Automated Extraction of Structured Data from HTML Documents

by

Maciej Stachowiak

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electircal Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Author .....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by .....
Boris Katz
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automated Extraction of Structured Data from HTML Documents

by

Maciej Stachowiak

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electircal Engineering

Abstract

In this thesis, I designed and implemented a system for automated extraction of structured data from HyperText Markup Language documents. The system is designed to flexibly parse a set of documents that have the same overall structure but differing data content. The system consists of three main components: a pattern matching engine which reads a document and a pattern template and outputs a set of bindings for the pattern variables in the template; a difference-detection utility which finds the differences between multiple documents in a set; and a graphical front-end that uses the difference finder to aid the generation of a template to match a set of documents. Several data sources on the World Wide Web were processed within this environment, showing a marked increase in productivity over fully human-generated ad-hoc parsers.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist

Acknowledgments

Boris Katz provided invaluable support and a large part of the inspiration for the work, and many suggestions on how to extend it. Deniz Yuret provided invaluable assistance by trying out and commenting on an early version of the system. Rebecca Schulman assisted in a survey of previous work in the area.

Contents

1 Introduction **9**

1.1 Motivation 10

1.2 Intended Use 11

1.3 Architecture of the System 12

1.3.1 The Pattern Matcher 12

1.3.2 The Difference Generator 13

1.3.3 The Graphical Interface 13

1.4 Pre- and Postprocessing 13

1.4.1 Preprocessing 13

1.4.2 Postprocessing 14

1.5 Realtime Use vs. Pre-Extraction 14

2 The Pattern Matcher **16**

2.1 Parsing 16

2.2 The Template Language 18

2.2.1 The `<variable>` Tag 18

2.2.2 `<optional>` and `</optional>` 19

2.2.3 `<repeated>` and `</repeated>` 19

2.2.4 `<docstart>` and `<docend>` 20

2.2.5 `<alternative>`, `<altitem>` and `</alternative>` 21

2.2.6 `<varstart>` and `</varstart>` 21

2.2.7 Nesting Requirements 21

2.3 The Matching Engine 22

2.4 Output 22
2.5 Possible Direct Applications 23

3 The Difference Detector **25**
3.1 Design and Output Format 25
3.2 A Clever Implementation 26
3.3 Use With the Matcher 27

4 A Graphical Interface **29**
4.1 Motivation 29
4.2 Design 30
4.3 Usage 31

5 Experiences with the System **32**
5.1 Generating Templates by Hand 32
5.1.1 The CIA World Factbook 32
5.1.2 The Internet Movie Database 33
5.1.3 Biography 34
5.2 Using the Difference Tool 34
5.2.1 Retrying the Old Sites 34
5.2.2 The All-Music Guide 35
5.3 Using the GUI 35
5.3.1 Test Cases 35
5.3.2 Amazon 35

6 Conclusions and Future Work **37**
6.1 Advantages over Previous Approaches 37
6.2 Shortcomings 38
6.3 The Regular Expression Paradigm 39
6.4 The Information In Differences 40
6.5 Future Work 40

A Template Language Operators **43**

List of Figures

List of Tables

A.1 Template Langauge Operators 44

Chapter 1

Introduction

The World Wide Web contains a great deal of information in the form of Hypertext Markup Language (HTML)[6] documents, but historically has lacked any kind of effective interface for accessing this data in a useful and unified way. We describe a novel system for extracting structured data from sets of similar HTML documents with a minimum of human intervention, potentially enabling the presentation of available data sources through powerful central interfaces such as intelligent search engines or natural language query systems.

This first chapter gives a general overview of the system and motivations for it. The system is comprised of three main parts, a pattern-matching engine, a difference-detecting utility that may assist in the generation of templates for the pattern-matcher, and a graphical interface for generating templates for the pattern matcher.

Chapter 2 gives a description of the pattern-matching engine's design and implementation, as well as discussing the possibilities for direct usage of this tool alone.

Chapter 3 covers the design and implementation of the difference detection tool, and discusses how it helps generate pattern templates for sets of similar web pages.

In Chapter 4 a graphical front-end that helps a user generate templates through the use of the diff utility and some heuristics is described.

Some experiences using the system in various ways are described in Chapter 5. Hand-generation of pattern templates is compared to use of the various automated

tools, and effectiveness of the system as a whole is studied.

Conclusions and possibilities for future work are discussed in Chapter 6. Although the task of extracting online data was much simplified compared to other methods while maintaining thoroughness and robustness, it would still be possible to make significantly less work for the user by putting more intelligence into the system.

1.1 Motivation

There are many sites on the World Wide Web that provide information about a particular topic in the form of web pages in a common format with what may be considered a number of several distinct fields. Such sites usually provide only the most primitive of search interfaces. Examples of such data sources include the Internet Movie Database [4] , Biography [8] and the CIA World Factbook [1]. These sites all provide a large set of HTML documents on a particular topic, each containing a portion of the information in a regular, structured way.

Traditional indexing techniques cannot support truly intelligent interfaces to these data sources, because much of the information each document, at least insofar as the relations between distinct data items are concerned, is contained within the layout of the document, not the text. It seems that to provide an interface similar to a relational database, where particular properties of given objects can be asked for directly (such as the director of a named film, or the capital of a particular country), it would be most fruitful to somehow parse these documents into their constituent fields.

The typical way of dealing with such data sources takes one of two forms, ad-hoc hand-generated parsers, and automated heuristic-based techniques. Writing an ad-hoc parser for each such data source is a fairly labor-intensive process. Each document's special layout must be handled individually. This approach has the problems that such parsers tend to stop working when the layout of the document set changes even slightly, and that it does not generalize very well, but it is guaranteed to extract all fields of interest in a nice format.

