-tête HTTP `User-Agent:` indiquant un robot Web classique (la version ordinateur de bureau est alors servi) puis en crawlant la version mobile avec un `User-Agent:` de navigateur mobile.

Notre système a non seulement pu détecter les changements de structure d'une version à l'autre mais aussi, en utilisant le contenu déjà crawlé, a pu corriger les actions de crawl échouant. Le tableau A.2 présente une application Web exemple qui a à la fois une version ordinateur de bureau et mobile, avec une liste partielle des changements structurels dans les motifs des deux versions. Notre système a été capable de corriger automatiquement ces

Table A.2. – Exemples de changements de motifs structurels : version ordinateur de bureau et mobile de `http://www.androidpolice.com/`

Version ordinateur de bureau	Version mobile
div[@class='post_title']/**h3**/a	div[@class='post_title']/**h2**/a
div[@class='post_info']	div[@class='post_author']
div[@class='post_content']	div[@class='content']

changements de structure en terme à la fois de navigation et d'extraction, atteignant une parfaite concordance entre les contenus extraits dans les deux crawls.

Adaptation à une nouvelle application Web

Ainsi qu'indiqué précédemment, nous avons expérimenté notre système sur 100 applications Web, en partant d'une base de connaissances contenant des informations sur une version spécifique des trois systèmes de gestion de contenu considéré. Parmi les 100 applications, 77 n'ont pas requis d'adaptation, ce qui illustre le fait que de nombreuses applications Web partagent les mêmes modèles. Les 23 applications restantes avaient une structure qui différait de celle indiquée par les actions de crawl de la base de connaissances ; l'AAH a appliqué la technique d'adaptation avec succès sur ces 23 cas. La plupart des adaptations consistaient en affaiblissant l'attribut @class ou @id plutôt qu'en remplaçant le nom d'un élément. Quand il y avait un changement de nom d'élément, c'était la plupart du temps de span vers div vers article et vice-versa, ce qui est assez simple à adapter. Il n'y a eu aucun cas dans le jeu de données où plus d'un affaiblissement pour chaque étape d'une expression XPath fût requis ; en d'autres termes, seulement des expressions meilleur-cas furent utilisées. Dans deux cas, l'AAH ne fut pas capable d'adapter l'ensemble des actions d'extraction, mais les actions de navigation fonctionnaient toujours ou ont pu être adaptées, ce qui veut dire que le site Web a pu être crawlé, mais qu'une partie du contenu structuré était manquant de l'extraction.

Comme exemple d'affaiblissement, dans l'application `http://talesfromanopenbook.wordpress.com/`, l'extraction du titre de message par l'action div[@class='post']/h2[@class='post-title'] échoua mais l'affaiblissement div[@class='post']/h2[@class='storytitle'] fut trouvé.

Conclusions

Dans l'archivage du Web, les ressources rares sont la bande passante, le temps de crawl et l'espace de strockage, plutôt que le temps de calcul [Mas06]. Nous avons montré que le crawl d'applications Web basé sur les applications peut aider à réduire la bande passante, le temps et l'espace de stockage (en nécessitant moins de requêtes HTTP pour crawler une application Web entière, évitant les doublons) tout en utilisant des ressources de calcul limitées (pour appliquer les actions de crawl sur les pages Web). Le crawl basé sur les applications aide également à ajouter de la sémantique aux archives Web, augmentant leur valeur pour les utilisateurs.

Notre travail peut être étendu de plusieurs manières, que nous allons explorer dans des travaux ultérieurs. Tout d'abord, nous pouvons enrichir le langage de motifs que nous

utilisons afin d'autoriser des règles de détection et d'extraction complexe, vers un support complet de XPath ou même d'un langage de navigation Web plus puissant comme OXPath [FGG+11], autorisant le crawl d'applications Web complexes faisant usage d'AJAX ou de formulaires Web. Il y a un compromis, cependant, entre le pouvoir expressif du langage et la simplicité de l'adaptation aux changements de structure. Deuxièmement, nous voulons allers vers une base de connaissances d'applications Web construite automatiquement, soit en demandant à un être humain d'annoter les parties d'une application Web à extraire ou crawler [KSG+12b], avec des techniques d'apprentissage semi-supervisées, soit même en découvrant de manière complètement non supervisée de nouveaux types d'applications Web (c.-à-d., d'applications Web) par une comparaison de la structure de différents site Web, pour déterminer le chemin optimal de crawl par échantillonnage, dans l'esprit de iRobot [CYL+08].

