Extracting XML Data from HTML Repositories

by

Ruth Yuee Zhang

M.Eng., Beijing University of Aeronautics and Astronautics, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Statistics)

We accept this thesis as conforming
to the required standard

The University of British Columbia

August 2004

Library Authorization

Ruth Yuee Zhang	10/08/2004
Name of Author *(please print)*	Date (dd/mm/yyyy)

Title of Thesis: Extracting XML Data from HTML Repositories

Degree: Master of Science Year:

Department of Statistics
The University of British Columbia
Vancouver, BC Canada

Abstract

There is a vast amount of valuable information in HTML documents, widely distributed across the World Wide Web and across corporate intranets. Unfortunately, HTML is mainly presentation oriented and hard to query. While XML is becoming a standard for online data representation and exchange, there is a huge amount of legacy HTML data containing potentially untapped information.

We develop a system to extract desired information (records) from thousands of HTML documents, starting from a small set of examples. Duplicates in the result are automatically detected and eliminated. The result is automatically converted to XML. We propose a novel method to estimate the current coverage of results by the system, based on capture-recapture models with unequal capture probabilities. We also propose techniques for estimating the error rate of the extracted information and an interactive technique for enhancing information quality. To evaluate the method and ideas proposed in this paper, we conduct an extensive set of experiments. The experimental results validate the effectiveness and utility of our system, and demonstrate interesting tradeoffs between running time of information extraction and coverage of results.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deep appreciations to my supervisors Professor Ruben H. Zamar and Professor Laks V.S. Lakshmanan for their excellent guidance and immense help. Without their support and encouragement, this thesis would not have been completed. I also want to thank Dr. Lang Wu for being my second reader and for his invaluable advice on my thesis. I am grateful to the faculty and staff in the department for providing such a friendly and motivating academic environment. Many thanks go to my friends: Mike Danilov, Yinshan Zhao, Weiliang Qiu, Kunling Wu and Aihua Pu and all of other graduate students for their great help in difficult times. Last but not least I would like to thank my parents, my husband and my son for their endless love.

RUTH YUEE ZHANG

The University of British Columbia
July 2004

Chapter 1

Introduction

1.1 Motivation

There is a vast amount of valuable information widely distributed across the World Wide Web and across corporate intranets. Until now, the typical ways to get information from the Web are hyperlink browsing and keyword searching, which return web pages (HTML documents) as results to the user. HTML documents are mainly presentation and human oriented. The user has to browse these documents manually to get the desired information. A simple example is to get the authors who have more than 10 books published. There is no way to get the answer by browsing or keyword searching.

Compared to HTML, XML, the de facto standard for data exchange, is both machine and human readable and can be easily queried and transformed. Thus, extracting information from HTML documents and translating the extracted information into XML is valuable and attractive. A considerable research has been dedicated to the information extraction from web pages. Some related works such as TSIMMIS ([17], [18]), RoadRunner ([11]), EXALG ([2]) and Lixto ([7]) will be discussed in Chapter 6.

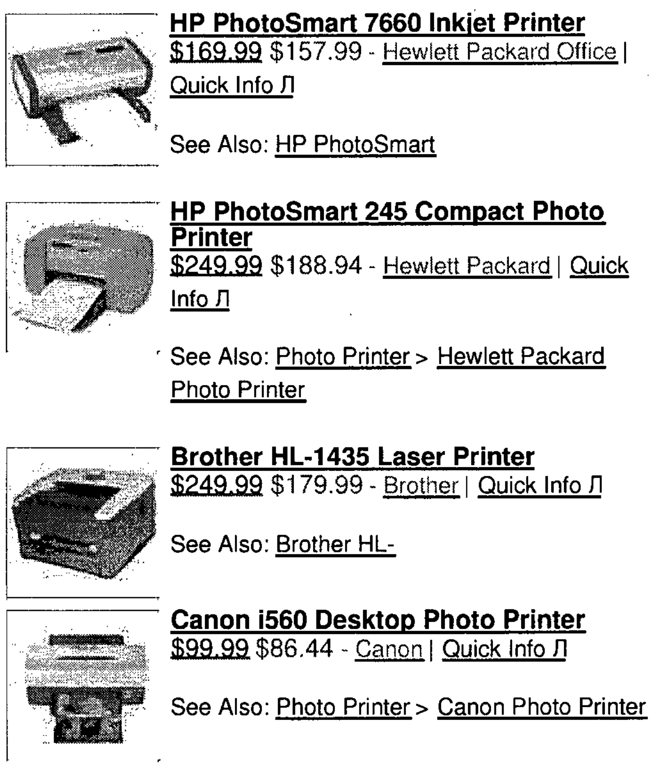
For example, we could generate up-to-date XML data about job postings including `title`, `company`, `qualification`, `location` and `salary` from a large HTML repository such as (any fraction of) the World Wide Web. Similarly, we could generate a large up-to-date list of books, including the `author`, `title`, `publisher`, `year` and `price`. Or we could also generate structured catalogues consisting of `models`, `brands` and `prices` of printers, computers, cameras and cars available from a variety of online stores or dealerships. Then a variety of queries and analyses could be performed on the data so extracted and represented. For example, we could get a list of printers that are less than $100 by a simple query on extracted XML printer-data, and a list of authors who have more than 10 books published by a simple query on extracted XML book-data.

The vision behind our project is to develop a general purpose system for methodically and systematically converting large HTML collections into XML, with some assurances about coverage and quality. An example web page is shown in Figure 1.1. Our system extracts the desired information from the web page and presents it in XML (Figure 1.2).

1.2 HTML and XML

HTML stands for Hyper Text Markup Language, which is used for creating hypertext documents on the WWW. An HTML file is a text file containing markup tags. The markup tags tell the Web browser how to display the page.

Start tags are written as `<tagname>`, and end tags as `</tagname>`. Tagnames are not case-sensitive, i.e. lower and upper case letters are treated as the same. For example, tag `<strong>` is considered the same tag as `<STRONG>`. Some tags have start and end tags. For example, text can be emphasized by placing it between



HP PhotoSmart 7660 Inkjet Printer
~~$169.99~~ $157.99 - Hewlett Packard Office | Quick Info Л

See Also: HP PhotoSmart

HP PhotoSmart 245 Compact Photo Printer
~~$249.99~~ $188.94 - Hewlett Packard | Quick Info Л

See Also: Photo Printer > Hewlett Packard Photo Printer

Brother HL-1435 Laser Printer
~~$249.99~~ $179.99 - Brother | Quick Info Л

See Also: Brother HL-

Canon i560 Desktop Photo Printer
~~$99.99~~ $86.44 - Canon | Quick Info Л

See Also: Photo Printer > Canon Photo Printer

Figure 1.1: An Example Web Page

```
<printers>
......
      <printer><model>HP PhotoSmart 7660 Inkjet Printer</model>
               <brand>Hewlett Packard Office</brand>
               <price>$157.99</prince>
      </printer>
      <printer><model>HP PhotoSmart 245 Compact Photo Printer</model>
               <brand>Hewlett Packard</brand>
               <price>$188.94</prince>
      </printer>
......
</printers>
```

Figure 1.2: Extracted XML Data

`<strong>` and `</strong>`. Some tags only have start tags and no end tags, e.g. `<img SRC="images/kremlin.gif" ALT="The Kremlin at dusk">`, which is used to include an image. Start tags may have attributes, which define some characteristic of that tag. For example, in previous example `SRC` gives the source of the image and `ALT` describes the content of the graphic.

There are some well-known problems with HTML files. HTML is designed to display data using a browser and to focus on how data look. There is a fixed collection of tags with a fixed semantics, i.e. fixed meanings of tags. Most HTML documents are invalid, i.e. not conforming to the specifications of HTML coding, which is a detail descriptions of HTML language. This problem is usually caused by incorrect nesting of elements, missing required attribute, non-standard attribute, omitted end tag, etc. All of the above make HTML documents presentation oriented and difficult to query.

XML stands for eXtensible Markup Language. Originally it was designed to meet the challenges of large-scale electronic data publishing. XML also plays an increasingly important role in the data representation and exchange on the Web and elsewhere. The main difference between XML and HTML is that XML is designed to describe data and to focus on what data are. The display of the data is described in a separate style sheet. XML documents are readable both by human and by computer. It allows users to define their own tags and their own document structure. As opposed to HTML, all XML documents must have a root tag, all tags in XML must have an end tag and XML tags are case-sensitive.

The tags created by the user can be documented in a Document Type Definition (DTD). The purpose of DTD is to define the structure, elements and attributes that are available for use in a document that complies to the DTD. Figure 1.3 is an example XML document with a Document Type Definition. All the names of tags are defined by the user. The DTD is interpreted as follows:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note    (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Figure 1.3: An Example XML File

!ELEMENT note: defines the element "note" as having four elements: "to, from, heading, body".

!ELEMENT to: defines the "to" element to be of the type "CDATA".

!ELEMENT from: defines the "from" element to be of the type "CDATA".

There are numerous supporting technologies on XML. XSL (eXtensible Stylesheet Language) is a style sheet language of XML. XSL can be used to define how an XML file should be displayed by transforming it into a format that is recognizable by a browser. One such format is HTML. Normally XSL does this by transforming each XML element into an HTML element. XQuery is an XML query language for retrieving and interpreting information from XML information sources. It is is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath, XQL, XML-QL, SQL, and OQL.

XML is not a replacement for HTML. In the future, it is most likely that XML will be used to structure and describe the Web data, while HTML will be used to format and display the same data.

There is enormous amount of information about HTML and XML on the Web. For example, `http://www.w3schools.com/` and `http://www.xmlfiles.com/` are good sources for learning more about HTML and XML.

1.3 Text Data Structure

In a survey ([13]) by Line Eikvil, the sources of data can be grouped into three types: free, semistructured and structured text.

Free text This refers to the natural language texts without any specific description of the structure. An enormous amount of information exists in natural language form. For example, news and articles reported in newspapers may contain information about the location of the headquarters of organizations, or about terrorist event such as the type of attack, the date, location, targets and effects on targets. Pharmaceutical research abstracts may contain information about new products, their manufacturers, patent information etc.

Natural Language Processing (NLP) techniques are used for extraction of information from free text. NLP usually applies techniques such as filtering, part-of-speech tagging and lexical semantic tagging to build extraction rules, which are based on syntactic and semantic constraints that help to identify relevant information.

Structured text This refers to the text following a predefined and strict format, such as textual information in a database or an XML document. Information in structured text can be easily and precisely extracted, given the format description.

Semistructured text This type of text is in-between free and structured text. The Web is a rich source containing semistructured text. It does not have a regular and static structure, and can not be queried or processed automatically like structured text. But there is some form of structure based on tokens and delimiters like tags in HTML documents. This is the type of source for our extraction system. Pages in the Web are categorized into three classes by [3]: multiple-instance, single-instance and loosely-structured sources.

The `multiple-instance sources` refer to sources that provide information on multiple pages with the same format. The information often comes from a database. Unfortunately in most of the cases, we cannot access these underlying databases freely. But the information on the Web is freely accessible. There are many sources on the Web falling in this category. An example of this kind of sources is Amazon.com. It shows the author, title, price and other items in the same format for all its book pages. Another example is the CIA World Fact Book. It provides information of 268 countries in the world, on a separate page for each country. All the pages are presented in exactly the same format.

The `single-instance sources` are single web pages which have some kind of clear organization. A job postings page of a company can be an example of this type of sources. The page usually has clearly identifiable sections such as job title, job description, location, posted date, etc.

The `loosely structured sources` are pages that do not show a structure or organization. Almost all personal homepages are loosely structured, where automatic information extraction becomes more difficult.

1.4 Questions Adressed

Some fundamental questions for our project are listed below.

1. From an HTML repository, which essentially ignores the meaning of data, how

are we supposed to determine which piece of data corresponds to what we are looking for? For instance, how do we know which piece of text in a page corresponds to a book title?