Another increasingly popular approach is to apply the techniques of data mining as generally used on databases, attempting to analyze the information itself without much specific human intervention. Much effort has been made to study applications of such approaches to various large and loosely structured world wide web sites, such as IBM's Olympics site[3].

We propose a novel approach to data extraction tuned specifically for dealing with such data sources, which tries to find a middle ground between the two general techniques described above. The core of the technique is a pattern-matching engine that can match an HTML document to a template. The template is matched as a pattern to the document; if the match succeeds, the pattern variables in the template are bound to corresponding fields in the document, and the result is returned for further processing. Such a template can be as thorough as hand-parser, but much less time-consuming to generate even by hand. Furthermore, other tools in the suite provide machine-assisted template generation; this reduces the overhead even more. It is possible that with sufficient work, the needed human input will be at some point quite minimal, perhaps limited to naming the various fields and making minor adjustments.

1.2 Intended Use

The principal target application for this work is the START natural language system (reference). This system has a relatively good ability to answer queries posed in natural English. However, the knowledge base available to START is somewhat limited. Recently, work has begun to leverage the numerous databases that are easily accessible via the World Wide Web; these data sources already exist and are nicely structured, and START would provide a convenient central interface for asking natural questions about a number of topics. The architecture to achieve this consists of several distinct projects, one of which is the current work.

In order to be able to answer questions about topics like films or famous persons, START needs to be able to understand things like movie titles, personal names and

dates on the lexical level. This requires a module to pre-process query text; identify such constructs, using lexical considerations, punctuation, and explicit lists of items in a particular category; and present these lexical constructs as tagged units that START can understand.

Second, START needs a database-like interface to make various queries about the relevant topics. Storing all the information directly in START's knowledge base would be infeasible. Thus, a need became clear to provide a uniform query interface for numerous data sources[7].

Finally, the information must originally come from somewhere and be made accessible to START in a useful form. That's where the data extraction system comes in. By allowing relatively easy and uniform parsing of online data into a simple structured form, the system can work with the query system in a number of ways to provide the actual information access.

1.3 Architecture of the System

The system is organized in three main parts, although they may need to be combined with other tools or "glue" scripts to provide some forms of needed functionality. These are the `webmatch` pattern matcher, the `webdiff` difference generator, and the `webtgen` graphical interface for generating templates.

1.3.1 The Pattern Matcher

The pattern matcher, `webmatch`, uses template files written in an HTML-like pattern language to parse fields out of HTML files. This pattern language is similar to regular expressions, but operates on the HTML element level and provides constructs for binding portions of the matched document as variables.

The program can operate on whole documents at a time, on sections of documents, or on collections of documents all at once. This flexibility allowed several different approaches to be investigated.

1.3.2 The Difference Generator

The difference generator detects differences between pairs of web pages and displays them in a useful format. This makes template generation easier, as it very clearly marks off the interesting, variable parts of the document set from the fixed parts. The output of the difference generator can be edited directly to generate a suitable template.

1.3.3 The Graphical Interface

The graphical interfaces uses some simple heuristics and the difference generator to assist in template generation. It also provides a convenient overall interface for editing templates.

The interface starts with a set of documents and finds the differences between them. It shows fixed parts and specially hilights the differing sections. A user can select one of these hilighted sections to view the pattern language fragment that was guessed for this section, and edit it if appropriate.

1.4 Pre- and Postprocessing

The data extraction system does not currently provide a complete end-to-end solution; various types of preprocessing and postprocessing may be needed in addition to the available tools.

1.4.1 Preprocessing

One preprocessing step that will be necessary in many cases is downloading the document or documents to be processed via a network protocol. Good tools exist to do this already, and combining them with the pattern matcher via a script seemed like a better and more general approach than adding specific support for downloading over the network. For example, the GNU "wget" program can download information from the Internet through several protocols. [cite]

Another possible use for preprocessing is to strip off extraneous parts of the document to make parsing easier. For instance, the beginning or end of each document could be clipped off, and they could be combined into one big file for faster processing; or non-standard tags that get in the way of parsing could be stripped out.

1.4.2 Postprocessing

Postprocessing may be needed for a number of reasons. The matching engine does not provide a built-in system for output formatting. This is because the possibility of optional or repeated fields makes a fully general output formatting system impractical, and the simple output format that is provided can easily be converted to a desired final output format by application-specific methods.

Another possible postprocessing step is the insertion of the data in question into a database, or otherwise provide it to some external program. This too would have to be handled by other programs or scripts.

One limitation of the system architecture is that it cannot process or split a single run of text that is not separated by HTML tags. This is a deliberate design decision; always operating on the level of HTML elements greatly simplifies the template language. However, there are some cases where a single run of text logically ought to be construed as more than one field, or where parts of a field should be pruned off to get a useful result. Again, application-specific postprocessing can be very useful here.

1.5 Realtime Use vs. Pre-Extraction

Once an appropriate template is generated, there are two possible general modes for using the system, realtime or offline. Each of these approaches has particular advantages and disadvantages.

Realtime usage would entail downloading and parsing a document whenever information from it is immediately needed. Such an approach is infeasible with hand-written parsers, because changes to the pages will break the whole system and require a long time to fix. However, it does have the advantage that if the information avail-

able is updated, the latest version will always be available. But if the target server is down or the document layout changes, there will still be downtime. True, it's easier to change a template than write a whole new parser, but until and unless this can be done entirely automatically, some downtime is bound to result. Also, this approach may be slow; it requires a transaction to a distant server followed by a fair amount of processing.