Self References

Journal Article

[1] Muhammad Faheem, Pierre Senellart, Vassilis Plachouras, Florent Carpentier, Julien Masanès, Thomas Risse, Patrick Siehndel, Yannis Stavrakas. ARCOMEM Crawling Architecture. *Future Internet*, 6(3), pp. 518-541, 2014.

[2] Muhammad Faheem, Pierre Senellart. Crawl intelligent et adaptatif d'applications Web pour l'archivage du Web. *Ingénierie des Systèmes d'Information*, 19(4), pp. 61-86, 2014.

Conference Articles

International Publications

[3] Muhammad Faheem, Pierre Senellart. Demonstrating Intelligent Crawling and Archiving of Web Applications. In *Proc. CIKM*, San Francisco, USA, October 2013.

[4] Muhammad Faheem, Pierre Senellart. Intelligent and Adaptive Crawling of Web Applications for Web Archiving. In *Proc. ICWE*, Aalborg, Denmark, July 2013.

[5] Muhammad Faheem. Intelligent crawling of Web applications for Web archiving. In *Proc. PhD Symposium of WWW*, Lyon, France, April 2012.

National Publications

[6] Muhammad Faheem, Pierre Senellart. Une dèmonstration d'un crawler intelligent pour les applications Web. In *Proc. BDA*, Nantes, France, October 2013. Demonstration.Conference without formal proceedings.

[7] Muhammad Faheem, Pierre Senellart. Collecte intelligente et adaptative d'applications Web pour l'archivage du Web. In *Proc. BDA*, Nantes, France, October 2013. Conference without formal proceedings.

Preprints

[8] Muhammad Faheem, Tim Furche, Giovanni Grasso, Christian Schallhart. OWET: A Comprehensive Toolkit for Wrapper Induction and Scalable Data Extraction. **Submitted for publication**.

[9] Muhammad Faheem, Pierre Senellart. Adaptive Crawling Driven by Structure-Based Link Classification. **Submitted for publication**.

External References

[AB09] Dirk Ahlers and Susanne Boll. Adaptive geospatially focused crawling. In *CIKM*, 2009.

[ACD+03] Einat Amitay, David Carmel, Adam Darlow, Ronny Lempel, and Aya Soffer. The connectivity sonar: Detecting site functionality by structural patterns. In *HT*, 2003.

[AF08] Hassan Artail and Kassem Fawaz. A fast HTML Web page change detection approach based on hashing and reducing the number of similarity computations. *Data Knowl. Eng.*, 66(2):326–337, August 2008.

[AGM03] Arvind Arasu and Hector Garcia-Molina. Extracting structured data from web pages. In *SIGMOD*, 2003.

[AH08] Jesse Alpert and Nissan Hajaj. We knew the web was big... `http://googleblog.blogspot.co.uk/2008/07/we-knew-web-was-big.html`, 2008.

[AKP07] G. Almpanidis, C. Kotropoulos, and I. Pitas. Combining text and link analysis for focused crawling-an application for vertical search engines. *Information Systems*, 32(6):886–908, September 2007.

[AM98] Gustavo O. Arocena and Alberto O. Mendelzon. WebOQL: Restructuring documents, databases and Webs. In *Proceedings of the Fourteenth International Conference on Data Engineering*, 1998.

[APM00] Allan Arvidson, Krister Persson, and Johan Mannerheim. The Kulturarw3 Project - The Royal Swedish Web Archiw3e - An example of "complete" collection of web pages. In *Proceedings of the 66th IFLA Council and General Conference*, August 2000.

[ARC13] ARCOMEM Project. `http://www.arcomem.eu/`, 2011–2013.

[BCM08] Claudio Bertoli, Valter Crescenzi, and Paolo Merialdo. Crawling programs for wrapper-based applications. In *IRI*, 2008.

[BCSV04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, July 2004.

[BDM11] Lorenzo Blanco, Nilesh N. Dalvi, and Ashwin Machanavajjhala. Highly efficient algorithms for structural clustering of large websites. In *WWW*, 2011.

