ABSTRACT

Data Extraction from Structured HTML Sources

By
Alexis Winston
Masters of Computer Science
California State University Chico
2004

The Tree Mapping System (TMS) makes use of a template to automatically extract data from a set of HTML documents sharing a common structure. This template is generated in a semi-automated manner with the user providing example documents from the document set. The user then marks regions of interesting content to be extracted and the system creates a template encoding the document structure. During extraction the system maps the template onto the documents in the set to locate the target data.

TMS employs an original mapping algorithm which calculates the similarity between nodes in trees representing the documents by comparing the node properties and the tree structures. This algorithm finds the mapping from the nodes in one tree to the nodes in the other which maximizes this similarity measure. The mapping algorithm is used to generate a template from a set of example documents, as well as to locate repeated regions in a single document, and finally to map a document to a template during extraction.
Data Extraction from Structured HTML Sources

Alexis Winston
CHAPTER I

INTRODUCTION

Background

The increased use of automation to assist in information processing tasks seems to have the potential to enable more efficient work. Automation can help reduce the amount of time and human effort required to perform tasks such as data entry, search, categorization of documents, and the aggregation of information from various sources. These tasks have components that clearly do not require human level intelligence to complete.

Data entry tasks where data is copied from one format to another may be automated by the creation of a script using knowledge of both formats to locate data in one format and convert it to the other. Human intervention may still be required to provide the understanding of the document semantics needed to create the script, and also to validate the results to detect mistakes.

Automation already plays a key role in the area of web search. Search engines use automation to help guide users to pages that may contain the information they are looking for. The user will typically still have to search through the results and associated pages manually to locate the actual information they are looking for. In all of these areas there is still room for the development of more intelligent agents to assist in the performance of these tasks.
One aspect that many of these tasks have in common is that they involve manipulation of information contained in human readable documents. These documents are designed to be read and understood by a human. They are difficult for a system without knowledge of the semantics of the document’s structure to work with.

It is important to recognize the distinction between the semantics of the document’s structure and the semantics of it’s content. Structural semantics include information such as the relationship between the rows and columns of a table, or a heading to the information that follows. Content semantics refers to the information contained in the document’s content itself. Human readers can infer the structural semantics of the document from established conventions, such as tables, repeated patterns within the text, and from an understanding of the content. In a sense the structural semantics represent the information contained in the structure of the document itself that may be useful or even necessary in understanding the content of the document. They can also help in organizing that content so that the reader can easily locate the information they are looking for.

In the example of the data conversion script, the writer of the script would need to understand the structural semantics of both formats and use this knowledge to create the script. The script would need to make use of the knowledge of the information contained in the structure of the first document to locate the field to be extracted and also use the structure of the second document to locate the field where the data should be placed.

One potential solution to the difficulties of working with human readable documents is to include machine-readable versions of the documents. Approaches such as
extensible markup language (XML) [1] seek to do this by isolating the content of the
document into fields that are easily located by the system and including templates to
render the information in a human readable form. Of course, XML by itself only provides
a common syntax. For a set of documents to be read by an automated system they must
all share a common set of structural semantics that is known to the system. The XML
syntax helps with this by making it easy for a provider of a document set to isolate the
structural semantics from the content by placing this information in the XML tags.

A good example of using machine-readable formats in automation is Rich Site
Summary RSS feeds [2]. RSS is a standard set of structural semantics, expressed in
XML, which allows web sites to provide XML versions of their content that can be read
by an automated system. These automated systems can then gather these RSS feeds from
many sources and combine them into a single document to present to the user. This can
make it easier for a user to monitor news from many sources using only a single
application.

Unfortunately, machine-readable documents may not always be a realistic
option. If the documents are created by a number of different authors or organizations
they may not have an incentive to provide machine-readable content. Additionally, legacy
documents may not have been created in a machine-readable format. A system capable of
dealing with human readable documents would be able to potentially extract data from a
wider range of sources.

Data Extraction

The focus of this work is on the problem of data extraction from human
readable documents, specifically from HTML documents. This paper describes the Tree Mapping System (TMS) which extracts data from HTML sources. There are numerous HTML documents distributed on the Internet. Some areas of interest include: web based discussion board, online stores, web based email clients, and online news sites.

HTML provides a syntax for explicitly defining the structure of a document that makes it possible to analyze that structure in a direct manner. In some ways this is similar to the role of XML; however, the HTML tags do not explicitly contain useful semantic information about their contents. Rather they contain information about its display that a human reader can use to infer the structural semantics.

The same approaches that apply to extracting data from HTML sources may also be applicable to other structured sources as well. For example, a scanned copy of a printed form potentially contains a structure similar to such a form in an HTML format. The difficulty is that there is no explicit representation of the structure of the document to draw on. A system might make use of a preprocessing step to analyze the layout and appearance of the document to construct such a representation.

This work also focuses on structured documents. It is certainly possible to create an HTML document that contains little or no structure; however, many documents do contain a great deal of structure which can aid in data extraction. Structure plays an important role in a human readable document. It can help readers locate and understand content more quickly, and place that content in a useful context. For example a field labeled “author” on a message board posting helps the reader understand the meaning of the contents of that field. The same information could be conveyed in an unstructured form by a line such as, “the author of this post is…” The structured version makes it
easier to find the information quickly. This is especially true if there are many documents sharing a common structure, or if the structure makes use of a common convention the reader has encountered before.