The other approach is to download all of the appropriate data into a relational database on local disk, using the parsing system essentially to process the data once for inclusion into the database, and possibly for periodic updates. This approach has the advantages of being fast and robust; there is no dependence on the external server staying up or providing documents in a particular format. Also, the timing of updates can be strictly controlled.

However, it may be considered a disadvantage not to have all of the latest data, especially for data sources like news sites or weather reports that change quite frequently. Further, it may require a huge amount of disk space to essentially duplicate the on-disk databases of a number of web-based data sources.

A caching approach is a possible compromise. It would balance speed against disk usage, and would provide something of a defense against remote server downtime. However, it could still be susceptible to failure when file formats change, and the cache would have to be carefully tuned to avoid giving stale results.

Chapter 2

The Pattern Matcher

This chapter discusses `webmatch`, the HTML pattern-matching engine.The pattern matcher parses an HTML document and a template wtritten in a special template language, and attempts to match the document to the template, while matching particular variables within the template. If the match is successful, bindings for the variables in the template result.

We describe here the parsing rules, the nature of the template language, and the implementation of the matching engine, as well as the output format.

Overall, the matching engine operates much like a regular expression matcher, such as the Unix **grep** utility. The template language is similar in power to regular expressions, but operates on the level of HTML elements rather than individual characters. Much of the power of the regular expression paradigm is leveraged effectively. Additionally, the ability to mark certain parts of the expression as variables to be bound allows separation into fields while matching the template.

2.1 Parsing

The pattern matcher parses HTML on a very shallow level. Individual HTML elements are identified and separated, but little attempt is made to interpret the elements semantically, or to consider their relative structure. There are three basic types of HTML elements considered by the parser, tags, comments and content.

Tags elements are most ordinary HTML tags, such as <p>, , or . A tag is delimited by the < and > characters, and starts with an initial tag type that contains no whitespace. Optionally, this may be followed by several attributes that are separated by whitespace from the tag identifier and each other. A tag has the form *name=value*. Optionally, the value may be surrounded by double quotes, in which case it may contain whitespace, or the characters = or > without the special meanings of these characters being invoked. The parser creates a data structure for each tag that includes the tag name and a set of name-value pairs that represent the attributes.

HTML has a special syntax for comments, even though they appear mostly like tags. Comments are delimited by <!-- and -->, and any internal < or > characters are ignored, as are any quote characters. For instance, "<!-- this is> a strange <comment -->" is a single HTML comment, not two tags surrounding some text. The parser specially handles this case, creating a tag with a name of "!-" and a single attribute named "comment" which contains the rest of the comment body. Whitespace in comments is canonified in the same way as that in content elements, described below.

A content element is simply a run of ordinary text not containing any tags or comments. In a content element, whitespace must be handled specially. In nearly all HTML contexts, any amount of any kind of whitespace is equivalent for purposes of layout, and leading or trailing whitespace between text and a tag is redundant. This means that to compare and match documents effectively, the whitespace in content elements must be modified to achieve a canonical representation. In particular, it seemed most convenient to strip all leading and trailing whitespace, and replace each internal sequence of whitespace with a single space character. This is in some sense the simplest format, and it has the side-effect of removing all newline characters from each content element, which turned out to be quite useful.

HTML has several additional requirements for well-formedness, for instance, some types of tags, such as , need to have a corresponding close tag, with the name prefixed by a slash, in this case, . Some other tags may only legally appear

in certain contexts. For example, <li> may only appear inside an appropriate list context, such as between <ul> and . However, the parser does not enforce any of this structure. Further, many documents actually found on the world wide web are not actually well-formed, so considering this structure would actually be counterproductive. Thus, the parser doesn't consider the actual semantics of the tag names, makes no attempt to make sure they are valid tags, and does not check for end tags corresponding to start tags.

2.2 The Template Language

The template language of the pattern matcher is essentially identical to HTML, but uses several tags that are not part of any HTML standard to represent the portions of the template that may vary between documents, and to mark the parts of a matched document that should be reported as variable bindings.

Here we describe the various special tags in the order they were added to the template language, along with typical use and the original motivation for adding each to the language.

2.2.1 The <variable> Tag

Initially, the only special construct of the template language was the <variable> tag. A template was essentially an HTML document with some elements or sequences of elements replaced by <variable>. A <variable> construct may match zero or more HTML elements; various attributes for the tag indicate what types of elements are allowed in the matching sequence.

This tag has a name attribute that is required for output purposes; if omitted, the variable binding will be ignored and not printed at all. For example, <variable name=my_favorite_variable> will result in a variable that will be named my_favorite_variable in the output.

Originally, a variable tag could only match a single content element in an HTML document. However, this turned out to be much too limiting. While it allows sepa-

rating out fields from a document set with an extremely rigid structure, where only the content varies and the tags remain exactly the same, it turns out that this was insufficient for many document sets. As a result, the option `allowed_tags` was added. Specifying a comma-delimited list of HTML tag names for this attribute allows for varying HTML tag constructs to be matched but ignored, or for fields to actually contain varying HTML formatting. Additionally, the `no_content` attribute may be set, indicating that only the specified tags may be matched, but content elements may not be. Finally, the `negate` attribute, if set, allows all types of tags *except* those listed to be matched; however, the meaning of `no_content` is not reversed in this case. Thus, `<variable name=foo negate=true>` will match an arbitrarily long sequence of any type of HTML element.

2.2.2 <optional> and </optional>

Limitations were soon discovered with the `<variable>`-only scheme: many document sets have optional sections that are present only in some documents of the set. We can't always just match them as a single variable, because often the section will contain one or more fields we want to parse out separately.