2. Suppose we have a system that extracts data from the web. Can we say anything about *how large a proportion* of the actual data that it can successfully extract? This question is important for two reasons. Firstly, being able to estimate the amount of results that could be reached by a system would give us a means by which we could discriminate between competing systems. It allows systems to be calibrated using some metric. Secondly, when a system involves a loop (as in our case), where in each iteration it extracts some information, the ability to predict *coverage* is *critical.* As we will see, each iteration is an expensive operation since it involves accessing a large collection of HTML files, looking for occurrences of records, and analyzing them. If at some point we know that the coverage is close to what the user desires, we can terminate the iteration.

3. An equally important question is what can we say about the *quality* of the information extracted. Firstly, if we can estimate the information quality of an information extraction system, that gives us yet another metric for calibrating systems. Secondly, it forms the basis for data cleaning. Specifically, if we can estimate the *error rate* in the extracted information, this can facilitate the subsequent user decision on how to cope with the quality of the information obtained, as addressed by the next question.

4. Finally, if the estimated error rate is unacceptable to the user, how can we *clean* the data to eliminate as many errors as possible.

These were the questions that motivated the current work.

1.5 Evaluation Metrics

We propose two metrics, *coverage* and *error rate*, to evaluate the result of an information extraction system. The coverage and error rate of a set of extracted records are defined as

$$\text{coverage} = \frac{\text{number of records in the set}}{\text{number of records findable by the system}}, \tag{1.1}$$

$$\text{error rate} = \frac{\text{number of erroneous records in the set}}{\text{number of records in the set}}. \tag{1.2}$$

It is worth mentioning other two evaluation metrics, recall and precision, which are used by the Message Understanding Conferences (MUC) to evaluate information extraction systems.

$$\text{Recall} = \frac{\text{number of correctly extracted records}}{\text{number of records in answer key}}, \tag{1.3}$$

$$\text{Precision} = \frac{\text{number of correctly extracted records}}{\text{number of extracted records}}. \tag{1.4}$$

Recall may be crudely interpreted as a measure of the fraction of the information that has been correctly extracted, and precision as a measure of the fraction of the extracted information that is correct. They are inversely related to each other, meaning that by allowing for a lower recall you can achieve a higher precision and vice versa.

The notion of coverage is different from recall. In the case of recall, the denominator is the number of records in the given answer key, which is known in advance. In the case of coverage, the denominator is the number of all records that could be found by the system, which is unknown and must be estimated. The numerator in recall is the number of correctly extracted records, while in coverage it is the number of all extracted records.

The functions of recall and coverage are different. Recall is used to experimentally evaluate various information extraction systems. The information sources are restricted to a fixed range and all the useful records (i.e. answer key) are known, usually gathered manually. Conversely, coverage is used to measure the proportion of all findable records have been extracted in the current iteration. Given the large scale of the Web and other HTML repositories, we cannot know exactly how many records are included in these sources. For the same reason, the information extraction system is computationally expensive and usually takes a huge amount of time to reach convergence. Coverage can estimate how much information could be extracted when convergence could be reached, and what proportion of all findable information has already been extracted.

Error rate is equivalent to (1-precision).

1.6 Overview

We present a system for extracting records from a repository of HTML files and for converting them to XML. For example, a record can be a job posting that has five attributes: `title, company, qualification, location` and `salary`. Thousands of job postings are scattered on the Web in unstructured or semi-structured form. Our system can extract and store them in relational form or as XML data.

The procedure consists of two phases.

- Phase I includes extraction of records and their translation to XML. During this phase, duplication of information is detected automatically and eliminated. In addition, using statistical techniques, we are able to estimate the coverage of the result set with respect to the source HTML data set. All these steps are done automatically.

- Phase II is aimed at assessing and enhancing the quality of the extracted information. This phase is interactive. The system provides random samples to users, from which users provide feedback by pointing out the erroneous information. Based on the feedback, the system automatically "cleans" up previously extracted information.

Records extraction in phase I is built on the framework of the method DIPRE (Dual Iterative Pattern Relation Expansion) pioneered by Brin ([8]). DIPRE focuses on extracting a relation of pairs (two attributes such as author and title of a book). Figure 1.4 shows the structure of this method. Starting from a small seed set of example tuples (records), the program finds the occurrences of those seeds on a web page repository. These occurrences are grouped, and patterns are recognized from these groups of occurrences. Next, the program searches the repository to find all occurrences that could be matched with at least one of the patterns. Records could be extracted from these occurrences according to the content of the patterns. The first iteration yields a set of records which usually have more records than those in the seed set. We take this set as the seed set and find all their occurrences, generate more patterns then get more records matching these patterns. This procedure could be repeated until some termination criterion is met. This criterion may be "convergence" (no new records are found) or that the scale of the results is large enough.

One attractive feature of DIPRE is that records are extracted from thousands of web pages which are usually structured very differently, as opposed to one web page or even several similarly structured web pages. We implement this framework, extend it to support multivariate records involving more than two attributes, and improve it in several ways. Because the scale of the repository is usually very large, it is not feasible to run the iterative procedure until convergence, i.e. until no more new records are found. The user could set up a desired coverage. The program will estimate the coverage of the result of each iteration and continue until the desired coverage is achieved.

Owing to the large scale of heterogeneous information resources, it is inevitable that there will be erroneous records contained in the results. In phase II we assess the quality of results by estimating the error rate. If the user decides to carry out this step, our system will present a random sample of extracted records. The user then points out whether a record is a correct record or an error.

We make the following contributions in this project.

- Building on the DIPRE method, we develop techniques for extracting records from thousands of HTML documents and presenting them in XML format. Occurrences and patterns are defined in a more flexible way compared to DIPRE, without increasing the error rate in the results. At the end of each iteration, we select the most reliable records to start the next iteration. Experiments show that these techniques can improve the efficiency, produce high coverage and low error rates.
- We propose a methodology to evaluate the coverage achieved at the end of each iteration. Statistical methods based on capture-recapture models with unequal capture probabilities are applied to estimate the coverage rate of the extracted results. This can be used to determine when to stop the iterations, based on a user specified coverage threshold.
- We propose a method for estimating the error rate of the results extracted, by having the user identify correct and erroneous records in a random sample. The size of the sample is automatically determined based on the required level of confidence in the estimated error rate.
- We propose an interactive method to enhance the quality of the results. Based on the interactive feedback from the user, our method is able to track down

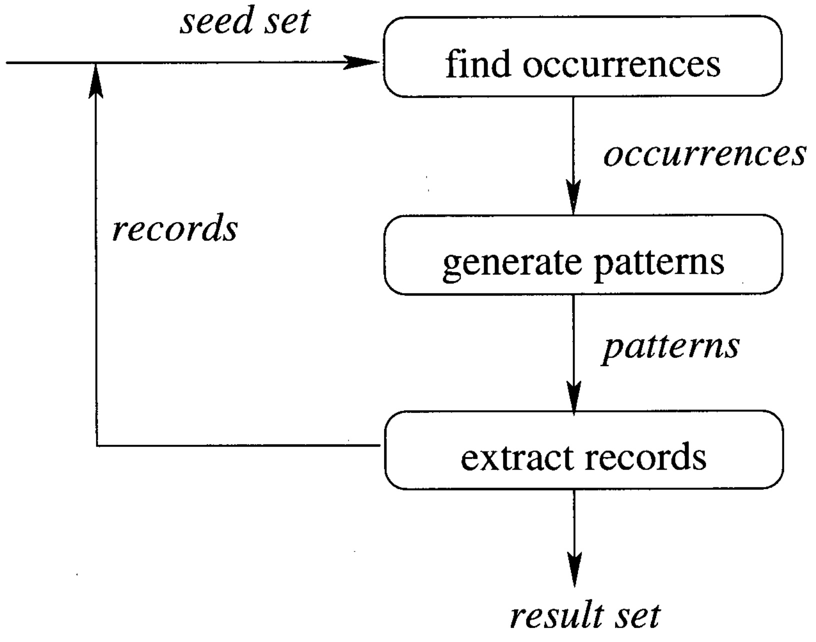


Figure 1.4: The General Structure of DIPRE

the patterns that led to errors, thus flagging problem patterns and problem sets, which include all records that match problem patterns. Our system can estimate the error rate for the problem set, and clean up erroneous records in it.

The rest of the thesis is structured as follows. In Chapter 2, we define the key concepts and describe the strategy for extracting records from HTML files and converting them to XML. Chapter 3 describes two implementation methods. In Chapter 4, we discuss the coverage estimation and data cleaning in details. Chapter 5 gives some experimental results. In Chapter 6, we briefly describe some related work. Finally, Chapter 7 discusses some future work.

Chapter 2

Records Extraction

Our goal is to extract records from a large collection of web pages, starting from several examples of target records. A record (for example, a book) usually has several attributes (for example, author, title, publisher, year and price). In this chapter we generalize two concepts — occurrences and patterns, which have been introduced by [8].

We will also describe the four steps in an iteration of the extracting procedure: (i) finding occurrences, (ii) generating patterns, (iii) extracting records and (iv) eliminating duplicates. The first and third steps are implemented using two different methods, parsing HTML and using regular expressions. The implementation of these two steps will be described in detail in Chapter 3. In this chapter, we briefly introduce the general procedures for the first and third steps. The second and fourth steps are the same for the two different implementation methods. We will discuss these two steps in detail in this chapter.

2.1 Occurrences

An occurrence of a record is the context surrounding attributes of this record in a web page (an HTML file). It is defined as a tuple over *url, order, prefix, separators, suffix* and *values of attributes of the record*, where:

url is the URL of the web page that the record appears on.

order is a number indicating the order that the attributes show up in. For example, there are six possible orders for records with three attributes.

prefix is the names of tags and contexts preceding the attribute appearing first.

separators are the names of tags and context between attributes.

suffix is the names of tags and contexts following the last attribute.

2.2 Patterns

A pattern is used to summarize a group of occurrences. The methods to summarize those occurrences play a critical role in our program. Starting from the same initial examples, different methods to generate patterns produce results which are very different in terms of extracted information quality and quantity.

We follow Brin's ([8]) simple principle to generate patterns. It is defined as a tuple over (*urlprefix, order, prefix, separators, suffix*). The occurrences are first grouped by order and separators. Patterns are generated for each of these groups. For each group, the *order* and *separators* of a pattern are the same as order and separators of occurrences in the group. The *urlprefix* of a pattern is the longest matching prefix of all url's, the *prefix* of a pattern is the longest matching suffix of all prefixes, and the *suffix* of a pattern is the longest matching prefix of all suffixes.

For example, two paragraphs in Figures 2.1 are part of HTML scripts of two web pages, which contain information of printers with three attributes: model, manufacturer and price. The URLs of web pages are shown in the first line of each paragraph. For the following two records,

- **R**ecord 1

  attribute1: Canon i70 Color Bubble Jet Printer

  attribute2: Canon

  attribute3: $217.34

- **R**ecord 2

  attribute1: HP LaserJet 1012 Printer

  attribute2: Hewlett Packard

  attribute3: $149.99

two occurrences, which are shown in Figure 2.2, can be found from HTML scripts in Figures 2.1. According to the principle to generate patterns, a pattern can be generated from these two occurrences. The generated pattern is shown in Figure 2.3.

New occurrences are found by matching the patterns. An occurrence is said to match a pattern if there is a piece of text that appears in a web page whose url matches the urlprefix*, and the text matches `*prefix *separators* suffix*` where '*' represents any string. A new record can be extracted from an occurrence. Values of attributes are extracted and assigned according to the order of the pattern.