[Ber00] Michael K. Bergman. The deep web: Surfacing hidden value, 2000.

[BF04] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. In *Proceedings of the 19th Brazilian Symposium on Databases*, 2004.

[BF05] Luciano Barbosa and Juliana Freire. Searching for hidden-web databases. In *WebDB*, 2005.

[BF07] Luciano Barbosa and Juliana Freire. An adaptive crawler for locating hidden-Web entry points. In *WWW*, 2007.

[BGV92] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, 1992.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, 1998.

[Bru13] Geoff Brumfiel. The first web page, amazingly, is lost. `http://www.npr.org/2013/05/22/185788651/the-first-web-page-amazingly-is-lost`, 2013.

[BT06] Steve Bailey and Dave Thompson. UKWAC: Building the UK's first public web archive. *D-Lib Magazine*, 12(1), 2006.

[Bur97] Mike Burner. Crawling Towards Eternity: Building an Archive of the World Wide Web. *Web Techniques Magazine*, 2(5), 1997.

[BYCMR05] Ricardo Baeza-Yates, Carlos Castillo, Mauricio Marin, and Andrea Rodriguez. Crawling a country: Better strategies than breadth-first for web page ordering. In *WWW Special Interest Tracks and Posters*, 2005.

[BYKS07] Ziv Bar-Yossef, Idit Keidar, and Uri Schonfeld. Do not crawl in the dust: Different urls with similar text. In *WWW*, 2007.

[CGM00] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB*, 2000.

[Chi01] Boris Chidlovskii. Automatic repairing of Web wrappers. In *WIDM*, 2001.

[CKGS06] Chia-Hui Chang, Mohammed Kayed, Moheb Ramzy Girgis, and Khaled F. Shaalan. A survey of web information extraction systems. *IEEE Trans. on Knowl. and Data Eng.*, 18(10):1411–1428, 2006.

[CM07] Kumar Chellapilla and Alexey Maykov. A taxonomy of javascript redirection spam. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, 2007.

[CMB+11] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion,

Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. GATE, 2011.

[CMM01] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[CMM02] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: automatic data extraction from data-intensive web sites. In *SIGMOD*, 2002.

[CMM03] Valter Crescenzi, Paolo Merialdo, and Paolo Missier. Fine-grain Web site structure discovery. In *WIDM*, 2003.

[CMM05] Valter Crescenzi, Paolo Merialdo, and Paolo Missier. Clustering Web pages based on their structure. *Data Knowl. Eng.*, 54(3):279–299, 2005.

[Col08] Stephen Coleman. Blogs and the new politics of listening. *The Political Quarterly*, 76(2):272–280, 2008.

[CvdBD99] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11–16):1623–1640, May 1999.

[CWW09] Warwick Cathro, Colin Webb, and Julie Whiting. Archiving the web: The pandora archive at the national library australia. *National Library of Australia Staff Papers*, 2009.

[CYL+08] Rui Cai, Jiang-Ming Yang, Wei Lai, Yida Wang, and Lei Zhang. iRobot: An intelligent crawler for Web forums. In *WWW*, 2008.

[DAF+03] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, December 2003.

[DBP94] P. M. E. De Bra and R. D. J. Post. Information retrieval in the world-wide web: Making client-based searching feasible. In *WWW*, 1994.

[DCL+00] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused crawling using context graphs. In *VLDB*, 2000.

[dK13] Maurice de Kunder. The indexed Web. `http://www.worldwidewebsize.com/`, 2013.

[DMSW09] Dimitar Denev, Arturas Mazeika, Marc Spaniol, and Gerhard Weikum. Sharc: Framework for quality-conscious web archiving. *Proc. VLDB Endow.*, 2(1):586–597, August 2009.

[Edm67] J.R. Edmonds. Optimum branchings. *J. Research Nation Bureau of Standards*, 71B, 1967.

[EM03] Marc Ehrig and Alexander Maedche. Ontology-focused crawling of web documents. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, 2003.

[Ete08] Elika J. Etemad. Cascading style sheets (CSS) snapshot 2007. Working draft, W3C. `http://www.w3.org/TR/css-beijing`, May 2008.

[FB10] Emilio Ferrara and Robert Baumgartner. Automatic wrapper adaptation by tree edit distance matching. In *Combinations of Intelligent Methods and Applications*. Springer, 2010.