As a result, the `<optional>` and `</optional>` tags were added to the template language. Anything between `<optional>`/`</optional>` pair could be either matched in full or not matched at all, as long as the whole pattern is satisfied. This allowed a significantly greater range of document sets to be parsed successfully.

2.2.3 <repeated> and </repeated>

Often sections with a similar structure repeat an arbitrary number of times or a variable number of times in some range. Although a similar effect can be achieved by multiple sequential uses of the `<optional>` construct, in many cases it is much more clear to explicitly mark the possible repetition.

As a result, the `<repeated>` and `</repeated>` tags were added. Anything between `<repeated>` and `</repeated>` may be matched any number of times, from zero up

to a theoretically unbounded number. The optiona `min` and `max` attributes can set upper and lower bounds; for instance, `<repeated min=3 max=7><b></repeated>` will match between 3 and 7 instances of `<b>` in a row, inclusive.

2.2.4 <docstart> and <docend>

Originally, the system was designed so that the document as a whole, from start to end, had to be matched, or else the match would fail. This turned out to be a stifling restriction for a number of reasons.

First of all, it is often desirable to extract pieces of a document that appear in a particular context, but totally ignore everything else. This may simplify getting at the data that is actually of interest.

Second, trying to write templates by hand was very difficult when only the whole document could be matched. The matching system had no way to report where in the pattern the match failed, so the obvious way to work was to write partial templates that would attempt to match only the beginning of the document, and periodically test. But since templates could only match an entire document, it was necessary to truncate the documents being used for testing purposes to only contain as much as the template could match. This was extremely inconvenient.

To address both of these concerns, the default behavior was changed so that if the pattern matched any contiguous subset of the document, the match would succeed. The `<docstart>` and `<docend>` tags can provide some or all of the original behavior. `<docstart>` will only match successfully at the beginning of the document (but will consume no part of the input). `<docend>` does the corresponding thing for the end of the document.

By putting a `<docstart>` tag at the beginning of the template, one can force it to match only some prefix of the document being matched. This is useful for incrementally developing templates intended to match the whole document. And by adding `<docend>` at the end of the template as well, the template can be forced to match a whole document only.

2.2.5 <alternative>, <altitem> and </alternative>

At this point, the template language had almost the full power of regular expressions, but on the HTML element level rather than the character level. The only missing construct was the alternative operator, typically written as "—". This operator was not found to be needed often in practice; however, experience with ordinary regular expressions operating on text shows that this operator can be very useful in cases that are hard to handle otherwise. As a result, the <alternative>, <altitem> and </alternative> tags were added to allow sets of possible alternatives to be specified. <alternative> marks the start of a set of alternatives, <altitem> indicates the end of one alternative and the start of the next, and </alternative> indicates the end of a set of alternatives. Exactly one alternative must be matched.

2.2.6 <varstart> and </varstart>

In some cases, more precision was needed in describing structure of a pattern variable than just "an arbitrary sequence containing these elements". Often, the looseness of the possible constraints made it tricky to write a template that would match things in the desired way. As a result, <varstart> and </varstart> were added. Any valid template language constructs may be placed between these tags, and the total will be bound to a variable. The name attribute of the <varstart> tag specifies the name of the variable to be used. This construct also allows the possibility of nested variables.

2.2.7 Nesting Requirements

All matching pairs of start and end tags, such as <optional> and </optional>, must nest properly; that is, when a template language end tag appears, all start tags between it and its corresponding start tag must be closed. For instance, if an <optional> tag is followed by a <repeated> tag, the corresponding </repeated> must appear before the corresponding </optional>. This is necessary for the pattern matcher to have reasonable semantics, but is not currently enforced.

2.3 The Matching Engine

The core of the pattern matching engine is implemented as a finite state automaton which is essentially a greedy backtracking regular expression engine.

The "greedy" criterion results in the engine outputing the "leftmost longest" possible match. This means that, starting from the left, it will try to match as many elements as possible to the first construct, and then consider the next element to the right, and so on, and choose the leftmost member of each set of alternatives. At each possible choice point, the engine marks what choice it made on the backtracking stack. This becomes important when the greedy matching algorithm starts to fail, At that point, backtracking kicks in.

If the original greediest choice reaches a point where the current template element cannot possibly match the current document element, and it is impossible to move the position in the template forward otherwise, the engine skips back to the last choice point and matches one less element in a `<variable>`, repeats a `<repeated>` section one time fewer, skips an `<optional>` section, or takes the next branch of an `<alternative>` construction. If all choices for a choice point are exhausted, the engine backtracks still further. If the engine gets back all the way to the beginning of the document, it will try starting at the second element rather than the first and so on, as long as a match remains possible.

As the engine proceeds, it marks the start and end of the range of each variable on the backtrack stack. This allows variable bindings to be reconstructed during the output phase while adding little extra overhead or complexity.

2.4 Output

For the output stage the backtracking stack is first reversed to form a queue of the various choice points and variable range markings, in the order in which they were originally matched.

This queue is then traversed. Each time a `<variable>` or `<varstart>` construct is

encountered, the engine determines what set of elements in the document are matched by it. For <variable> this can be done immediately.

However, for <varstart>, more work needs to be done. The tags in the current variable being matched must be saved in a temporary stack until the corresponding </varstart> is encountered, since other template language constructs, including other nested <varstart>/</varstart> pairs, may be encountered.

When the full binding for a variable is finally determined, it is printed in the following format: first the variable name is printed, then the character "=". Finally, all of the HTML elements in the tag are printed in sequence. Ordinary tags and HTML comments are printed as usual, and any content elements are printed in the whitespace-canonified form.