Next we describe the four steps in an iteration of our extracting procedure.

```
www.1-electronics-store.com/product+Canon-i70-Color-B00008CMU9.html
  ...<tr><td align="center">...</td><td>
  <a href="/product+Canon-i70-Color-B00008CMU9.html"><b>
  Canon i70 Color Bubble Jet Printer</b></a><br>
  Manufactured by <a href="/man+Canon.html">
  Canon</a><br>Price: $217.34</a></td></tr>...

www.1-electronics-store.com/product+HP-LaserJet-1012-B0000C1XHY.html
  ...<tr><td align="center">...</td><td>
  <a href="/product+HP-LaserJet-1012-B0000C1XHY.html"><b>
  HP LaserJet 1012 Printer</b></a><br>
  Manufactured by <a href="/man+Hewlett-Packard.html">
  Hewlett Packard</a><br>Price: $149.99</a></td></tr>...
```

Figure 2.1: Example of Finding Printers: HTML Scripts from two HTML files

```
url: www.1-electronics-store.com/product+Canon-i70-Color-B00008CMU9.html
attribute1: Canon i70 Color Bubble Jet Printer
attribute2: Canon
attribute3: $217.34
order: 0
prefix: <td><a><b>
separator1: </b></a><br><a>
separator2: </a><br>Price:
suffix: </a></td></tr>

url: www.1-electronics-store.com/product+HP-LaserJet-1012-B0000C1XHY.html
attribute1: HP LaserJet 1012 Printer
attribute2: Hewlett Packard
attribute3: $149.99
order: 0
prefix: <td><a><b>
separator1: </b></a><br><a>
separator2: </a><br>Price:
suffix: </a></td></tr>
```

Figure 2.2: Example of Finding Printers: Occurrences

```
urlprefix: www.1-electronics-store.com
order: 0
prefix: <td><a><b>
separator1: </b></a><br><a>
separator2: </a><br>Price:
suffix: </a></td></tr>
```

Figure 2.3: Example of Finding Printers: A Pattern

2.3 Finding Occurrences

The goal of this step is to find occurrences of the records in the seed set. For each web page, the program go through all the records to find occurrences of the records. An occurrence is obtained by extracting the prefix, separators and suffix as we described and set the proper order. We also record the url of the web page as the url of the occurrence.

To define occurrences precisely, we need to set up some restrictions on the prefix, separators and suffix. In the current system, the prefix and suffix could have at most three tags. The length of the separators, i.e. the distance between two adjacent attributes of a record can not exceed a certain given limit. Otherwise we assume these two attributes are too far from each other and may not belong to the same record. We experimentally found that 300 characters is a good limit for the regular expression approach, and 15 tags is a good limit for the parsing HTML approach.

Details of our implementations can be found in Chapter 3.

2.4 Generating Patterns

The basic algorithm for generating patterns is shown in Figure 2.4.

```
1. $(O_1, \ldots, O_k)$ = GroupOccurrences($O$)
2. for $i = 1$ to $k$
   if $|O_i| == 1$ then next;
   else GenOnePattern($O_i$)
   (a) $p.urlprefix$ = FindUrlprefix(all urls of $O_i$)
   (b) $p.prefix$ = FindLongestSuffix(all prefixes of $O_i$)
   (c) $p.suffix$ = FindLongestPrefix(all suffixes of $O_i$)
   (d) if $p.urlprefix$==NULL
           SubGroupbyUrl($O_i$), goto step 2
       elseif $p.prefix$==NULL
           SubGroupbyPrefix($O_i$), goto step 2
       elseif $p.suffix$==NULL
           SubGroupbySuffix($O_i$), goto step 2
       else
           $p.basedoccur$=$|O_i|$, push($P$, $p$)
```

Figure 2.4: The Algorithm for Generating Patterns

The input is the set of all occurrences, which is denoted by O, found in the previous step. First, these occurrences are partitioned into groups $O_1, \ldots, O_k$ according to the order and separators. The occurrences in each group have the same order and separators. Then for each of those groups that have more than one occurrence, GenOnePattern() is called to generate a pattern p.

GenOnePattern() works in the following way. The urlprefix of p is the longest matching prefix of all the urls in the group. The prefix of p is the longest matching suffix of all prefixes of the occurrences in the group. The suffix of p is the longest matching prefix of all suffixes of the occurrences. If the urlprefix is null, no pattern is generated. This group of occurrences has to be subgrouped by prefixes of the urls, that is, in each subgroup either occurrences have a common prefix of urls or there is only one occurrence in that subgroup. GenOnePattern() is called again for these subgroups. If the prefix is null, this group is subgrouped by suffixes of the prefixes, i.e. occurrences in each subgroup have a common suffix of prefixes. The same action is taken if the suffix is null.

If none of them is null, a pattern p is generated. The number of occurrences that support this pattern is assigned to $p.basedoccur$. The output is a set of patterns P.

Here is an example of subgroup of occurrences by suffixes of prefixes. The prefixes of occurrences in a group are `<b></b><i>`, `<a></b><i>`, `<b></b><br>`. The first two occurrences have the common suffixes of prefixes, so they are going to be in a subgroup after the SubGroupbyPrefix procedure applied. The third occurrence in another subgroup.

2.5 Extracting Records

In this step, the program searches through the HTML repository for texts matching any of the patterns obtained in previous step. Then records are extracted from those texts. Searching is targeted at the patterns and the HTML files, of which the url of the HTML file is matched with the urlprefix of the pattern.

The index of the pattern that a record matches is saved with this record. *It plays a key role in data quality assessment and enhancement.*

Going over all the HTML files and patterns, the system produces a set of records as well as indexes of patterns they match.

2.6 Eliminating Duplicates

It happens very often that some records are found more than once. The most basic method to eliminate duplicates is to compare the values of corresponding attributes of two records. If all of them are exactly the same, the two records are duplicates. One of them is dropped from the result and the number of occurrences of this record is incremented by 1.

Usually there are some duplicates that can not be identified by merely comparing the values of attributes directly. For example, the author of a book may appear in a different format, such as "Isacc Asimov" and "Asimov, Isaac". The title of a book may be

- "The Weapon Shops of Isher", "Weapon Shops of Isher" or "Weapon Shops of Isher, the".
- "Guns, Germs and Steel" or "Guns, Germs, and Steel".
- "Tiger! Tiger!" or "Tiger, Tiger".

To detect those kinds of duplicates, we apply a fairly simple approximate string matching method. Values of two attributes are compared in the following way.

1. Spaces, punctuations and symbols other than English letters and numerals (`[a-zA-Z0-9]`) are removed.
2. Stopwords, i.e. words that are common in a full-text file but have little value in searching, such as `a, an, the, for, to, in, on, and,` etc. are removed.
3. All letters are translated to capital case.
4. Each value is translated into an alphabetical ordered sequence only consisting of capital letters (`A-Z`) and numerals (`0-9`).

The corresponding two sequences are compared to decide whether the two values are the same.

After the four steps of an iteration, a set of records is obtained. To track the relationships between occurrences and patterns, for each record, we store a list of indices of patterns that this record matches.

Intuitively, the records that have more occurrences are more likely to be correct. The records that have more than one occurrence can be selected as the seed set of the next iteration. One of the advantages of this method is that, the next iteration starts from more reliable records and less noise will be added to the pattern generation process. This will also reduce the number of erroneous records in the resulting database. Another advantage is that the next iteration starts from a smaller number of records therefore saves running time. Because records with more occurrences are selected, patterns are more likely to be generated from these records. In our experiments, we noticed that when we use a set of records that have more than one occurrence as seed into the next iteration, it runs faster and returns almost the same number of new records as if we had used all records as seed.

Finally, the extracted records are converted to XML.

Chapter 3

Implementation and Performance

There are two computationally expensive procedures in our system, finding occurrences and extracting records. Both of them have to search through a large repository of HTML documents. Two methods, parsing HTML documents and regular expressions can be applied to implement these two procedures.

An HTML parser is a processor that reads an HTML document and breaks it up into a set of different kinds of markup tags and plain text. The technique of parsing HTML is widely used to extract useful information from HTML documents. There are various HTML parsers available on the Web. For example, Perl modules HTML::Parser ([20]) and HTML::TokeParser ([21]), and Java library HTMLParser ([19]).

The second method used is regular expressions. A regular expression is a string of characters which tells the searching program what kind of string (or strings) you are looking for, e.g. all the words starting with letter *A*. Regular expressions are an extremely powerful tool for manipulating text data. They are now standard features in a wide range of languages and popular tools such as Perl, Java, PHP, Python, and Emacs. The complete overview of the syntax and semantics of regular expressions can be found in many related books and web sites such as [9], [29] and [12].

In this chapter, we will discuss these two methods in some detail. Algorithms for implementing these two procedures are also presented.

3.1 Parsing HTML Documents

In this section, we first introduce parsing HTML script by Perl module HTML::TokeParser. Then the algorithms of finding occurrences and extracting records, which use the technique of parsing HTML, are described.

3.1.1 HTML::TokeParser

HTML is widely regarded as the standard publishing language of the World Wide Web. HTML uses tags such as `<h1>` and `</h1>` to structure text into headings, paragraphs, lists, hypertext links etc. The HTML::TokeParser module in Perl parses an HTML document into a set of tokens. Each token represents either a tag or plain text between two tags. By doing this, it separates the tags from the plain text and recognizes different types of tags.

As we mentioned in Section 1.2, start tags may have attributes. Attribute names and values of a start tag usually relate to the specific records. To find common patterns from context around similar records, we ignore attributes of start tags, and only consider the type of the tokens, the name of a tag, and the content of the text.

For example, parsing the first part of the HTML script in Figure 2.1 gives the following tokens. The first token corresponds to the start tag with tag name "tr". The 4th token corresponds to the end tag with tag name "td". The 8th token corresponds to the text "Canon i70 Color Bubble Jet Printer".

1	S	tr
2	S	td
3	T	...
4	E	td
5	S	td
6	S	a
7	S	b
8	T	Canon i70 Color Bubble Jet Printer
9	E	b
10	E	a
11	S	br
12	T	Manufactured by
13	S	a
14	T	Canon
15	E	a
16	S	br
17	T	Price: $217.34
18	E	a
19	E	td
20	E	tr

3.1.2 Algorithm Implementation

- The algorithm of `Finding Occurrences` is implemented as follows.

  The input is a set of records and HTML files. The Output is a set of occurrences. Each occurrence has tuples: values of attributes of records, url, order, prefix, separators, suffix.

For each HTML file, parse the HTML file into a set of *Tokens*. Each token has three values: *index* (the position of this token), *type* (the type of this token) and *content*.

For each record,

1. Set up a *Flag* to represent the current matching status. *Flag*=0 means new occurrence, and *Flag*=i ($i \neq 0$) means that there are i attributes of the record matched.
2. Set up an *orderFlag* to represent the order of the attributes of the record, *orderFlag*=i means that the current matched attribute is the ith attribute of the record.
3. Create an array *Occur* to store all the occurrences. Each *Occur* has attributes as: *url, Order, Prefix, Separator, Suffix* and the values of attributes of the record.
4. Scan through the *Tokens*. For each token,
    (a) If it is not a text-token, go to next token; Otherwise, if the *Content* of the token is matched with the value of the ith attribute of the record, check the *Flag*.
    (b) If the *Flag* is 0 which means this is a start of a new occurrence. Set the *Flag* to 1. Record the *Index* for this token. Set *Occur.prefix* as the names of three proceeding tags.
    (c) Else, i.e. if the *Flag* is j ($j \neq 0$), which means there are already j attributes of the record matched. We check the distance between current matching and previous matching.
        - If the difference of *index* between consecutive matching is large than 15, then we treat this matching as a start of a new occurrence, do what we do in the previous item for new occurrence.

    - Else treat this as a matching of an attributes. Set the *Flag*=$j+1$. Set *Occur.separator[j]* as the names of the tags between two consecutive matching. Set *orderFlag[Flag]=i*

   (d) If *Flag* is equal to the number of attributes of the record,

    - Set *Occur.suffix* as the names of the three following tags.
    - Set the *Occur.order* according to *orderFlag.*
    - Push this occurrence into *Occur.*

  As we reach the end of the HTML file, we got an array of occurrences: *Occur.* Examples of occurrences are shown in Figure 2.2.