Structure can also aid in the generation of documents. Sets of structured documents are generally created by a person, or software, or a combination of both, following a template and filling in the information. This simplifies data entry and helps avoid mistakes or omissions and in the case of software it makes it possible to generate dynamic human readable documents from a database. If the data extraction agent does not have direct access to the machine-readable information in the database, the process of reading the human readable versions could allow the database to be rebuilt.

With structured documents, a data extraction system can attempt to separate the content from the structure. It can then analyze the structure for semantic information that can be useful for locating specific elements of the content. Finally, the content can be extracted and marked with the semantics inferred from the structure.

Approaches to Data Extraction

From Web Pages

One classic approach to data extraction is to use regular expressions to represent the structure of the document and locate content sections [3]. Regular expressions can be used to define complex text patterns for a wide variety of extraction tasks. Regular expressions have an important advantage in that they do not depend on the HTML tags in the document. Patterns that are contained purely in the text of the document can be exploited. However, they do have a number of disadvantages. First, they...
typically have to be designed by hand for a specific document set. The programmer will have to examine the raw HTML code of a representative document to look for patterns that can be used as boundary markers around the desired content. This can be a time consuming process especially when working with complex documents. Another limitation is that regular expressions do not make any attempt to separate the document structure from the content or to take advantage of the inherent semantics of the HTML itself. Small variations in the document structure can cause mistakes, or the patterns could potentially match against text in the content area itself if they are not carefully constructed.

Another approach to data extraction from HTML documents is to use the information in the HTML tags of the document to help determine its structure [3]. The nested HTML tags can be parsed to form a tree. This is referred to as a document object model (DOM) tree [4]. Each tag also contains various properties that are attached to the node it represents in the tree. This tree construct provides a convenient method for representing the structure of the document and comparing differences between pages. Also, since the ordering and properties of the tags control the layout and appearance of the document when it is rendered into human readable form for the user, there is a direct connection between the structure presented to a human reader and the tree construct analyzed by the extraction system.

There may still be a useful role for regular expressions in an HTML-based system. Portions of the structure of an HTML document are sometimes contained only in the text between tags. For example, consider the following piece of HTML: “<TD>Name: Sam</TD>”. The text “Name:” is part of the structure of the document, while the
name “Sam” is the content. Regular expressions are capable of finding patterns such as this. Since the DOM tree deals only with the tags and treats text as a property of its containing tag, regular expressions could be used to match patterns that are hidden from the HTML system. The disadvantages of regular expressions are ameliorated somewhat in this case because the body of text to be searched for a pattern is greatly reduced to only the text contained between a pair of tags.

In addition to the decision of how to represent the document structure, it is important to consider how that representation will be generated. Creating templates by hand is time consuming and prone to errors. The system presented in this work provides automation to assist in the generation process. In general, there is a trade off between automation and generalization. The less semantic knowledge the user is required to provide and the more assumptions the system is required to make, the more difficult it becomes to create a general system capable of handling the largest set of documents. An ideal system would obtain from the user the minimum required semantic information about the document set in order to map it, and automate as much of the task as possible. Especially important is the ability to deal with minor changes in structure and noisy data with minimum supervision.

Finally, it may be useful to consider the semantics of the document’s content as well. Certainly human readers use their understanding of the document’s meaning to help understand its structure. Although it is beyond the scope of this work, it might be possible to use natural language processing techniques on the text of the document to help map the structure. This would be especially useful for any fully automated system.
Limitations

This work focuses on the problem of data extraction, which is a useful first step to prepare data for a variety of other interesting data manipulation tasks. It deals with the issue of reading human readable HTML documents that are fairly well structured. Templates are generated with a semi-automated system assisting a user.

Usage

There are a variety of possible uses for TMS. For example, it could be used by an email client to interface with a web based email system. TMS would be used to create a template for the web-based system, and then that template would be used to extract the email. TMS could also be used to track prices at an online store or create a database of news stories from a news website. It could be used to monitor a web forum, or extract data from an online auction site for data mining purposes. TMS is useful in any situation where automated interaction with a set of structured web pages is needed.

Terms

- HTML: HyperText Markup Language is a standard syntax for defining the appearance and structure of a human readable formatted text document. HTML documents are commonly used to present information on the Internet.
- Document object model (DOM): the tree used to represent the structure of an HTML document.
- Template: a model representing the common structure of a set of documents that can be used to locate and extract the contents of the document.
• Structural semantics: the information contained in the structure of the document itself, such as the relationship between rows and columns of a table.

• Structure / Content: Structure represents how a document is displayed while content represents the information within the document. Structure is generally common across a document set while content varies greatly from document to document.

• XML: A standard syntax for defining document structure [1].

• RSS: A standard document structure defined in XML used for creating machine readable documents [2].