This results in output that is very easy to parse. There is always exactly one binding per line (since all newlines have been replaced with spaces), and the name can be separated very easily from the value just by splitting on the "=" character. This allows for easy parsing and possible further processing by other tools.

2.5 Possible Direct Applications

It is possible to directly apply the pattern matcher by hand-writing templates in the template language to match the document set. In fact, this was the only use originally envisioned, before the graphical interface was designed. There are several options for structuring a template.

The engine may be used to match whole documents only. Appropriate use of <docstart> and <docend> can guarantee this. This is useful when it is necessary to ensure very accurate matching, and an appropriate warning if some document in the set does not match the expected structure exactly.

Alternatively, the template may be designed in the form of one or more context segments. A context segment is simply a <variable> construct with a few tags and sections of text on either side to identify what context it will appear in. Such a context segment has the advantage that it may do the right thing for a document

even if parts of it that are uninteresting to the user are in a different format than expected. Further, it allows less work. However, it may be less accurate; it may be that sometimes segments other than the one intended will be matched.

Also, as mentioned above, it may be useful to force a match at the beginning of the document while incrementally designing a template intended to match whole documents only.

Chapter 3

The Difference Detector

Although writing templates for the pattern engine by hand is already quite an improvement in productivity over writing completely ad-hoc parsers, it still involves an undue amount of manual labor. In particular, a human being has to look at a set of documents and determine what is the same about them, and what is different, in order to write an effective template.

However, the technology to determine differences between documents automatically is well known. The standard Unix `diff` utility (cite) does this, operating on the level of lines. Given two documents, it determines what lines would have to be deleted from or added to the first to result in the second. `wdiff` [5] operates similarly, but on the level of words.

A similar tool that operates on the HTML element level, `webdiff`, is desribed here, as is its applicability to generating templates for `webmatch`.

3.1 Design and Output Format

The desire was for a tool that generally works like `diff`. It should compare two HTML documents and show what changes would have to be made to the first to result in the second. However, most `diff` output formats are inappropriate for the desired application for `webdiff`, so a new format had to be designed.

Most `diff` output formats only show the actual points of difference between files,

and perhaps a few lines of surrounding context, generally as separate "hunks" which relate to one particular area of difference. However, for the output of "webdiff" to be maximally useful, it should generate one merged output that contains all the lines of the original input, and specially marks those parts that appear in only one document or the other. One simple way of doing this is to print tags one per line in the output, and prefix each tag with a space, a minus, or a plus, depending on whether it was present in both documents, present in only the first, or present in only the second. This captures the idea of tags being taken away from or added to the first document to generate the second.

Another important difference in how `webdiff` should work is that it should consider differences with HTML-element level granularity, rather than line-level. This can imply either larger or smaller chunks. For instance, if the text "<h1>Some Header</h1>" appeared on a single line, it should be considered as three separate HTML elements, not a single line. Conversely, if some multi-line run of text that contains no HTML tags or comments occured, it should be treated as a single HTML element, even though it consists of multiple lines. This requires the input to be parsed in quite a different way than the way a normal `diff` implementation would do it.

3.2 A Clever Implementation

Available diff implementations work quite well for what they do, and some, such as the freely available GNU `diff`[2], can generate a minimal set of differences between two files. It would be a waste to entirely reimplement these well-tuned algorithms for only two relatively small changes, a different level of granularity, and a different output format.

The solution is an implementation as a front end to GNU `diff`. This way, effort need not be wasted reimplementing the diff algorithms themselves, and only the key areas of input and output can be adressed.

The front-end first parses each HTML document on a shallow level in the same way as `webmatch`, stripping whitespace as usual. Each individual HTML element will

then contain no newlines. Also, the attributes are stripped from all HTML tags, since they are ignored when matching a document to a template. The elements are then written out one per line to a pair of temporary files. These files can then be compared by an external diff process, resulting in a listing of differences on the HMTL element level.

However, the output still needs to be parsed and reformatted. Fortunately, the "unified" output format of GNU `diff` makes this relatively easy. Each hunk of differences is prefixed by an indication of the range in each file that it corresponds to. The hunk itself consists of a set of lines prefixed by a space, a plus or a minus depending on wether they occur in both files, only the second, or only the first respectively. All tags up to start of the current hunk since that are past the end of the last hunk can simply be printed out directly, each prefixed by a space. The body of the hunk can then be printed out as-is; all the lines in it will already correspond to tags in the appropriate output format. The next hunk can then be processed, until no hunks remain. After that, all the tags in the first document after the last hunk can be printed out.

This results in the desired output format described above.

3.3 Use With the Matcher

Use of `webdiff` with the `webmatch` pattern matcher is very helpful in generating templates quickly and effectively.

Using `webdiff`, it is possible to take two representative documents from a document set and quickly generate a first pass at a template. `webdiff` is run on the two documents. The output is then edited to replace the portions that differed in the original documents, now marked with + or - characters. This eliminates the tedious searching for differences by hand, and the sections that need to be modified to get a working template are already marked.

In fact, the types of differences even give a hint as to what kind of template language operators should be used for various kinds of differences. Generally, if both

documents have some differing elements that are in the same place, a `<variable>` tag is most likely appropriate. However, if only one document has some extra sequence of elements in one particular place, an `<optional>` construct is probably called for.

However, there are still some drawbacks. The process cannot be done completely mechanically, as it is possible that `webdiff` may generate a set of differences that, while minimal, are not the most intuitive, and not the ones a human would intend. Also, there is no easy way to detect when `<repeated>` or `<alternative>` structures would best capture the differences between two documents.