[FGG+11] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Jon Sellers. OXPath: A language for scalable, memory-efficient data extraction from Web applications. *PVLDB*, 4(11), 2011.

[FGG+12] Tim Furche, Georg Gottlob, Giovanni Grasso, Omer Gunes, Xiaoanan Guo, Andrey Kravchenko, Giorgio Orsi, Christian Schallhart, Andrew Sellers, and Cheng Wang. Diadem: Domain-centric, intelligent, automated data extraction methodology. In *WWW*, 2012.

[FMNW03] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *WWW*, 2003.

[Fre98] Dayne Freitag. Information extraction from HTML: Application of a general machine learning approach. In *In Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

[GBE96] Terrance Goan, Nels Benson, and Oren Etzioni. A grammar inference algorithm for the world wide web. In *AAAI*, 1996.

[Gil05] Jim Giles. Internet encyclopaedias go head to head. *Nature*, 438, 2005.

[GLM06] Weizheng Gao, Hyun Chul Lee, and Yingbo Miao. Geographically focused collaborative crawling. In *WWW*, 2006.

[GLZZ06] Yan Guo, Kui Li, Kai Zhang, and Gang Zhang. Board forum crawling: A Web crawling method for Web forums. In *Web Intelligence*, 2006.

[GM99] Stéphane Grumbach and Giansalvatore Mecca. In search of the lost schema. In *ICDT*, 1999.

[GMC11] Daniel Gomes, João Miranda, and Miguel Costa. A survey on web archiving initiatives. In *Proceedings of the 15th International Conference on Theory and Practice of Digital Libraries: Research and Advanced Technology for Digital Libraries*, 2011.

[GMM+11] Pankaj Gulhane, Amit Madaan, Rupesh Mehta, Jeyashankher Ramamirtham, Rajeev Rastogi, Sandeep Satpal, Srinivasan H. Sengamedu, Ashwin Tengli, and Charu Tiwari. Web-scale information extraction with vertex. In *ICDE*, 2011.

[GMS14] Georges Gouriten, Silviu Maniu, and Pierre Senellart. Scalable, generic, and adaptive systems for focused crawling. In *HT*, 2014.

[GPT05] David Gibson, Kunal Punera, and Andrew Tomkins. The volume and evolution of Web page templates. In *WWW*, 2005.

[GS05] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW*, 2005.

[GS12] Georges Gouriten and Pierre Senellart. API Blender: A uniform interface to social platform APIs. In *WWW*, Lyon, France, 2012. Developer track.

[GVD04] Pierre Genevès and Jean-Yves Vion-Dury. XPath Formal Semantics and Beyond: a Coq based approach. In *TPHOLs*, 2004.

[HJM+98] Michael Hersovici, Michal Jacovi, Yoelle S. Maarek, Dan Pelleg, Menachem Shtalhaim, and Sigalit Ur. The shark-search algorithm. an application: tailored web site mapping. *Computer Networks and ISDN Systems*, 30(1-7):317–326, 1998.

[HN99] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *WWW*, 2(4):219–229, April 1999.

[HNVV03] Maria Halkidi, Benjamin Nguyen, Iraklis Varlamis, and Michalis Vazirgiannis. Thesus: Organizing web document collections based on link semantics. *VLDB*, 12(4):320–332, 2003.

[HXG+13] Yeye He, Dong Xin, Venkatesh Ganti, Sriram Rajaraman, and Nirav Shah. Crawling deep web entity pages. In *WSDM*, 2013.

[Inta] Internet Archive. `http://crawler.archive.org/index.html`.

[Intb] Internet Memory Foundation. `http://internetmemory.org/en/`.

[ISH+96] Michiaki Iwazume, Kengo Shirakami, Kazuaki Hatadani, Hideakl Takeda, and Toyoaki Nishida. IICA: An Ontology-based Internet Navigation System. In *AAAI*, 1996.

[ISO] ISO. ISO 28500:2009, Information and documentation – WARC file format.

[JSYL13] Jingtian Jiang, Xinying Song, Nenghai Yu, and Chin-Yew Lin. Focus: Learning to crawl Web forums. *IEEE Trans. Knowl. Data Eng.*, 2013.