• Regular Expressions: Regular expressions are a method of defining a complex text pattern to be located within a body of text. For example, the pattern [0-9]+ would match a number of one or more digits.
Overview

There have been a number of different approaches to data extraction from HTML sources in the past few years. Many of them make use of some sort of assisted learning algorithm to generate a series of rules that can be applied to the documents in the set to perform the data extraction operation. However they use a variety of different features within the documents, different rule representations, and different learning algorithms to accomplish the task. A few of these approaches are presented here.

Related Work in Data Extraction

A system developed by Rahardjo and Yap makes use of the concept of “minimum edit distance” to detect differences between a pair of example HTML documents [5]. This system does make use of HTML structure, during an initial pass through the document, by treating HTML elements as tokens in the edit distance calculation. In this way it can detect elements which have been added, removed, or replaced. After performing the initial pass it does a more fine grained comparison where the text in individual matched HTML nodes are checked for changes, again using the “minimum edit distance” measure.

The “minimum edit distance” for a pair of strings is calculated by counting the minimum number of operations needed to transform one string into the other. These
operations can be: the insertion of a token, the removal of a token, or the replacement of
one token by another. For the first pass the tokens are entire HTML elements. Once areas
of difference are found the second pass uses individual words as tokens. Once the
differences between the documents are located both documents are displayed to the user
side by side with the different areas marked. The system assumes that some of these
differences will contain the interesting content the user is after. The user then selects
which marked differences are actually relevant, and those differences are used to perform
the extraction.

The approach taken by Rahardjo and Yap is similar in some ways to TMS. First, they do make use of HTML structure by using HTML tags in their “minimum edit
distance” calculation. They also do an automated analysis of a set of documents. Finally,
they allow the user to mark the content to be extracted. However there are also some
important differences. They do not make use of repeated patterns within the text as TMS
does with the concept of regions. The ability to define regions is very useful when
creating extraction templates with TMS. Also, their “minimum edit distance” is used to
locate differences between documents, while TMS uses a similarity measure to locate
common structure between the documents.

The NoDoSE (Northwestern Document Structure Extractor) system ignores
the semantics of the HTML structure and looks for patterns in the raw text itself [6]. The
theory here is that extracting data from non-HTML text sources is harder, and that a
system that can function on any text document will be able to handle HTML as well.
NoDoSE uses a graphical user interface to allow users to build a grammar to define the
document structure by marking up an example document. As the grammar is built,
NoDoSE applies it to the rest of the document, and if NoDoSE incorrectly marks part of the document the user can correct those mistakes, which updates the grammar. This process of using the existing grammar to mark up the remainder of the document as the grammar is built up, allows the user to avoid marking up the entire document in some cases.

NoDoSE creates a tree representing the structure defined by the user and then generates potential rule sets that might generate the example tree from the example document. These rules are evaluated for errors by NoDoSE and the best one is applied by NoDoSE to the rest of the document or other documents. The user checks the output of the parse and changes any errors, creating a new example. New rules can then be generated and evaluated by NoDoSE, and later the user, against the new example.

NoDoSE is designed to deal with any generic text format, not specifically HTML. This gives NoDoSE the ability to extract data from non-HTML documents that TMS does not work with. Both NoDoSE and TMS use a semi-automated approach where the user makes use of a GUI to generate a grammar or template for extraction. However, TMS is able to use the more explicit structure of an HTML document to assist the user in generating a template for the document.

The WhizBang Labs Wrapper Learner is a hybrid system that makes use of both text and HTML structure and also analyzes tabular data found in the page[7]. This system uses independent agents called “builders” to analyze each of these elements of the document and then uses a master learning algorithm to combine the results of the individual “builders”. If the rules generated by any one of these “builders” are capable of correctly parsing the document set by themselves then those rules are used; if not, those
rules are used in combination with other rules from different “builders”. The example documents in the training set are initially marked up by the user to indicate regions to extract.

One interesting aspect of this system is the way it makes use of the page’s appearance as it is displayed to the user. This “builder” is only used on tables, and only on tables that contain tabular data, as opposed to HTML tables that are used for document layout purposes. Instead of viewing the table in its HTML tree structure, it considers the table as a grid of rows and columns. The “builder” also takes the step of replacing the explicit HTML tags that control the text appearance with a representation of that appearance. This is to account for different HTML tags that can result in the same appearance. The “builder” uses this information to create rules for extracting data from the table.
CHAPTER III

Methodology

Overview

The approach presented in this work is based on generating a template for the document set. Conceptually, the template represents the common structure of the documents in the set with the actual content of the documents removed. The process of extraction is then reduced to mapping the nodes in the document to the nodes in the template. The locations of interesting content are stored in the template and after the mapping process, this content can be retrieved. This process mirrors the creation of dynamic web documents where data is taken from a database and used to fill out a template to create the final document, which is then presented to the user.

Structurally, these templates are trees composed of nodes similar to the nodes in the document’s DOM tree. There are two basic types of nodes. Element nodes represent HTML elements in the document. Region nodes represent a repeated pattern in the document, which allow the template to represent substructures in the document that can occur a variable number of times. An example of such a document would be a list of objects. Each object in the list is an instance of the region representing the object in the template.