Despite these drawbacks, it is fruitful to examine how much further the process could be automated with this approach.

Chapter 4

A Graphical Interface

Although `webdiff` makes generating templates for `webmatch` easier than doing it completely by hand, it still seems that there is more human labor in the process than there should be. Ideally, a program could examine a specified set of documents and automatically generate a template for parsing them, with no human intervention. Unfortunately, this goal seems to be impossible to achieve without much deeper understanding of the document contents.

However, it is possible to design a graphical interface which guesses as much as it can, and lets a human operator easily add the missing portions, or adjust the automatically generated results.

4.1 Motivation

In the ideal case, a complete template would be generated completely automatically, with no human intervention. However, this is not entirely possible. In order for a template to be truly useful for many applications, it must use meaningful variable names. There is no way generally to infer appropriate variable names solely from a document or set of documents. In some cases, important fields are appropriately labelled, but often they are not.

Some more practical issues that potentially could be overcome are also difficult to deal with properly. First, the diff engine can fail to detect differences in the way

that would be most meaningful in context. Usually the minimal set of differences is also the most useful, but not always. Second, differences between documents may be of several different types. A difference may be a meaningful field variable, or an optional section that may contain one or more variables. In some cases it may be best represented by `<repeated>` or `<alternative>` constructs. But it can be difficult to tell when these would be appropriate.

On the other hand, providing a good first guess for a human operator who fills in the variable names and adjusts the template through an easy to use interface would still be of significant benefit. Thus, a graphical interface was developed which uses the difference generator and some simple heuristics to guess as much as it can, and then allows adjustment.

4.2 Design

The basic design allows the user to launch the graphical front-end on a pair of files from the same document set. The program uses `webdiff` to find the differences and displays them, specially formatted, in a text window.

Contiguous sections that include lines starting with a plus or minus character, in other words, lines that indicate a difference between the two documents, are highlighted by being displayed in reverse video.

When the user clicks on a highlighted section, a dialog with several options is presented. The user may extend or shrink the highlighted section in either direction, or split it into two different sections. The user can also type in the template language constructs used for the section, optionally starting with a predetermined default.

Currently, the GUI interface only tries to detect opportunities for two specific constructs. If a highlighted section includes lines from both documents, it is replaced by default with a `<variable>` tag that has an `allowed_tags` attribute which allows all the tags that appear in that section to be matched. When this heuristic is appropriate, the user just needs to add an appropriate variable name. Alternatively, if a section contains lines from only one of the two documents, the system tries to surround it

with `<optional>` and `</optional>` by default. In this case, it may be necessary to mark some internal portions as a variable if that is the intended use.

4.3 Usage

Use of the graphical interface is generally even faster and more convenient than starting with the results of `webdiff` and editing them by hand. The sections that need to be changed are clearly highlighted, and good initial suggestions are given for what to replace them with.

However, it may make it more difficult to generate a very accurate template in some cases. If the set of differences detected is not the most intuitive and appropriate one for the document set, a user who blindly accepts the program's suggestions is not likely to get the result he really wants.

Chapter 5

Experiences with the System

Using the data extraction system on several data sources demonstrated a number of benefits and shortcomings of the system. At all times, productivity was much higher than when writing ad-hoc parsers by hand. A number of improvements made in the course of testing, and the possibility of a number of knowledge base applications was seen. The `webdiff` and graphical interface tools were also found to be quite helpful.

5.1 Generating Templates by Hand

Generating templates by hand turned out to be feasible, but a bit more time-consuming than initially expected. This was largely because the size of a typical data-containing web page was gravely underestimated, and because simple-minded techniques were applied initially.

5.1.1 The CIA World Factbook

The first template attempted was for the CIA World Factbook site. The CIA World Factbook provides numerous types of information about the nations of the world. Experiences here led to a number of improvements in the program and in the methodology for applying it.

The first attempt at writing a template for the factbook was using the `<variable>`

operator only. This failed, because a number of documents had sections that were simply not present in others, but that contained potentially useful information.

As a result of this, the <optional> operator was added to the language. This allowed the rest of the template to be generated effectively. The effort was somewhat time consuming, as the typical document was very long and repetitive.

It was noted, however, that the general layout of the document was very consistent from section to section - typically, something that could be taken as a filed name would be found in a particular structural relationship to the appropriate filed value. As a result, a much shorter template was written which used the <repeated> operator - this eliminated all concern about optional fields, since all fields were treated generically, and thus all were effectively optional. Although this approach worked in this particular case

5.1.2 The Internet Movie Database

The Internet Movie Database was a lot less regular than the CIA World Factbook, so the trick of using <repeated> to parse most of the document was not available. Also, this more complex structure indicated that it would have been much harder to write an ad-hoc parser for this data source. Nontheless, a pattern matching template did not take very long.

Although the whole document could not be processed with a single simple <repeated> construct, there were several places where particular sections repeated an arbitrary number of times with different data. Thus, the usefulness of <repeated> was demonstrated in this context too.

Some information pertaining to a particular IMDB entry was found in separate HTML documents linked to from the original. This indicated that a more complex approach that was able to parse out these additional URLs from the main document, and download and parse the additional documents, would be even more useful.

5.1.3 Biography

The Biography site provides biographical information on historical figures. It was the first document set to which someone other than the original author successfully applied the pattern matcher, demonstrating that the template language is not entirely idosyncratic.