- The algorithm of `Extracting Records` is implemented as following.

  The input is a set of patterns and HTML files. The output is a set of extracted records/results.

  We restrict the matching only within the HTML files of which URLs match the urlprefix of the pattern. For HTML file, if there is a patten p such that $p.urlprefix$ could be matched with the URL of HTML file, the HTML file is parsed into a set of tokens.

  For each token,

  1. Matching the prefix of the pattern
     - If the token is a tag,
       * If the name of the tag matches with the ith tag in $p.prefix$, set $matchflag = i$
       * Else set $matchflag = 0$, goto next token.
     - If the token is not a tag,
       * If all the tags of $p.prefix$ have been matched, return the index of this token $index1$.
       * Otherwise, set $matchflag = 0$, go to next token.

2. Matching the separators of the pattern. If it is success, return the indexes of the tokens right after the last matched tags of the separators. Otherwise go to next token.
3. Matching the suffix of the pattern. If it is not success, go to next token; Otherwise,
4. Extracting the record. Now the whole pattern is matched, the values of attributes of a record can be extracted according to the indexes returned from each matching step above. The order of the attributes is determined by the order of the pattern. The index of the pattern is recorded with the record.
5. The next iteration starts from the token next to the end of current matching.

3.2 Regular Expression

3.2.1 Introduction

The other approach to implement the algorithms is to use regular expressions. A regular expression is a string of characters for matching strings that follow some pattern. Regular expressions are made up of normal characters and metacharacters. Normal characters include upper and lower case letters and digits. The metacharacters have special meanings.

In the simplest form, a regular expression is just a word to search for. For example, suppose we want to search for a string with the word "cat" in it. The regular expression would simply be "cat". The words "cat", "catalog", or "sophisticated" would match this regular expression. A simple example using metacharacters is the regular expression `'a[0-9]+'`. It matches `'a'` followed by one or more digits, where `'[]'` is a metacharacter to look for one of a set of characters, `'+'` is a pattern quantifier that matches one or more items.

There are two regular expression operators within Perl. The matching operator, `m//`, is used to match a string to a regular expression, which returns a true or false value. The substitution operator, `s///`, substitutes one expression for another. For example, the following operations

```
$string=``my cat'';
$string=~s/cat/dog/;
```

substitutes "dog" for the "cat" of $string. As a result, the $string becomes "my dog";

Next we briefly introduce some of the metacharacters of regular expressions that are used in our program.

^	Matches the beginning of a string.
$	Matches the end of a string.
\d	Matches a digit character.
\s	Matches a white-space character (space, tab).
.	Matches any character except the newline character.
`[]`	Matches any one of the characters between the brackets. For example, the regular expression r[aou]t matches rat, rot, and rut, but not ret. Ranges of characters can be specified by using a hyphen. For example, the regular expression [0-9] means match any digit. To match any character except those in the range, the complement range, use the caret as the first character after the opening bracket. For example, [^0-9] will match any character except digits.

***** Matches 0 or more repetitions of the previous term.

+ Matches 1 or more repetitions of the previous term.

{n} Matches exactly n times.

{n,m} Matches at least n times but no more than m times.

() Groups expressions together. Each group is recorded in the special variables $1, $2, $3 and so on, in the sequence of parentheses. For example, suppose $date="04,28,2004", the following substitution operation

```
$date=~s/(\d+),(\d+),(\d+)/$3$1$2/;
```

changes the $date to "20040428", where variables $1, $2, $3 store strings that match regular expressions in the first, second and third parentheses, i.e. "04", "28" and "2004" separately.

g Modifier. Matches/replaces globally every occurrence within a string, not only the first.

i Modifier. Matches are not case-sensitive.

s Modifier. Allows use of '.' also to match a newline character.

? Non-greedy Matching. By default, Perl uses greedy matching. For example,

```
$str=''Hello there, Nothing here'';
```

the substitution operation

```
$str=~s/H.*here/Hello/;
```

substitutes "Hello" for the part matching "H.*here" in `$str`. By default, Perl takes as much as it can into the match. So the part matching "H.*here" in

$str is "Hello there, Nothing here", and the $str becomes "Hello". If we use non-greedy matching,

```
$str=~s/H.*?here/Hello/;
```

after 'H' is matched, '.*' matches the minimum length before the 'here' is matched. So the part matching "H.*here" in $str is "Hello there", which is then substituted by "Hello". Now $str becomes "Hello, Nothing here".

3.2.2 Algorithm Implementation

Here are some examples of part of the regular expressions we use.

- `(<[^<>]*?>[^<>]*?){3}`

  Matches three consecutive tags, which may or may not have attributes. Plain text is allowed between tags.

- `~s#<(/?\w*)\s?[^<>]*?>#<$1>#gi`

  Extracts the name of a tag. For example, after the substitution, `<td align="center">` turns out to be `<td>`.

- `\s*([^<>]*?)\s*`

  Matches plain text (i.e. no tag is in-between), which may have some white-spaces at the beginning or end.

- `~s#>#[^<>]*?>\\s*[^<>]*?#g`

  Substitutes > by `[^<>]*?>\s*[^<>]*?` globally. For example, after the substitution, `<a><b>` turns out to be

  `<a[^<>]*?>\s*[^<>]*?<b[^<>]*?>\s*[^<>]*?`

  which could be matched by

  `<a href="Canon-i80-Color-B0001DBGSQ.html"><b>`

- `Finding Occurrences`

  For each record, regular expressions are constructed by combining its attributes in different orders. For example, a regular expression of a printer for order 0 is

  ```
  ((<[^<>]*?>[^<>]*?){3})Canon i70 Color Bubble Jet Printer
  (.{1,300}?)Canon(.{1,300}?)\$217.34(([^<>]*?<[^<>]*?>){3})
  ```

  Then the program searches through all HTML files for texts that match these regular expressions. An occurrence is extracted from each of the matching texts. The same occurrence shown in Figure refex:occur can be obtained by this approach from the HTML script in Figure 2.1.

- `Extracting records`

  For each HTML file, each pattern,

  1. If the URL of current HTML file matches the urlprefix of the current pattern, perform the task described in point 2 below; otherwise move on to the next pattern.
  2. Match the regular expression "prefix*separators*suffix".

     For example, for the pattern shown in Figure 2.3, the following regular expression is generated.

     ```
     <td[^<>]*?>\s*[^<>]*?<a[^<>]*?>\s*[^<>]*?<b[^<>]*?>\s*[^<>]*?\s*
     ([^<>]*?)
     \s*</b[^<>]*?>\s*[^<>]*?</a[^<>]*?>\s*[^<>]*?
     <br[^<>]*?>\s*[^<>]*?\s*<a[^<>]*?>\s*[^<>]*?\s*
     ([^<>]*?)
     \s*</a[^<>]*?>\s*[^<>]*?<br[^<>]*?>\s*[^<>]*?Price:\s*
     ([^<>]*?)
     \s*</a[^<>]*?>\s*[^<>]*?</td[^<>]*?>\s*[^<>]*?</tr[^<>]*?>\s*[^<>]*?
     ```

For each of the matching texts, the texts between the prefix and the first separator, between two adjacent separators, and between the last separator and the suffix, which correspond to the expressions in the brackets in the regular expression, are extracted out and assigned to the attributes of a record according to the order of the pattern. The index of the pattern is also recorded with this record.

3.3 Performance

Both of the two approaches, parsing HTML and regular expressions, have advantages and disadvantages. Regular expressions are amazingly powerful and deeply expressive. However, constructing the regular expressions is tedious and error-prone. We have to worry about character level trivia, such as space, newline, single and double quotes. From the examples in previous section, we can see that the regular expressions that we use are very complicated. Moreover, there are some rare cases that regular expressions may fail. For example, applying the regular expression

```
(<[^<>]*?>)
```

on the following HTML script,

```
<IMG SRC = "foo.gif" ALT = "A > B">
```

we get the wrong result

```
<IMG SRC = "foo.gif" ALT = "A >
```

instead of the whole tag.

The HTML TokeParser module in Perl takes care of the tedious character level work. It extracts and distinguishes six types of tokens. Compared to regular expression, the approach of parsing HTML is more robust. Also, it is more reason-

Table 3.1: Computational Times

method	attributes	seeds	files	occurs(time)	patts	results(time)
Parser	2	3	242	161(48)	15	357(55)
Regex	2	3	242	97(27)	7	372(0.21)

able to control the distance between consecutive attributes by the number of tags than by the number of characters.

Experiments are conducted to compare the computational times of different approaches. The results are shown in Table 3.1. The columns are the method used, the number of attributes of each record, the number of records in the seed set, the number of HTML files, the number of occurrences found and the time it costed in seconds, the number of patterns, and the number of records extracted and the time it costed in seconds. The first row corresponds to the parsing HTML approach. The second row corresponds to the regular expression approach for records with two attributes.

The experiments are based on the same seeds and the same HTML files. The numbers of records in the results are very similar, and there is no fake records in both of the two results, but the computational time costed by parsing HTML approach is much more than the computational time costed by regular expression approach, especially in the step of extracting records. The reason for the poor timing performance of parsing approach is the huge number of tags in an HTML document. It often happens that some tags in the beginning of a pattern are matched before the matching failed. The program have to keep track of each matching attempt, and then to go back to the starting point when matching attempt has failed at some point.

Both of the two steps, finding occurrences and extracting records, search through a large repository of HTML documents, therefore they are very time-consuming. The computational time is a critical issue for our system. From this point of view, regular expressions are preferred.

Chapter 4

Statistical Analysis

One of our goals is to capture a high percentage of all the records available in an HTML repository. To that effect, we implemented an automatic procedure to estimate, after each iteration, the percentage of all the records available in the repository that are already included in our result (result = set of recovered records). Another goal is to minimize the number of errors in our result. To that effect, we implemented an interactive (semi-automatic) procedure to estimate and reduce the percentage of erroneous records.

In what follows, we give some details of the statistical procedures employed in our system.

4.1 Coverage Estimation

Given the large scale of the Web and other HTML repositories, searching for records is computationally expensive. Therefore we wish to minimize the number of iterations, especially when using large seed sets. Instead of running our system until full convergence (i.e. until no new record is found) we stop when a given coverage target has been met.

We define the "population size" N as the number of records our system could possibly find, i.e., the size of the result set after full convergence. In principle, our searching procedure always converges because the number of records found in each iteration cannot decreases, and there is a finite set of records in the repository. Convergence could be reached either when all records have been found, or no new record could be found in the last iteration.

Let N_i be the size of the result set after the ith iteration and the current coverage is defined as

$$C_i = \frac{N_i \times 100}{N}. \qquad (4.1)$$

Obviously, the denominator in (4.1) is unknown and must be estimated in order to estimate C_i.

In the next section, we discuss the adaptation of "capture-recapture" models and techniques – widely used in Biology to estimate the size of wildlife populations – to estimate the size of (the no less wild) population of "findable" records.

4.1.1 Capture-Recapture Models

The capture-recapture models were originally proposed by biologists to estimate the size of wildlife populations. The basic idea is to set traps to capture some animals and release them after they have been marked. A second trapping is conducted after the animals have had enough time to return and mix back with their population. The number of recaptured animals can then be used to estimate the size of the population. Capture-recapture models and techniques have been used in other areas such as estimating the size of the indexable Web and the coverage of search engines ([26]). In our case, instead of animals we "capture" records and each trapping occasion corresponds to an extracting occasion, i.e. an iteration of our extraction system. Starting from a randomly selected seed set of records (a trap), we run the next iteration of the extraction program to get a larger set of records (captured objects).