TMS must perform two separate, but related, tasks: generation and extraction. Template generation is the process of creating a template for a document set
from a number of example documents. A basic template is generated from the example set and then a graphical interface allows a user to select repeated regions and mark content for extraction. Data extraction is the process of using the template to extract data from other documents in the document set.

The mapping process itself makes use of multiple layers of templates, ranging from a very specific representation of the DOM tree of the document to a general master template for the entire document set. Only this master template is actually stored. The other templates are used as temporary processing structures while mapping the document, either in template generation or data extraction. There are four layers in all and each is linked to the template above and below it. Each node in a lower layer is mapped to at most one node in the layer above it, however, a node in a higher layer may be mapped to multiple lower layer nodes.

Fig. 1 Template Overview.
The overall goal of the system is to map the master template to each
document. This is accomplished by generating the intermediate template trees and
mapping the nodes between templates. Each layer has a specific purpose. The DOM tree
is an exact representation of the document. The expanded template is an annotated
version of the DOM tree that can contain additional nodes representing aspects of
document structure. The abstract template is more generalized. Each of its nodes can map
to many nodes in the expanded template tree. Expanded template nodes that map to the
same abstract template node represent repeated structure within the document. Finally the
master template allows multiple documents in the set to be generalized into a single
template. It combines the information in the abstract templates for every example
document in the set and contains optional nodes that may only occur in some documents.

The lowest layer is the DOM tree of the document. TMS makes use of Internet
Explorer™ to parse the document into this DOM structure. Using Internet Explorer™ for
this task provided two main advantages. First, it allowed me to focus on the task of data
extraction without writing my own web browser. Second, using a popular browser
ensured that TMS would parse documents in the same way that users would view them.
Internet Explorer™ provides a powerful and fairly well documented API. The DOM tree
layer is not actually a template, but rather an internal structure generated by the parser.

The second layer is the expanded template. Each element node in this layer
maps exactly to one node in the document, however it also contains region nodes for each
instance of a region that has been located in the document. The mapping between the

---

1 Internet Explorer is a registered trademark of Microsoft Inc.
DOM tree and the expanded template is straightforward, since each node in the DOM
tree has a trivial one to one relationship with a node in the expanded template.

The third layer is the abstract template. This layer contains the generic
template we are mapping to. It contains a single region node for each region we are
looking for in the document. Each of these region nodes is mapped to every instance of
that particular region in the expanded template, and each of its children is mapped to the
child nodes of every instance. In this way every occurrence of these nodes in the
expanded template is mapped to a single abstract representation of that node in the
abstract template.

The highest layer is the master template. This is identical to the abstract
template, but while there is one abstract template for each example document in the set,
there is only one master template for the whole set. Its nodes are mapped to the
corresponding nodes in each abstract template.

In many cases there are predictable variances between documents in the set.
For example, Hotmail\textsuperscript{TM}\textsuperscript{2} displays additional fields in an email document if the email
contains a CC field or an attachment. These fields are structured, but they do not occur in
every document. In cases like this the abstract template must contain nodes that will be
unmapped in some documents. Nodes we expect to be unmapped are marked as optional
in the template. This allows the mapping algorithm to prefer to leave these nodes
unmapped over other nodes that are expected to occur in every document. The system
also uses this optional node mechanism to handle nodes that are different in every
document, such as HTML tags contained in the document content itself.

\textsuperscript{2} Hotmail is a registered trademark of Microsoft Inc.
Template Generation

The template generation process is divided into two steps, and the first is completely automated. The user provides a number of example documents to the system, which are merged to form the initial template. This template is basically a union of the nodes in each example, using a mapping algorithm to determine identical nodes and only add a single example of them. This mapping algorithm is described in detail in a later section. For now it is important to understand that it finds a mapping between nodes in two given trees that maximizes an average similarity value. This algorithm is used both in generation and extraction when two trees of nodes need to be mapped to each other or merged together. During a merge, two mapped nodes can be replaced by a single node representing both.

For each example document, the DOM tree is copied to create initial expanded and abstract templates for that document. The initial mappings between these template trees are trivial as each node is simply mapped to the node it was copied from in the earlier tree. The abstract template for the first example document is then copied to create the initial master template for the document set. The abstract templates for each additional example are merged into the master template, using the mapping algorithm to determine if nodes are identical. If a new node is found which does not map to a node in the master template, this new node is added to the master, but marked as optional. Likewise, if a node in the master is not mapped to a node in the example then it too is marked as optional. Once all examples are added, the resulting master template contains a union of all of the nodes in each example.
One limitation of the current approach is that the order in which the examples are added is potentially significant in some cases. If two examples are merged each containing a node not found in the other and these nodes are adjacent, the ordering of these optional nodes in the result is arbitrary. Obviously, with only these two examples there is not enough information to determine the proper ordering of these two nodes, however if a third example is added containing both optional nodes the order could be found. Unfortunately the system in its current form can not reorder these nodes to account for the new example. If the third example were added first however, the ordering would be correct. To correct this problem the mapping algorithm would need to be modified to consider all possible orders for sequential optional nodes. However that would increase the search space of the mapping algorithm, because the optimum subsequence for a parent with adjacent optional nodes would have to be found for each permutation of those adjacent nodes. For this version, this problem can be avoided somewhat by paying attention to the order of the examples and providing the most general example first. This issue never actually appeared in any of the document sets that were examined.