[JTG03] Judy Johnson, Kostas Tsioutsiouliklis, and C. Lee Giles. Evolving strategies for focused web crawling. In *Proceedings of the 20th International Conference on Machine Learning*, 2003.

[Jup12] Emily Jupp. Obama's victory tweet 'four more years' makes history. *The Independent*, November 2012. `http://ind.pn/RF5Q6O`.

[KFJ06] Pranam Kolari, Tim Finin, and Anupam Joshi. SVMs for the blogosphere: Blog identification and splog detection. In *AAAI*, 2006.

[KLHC04] Hung-Yu Kao, Shian-Hua Lin, Jan-Ming Ho, and Ming-Syan Chen. Moining Web informative structures and contents based on entropy analysis. *IEEE Trans. Knowl. Data Eng.*, 2004.

[Koe03] Wallace Koehler. A longitudinal study of web pages continued: a consideration of document persistence. *Inf. Res.*, 9(2), 2003.

[KSG+12a] Jochen Kranzdorf, Andrew Sellers, Giovanni Grasso, Christian Schallhart, and Tim Furche. Visual OXPath: Robust Wrapping by Example. In *WWW*, 2012.

[KSG+12b] Jochen Kranzdorf, Andrew Jon Sellers, Giovanni Grasso, Christian Schallhart, and Tim Furche. Visual OXPath: robust wrapping by example. In *WWW*, 2012. Demonstration.

[Kus99] Nicholas Kushmerick. Regression testing for wrapper maintenance. In *AAAI*, 1999.

[Kus00a] Nicholas Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118:15–68, 2000.

[Kus00b] Nicholas Kushmerick. Wrapper verification. *WWW*, 3(2):79–94, March 2000.

[KWD97] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *IJCIA*, 1997.

[LdSGL04] Juliano Palmieri Lage, Altigran S. da Silva, Paulo B. Golgher, and Alberto H. F. Laender. Automatic generation of agents for collecting hidden web pages for data extraction. *Data Knowl. Eng.*, 49(2):177–196, May 2004.

[LJM06] Hongyu Liu, Jeannette Janssen, and Evangelos Milios. Using hmm to learn user browsing patterns for focused web crawling. *Data Knowl. Eng.*, 59(2):270–291, November 2006.

[LL01] Mengchi Liu and Tok Wang Ling. A rule-based query language for HTML. In *DASFAA*, 2001.

[LL06] Christoph Lindemann and Lars Littig. Coarse-grained classification of Web sites by their structural properties. In *CIKM*, 2006.

[LL07] Christoph Lindemann and Lars Littig. Classifying Web sites. In *WWW*, 2007.

[LLWL09] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: Scaling to 6 billion pages and beyond. *ACM Trans. Web*, 3(3):8:1–8:34, July 2009.

[LMK03] Kristina Lerman, Steven N. Minton, and Craig A. Knoblock. Wrapper maintenance: A machine learning approach. *J. Artificial Intelligence Research*, 2003.

[LMWL06] Yu Li, Xiaofeng Meng, Liping Wang, and Qing Li. RecipeCrawler: Collecting recipe data from WWW incrementally. In *Advances in Web-Age Information Management*. Springer, 2006.

[LN01] Seung-Jin Lim and Yiu-Kai Ng. An automated change-detection algorithm for HTML documents based on semantic hierarchies. In *ICDE*, 2001.

[LNL04] Zehua Liu, Wee Keong Ng, and Ee-Peng Lim. An automated algorithm for extracting Website skeleton. In *DASFAA*, 2004.

[LST13] Wee-Yong Lim, Amit Sachan, and Vrizlynn L. L. Thing. A lightweight algorithm for automated forum information processing. In *Web Intelligence*, 2013.

[Mas06] Julien Masanès. *Web archiving.* Springer, 2006.

[MB06] Jim Melton and Stephen Buxton. Querying XML: XQuery, XPath, and SQL/XML in Context, Morgan Kaufmann Publishers Inc., 2006.

[MC02] James C. Mulvenon and Michael Chase. *You've Got Dissent! Chinese Dissident Use of the Internet and Beijing's Counter Strategies.* Rand Publishing, 2002.

[MHL03a] Xiaofeng Meng, Dongdong Hu, and Chen Li. Schema-guided wrapper maintenance for web-data extraction. In *WIDM*, 2003.