We now briefly introduce the basic ideas underlying capture-recapture procedures. In the first extracting occasion of the capture-recapture experiment, a number n_1 of individuals or records are captured, marked and released. At a later time, the second extracting occasion, a number n_2 of individuals or records are captured, of which, say m_2 have been marked. The Petersen estimator for the population size is based on the observation that the proportion of marked individuals in the second sample (recaptured sample) should be close to the same proportion of marked individuals in the total population, i.e.

$$\frac{m_2}{n_2} = \frac{n_1}{N}$$

Thus the estimator for population size is

$$\hat{N} = \frac{n_1 \times n_2}{m_2} \tag{4.2}$$

The estimate variance of this estimator is

$$\mathrm{var}(\hat{N}) = \frac{n_1 n_2 (n_1 - m_2)(n_2 - m_2)}{m_2^3}. \tag{4.3}$$

The general assumptions for this basic capture-recapture model are: (i) the population is closed; (ii) all marks are correctly noted and recorded; (iii) marks are not lost; (iv) each individual has a constant and equal capture probability on each extracting occasion.

The first assumption means that the size of the population is constant over the period of experiment. This is a strong assumption and usually not true in biological populations. Nevertheless, this assumption is valid when we deal with a static repository (e.g., a fragment of the web crawled and stored). Assumptions 2 and 3 are also always true in our setting. However, the last assumption is strongly violated here. Different records usually have different probabilities of being found (captured). We applied two methods to solve capture-recapture models with unequal capture probabilities. We first introduce the post-stratification method, which is transparent and easy to understand. A more complicated method, the generalized jackknife, proposed by Burnham ([10]), is discussed later.

4.1.2 Post-stratification Method

When a web page contains a large number of records, it is more likely that records from this page follow (a small number of) patterns, because the contexts of records in the same web page are usually similar. Consequently the records in this page are more likely to be found (by matching these patterns). Therefore, depending on the patterns that a record matches, it becomes easier or harder to be found. For a pattern, we define its score as the number of records matching this pattern. Patterns with higher scores are associated with higher capture probabilities.

For each extracting occasion, we obtain a set of patterns and a set of records. Together with each record, the program keeps a set of indexes of patterns which are matched by this record. The score of a pattern can be obtained by scanning through the set of records and counting how many times a pattern has been matched.

Based on the scores of patterns, we set up a score for each record. The score of a record is the maximum of scores of all the patterns which are matched by this record. The higher the score of a record, the larger the capture probabilities of this record.

The post-stratification method is to stratify the captured records by their scores into several strata. Capture probabilities of records in the same strata are assumed to be the same. Thus the basic capture-recapture model can be applied in each stratum to estimate the population size of this stratum. The total estimated

```
1. Calculate ScoresofPatterns($R_1$), ScoresofPatterns($R_2$)
2. Calculate ScoresofRecords($R_1$), ScoresofRecords($R_2$)
3. $R$ = Combine($R_1$, $R_2$), for each $r$ in $R$
   If $r$ only in $R_1$, $r.mark = 1$
   If $r$ only in $R_2$, $r.mark = 2$
   If $r$ in both $R_1$ and $R_2$
   $r.mark = 3$
   $r.score = \max(r.score \text{ in } R_1, r.score \text{ in } R_2)$
4. $(G_1, G_2, G_3)$ = StratifyByScore($R$)
5. For each $G_i$, $i = 1, 2, 3$
   $x_{ij}$ = the number of $r$ that $(r.mark = j)$, $j = 1, 2, 3$
   $n_i = x_{i1} + x_{i3}$, $m_i = x_{i2} + x_{i3}$, $x_i = x_{i3}$
   $\hat{N}_i = n_i \times m_i / x_i$
6. $\hat{N} = \hat{N}_1 + \hat{N}_2 + \hat{N}_3$
```

Figure 4.1: Algorithm of Post-Stratification

population size then will be the sum of the population sizes of the strata.

The analysis is based on two extracting occasions. We get two sets of patterns P_1 and P_2, and two sets of records R_1 and R_2. The algorithm for estimating the population size based on two extracting occasions is shown in Figure 4.1.

From the discussion above, the score can be obtained for each pattern, as well as for each record. Then the two sets of records are combined together by the following strategy. For each record: (i) if it is in both sets, it is marked 3 and the score of this record is the largest of its two scores; (ii) if it is only in set 1, it is marked 1 and its score is the same as in set 1; (iii) if it is only in set 2, it is marked 2 and its score is the same as in set 2.

Based on the compound set, the records are stratified by their overall scores. For example, records can be stratified to three strata according to their scores, greater than 10000, 1000–10000, and less than 1000. Parameters and variables manipulated by the algorithm in Figure 4.1 are as follows:

- x_{i1}, x_{i2}, x_{i3} denote the number of records in ith ($i = 1, 2, 3$) stratum, which are only in set 1, only in set 2, in both sets respectively.
- n_i is the number of records in the ith stratum captured in set 1.
- m_i is the number of records in the ith stratum captured in set 2.
- x_i is the number of recaptured records.
- $\hat{N}_i$ is the estimated population size of ith stratum.
- $\hat{N}$ is the estimated population size.

4.1.3 Generalized Jackknife Estimate

There are detailed discussions on the capture-recapture models on closed animal populations with unequal capture probabilities in [28]. Three types of models with unequal capture probabilities for closed populations are based on the following assumptions:

M_t: Every individual has the same capture probability for a given occasion, but capture probabilities can vary at each sampling time.

M_b: The initial capture probabilities for all individuals are the same, adjust for a change in capture probabilities caused by a response to trapping.

M_h: Each individual has its own capture probability independent of all other individuals. This probability keeps the same at each trapping occasion.

The capture probabilities for different records are usually different. For each record the capture probability is constant at each extracting occasion. So the model M_h is suitable here.

Burnham and Overton introduced the model M_h and developed a nonparametric estimator of population size based on the generalized jackknife in [10]. We briefly describe the estimation procedure here.

Suppose we do the extracting t times, the basic data are the extracting histories of records, denoted by $x_{ij}, i = 1, \ldots, N, j = 1, \ldots, t$, where N is the population size,

$$x_{ij} = \begin{cases} 1 & \text{if the } i\text{th record is captured on the } j\text{th extracting occasion,} \\ 0 & \text{otherwise.} \end{cases}$$

The unknown parameters are population size N and the capture probabilities of records $p_1, \ldots, p_N$. The model assumes that $p_1, \ldots, p_N$ are a random sample from an arbitrary probability distribution and all x_{ij}'s are independent.

The capture frequencies are defined as $f_0, f_1, f_2, \ldots, f_t$, where for $j = 1, \ldots, t$, f_j is the number of records that have been caught exactly j times in all the t times extracting, and f_0 is the number of records never captured. The capture frequencies have multinomial distribution. The number of all different records captured in the t times extracting occasions is $S = \sum_{j=1}^{t} f_j$. It has been proved in [10] that the set of capture frequencies f_j is a sufficient statistic for the data x_{ij}.

The kth ($k < t$) order jackknife estimator, given by Burnham in [10], is

$$\hat{N}_k = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \begin{pmatrix} k \\ i \end{pmatrix} (n-i)^k \hat{N}_{(t-i)} \tag{4.4}$$

where the $\hat{N}_{(t-i)}$ is an estimator of population size based on all combinations of $t-i$ extracting occasions. Let $j_1, \ldots, j_i$ be a combination of i integers from the set $\{1, \ldots, t\}$, define $\hat{N}_{t-i,j_1,\ldots,j_i}$ as the estimator based on the $t-i$ extracting occasions remaining after $j_1, \ldots, j_i$ extracting occasions are dropped, then

$$\hat{N}_{(t-i)} = \begin{pmatrix} t \\ i \end{pmatrix}^{-1} \sum_{j_1,\ldots,j_i} \hat{N}_{t-i,j_1,\ldots,j_i} \tag{4.5}$$

which can be calculated by

$$\hat{N}_{(t-i)} = S - \begin{pmatrix} t \\ i \end{pmatrix}^{-1} \sum_{r=1}^{i} \begin{pmatrix} t-r \\ i-r \end{pmatrix} f_r. \tag{4.6}$$

From (4.4) and (4.6) we can see that the $\hat{N}_k$ is a linear combination of the capture frequencies,

$$\hat{N}_k = \sum_{i=1}^{t} a_{ik} f_i. \tag{4.7}$$

The coefficients a_{ik} can be obtained from (4.4) and (4.6).

The estimated variance of this estimator is

$$\widehat{\text{var}}(\hat{N}_k) = \sum_{i=1}^{t} a_{ik}^2 f_i - \hat{N}_k$$

The variance of $\hat{N}_k$ increases as k increases. Therefore if two estimators are not significantly different, we take the one with the smaller order. We test the hypothesis that there is no significant difference between ith and $(i+1)$th order jackknife estimates, that is

$$H_0 : E(\hat{N}_i - \hat{N}_{i+1}) = 0, H_1 : E(\hat{N}_i - \hat{N}_{i+1}) \neq 0$$

The test statistic is given in [10]. Under the null hypothesis the test statistics has approximately a standard normal distribution. If we can not reject the null hypothesis, we take $\hat{N}_i$ as the estimation of population size. If all of these estimators are significantly different, we take the $(t-1)$th order jackknife estimator as the final estimator.

```
1 system: Generate a random sample $R$ of size $n$ from the result $W$.
2 user: Identify true records $T$ and erroneous records $E$, using user feedback.
3 system: Estimate the error rate of $W$.
4 user: If estimated error rate is acceptable, stop else continue.
5 system: $PP$ = FindProblemPatterns($E$).
    For each $p$ in $PP$
        (1) system: $B$ = ProblemSet($p$), $m = |B|$
        (2) user: Determine to discard or keep $B$. If we decide to keep $B$,
        (3) system: $R(p)$=RandomSample($B$)
        (4) user: Identify true records $T(p)$ and erroneous records $E(p)$.
        (5) system: $r(p)$=ErrorRate($p$)
        (6) user: Determine to discard, keep as it is or clean $B$. If we decide to clean,
            continue.
        (7) system: Clean(B)
```

Figure 4.2: Quality Assessment and Enhancement Algorithm

4.2 Quality Assessment

The error rate for a set of records is defined as the number of erroneous records over the total number of records in the set,

$$r = \frac{\text{\# of erroneous records}}{\text{\# of records}}.$$

Unfortunately, the occurrence of errors in the resulting database of records is unavoidable. On the other hand, a high error rate can greatly diminish the usefulness of the database. Therefore, it is important to consider steps to estimate and reduce r.

The first step to control r is to obtain an estimate of its magnitude. We call this step "Quality Assessment". If, as a result of this step, r is considered too large, then measures are implemented to reduce it. We call this second step "Quality Enhancement" and discuss it in the next section.

For interactive quality assessment and enhancement, the system initiates the dialogue described in Figure 4.2. The first three steps are for the quality assessment of the result. The following steps are to enhance the quality of the result. First the system generates a sample R of n records that are randomly selected from the resulting database W of extracted records. The sample size is determined automatically by the system, based on the required level of confidence in the estimated error rate. Then the user examines R manually to identify erroneous records. The error rate is estimated by the proportion of errors in W and reported together with a 95% confidence interval for r. Based on the estimated error rate, the user decides whether just to accept W as the final result or to do further quality enhancement steps. The estimation of r is further discussed below. Quality enhancement is further discussed in the next section.

The estimate of the error rate, $\hat{r}$, can be obtained using simple random sampling. A relatively small number of records, n, are randomly selected from the database and submitted to the user for inspection. The user manually checks the records and identifies erroneous ones. The number of erroneous records, denoted by n_e, has hypergeometric distribution with parameters N, n and r and an unbiased estimate of r is

$$\hat{r} = \frac{n_e}{n}.$$

The estimated standard error of this estimator is given in [31],

$$\widehat{\mathrm{se}}(\hat{r}) = \sqrt{\frac{N-n}{N}\frac{\hat{r}(1-\hat{r})}{n-1}}.$$

A key issue is to choose the number of records needed to estimate the error rate with a desired precision (upper bound on the estimate standard error). If we want $\mathrm{se}(\hat{r}) < \beta$, then we need

$$n > \frac{Nr(1-r)}{(N-1)\beta^2 + r(1-r)}, \tag{4.8}$$

where N is the total number of records in the database.