Once the basic template is generated the user must mark regions and extraction targets on the master template. These extraction targets indicate content to be extracted using the template. The user can select any node in the template and add such a target to it. To add a region, the user selects the first example of the region and the mapping algorithm is used to find subsequent examples whose similarity value compared to the first example is above a threshold. All of these examples are merged in a manner similar to the method used above to merge the example documents. All of the examples are removed from the master template and a new region node is added, with the merged
nodes as its children. The process of selecting nodes for both of these tasks is done using a graphical user interface that displays both a structural view of the template tree and a view of the document itself in human readable form. Selecting a node from either view will highlight the node's location in both views making it easy to locate the desired node.

As regions are added to the master template each example document must be remapped to the new template. Once the master template is finished it is saved as an XML file.

Data Extraction

The extraction process uses the same mapping algorithm used in generation, but the general process is somewhat simpler. The master template is loaded from a file. Then each document to be extracted is parsed into a DOM tree. This tree is copied into an expanded template, whose nodes are mapped back to that document's tree. For each document the master template is copied into an abstract template. Finally the mapping algorithm is used to find the best mapping between the nodes in the expanded template and the abstract template. When the mapping involves a region node from the abstract template, a new region instance node is added to the expanded template.

When this process is completed the expanded template tree is traversed looking for extraction targets. The nodes containing these targets can then have their text returned or a link to the HTML node itself can be returned for use in scripting. For returning the extracted data TMS uses a tree structure where each region instance is a

```java
calculate_node_similarity(Node A, Node B) {
    if(A.type != B.type) base = 0
    else base = (properties weight) * (properties the same) / (total properties) + (1 - properties weight)

    if(the nodes have no children) return base

    for(each child in A) {
        for(each child in B) {
            similarity_table[child of A][child of B] = calculate_node_similarity(child of A, child of B)
        }
    }

    subsequence = find_best_sequence(0, 0, child count for A, child count for B, similarity_table)
    return (children weight) * (average similarity of best subsequence) + (1 - children weight) * base
}

find_best_sequence(position, start, length, end, table) {
    // position is the first position of the subsequence we want to find
    // start is the lowest integer we may use in the sequence
    // length is in length of the total sequence
    // end is one greater than the highest integer we may use in the sequence
    // table is a table of node similarities

    unmapped_total = 0
    best_value_found = 0

    for(i = start; i < end; i++) {
        value_sum = table[position, i] * 2 + (unmapped_total * (unmapped threshold) )
        if(i + 1 >= end) value_sum += (length - (position + 1)) * (unmapped threshold)
        else if(position + 1 >= length) value_sum += (end - (i + 1)) * (unmapped threshold)
        else value_sum += find_best_sequence(position + 1, i + 1, length, end, table)

        if(best_value_found < value_sum) best_value_found = value_sum
        unmapped_total += 1
    }

    if(position + 1 >= length) value_sum = (end - start) * (unmapped threshold)
    else value_sum = find_best_sequence(position + 1, start)

    value_sum += (unmapped threshold)

    if(best_value_found < value_sum) best_value_found = value_sum

    return best_value_found
}
```
subtree and targets within that region are leaves. Targets outside of any region are children of the tree’s root.

Mapping Algorithm
The mapping algorithm developed for TMS returns the best possible mapping between nodes in two trees. This mapping is then used to merge the trees in generation or to link them together in extraction. The algorithm searches through all valid mappings and selects the one with the highest average similarity value. In order to reduce the search space it makes a few assumptions. First, it assumes that if two nodes are mapped together, then the only possible mappings for their children will be with each other. This means that if nodes A and B are mapped together, any child of A must be mapped to a child of B or be unmapped, and the same is true for the children of B. This allows the algorithm to deal with extra nodes inserted into the tree, or even missing nodes, especially if they have been marked as optional. However, it cannot deal with extra nodes inserted as parents of important nodes. If examples containing this behavior are presented in the template generation stage, both versions will appear as options in the template and the mapping function can select whichever version matches.

Second, the mapping algorithm assumes that the children will be mapped in order, although any children of either A or B may be left unmapped. This helps simplify the search, and it makes the order of the children of a node part of that node’s signature. Unfortunately it does lead to the problem where the order of adjacent optional nodes could not be determined.

Finally, it makes the assumption that the roots of both trees are mapped to each other. Since our templates always begin at the root of each document, with the “body” node, this is a reasonable assumption. In addition to those fixed assumptions, the algorithm makes use of several tuning factors which can be altered and are stored with the template. These factors will be explained in the description of the algorithm below.
The mapping algorithm has two parts which work together to determine the similarity of a pair of nodes and their children. The first part, the similarity calculator, is a recursive algorithm that finds the similarity of two nodes based on their properties, and the best mapping of their direct children. It makes use of the second component, the sequencing algorithm, to find this best mapping. The similarity calculator supplies the sequencing algorithm with a table containing the similarity values for each pair of children. The calculator determines the values for this table by recursively calling the calculator to determine the similarity between each pair of nodes in the table.