A somewhat different approach was taken for this data set than the other two. All the data was initially put into one big file, mildly preprocessed by a perl script to strip off irrelevant headers and trailers and remove some invalid HTML tags. Then, a template matching a single entry was surrounded with a `<repeated>` construct and matched to the document.

The first pass at a template took on the order of a few minutes once the data was loaded and preprocessed. This parsed a relatively small fraction of the data successfully. However, after a few simple incremental changes to the template and a few bug fixes to the parser, the whole data set was processed successfully.

This was again a much more efficient way to work than doing everything by hand.

5.2 Using the Difference Tool

Applying `webdiff` to several sets of pages showed that this made generating parsers even easier. The output could be almost directly translated into a suitable template.

5.2.1 Retrying the Old Sites

The first real test of `webdiff` was to try it on some of the sites that templates had been written for already, and see if the resulting set of differences corresponded well to what had already been chosen as the significant fields.

This worked best with the Interned Movie Database, which had a fairly complex and non-homogenous layout, and enough structure to match to. Biography and the CIA World Factbook did not fare quite as well, but overall, the results seemed to correlate fairly well with the handcrafted templates.

5.2.2 The All-Music Guide

A new document set, The All-Music Guide, was then tested. The results were a moderate success. The fields were marked off fairly well for subsequent editing, but many of the fields were essentially lists of a variable number of elements which could only be properly handled with a `<repeated>` construct. The specific differences between two documents gave no help here.

5.3 Using the GUI

The graphical interface showed some promise during testing, but did not overwhelmingly improve productivity, and indicated a need for further work to fulfill its full potential.

5.3.1 Test Cases

The graphical interface was first tested on simple, handcrafted test documents. It managed to deal with these relatively well, picking out the intended differences and highlighting them. However, the test documents were extremely simple and to some degree designed to work well with the available heuristics, so this is not surprising.

5.3.2 Amazon

The Amazon.com online bookstore was then tested. Although primary a catalog and point of sale, the site has a fair amount of information about each book listed.

The GUI detected and hilighted all of the key differences between the documents. However, much of the typical Amazon.com entry is not useful, as it provides either unimportant marketing and sales information, or reviews which are relatively useless for knowledge base purposes. Thus, a lot of modification had to be done to the GUI's original guess template.

Although the graphical interface may provide some level of comfort for new users, it did not turn out to be very helpful for dealing with the system's potential com-

plexities.

Chapter 6

Conclusions and Future Work

The HTML data extraction system proved itself to be a remarkable improvement over previous technologies. It is much more suitable for use in redirecting data sources through intelligent knowledge base interfaces for a number of reasons.

However, the system still has notable shortcomings, both in the original design and in the implementation. This leads to a number of areas for potential future improvement. Further, a number of interesting conclusions were reached about processing structured documents.

6.1 Advantages over Previous Approaches

Compared to the previous approach of writing a specific ad-hoc parser for each document set, the methodology enabled by the data extraction system is much improved in a number of ways, while extracting data as precisely and thoroughly as completely handmade tools.

First of all, development of parsers for new document sets or for changed versions of existing ones is much faster. While writing a parser for even the relatively simply structured CIA Factbook document set took about two hours under the old methodology, under the new system it can take as little as fifteen minutes, depending on the complexity of the data source, even when writing the template completely by hand. Although this seems like a relatively small gain, it makes a big difference in practice.

It is much more practical to interface to a large number of data sources, and greater flexibility results from the faster turnaround time.

The increased flexibility of the system is also a great benefit. Previous parsing tools were set up to expect the input to be arranged in a particular way, for example, for all of the documents to be concatent into one big file and be processed as a single batch job. The new system separates out the parsing functionality itself from such considerations through a clean, modular separation around the parsing engine. This allows the approach being taken with a particular document set to be changed with very little effort.

The rapid turnaround and flexibility also makes practical the use of the system in a "realtime" way - downloading data off the net and parsing it as needed, which was not feasible before. The ability to parse a single document very fast, rather than operating in batch more, was an imporatant consideration. Also, the reduced turnaround time makes the possibility of the data source changing much less of a threat. System downtime would be minimized to the time needed to make and test a few relatively small changes.

In addition, the new parsing system is potentially more robust. Appropriately written templates can actually ignore much of the fixed structure of the document and locate data fields by looking for very specific contextual hints. This means that unexpected differences in one document or a change in the whole set are more likely to be handled gracefully.

6.2 Shortcomings

The system still has a fair number of shortcomings, both in the actual functionality offered by the system, and in the amount of human labor still needed to keep it working.

One major shortcoming, for which there is no particularly good workaround, is that the system can't parse within a single content element; separating fields can only be based on intervening HTML tags. There are cases where this is very desirable.

For example, the coordinates of a city may be given as a longitude and a latitude, separated by a comma. It may be desirable to separate this into two distinct fields, but the current system cannot do that by itself. Instead, a specific ad-hoc postprocessing step would have to be added. Similarly, in some cases it may be desirable to remove some extraneous parts of a file entirely before using it. Perhaps in some cases there is no tag between a redundant label and the varying information. Or perhaps the URL needs to be parsed out of an HTML `<A>` tag in order to download specific additional documents.

Other problems are more easily addressed. For example, the need for "glue" scripts to take a request, download the appropriate document, run the parsing engine, and do something appropriate with the results, is not a big problem. In fact, several sample scripts that could easily be generalized for particular forms of use have been developed in the course of working with the system already.

But another big problem area is that even when using the graphical interface, a great deal of human input and judgement is needed to get a good pattern template. This is partly becuase of the current limitation that the system only compares two documents at a time, and does not have very advanced heuristics for figuring out how to parse them.