[MHL03b] Xiaofeng Meng, Dongdong Hu, and Chen Li. Schema-guided wrapper maintenance for Web-data extraction. In *WIDM*, 2003.

[Mih96] George Andrei Mihaila. WebSQL - an SQL-like query language for the World Wide Web. Master's thesis, University of Toronto, 1996.

[MJC+07] Jayant Madhavan, Shawn R. Jeffery, Shirley Cohen, Xin (Luna) Dong, David Ko, Cong Yu, and Alon Halevy. Web-scale data integration: You can only afford to pay as you go. In *CIDR*, 2007.

[MKK+08] Jayant Madhavan, David Ko, Lucja Kot, Vignesh Ganapathy, Alex Rasmussen, and Alon Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[MKSR04] G. Mohr, M. Kimpton, M. Stack, and I. Ranitovic. Introduction to heritrix, an archival quality web crawler. In *Proceedings of the 4th International Web Archiving Workshop*, 2004.

[MMK99] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third Annual Conference on Autonomous Agents*, 1999.

[MPSR01] Filippo Menczer, Gautam Pant, Padmini Srinivasan, and Miguel E. Ruiz. Evaluating topic-driven web crawlers. In *SIGIR*, 2001.

[NCO04] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web?: The evolution of the web from a search engine perspective. In *WWW*, 2004.

[NH02] Marc Najork and Allan Heydon. High-performance web crawling. In James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 25–45. Kluwer Academic Publishers, 2002.

[Nie11] Michael A. Nielsen. *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press, 2011.

[OD08] Xavier Ochoa and Erik Duval. Quantitative analysis of user-generated content on the Web. In *WebEvolve*, 2008.

[OFG97] Edgar Osuna, Robert Freund, and Federico Girosi. An improved training algorithm for support vector machines. In *Workshop on Neural Networks for Signal Processing*, 1997.

[ON10] Christopher Olston and Marc Najork. Web crawling. *Found. Trends Inf. Retr.*, 4(3):175–246, March 2010.

[OP08] Christopher Olston and Sandeep Pandey. Recrawl scheduling based on information longevity. In *WWW*, 2008.

[PCM+13] Vassilis Plachouras, Florent Carpentier, Julien Masanés, Thomas Risse, Pierre Senellart, Patrick Siehndel, and Yannis Stavrakas. An architecture for selective web harvesting: The use case of heritrix. In *Proceedings of the 1st International Workshop on Archiving Community Memories*, 2013.

[PDO04] Sandeep Pandey, Kedar Dhamdhere, and Christopher Olston. Wic: A general-purpose algorithm for monitoring web information sources. In *VLDB*, 2004.

[PKTK12] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *WWW*, 2012.

[PPV08] Ioannis Partalas, Georgios Paliouras, and Ioannis Vlahavas. Reinforcement learning with classifier selection for focused crawling. In *Proceedings of the 2008 Conference on Artificial Intelligence*, 2008.

[PtWHWG00] Steven Pemberton and the W3C HTML Working Group. XHTML 1.0: The extensible hypertext markup language. Recomendation, W3C, 2000.

[RDP+12] Thomas Risse, Stefan Dietze, Wim Peters, Katerina Doka, Yannis Stavrakas, and Pierre Senellart. Exploiting the social and semantic web for guided web archiving. In *Theory and Practice of Digital Libraries*, volume 7489, pages 426–432. Springer, 2012.

[RFC] RFC 1738. Uniform Resource Locator (URL). `http://www.faqs.org/rfcs/rfc1738.html`.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. Recomendation, W3C, 1999.

[Roy12] Royal Pingdom. WordPress completely dominates top 100 blogs. `http://goo.gl/eifRJ`, 2012.

[RPAH07] Juan Raposo, Alberto Pan, Manuel Álvarez, and Justo Hidalgo. Automatically maintaining wrappers for semi-structured Web sources. *Data Knowl. Eng.*, 2007.

[SA99] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy Web data-sources using W4F. In *VLDB*, 1999.

[SDM+09] Marc Spaniol, Dimitar Denev, Arturas Mazeika, Gerhard Weikum, and Pierre Senellart. Data quality in web archiving. In *Proceedings of the 3rd Workshop on Information Credibility on the Web*, 2009.