To get n, we need an "initial estimate", $\hat{r}_0$, of r and the desired precision, β. When there is no information about r, $\hat{r}_0 = 50\%$ can be used to obtain a conservative sample size n. However, the actual error rate of the extracted result should be much less than 50%. Otherwise we would need to reconsider the pattern generation rule. In our implementation, $\hat{r}_0 = 20\%$, $\beta = 2\%$ are for default determination of n.

By central limit theory, the estimator $\hat{r}$ is approximately normally distributed with mean r and variance var($\hat{r}$). A 95% confidence interval for the error rate r is

$$\hat{r} \pm 1.96 \times \widehat{\text{se}}(\hat{r}).$$

4.3 Quality Enhancement

The quality enhancement procedure first finds the problem patterns, that is, patterns that are matched by erroneous records. Recall that each record in the result has a set of indexes of patterns that it matches. For each problem pattern p, all records that match this pattern are called problem set B. The system will report features of B such as the number of records and the percentage significance of this set compared to the resulting database W. Based on this report, the user either decides to accept the result or to perform quality enhancement. For example, if there are only few records in B, or the proportion of a problem set B is only 0.1%, the user may decide to discard the entire problem set from the result. On the other hand, if the size or relative size of B is large, the user may instruct the system to improve the quality of this problem set B.

The error rate for the problem set B can be obtained using the same random sampling method described before. Again, the user must decide here the future course of action. If the error rate is very high, this problem set should probably be discarded. If the error rate is very low, we may keep this set as it is. Otherwise, when the error rate of a problem set is moderate and the size of the set is not trivial

```
$V$=values of attribute $k$ of $(W - B)$.
For each $r$ in $B$
  1. Spellingcheck($r$). If pass,
  2. $v = r.attribute[k]$
  3. if($v$ not in $V$) and (occ($r$)==1)
     $r$ is an error
     else $r$ is true
```

Figure 4.3: The Algorithm of Clean(B)

compared to the whole set, the user can ask the system move to the "cleaning" step.

Spell-checking is a simple but very powerful tool to filter out erroneous records containing alphanumeric values. The user can specify the spelling constraints on attributes of records. For example, names of people or products usually start from capital letters. Prices usually begin with currency signs and then digits.

We propose another effective method of finding erroneous records in a problem set B. The basic algorithm is shown in Figure 4.3.

Based on the fact that the number of records in the result is substantial, some attributes of records are very likely to have repeated values in the result. For instance, an author usually appears more than once in a large set of books. A manufacturer or brand usually appears more than once in a large set of printers. Our method is to verify whether the value of such a specific attribute of a record is valid, by checking whether this value has been repeated in some other records.

Taking searching for books as an example, V is the set of values of authors from all records other than the current problem set. The value (v) of the author of each record in the problem set is compared to the values in V. If we cannot find v in V, this record is likely to be an error.

In addition, we check whether this record matching one pattern or more than one pattern. If it matches only this problem pattern, we assume that it is an error.

Notice that it is possible that some correct records may be wrongly treated as errors. The larger the number of records in the result W, the smaller this probability. Experiment results in Section 5.3 show the effectiveness of this method.

Chapter 5

Experimental Results

We successfully applied our prototype to several domains such as printers (model, manufacturer, price), papers (author, year, title, journal) and books (author, title, price). Comprehensive experiments were conducted for the simple case of two attributes: authors and titles of books. The results of these experiments are presented in this section.

5.1 Searching for Books

The experiments ran on a collection of 16128 HTML files used as a testbed. These files were obtained by searching the Web using Google. For each of 370 randomly selected books, we run the Google search automatically, setting the author and title of a book as the keywords. For each book the first 50 web pages of the searching results were selected. We took the union of the 370 sets of urls and successfully downloaded 16128 HTML files.

We started the experiment with 3 popular books, shown in Table 5.1. The first iteration produced 266 occurrences and 19 patterns. Matching these patterns

Table 5.1: Seeds of Books

Author	Title
Isaac Asimov	The Robots of Dawn
David Brin	Startide Rising
Charles Dickens	Great Expectations

Table 5.2: Result of the First Iteration

related occurrences	number of books
exactly 1 occurs	813
more than 1 occurs	454
more than 2 occurs	122
more than 3 occurs	74
more than 4 occurs	57
more than 5 occurs	43
more than 6 occurs	34
more than 7 occurs	26
more than 8 occurs	19
more than 9 occurs	15
more than 10 occurs	11

over all the HTML files produced 1267 books. For these 1267 books, the number of occurrences related to the number of books are shown in Table 5.2.

The key issue is how to choose the seed sets for the following iterations. Three methods of choosing seed sets were used in our experiments.

Blind: This method takes all of the results from the previous iteration as the seed set for the next. Because the number of seeds is large, only the second iteration was performed. Starting from the 1267 books, 16525 occurrences are found, 880 patterns are generated and 45034 books were extracted.

GoodSeed: This method chooses books with more than one occurrence from the results of previous iteration as the seed set. For the same reason as above, only the second iteration was performed. There were 454 books that have more than one occurrence out of the 1267 books. Starting from these 454 books, we found 9200 occurrences, 588 patterns and 40715 books.

TopSeed: This method chooses some number of books that have the most number of occurrences from the results of previous iteration. To show the relationship between the running time and the coverage of the results, the numbers of seeds are set to be 50 more than the number of seeds of previous iteration. That is, the number of seeds for the second iteration is 53, for the third iteration is 103, ..., four more iterations were performed. The number of books obtained is 39412. These results are shown in Table 5.3.

By doing the above experiments, we can compare three quantities (the coverage, error rate and running time) of these three methods. These results are shown in Table 5.6. Columns represent names of seed sets, numbers of extracted results, estimated coverges of those results, 95% confidence intervals of error rates and times costed in hour. The `Blind` has a little higher coverage but its error rate is the largest and the running time is much longer than the other two methods. The `GoodSeed` has the shortest running time. What is very impressive is that, the estimated error rate for `TopSeed` is very low (zero). Because in `TopSeed` method, the seeds for each iteration are selected with high positive confidence, the results are highly reliable. The running time of `TopSeed` is less than half of that of `Blind`. In general, the last two methods are better than `Blind`. `TopSeed` took more than `GoodSeed` because we ran more iterations, but it offers the least error (zero in this case). The coverage of `GoodSeed` and `TopSeed` are about the same.

We plot the coverage versus running time for `TopSeed` in Figure 5.1. It shows that once the coverage becomes large, it takes much more time to improve it.

An example of part of an XML document generated by our system is shown in Figure 5.2.

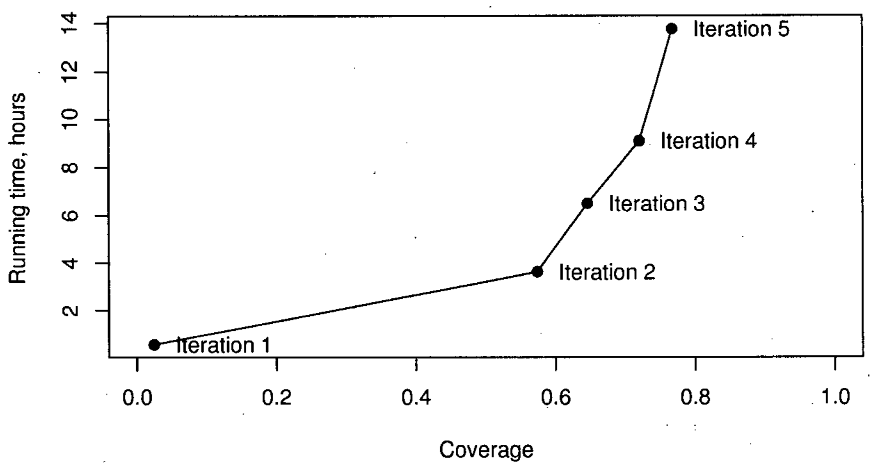


Figure 5.1: The Coverage Versus Running Time

```
<book><author>A. Merritt</author>
      <title>The Moon Pool</title>
</book>
<book><author>Isaac Asimov</author>
      <author>Robert Silverberg</author>
      <title>Nightfall</title>
</book>
```

Figure 5.2: Example of Part of XML File

Table 5.3: Searching Results by Choosing Seeds with Most Number of Occurrences

iteration	books	run time(h)	# of results
1	3	0.55	1267
2	53	3.62	29471
3	103	6.5	33167
4	153	9.08	36985
5	203	13.75	39412

Table 5.4: Results of 10 Extracting Occasions

NO.	occurs	patterns	# of results
1	388	39	31482
2	421	38	31053
3	368	39	31609
4	424	34	31199
5	449	44	32143
6	716	79	32736
7	1016	100	34694
8	458	48	32388
9	408	52	31704
10	499	53	31992

5.2 Coverage Estimation

The analyses in this section are based on the 10 extracting occasions. In this experiment, 10 seed sets are randomly selected from the result of `Blind` which contains 45034 books. All seed sets have the same number of books $n = 200$. For each seed set, we run one iteration of the extracting program to get a set of books. The number of occurrences, patterns and books in the results are shown in Table 5.4. The total number of different books captured is 40933. As we described in Section 4.1, these experiments are 10 extracting occasions of a capture-recapture study. The coverage estimation will be based on these results.

5.2.1 Post-stratification

The post-stratification method discussed in Section 4.1.2 is applied on each pair of 10 extracting occasions, $\begin{pmatrix} 10 \\ 2 \end{pmatrix} = 45$ pairs in total. The estimated population size is the average of these 45 estimations.

For each stratum, the Equation 4.2 is used to estimate the stratum size $\hat{N}$.

Table 5.5: Post-Stratification Result for Two Sets of Books

	only in set1	only in set2	both	total	estimation
1st stratum	39	1128	27427	28594	28596
2nd stratum	2398	1676	945	5019	9272
3rd stratum	744	911	56	1711	13814
total	3181	3715	28428	35324	51682

We can see that, when the recapture rate is high, i.e. the number of the recapture records m_2 is close to the number of records in the extracting occasion n_1 and n_2, $\hat{N}$ is close to m_2. When the recapture rate is low, i.e. the number of the recaptured records m_2 is small compare to n_1 and n_2, $\hat{N}$ tends to be large compared to n_1 and n_2.

We find empirically that books with scores more than 600 are very likely to be captured in each extracting occasion, therefore those books should belong to one stratum. Books are stratified into three strata according to their scores, greater than 600, between 300 and 600, and less than 300.

For the result sets of the first and second extracting occasion, for example, the post-stratification results are shown in Table 5.5. The first row is the numbers of records belong to the first stratum. In the first row, from left to right, the numbers are numbers of records captured in the first extracting occasion but not captured in the second occasion (x_{i1}), captured in the second extracting occation but not captured in the first occasion (x_{i2}), captured in both occasions (x_{i3}), the total number of records in the first stratum ($x_{i1} + x_{i2} + x_{i3}$), and the estimated size of the first strata ($\hat{N}_{11}$) respectively. The second and third rows are similar except they are for the records belong to the second and third strata. For each stratum, the basic capture-recapture model is applied. This gives the following estimated strata size,

$$\hat{N}_{11} = 28596, \hat{N}_{12} = 9272, \hat{N}_{13} = 13814$$

the estimated population size and the standard error of $\hat{N}_1$

$$\hat{N}_1 = 51682.$$

The final estimation of the population size is the average of estimates based on 45 pairs of extracting occasions.

$$\hat{N} = \frac{1}{45}\sum_{i=1}^{45}\hat{N}_i = 52448.$$

The jackknife variance estimator is applied to estimate the variance of $\hat{N}$. The estimator is formed as follows:

$$\mathrm{var}(\hat{N}_J) = \frac{t-1}{t}\sum_{l=1}^{t}(\hat{N}_{(-l)} - \hat{N})^2 \tag{5.1}$$

where $t = 10$, $\hat{N}_{(-l)}$ is the estimated population size without the lth $(l = 1, \cdots, t)$ trapping occasion, that is,

$$\hat{N}_{(-l)} = \frac{1}{\begin{pmatrix} 9 \\ 2 \end{pmatrix}} \sum_{i=1, i\neq l}^{10} \sum_{j=i+1, j\neq l}^{10} \hat{N}_{(ij)}. \tag{5.2}$$

The estimated standard error is $\hat{\mathrm{se}}(\hat{N}) = 7627$. The estimated coverage can be calculated based on the $\hat{N}$. For example, the estimated coverage of the result of `Blind` is $45034/52448 = 85.9\%$.