The similarity calculator takes two nodes, determines the best way to map their subtrees together, and then returns a value between zero and one that represents how similar those two nodes are. To simplify matters this paper will only consider element nodes, and not region nodes, to begin with. The algorithm begins with a base similarity which is zero if the nodes are different HTML element types, such as an “a” element and a “table” element, or one if they are the same. This is averaged with a value based on the similarity of their HTML properties. The weights used in this average are based on a tuning factor that determines how important HTML properties are in determining similarity. There is also an optional flag that causes the similarity calculator to automatically reject mappings between different HTML element types. This can potentially speed up the process somewhat.

Next, the similarity calculator compares the children of the nodes. It recursively calls the similarity calculator to determine the similarity between each child of the first node with each child of the second node. This will also recursively generate a set of subtree mappings for each pair of children that represent the best possible mapping if
those nodes were mapped together. If two nodes without children are compared then only their base similarity is calculated. If a node with children is compared to one without children, then all of the children of that node are considered unmapped.

The similarity values are stored in a table which is passed to the sequencing algorithm that will determine the overall mapping between the children of these nodes that will maximize their average similarity value. This average similarity is then combined with the base similarity of the parent nodes using another tuning factor for the weight.

The sequencing algorithm developed for TMS uses a dynamic programming approach to locate the mapping of children with the highest average similarity. Since the number of children of each parent is fixed, maximizing the sum of their similarities will also maximize the average similarity. Unmapped nodes are given a base similarity value depending on if they are marked as optional or not; optional nodes have a higher base. Both the unmapped and optional base values are tuning factors as well.

The sequencing algorithm works by testing each valid mapping between the

\[ P = \text{Position of the first element in the subsequence} \]
\[ S = \text{Lowest integer other than } -1 \text{ that may be used in the sequence} \]
\[ L = \text{Length of sequence (the number of children of node A)} \]
\[ E = \text{All integers in the sequence must be less than } E \]
\[ (\text{the number of children of node B)} \]
\[ u = \text{Base value for unmapped nodes} \]

\[ f(P, S, L, E) = \begin{cases} 
\max( g(P, S, L, E) u + f(P + 1, S, L, E) & P < L \\
u * (E - S) & P \geq L 
\end{cases} \quad (1) \]

\[ g(P, S, L, E) = \max( h(j, P, S, L, E) : S \leq i < E ) \quad (2) \]
\[ h(i, P, S, L, E) = u * (i - S) + (2 * \text{similarity}(P, i)) + f(P + 1, S + 1, L, E) \quad (3) \]
children of a given node A and the children of another node B. Each of these possible mapping combinations is represented by a sequence of integers of length L where L is the number of children of A. Each integer in the sequence must be between 0 and E where E is the number of children of B, or –1, which represents an unmapped node. With the exception of these unmapped nodes, the numbers in the sequence must always be increasing and can not repeat. Any sequence following these rules can be interpreted as a mapping between the children of A and the children of B, where the first integer in the sequence is the index of the child of B to which the first child of A should be mapped.

The second integer stores the mapping for the second child of A and so on. Any values of
–1 indicate an unmapped child, as do any integers that are skipped in the sequence.

Each of these sequences has a similarity value associated with it that is calculated as follows. Any integer in the sequence which is not –1 is looked up in the mapping table generated earlier to determine the similarity of the mapping between that child of A and that child of B. This value is multiplied by two since both nodes in the pair have this mapping. For each –1 in the sequence, or for any integer between 0 and E which does not occur, a base value is added depending on if that node has been marked as optional or not. The best mapping is given by the sequence that maximizes the sum of these values.

If the ith child of B is a region node then:

\[
\begin{align*}
    k(i, P, S, n, L, E) &= \max_{1 \leq j < L - P} \left( u \cdot (i - S) + \sum_{j=1}^{L-P} \left( j + n \right) \cdot f(P, i, P + j, E) + f(P + j, S + n, L, E) \right) \\
    h(i, P, S, L, E) &= \max\left( k(i, P, S, n, L, E) : 0 \leq n \leq 1 \right)
\end{align*}
\]  

(4) (5)

The value of the best subsequence running from position P until the end of the sequence composed only of integers between S and E can be found by considering each potential integer i, between S and E, combined with the best subsequence starting at P + 1 using integers between i + 1 and E. This value is found using equation 1. For each such integer i the best possible value if P is mapped to i is twice the value of that mapping plus the sum of the unmapped value of any skipped nodes plus the value of the best subsequence from P + 1 and i + 1. The sequencing algorithm also needs to account for the case where this node is left unmapped in which case the value is the unmapped value of P plus the subsequence running from P + 1 and S since no nodes in set B were consumed.
If \( S = E \) then there are no valid choices for \( i \) so \( P \) must be unmapped. Starting with \( P = 0 \) and \( S = 0 \) this will give us the best subsequence possible. To avoid recalculating subsequences the values are stored in a lookup table after the first calculation.

Now consider regions. Each region has a signature composed of its children nodes. Individual child nodes are not mapped to a whole region, but rather to the children in its signature. Therefore, instead of calculating the similarity values between a region node and an element node we calculate the similarity between the element node and each child of the region. These similarity values are stored in another table which is passed into the sequencing algorithm like the regular element to element similarity table.