[SDNR07] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, 2007.

[SFG+11] Andrew Sellers, Tim Furche, Georg Gottlob, Giovanni Grasso, and Christian Schallhart. Exploring the Web with OXPath. In *LWDM*, 2011.

[Sig05] Kristinn Sigurðsson. Incremental crawling with Heritrix. In *IWAW*, 2005.

[SMMB+08] C. M. Sperberg-McQueen, Eve Maler, Tim Bray, Jean Paoli, and FranÃ§ois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Recomendation, W3C, 2008.

[SMSK07] N. Sawa, A. Morishima, S. Sugimoto, and H. Kitagawa. Wraplet: Wrapping your Web contents with a lightweight language. In *SITIS*, 2007.

[Sod99] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272, 1999.

[SS02] V. Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of the 18th International Conference on Data Engineering*, pages 357–368, 2002.

[SSWC10] J.-Y. Su, D.-J. Sun, I-C. Wu, and L.-P. Chen. On design of browser-oriented data extraction system and plug-ins. *JMST*, 18, 2010.

[THCG05] Thanh Tin Tang, David Hawking, Nick Craswell, and Kathy Griffiths. Focused crawling for both topical relevance and quality of medical information. In *CIKM*, 2005.

[The09] The Future Buzz. Social media, Web 2.0 and internet stats. `http://goo.gl/H0FNF`, 2009.

[The12] The Apache Software Foundation. `http://hbase.apache.org/`, 2012.

[Twi11] Twitter. Historical data not working. `https://dev.twitter.com/discussions/2483`, 2011.

[Twi13] Twitter. GET. `https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline`, 2013.

[VdSdMC06a] Márcio L. A. Vidal, Altigran Soares da Silva, Edleno Silva de Moura, and João M. B. Cavalcanti. GoGetIt!: a tool for generating structure-driven Web crawlers. In *WWW*, 2006.

[VdSdMC06b] Márcio L. A. Vidal, Altigran Soares da Silva, Edleno Silva de Moura, and João M. B. Cavalcanti. Structure-driven crawler generation by example. In *SIGIR*, 2006.

[W3C98] W3C. Document Object Model (DOM) Level 1 specification, W3C Recommendation. `http://www.w3.org/TR/REC-DOM-Level-1`, 1998.

[W3C99] W3C. XML path language (XPath) version 1.0. `http://www.w3.org/TR/xpath/`, 1999.

[W3C07a] W3C. An XML Query Language. Recommendation, W3C. `http://www.w3.org/TR/xquery/`, 2007.

[W3C07b] W3C. XML Query (XQuery) Requirements. `http://www.w3.org/TR/xquery-requirements`, 2007.

[W3C08] W3C. Extensible markup language (XML) 1.0 (fifth Edition). `http://www.w3.org/TR/REC-xml/`, 2008.

[W3C09] W3C. Web application description language. `http://www.w3.org/Submission/wadl/`, 2009.

[WL02] Jiying Wang and Frederick H. Lochovsky. Data-rich section extraction from HTML pages. In *WISE*, 2002.

[WL03] Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *WWW*, 2003.

[Wor12] WordPress. WordPress sites in the world. `http://en.wordpress.com/stats/`, 2012.

[WYL+08] Yida Wang, Jiang-Ming Yang, Wei Lai, Rui Cai, Lei Zhang, and Wei-Ying Ma. Exploring traversal strategy for web forum crawling. In *SIGIR*, 2008.

[XYG+11] Yingju Xia, Yuhang Yang, Fujiang Ge, Shu Zhang, and Hao Yu. Automatic wrappers generation and maintenance. In *PACLIC*, 2011.

[YT12] Hwei-Ming Ying and V.L.L. Thing. An enhanced intelligent forum crawler. In *CISDA*, 2012.

[ZKK08] Hai-Tao Zheng, Bo-Yeong Kang, and Hong-Gee Kim. An ontology-based approach to learnable focused crawling. *Information Sciences*, 178(23):4512 – 4522, 2008.

[ZL05] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *WWW*, 2005.

[ZSWW07] Shuyi Zheng, Ruihua Song, Ji-Rong Wen, and Di Wu. Joint optimization of wrapper generation and template detection. In *KDD*, 2007.