5.2.2 Generalized Jackknife

For these $t = 10$ extracting occasions, the capture frequencies $f_1, \cdots, f_{10}$ are 3320, 2155, 2203, 1241, 921, 1152, 1421, 1393, 2081 and 25046.

The generalized jackknife estimates from 1st to 9th order can be calculated by equation (4.4) and (4.6). They are 43921, 45045, 45904, 46891, 48027, 49184, 50202, 50950 and 51342.

The test statistics to test whether there is significant difference between two consecutive order jackknife estimates are calculated. The results show that there are significant differences between these estimates. We take the 9th jackknife estimates as the estimation of the population size, $\hat{N} = 51342$. The estimated standard error $\hat{\text{se}}(\hat{N}) = 634$. The estimated coverage of the result of `Blind` is 45034/51342=87.7%.

The post-stratification method is simple to implement and understand. However, it assumes that records in each stratum have the same capture probabilities. When this assumption is highly violated, the estimates may be baised. Also, the standard error of the post-stratification estimator is much larger than the generalized jackknife estimator. In the current system, the generalized jackknife estimator is used to estimate the coverage.

5.3 Quality Assessment and Enhancement

First we estimate the error rates of the results of the three methods: `Blind`, `GoodSeed`, `TopSeed` and the book set obtained by Sergey Brin, which is posted on the web. Because all of these sets have very good quality, the error rates are much less than 10%. By choosing $r = 10\%$ and $\beta = 2\%$ in Equation (4.8), we get the number of records needed to estimate the error rates $n = 225$. All these samples are manually checked. The results are shown in Table 5.6. All of the error rates are very low. In practical situations these results may satisfy the user's requirement. However, to apply and evaluate our quality enhancement methods, we look through the result of `Blind` and find several erroneous books. From each of these errors, more erroneous books are detected using the method described in Section 4.3. The quality enhancement procedure is shown by the following examples.

There is an erroneous book with author "A Bad Spell in Yurt" and title "May 2001". This record is obtained by matching the problem pattern with index

Table 5.6: Performance of the Three Methods. (C.I. is 95% confidence intervals of the error rates)

sets	# of results	coverage	C.I.	time(h)
`Blind`	45034	87.7%	[0, 0.02]	59.04
`GoodSeed`	40715	79.3%	[0, 0.01]	17.25
`TopSeed`	39412	76.8%	0	33.5
Brin's set	15257	N/A	[0,0.03]	N/A

428. The total number of records (problem set) from this pattern is 73. Because the size of this problem set is relatively small (73/45034=0.16%), we investigate the whole set instead of a sample. Only 14 of them are true books and the error rate is 81%. We can discard the whole problem set from the result. Actually the errors in this set are easy to identify by spell checking. A book with a date as title is likely to be an erroneous book. By spell-checking we can detect and delete all errors in this problem set.

Another erroneous book has author "Away Melody" and title "A Far". Following the quality enhancement procedure, the problem pattern for this error is the pattern with index 151. The problem set B has 421 books in total, which is about 1% percent of the result of `Blind`. So we decide to keep this set and go to further steps. Manually checking a sample of 100 randomly selected books, 28 of them are errors. The reported confidence interval of the error rate is [0.2, 0.36]. The procedure Clean(B) is applied and 190 out of 421 books (45%) are marked as erroneous books. As we mentioned before, some of these 190 books are correct books and discarded wrongly as an error. The other 231 books are all correct books. So after the quality enhancement procedure, the error rate of the problem set becomes 0.

Chapter 6

Related Work

In general, an Information Extraction (IE) system is to extract useful information (records) from unstructured and semistructured text and transform the information into structured text such as databases and XML documents. In other words ([22]), IE systems go through the following general procedures:

- take unstructured and semistructured text as input and generate something like patterns or templates to locate the useful data.
- extract the useful data according to the patterns or templates.
- encode the extracted data in a structured form that is ready for further manipulation.

Informaiton Extraction from unstructured text, i.e. plain text, is one of the topics of Natural Language Processing (NLP). The paper [1] presents a system for information extraction from large collections of plain-text documents.

There has been much work on information extraction from semistructured text, which usually are HTML documents. The traditional approach for information extraction from the Web is to write specialized programs, called wrappers, that identify and extract data of interest. In the case of our system, patterns serve as wrappers.

In early stage, wrappers are generated manually, e.g. TSIMMIS ([17], [18]). The manually approach has many shortcomings, mainly due to the difficulty in writing and maintaining wrappers. Recently, many wrapper generation tools have been proposed to generate wrappers automatically, e.g. RoadRunner ([11]) and EXALG ([2]), or semi-automatically, e.g. Lixto ([7]). A common goal of wrapper generators is to generate wrappers that are highly accurate and robust, while requiring as little effort as possible from the users.

Laender et al. [25] is an excellent survey on recent web information extraction tools which compares them on several qualitative aspects such as degree of automation, support for complex structure, etc. In this chapter, we will introduce and briefly describe several information extraction systems. All these systems have the same goal, i.e. extracting data from a set of HTML pages and presenting the data into structured text. However the implementation methods are quite different.

6.1 TSIMMIS

A pioneer information extraction system TSIMMIS ([17], [18]) generates wrappers manually. Wrappers are in the form of specification files, which are simple text files written by users. The specification files describe the structure of HTML pages and identify the beginning and the end of the relevant data. The extractor executes the commands in a specification file to find and extract the interesting data into a set of variables. Finally the extracted data is converted into database objects. As a running example, the system is used for extracting weather data from several web sites.

Each specification file consists of a sequence of commands, which are writtern

```
1 [``root'',
   ``get('http://www.intellicast.com/weather/europe/')'',
   ``#'`],
2 [``temperatures'',
   ``root'',
   ``*<TABLE*<TABLE*</TR>#</TABLE>*'']
```

Figure 6.1: A Simple Example Specification File

by users according to the following specification rules. Each command defines one extraction step, in the form of *[variables, source, pattern]*, where *variables* are one or more variables that hold the extracted results, *source* is the part of the HTML script to be considered, *pattern* specifies how to find the text of interest within the source. The value of a variable can be used as input source for subsequent commands. By allowing this, TSIMMIS supports the hierarchical structure of web data.

An example of the specification file is shown in Figure 6.1. The first command defines the variable *root* as the entire source file whose URL is given by the source *get('http://www.intellicast.com/weather/europe/')*. The pattern "#" in the first command means that everything in the source is to be extracted and stored in the variable *root*. The extractor then executes the second command, which defines a new variable called *temperature* nested in the variable *root*. The value of *temperature* is specified by the result of applying the pattern

```
*<TABLE*<TABLE*</TR>#</TABLE>*
```

to the source variable *root*, i.e. entire HTML source file. The second command can be interpreted as, discarding everything until the first occurrence of `</TR>` in the second table definition and saving the data between `</TR>` and `</TABLE>` as variable *temperature*. The variable *temperature* will store part of the HTML file and could be an input source for subsequent commands, which can specify patterns within this part of HTML source file.

TSIMMIS is flexible, accurate and efficient. The specification files are straight forward and easy to understand. The drawback is that the extraction mechanism depends on users input of the specification file. When the structure of source files changes, the specification file need to be updated by the user. This is a problem for the Web sources, because different information sources (web sites) usually have very different structure, new sources appear everyday and the format of existing sources may change. Therefore, most of the recent research works are based on automatic or semi-automatic wrapper generation. Anyway TSIMMIS is an important system in the history of information extraction.

6.2 RoadRunner

Many web sites contain large sets of pages generated using a common template. For example, all book pages in Amazon present the author, title, price, etc. in the same way. RoadRunner ([11]) generates wrappers automatically to extract data from such template-generated web pages.

The input of the system is a number of sample pages taken from a class of web pages, for which we want to generate a wrapper. The system starts from any two pages. One is chosen as an initial version of the wrapper. The matching algorithm works on these two pages, an initial wrapper and a sample, to find a common wrapper for them. Then the matching algorithm is applied on this wrapper and another sample page to find a common wrapper. This procedure continue until a wrapper has been found for all the sample pages. Then the wrapper is applied on this class of web pages to extract the data. This system is applied on several template-generated web site such as Amazon and RpmFind.

In the following, the wrapper generation procedure is briefly explained through an example. Detail information about this project can be found in [30].

```
<html>Books of:<b>#PCDATA</b>
<ul>
     (<li><i>Title:</i>#PCDATA</li>)+
</ul></html>
```

Figure 6.2: The Wrapper

Union-free regular expressions (UFRE) are used to describe a wrapper. A union-free regular expression is defined over tokens and a special symbol `#PCDATA`. If a and b are UFRE, $a \cdot b$, $(a)+$ and $(a)?$ are also UFRE, where + means one or more occurrence and ? means zero or more occurrence. There is no union operators in UFRE. An example of a UFRE is shown in Figure 6.2.

A wrapper is generalized by solving mismatches between the wrapper and the sample page. Both the wrapper and the sample page are parsed into a list of tokens. A mismatch occurs when some token in the sample does not comply to the corresponding token in the wrapper. There are two kinds of mismatches, string mismatches and tag mismatches, which correspond to different methods to generalize the wrapper respectively. A simple matching example is shown in Table 6.1.

- `String Mismatches.` It happens when different strings occur in the corresponding positions of the wrapper and the sample. String mismatches are used to discover data fields. For example, in Table 6.1, there is a string mismatch in token 4, "John Smith" and "Paul Janes". In this case the wrapper is generalized by replacing "John Smith" by `#PCDATA`.
- `Tag Mismatches.` It happens when different tags, or one tag and one string occur in the corresponding positions of the wrapper and the sample. Tag mismatches are usually caused by repeated patterns. For example, a tag mismatch between token 19 in the wrapper and the sample comes from different number of books in the book lists, i.e. different number of the repeated pattern t =`<li><i>Title:</i>#PCDATA</li>`. In this case the wrapper is generalized

Table 6.1: One Simple Matching

Wrapper	Sample
01: <html>	01: <html>
02: Books of:	02: Books of:
03: <b>	03: <b>
04: John Smith	04: Paul Jones
05: </b>	05: </b>
06: <ul>	06: <ul>
07: <li>	07: <li>
08-10: <i>Title</i>	08-10: <i>Title</i>
11: DB Primer	11: XML at Work
12: </li>	12: </li>
13: <li>	13: <li>
14-16: <i>Title</i>	14-16: <i>Title</i>
17: Comp. Sys.	17: HTML Scripts
18: </li>	18: </li>
19: </ul>	19: <li>
20: </html>	20-22: <i>Title</i>
	23: Javascript
	24: </li>
	25: </ul>
	26: </html>

by replacing t by $(t)+$.

Usually the tag mismatches are more complicated than those we just discussed. The matching algorithm is recursive since more mismatches can be found and need to be solved when trying to solve one mismatch. The algorithm should also be able to backtrack. Usually there are several possible alternatives to solve a mismatch. When an alternative prove to be wrong, the program have to backtrack and resume the matching from the next alternative.

The matching algorithm has exponential computational time complexity with respect to the input length. Several pruning techniques are introduced to reduce the complexity. For example, only the shortest $k = 4$ candidate patterns are evaluated for each mismatching.