The sequencer proceeds as described above, except when it tries to map an element to a region. In that case, the region can actually consume multiple nodes in set \( A \) for a single region instance. Further, it may not consume any nodes in \( B \) because the region might be mapped again. To account for this last part there are two cases for this value of \( i \), one where the remaining subsequence starts at \( i + 1 \) (consuming the region) and one where it starts at \( i \) (and has another instance of the region). Whichever case results in a better sum is used. To determine how many nodes in \( A \) are consumed by the region, and which nodes are mapped to which, the sequencing algorithm is used again. A separate instance of the sequencer is run, mapping the remainder of set \( A \) to the signature of the region. The sequencing algorithm actually run a series of these subsequences where the length of the resulting sequence is set to each value between 1 and the remaining length of set \( A \). The sequencing algorithm has to do this because the sequences generated
by the sequencing algorithm are of fixed length, but the best length for the region subsequence is not known ahead of time.

However the length of the final sequence is no longer fixed, because the sequencing algorithm has consumed a number of additional nodes of set B depending on how many instances of the region it found. Since a sum is used to determine the value of a sequence, longer sequences might be favored, even if they result in leaving more nodes unmapped. To counter this, the value of a region subsequence is calculated by finding the average similarity of the nodes in the sequence and multiplying this by the actual number of nodes consumed. Since the region node is only consumed on its last instance the sequence length is again fixed.

Both the generation of the region similarity data and the creation of the subsequences are recursive, which allows for arbitrarily nested region nodes. There is some extra book keeping work involved with the regions, and care must be taken in calculating the average value of the region subsequence to discard regions composed only of unmapped nodes that do not consume the region.
CHAPTER IV

RESULTS

Using the Template Generator

The template generator is the component of the system that allows the user to create new templates. This is a relatively simple process, and a template for a page can generally be created in a few minutes. The user first needs to select a few example pages to create the template from. A standard web browser interface is presented to allow the user to browse to web pages and add them as examples. In general two or three example pages work fine. Since the mapping algorithm must be run on each example document multiple times during the course of template creation, the process will go faster with only a few documents. It is a good idea to take a moment here to consider the attributes of the document set and look for special cases that may have unusual formatting and include those as additional examples.

Once the examples have been selected, the system generates a master template from the examples using the process described in chapter three. This master template is then mapped to each example document. Examples can be selected for viewing from a tab bar in the interface listing all example documents. An example can be viewed either as the rendered document as it appears in a web browser, or as a tree structure. A split screen view shows both, and is very useful for locating nodes in the template. Clicking on a node will select
and highlight it in both views. Information about that node and its linked counterparts in the various template layers is also displayed. Next, the user locates areas of interest in one or more of the example documents. Often these areas contain lists of items that can be represented as region nodes, as described in chapter three. In this case, the user can add a region by selecting the nodes which make up the first instance of the region. TMS will
locate other examples of that region in the master template using the mapping algorithm
and alter the master template accordingly. All of the examples are then automatically remapped by the system to this new master template.

After the regions are created the user can add extraction targets to mark the nodes that contain information to be extracted. This is as simple as clicking on a node to be extracted and pressing a button. Targets are added to single nodes only, but the text of the entire subtree below the selected node is extracted. Adding a target only affects the master template, although the targets appear on the examples as well, and therefore this does not require the examples to be remapped. Once the template is complete and the user is satisfied with it, it is saved as an XML file.

Sample Cases

The following are a few web sites where the system was used to create extraction templates. They represent several different domains where automated data extraction might be. All of these sites provide dynamic content presented with a standard structure.

Fig. 6. Expanded template for Google Image Search™ page.

Google Image Search™

3 Google™ is a registered trademark of Google Inc.
The Google Image Search™ [8] site served as a simple test case for the system. It is a special search option provided by the popular Google™ search engine [9], which returns images found on the web matching the search criteria. Thumbnails of these images, as well as information about them, are displayed in a grid fashion on the page. This is slightly more complex than the standard Google™ search results page because it requires nested regions: one for the rows and one for the items in the rows. There is also some variation in the items returned: some contain an extra link to similar images on the same site. This extra link appears as an optional node in the region. TMS was able to generate a template to extract from the page without any problems. The extracted data includes a link to each image as well as the name and size of the image.

Fig. 6 shows a portion of the expanded template for a document in the set. White nodes have been collapsed, their children hidden, for clarity. The body node has a series of row region nodes, each of which contains image and text region nodes. These names are arbitrary labels assigned by the user to describe each region. Within each nested region the outlined nodes are the nodes containing data to be extracted.

Invision Message Board

This sample case is a commercial message board package used by a number of different web sites [10]. For the test I used the community forum at the developer’s web site. I created templates for both the forum page, which displays all of the posted topics, and the topic page, which displays all the posts in the topic. These two templates are completely independent, created using different example sets. This forum package has a slightly unusual structure, including the use of HTML comment tags, but it didn’t present any problems for the template generator.
The Hotmail™ email service provides free web based email accounts, however these accounts can only be accessed through their web interface. Extracting from Hotmail™ was a challenge for an earlier version of TMS, which was intended to be integrated into an email client. This early system did not use multiple examples to generate optional nodes and was unable to create a template to handle the email view page. The email view page adds additional fields to the email, such as for attachments or multiple recipients, when needed. The template needs to be able to mark the nodes containing these fields as optional.