6.3 The Regular Expression Paradigm

A major point of investigation when developing the system was how applicable the regular expression paradigm was to parsing structured documents, considering things on the structured element level rather than the raw text level. It seems clear from the documents actually tried that it works quite well.

None of the documents required more than the power of regular expressions to match the whole document set. Further, while in a very few cases multiple fields were found in a straight run of text, causing the HTML element level of operation to be a problem, in general the regular expression idea matched very well the sorts of differences found among similarly structured documents.

In fact, it became clear that most document sets can be parsed quite simply - templates that work can be generated with just "variable" and "optional" constructs. And "repeated" handles most other cases of interest. This indicates that a very simple pattern language can be a very powerful tool for sifting through and organizing the information found in structured documents, just by taking advantage of the structure of the document itself.

6.4 The Information In Differences

Another key idea that has been demonstrated by this work is that a lot of information in a set of documents lies in what is different about them, and what is the same. The structures that do not vary are unlikely to be very informative, while those that differ from one document to another are highly likely to contain the useful information.

This idea seems sound both on the level of intuition and information theory. Indeed, this point was almost taken for granted when developing `webdiff` as an aid to generating `webmatch` templates. But the fact that just finding the differences between two documents using a computer program and semi-mechanically making some simple substitutions for the different parts allows you to distinguish and extract the interesting and useful information they contain is, in retrospect, a remarkable demonstration.

6.5 Future Work

Several directions of future work are seen. First, several of the limitations of the current system which prevent it from parsing documents in the desired way should be adressed. Second, more intelligence should be put into the system, reducing the burden on the user.

The need to postprocess various fields, either by removing extraneous parts or by splitting one field into many, is quite clear and should be adressed by the system directly. While making the system parse content elements one character at a time

rather than at one go may seem like a solution, it would violate the intentional and useful design decision of always parsing on the HTML level. The most reasonable approach to achieving field postprocessing seems to be to allow a `variable` tag to specify one or more special postprocessing expressions, along with associated names for the output variables for these. Key oprations that must be supported are matching by (conventional) regular expressions and pattern substitution. A subset of the `sed` text processing language may be appropriate here, in particular the "/" and "s/" operators.

End-to-end frameworks for downloading a data off the web, given a set of URLs or an algorithm for generating them, either live to get a specific field or fields from an individual document, or into a database for later offline usage, would be quite useful. A little glue code in the Perl language can typically handle this, but pre-made frameworks for common cases would still be helpful.

A lot of additional work can also be done to add more intelligence to the system to minimize human work.

Better heuristics in graphical interface would be one way of approaching this goal. Examining many documents rather than just the current two at a time, and properly integrating the results would allow for much more intelligent guesses about appropriate template structures. For example, nested optional sections, optional sections that contain variables, repeated structures, and variables not likely to contain interesting information could all be detected much better.

It would also be helpful to add the ability to test a template on a specific document from within the graphical interface, possibly providing suggested fixes in case of failure. Ultimately, testing on the full document set should be a standard part of the development process.

Another possible application of more artifical intelligence so that less natural intelligence is needed, is automatically detecting when a page format changes when using the system in realtime mode. This is partly a matter for the driver script, which should report a problem if documents from a given data source repeatedly fail to parse. But it may be even more useful to try to have a suggestion ready immediately by

comparing the pages to find their new differences, or perhaps comparing some saved data to a page in the new layout.

Appendix A

Template Language Operators

Table A.1: Template Langauge Operators

Operator	Attributes	Description
`<variable>`		Matches and binds a sequence of HTML elements.
	`name`	Specifies the name used for the variable in the output.
	`allowed_tags`	Specifies the HTML tags that may appear (as a comma-separated list).
	`no_content`	If specified, content elements may not appear.
	`negate`	If specified, any tags *except* those in allowed_tags will be allowed
`<optional>`		Begins an optional section.
`</optional>`		Ends an optional section.
`<repeated>`		Starts a section that may repeat.
	`min`	Specifies the minimum number of repetitions. (The default is 0.)
	`max`	Specifies the maximum number of repetitions. (The default is an unlimited number.)
`</repeated>`		Ends a repeated section.
`<docstart>`		Matches only at the start of the document.
`<docend>`		Matches only at the end of the document.
`<alternative>`		Starts a group of possible alternatives.
`<altitem>`		Separates alternatives in a set.
`</alternative>`		Ends a group of possible alternatives.
`<varstart>`		Starts a section that, when matched will be bound as a variable.
	`name`	Specifies the name used for the variable in the output.
`</varstart>`		Ends a range to be bound as a variable.

Bibliography

[1] Central Intelligence Agency. Cia world factbook. On the world-wide web at http://www.odci.gov/cia/publications/factbook/index.html.

[2] Paul Eggert David MacKenzie and Richard Stallman. *Comparing and Merging Files: Edition 1.3, for diff 2.5 and patch 2.1.* Free Software Foundation, 1994.

[3] Sara Elo-Dean and Marisia Viveros. Data mining the ibm official 1996 olympic web site. Technical Report 20714, IBM Research Center, Yorktown Heights, NY, February 1997.

[4] Mark Harding et al. Internet movie database. On the world-wide web at http://www.imdb.com.

[5] François Pinard. Wdiff manual.

[6] Dave Ragget, Arnaud LeHors, and Ian Jacobs. Html 4.0 specification. W3c recommendation, World Wide Web Consortium, April 1998.

[7] Levent M. Talgar. Modular knowledge integration interface for the start natural language engine. Master's thesis, Massachussetss Institute of Tehcnology, 1998.

[8] A&E Television. Biography. On the world-wide web at http://www.biography.com.