In some cases, the system fails to generate patterns. For web pages include disjunction patterns, UFREs can not represent union and the system might fails to generate wrappers. One example is the music bestsellers on Amazon. Some of pages have customer reviews and some do not. The corresponding templates are different. The system can not generate a wrapper, and no information is extracted from this class of web pages.

6.3 EXALG

Recent work EXALG ([2]) has proposed another approach for automatically generating wrappers to extract data from a class of template-generated web pages. The overall procedure of this system is similar to RoadRunner discussed in previous section. It takes a set of template-generated web pages as input, generates a wrapper automatically, and extracts the data in those pages according to the wrapper.

EXALG and RoadRunner are different in wrappers generation approaches.

In EXALG, wrappers are derived from LFEQs (Large and Frequent EQuivalence class). The definition of LFEQ will be discussed later in this section. EXALG has two stages. The first stage ECGM (Equivalence Class Generation Module) takes a set of web pages as input, and a set of LFEQs as output. The second stage ANALYSIS takes the results of the first stage as input and output a wrapper. The wrapper is then used to extract data from the web pages.

The system is applied on different input collections of web pages, e.g. Amazon, E-bay, Google etc. The detail of the experiments and results can be found in URL [15].

Two key concepts are used in EXALG, differentiating roles and equivalence class. EXALG distinguishes tokens by their roles. The *occurrence-vector* of a token $(f_1, \cdots, f_n)$ is the numbers of occurrences of this token in each of n web pages. An *occurrence-path* of a token is the path from the root to the token in the parse tree. The same tokens usually play different roles, if they have different occurrence-vectors in a large number of pages or different occurrence-pathes. For example, `<b>` before `Book` and `<b>` before `Reviewer` of web pages in Figure 6.3 are tokens playing different roles, which are called *dtokens* (differentiated tokens). All input web pages are first represented as a set of dtokens.

An equivalence class is a maximal set of dtokens having the same occurrence-vectors. The *support* of a dtoken is the number of pages in which this dtoken occurs. The *support* of an equivalence class is the common support of its dtokens. The size of an equivalence class is the number of dtokens in this class.

An equivalence class with large size and support is called a LFEQ (Large and Frequent EQuivalence class). *The basic intuition behind LFEQs is that it is very unlikely for LFEQs to be formed by chance. Almost always, LFEQs are formed by tokens associated with the unknown template used to create the input pages.*

We show the procedure of EXALG through an example. Figure 6.3 shows four input web pages.

In the first stage ECGM, two LFEQs of dtokens are found,
ε_1={`<html>,<body>,<b>,Book,Name,</b>,<b>,Reviews,</b>,<ol>,</ol>,</body>,</html>`}
ε_2={`<li>,<b>,Reviewer,Name,</b>,<b>,Rating,</b>,<b>,Text,</b>,</li>`}
Notice that the dtoken `<b>` in ε_1 is a different dtoken from the `<b>` in ε_2. The two dtokens `<b>` in ε_1 are also different dtokens from each other. For ε_1, each dtoken appears exactly once in each of the web pages, so the occurrence-vector is (1,1,1,1). There are 13 dtokens in ε_1, so the size of ε_1 is 13. The LFEQ ε_1 occurs in all of the four web pages, so the support is 4. For ε_2, the occurrence-vector is (1,1,2,0), the size is 12 and the support is 3. The tokens in the LFEQs are ordered and ε_2 is nested in ε_1.

The second stage builds an output wrapper using the LFEQs constructed in the first stage. The wrapper is shown in Figure 6.4, where '*' represents the location of the data in these web pages.

The paper also describes several cases that the system failed to extract data correctly. For example, pages contain a set of addresses encoded with the template {`<Name:*<br>,(Email:*<br>)?,(Organization:*<br>)?,(Update:*<br>)?>`}. Because the type constructors associate with a very few tokens, this template can not be discovered as a LFEQ by the system.

6.4 Lixto

A visual web information extraction tool *Lixto* is proposed in [7], [4], [6] and [5]. *Lixto* generates wrappers semi-automatically by providing the visual interface and browser-displayed example pages that allow a user to specify the desired extraction

```
<html><body>
  <b>Book Name</b>Databases
  <b>Reviews</b>
  <ol>
    <li>
      <b>Reviewer Name</b>John
      <b>Rating</b>7
      <b>Text</b>...
    </li>
  </ol>
</body></html>
```

```
<html><body>
  <b>Book Name</b>Query Opt.
  <b>Reviews</b>
  <ol>
    <li>
      <b>Reviewer Name</b>John
      <b>Rating</b>8
      <b>Text</b>...
    </li>
  </ol>
</body></html>
```

```
<html><body>
  <b>Book Name</b>Data Mining
  <b>Reviews</b>
  <ol>
    <li>
      <b>Reviewer Name</b>Jeff
      <b>Rating</b>2
      <b>Text</b>...
    </li>
    <li>
      <b>Reviewer Name</b>Jane
      <b>Rating</b>6
      <b>Text</b>...
    </li>
  </ol>
</body></html>
```

```
<html><body>
    <b>Book Name</b>Transactions
    <b>Reviews</b>
    <ol>
    </ol>
</body></html>
```

Figure 6.3: Input Web Pages

```
<html><body>
   <b>Book Name</b>*
   <b>Reviews</b>
   <ol>
      <li>
         <b>Reviewer Name</b>*
         <b>Rating</b>*
         <b>Text</b>*
      </li>
   </ol>
</body></html>
```

Figure 6.4: The Wrapper

patterns. Internally wrappers are these patterns which are represented by a declarative extraction language called *Elog*. The extractor performs the actual information extraction from one or several similar structured web pages by interpreting the *Elog* program. The user then can use the *XML Generator* to map the extracted information to XML. To extract information in differently structured web pages, the user needs to specify the patterns for each of them.

There have been numerous other works based on HTML-aware tools [27], natural language processing [16], wrapper induction [23], object models [24], and ontologies [14]. For a detailed discussion of these and other works, the reader is referred to [25]. Suffice it to say that none of these provides an analysis of objective metrics such as coverage or quality.

The most related work is what we introduced in the beginning, DIPRE proposed by Sergey Brin. Experiments are conducted based on a large repository of 24 million web pages. He found 15257 books from 5 examples of books.

In comparison, our experiments gain 45034 books based on 16128 web pages. Two main factors contribute to this large number of books. The first one is that, screening via a search engine, we focus on the web pages only related to books. The second one is the different definition of the occurrence and different pattern generation strategy. In our method, the criteria for occurrences and patterns are more relaxed without increasing false discoveries. Once again, to the best of our knowledge, ours was the first to provide a detailed analysis of coverage of results extracted and quality of extracted information as well as an interactive approach for quality enhancement.

Chapter 7

Future Work

The current experiments are based on a local web page repository. In future work, our system will be based on a large scale and up-to-date web page repository. Consequently, we need to improve the performance of our program. Our long-term goal is to extract a large set of records automatically, based on several example records given by the user. Users will be allowed to specify the structure of the output XML documents. We also plan to allow extraction of data with more structure as opposed to just flat records. Finally, it is interesting to ask whether we can scale up such techniques to a level where information extracted is in response to questions posed by the user. Our ongoing research addresses some of these questions.

Bibliography

[1] Agichtein, E., and Gravano, L. Snowball: Extracting relations from large plain-text collections. In *Proceedings of the Fifth ACM International Conference on Digital Libraries* (2000).

[2] Arasu, A., Garcia-Molina, H., and University, S. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data* (2003), ACM Press, pp. 337–348.

[3] Ashish, N., and Knoblock, C. A. Semi-automatic wrapper generation for internet information sources. In *Conference on Cooperative Information Systems* (1997), pp. 160–169.

[4] Baumgartner, R., Flesca, S., and Gottlob, G. Declarative information extraction, Web crawling, and recursive wrapping with lixto. *Lecture Notes in Computer Science 2173* (2001), 21.

[5] Baumgartner, R., Flesca, S., and Gottlob, G. The elog web extraction language. In *LPAR* (2001), vol. 2250 of *Lecture Notes in Computer Science*, Springer, pp. 548–560.

[6] Baumgartner, R., Flesca, S., and Gottlob, G. Supervised wrapper generation with lixto. In *The VLDB Journal* (2001), pp. 715–716.

[7] Baumgartner, R., Flesca, S., and Gottlob, G. Visual web information extraction with lixto. In *The VLDB Journal* (2001), pp. 119–128.

[8] Brin, S. Extracting patterns and relations from the world wide web. In *Selected papers from the International Workshop on The World Wide Web and Databases* (1999), Springer-Verlag, pp. 172–183.

[9] Brown, M. C. *Perl: The Complete Reference.* McGraw-Hill Osborne Media, Berkeley, 1999.

[10] Burnham, K. P., and Overton, W. S. Estimation of the size of a closed population when capture probabilities vary among animals. *Biometrika 65*, 3 (December 1978), 625–633.

[11] Crescenzi, V., Mecca, G., and Merialdo, P. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases* (2001), pp. 109–118.

[12] Deitel, H., Deitel, P., Nieto, T., and McPhie, D. *Perl How to Program.* Prentice Hall, 2001.

[13] Eikvil, L. Information extraction from world wide web - a survey. Tech. Rep. 945, Norweigan Computing Center, 1999.

[14] Embley, D. W., Campbell, D. M., Jiang, Y. S., Liddle, S. W., Ng, Y.-K., Quass, D., and Smith, R. D. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowledge Engineering 31*, 3 (1999), 227–251.

[15] Extracting Structured Data from Web Pages: Experiments. *http://www-db.stanford.edu/ arvind/extract/.*

[16] Freitag, D. Machine learning for information extraction in informal domains. *Machine Learning 39* (2000), 169–202.

[17] Hammer, J., Garcia-Molina, H., Cho, J., Crespo, A., and Aranha, R. Extracting semistructured information from the web. In *Proceedings of the Workshop on Management of Semistructured Data* (May 1997), pp. 18–25.

[18] Hammer, J., McHugh, J., and Garcia-Molina, H. Semistructured data: The TSIMMIS experience. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems* (1997), pp. 1–8.

[19] HTMLParser. *http://htmlparser.sourceforge.net/.*

[20] HTML::Parser. *http://www.perldoc.com/perl5.6/lib/HTML/Parser.html.*

[21] HTML::TokeParser. *http://www.perldoc.com/perl5.6/lib/HTML/TokeParser.html.*

[22] Janevski, A. Universityie: Information extraction from university web pages.

[23] Kushmerick, N. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence 118*, 1-2 (2000), 15–68.

[24] Laender, A. H. F., Ribeiro-Neto, B., and da Silva, A. S. Debye - data extraction by example. *Data and Knowledge Engineering 40*, 2 (2002), 121–154.

[25] Laender, A. H. F., Ribeiro-Neto, B. A., da Silva, A. S., and Teixeira, J. S. A brief survey of web data extraction tools. *SIGMOD Rec. 31*, 2 (2002), 84–93.

[26] Lawrence, S., and Giles, C. L. Searching the world wide web. *Science 280* (1998), 98–100.

[27] Liu, L., Pu, C., and Han, W. Xwrap: An xml-enable wrapper construction system for web information sources. In *Proceedings of the 16th IEEE International Conference on Data Engineering* (2000), pp. 611–621.

[28] Otis, D. L., Burnham, K. P., White, G. C., and Anderson, D. R. *Statistical inference from capture data on closed animal populations.* No. 62 in Wildlife Monographs. Wildlife Society, October 1978.

[29] Perl Documentation. *http://www.perldoc.com/.*

[30] The Road Runner Project. *http://www.dia.uniroma3.it/db/roadRunner/.*

[31] Scheaffer, R. L., Mendenhall, W., and Ott, L. *Elementary Survey Sampling.* Duxbury Press, Boston, 1986.