Unfortunately, the current system was not completely successful at creating a template to handle Hotmail™ either. The problem is that the nodes for each field are identical except for the text they contain, and since the system ignores the text of the document, it cannot determine which of the optional nodes it has encountered. One solution is to extract all of the fields in a generic manner, and leave determining what each field contains to a post processing step. A better solution might be to incorporate the text contained in a node in the measure of its similarity to other nodes calculated by the mapping function. However, this could slow down the mapping process.

Fig. 7 shows the header of a Hotmail™ message along with the associated subtree. The highlighted node is the “from” field of the header. All four field appear identical in the tree, and since the fields included in the header can vary from email to email, the system is unable to tell them apart.

\[\text{Hotmail™ is a registered trademark of Microsoft Inc.}\]
Google™ News

Google™ provides a service that searches news web sites for stories on a given topic [12]. Extracting from the Google™ news results was similar to extracting
Fig. 9. Abstract template mapped to a different news item.

Fig. 10. Master template used to map both news items.
from the Google™ image search site, however, there is more variation in the structure of the returned items. They may contain images or links to related stories for example. For the most part, this presented no problem for the template generator, however, the last link to a related story was often a link to another search page instead of to an actual story.

Fig. 11. A few adds from Newegg and the associated template code.
Detecting this would require considering the text contained in the node. The system does not necessarily need to detect these links to extract the data however. The system could just return all the related links without discriminating. Fig. 8 and fig. 9 appear fairly different, but both have been mapped as news items. Fig. 10 is a portion of the master template used to map both of them. The image is contained under a node that has been marked as optional, since it only occurs in some of the examples.

**Newegg**

Newegg is an online computer store that sorts its products into various categories, such as monitors or other components [13]. Each category has a page that displays special deals and links to search or browse for specific items. Two templates were created for this site. The first extracts the special deals from a category page, and the second extract the information about a specific product from its page.

Fig. 11 shows several of the ads from the Newegg store as well as a portion of the template to map them. The first template worked fine, although if it is applied to a page containing no special offers it will incorrectly find region instances on the page. These instances contain no nodes mapped to extraction targets however, so no data will be mistakenly extracted. The second template also worked. Since different products have different unique properties, these properties are extracted in a generic way. Both the name of the property and its value are isolated in a region instance and extracted as a pair.

**eBay™**

---

5 eBay is a registered trademark of eBay Inc.
eBay™ is an online auction service that lists various items for sale in a searchable database [14]. Creating a template to extract from these search results is interesting because the results can be returned in two visually similar, but structurally different, ways. Further, there is no obvious pattern to which structure would be returned for a given search. Fortunately, the generator was able to create a single template that could extract from both versions. Fig. 12 shows the master template for an eBay™ item. The three children of the “item” region are all optional nodes, allowing it to handle both kinds of items.

Fig. 12. Master template for an eBay™ item.
System Evaluation

Laender et al. [3] present seven features for evaluating data extraction systems. TMS falls into the category they describe as “HTML aware” as it uses information contained in the HTML structure of a document to generate a template.

The authors’ first criterion is degree of automation. The approach taken by TMS is “semi-automated”. It requires the user to supply example documents and mark regions of interest, but the actual work of template generation is performed by the program.

The authors’ second feature of interest is support for “complex objects”. These pieces of data that “present a complex structure”, may have nested structures or repeating parts. Because the templates in TMS attempt to match trees of nodes they are capable of dealing with these “complex structures” as long as they are formed of HTML elements. Any structure that occurs purely in text can not be directly identified by TMS, although it may still be extracted if the HTML node containing it can be identified independently.

The authors describe their third criterion as “page contents” and they distinguish between two types of pages. Pages with “semi-structured data” contain “...data items... implicitly formatted to be recognized individually.” Pages with “semi-structured text”, by contrast, “...bring free text from which data items can only be inferred.” With the first type, the structure of the page itself is sufficient to locate and extract the data. With the second type, however, the data is encoded in the natural language text of the document itself. As long as the structure is HTML-based TMS can
handle pages with “semi-structured data”. Since TMS does not employ any linguistic knowledge it is unable to separate data contained in “semi-structured text” from that text.

The authors’ fourth criterion is “ease of use”. This is a very subjective measure, however TMS does provide a graphical user interface that allows a user to interact with the example pages directly and see the results of applying the template to those examples. This is intended to make the system easy to use.

The fifth feature the authors look for is support for outputting data in XML [1]. TMS uses XML for storing its templates, however the extracted data is returned through a C++ interface. It is up to the user at this point to decide what to do with the data.

The authors’ sixth category is “support for non-HTML sources”. TMS provides no support for such sources. The mapping algorithm used in TMS could be applied to any tree structure with appropriate modifications. If structure in some other format could be converted into such a tree it is possible the TMS mapping algorithm could be applied, however that is beyond the scope of this work.

The final consideration is described as “resilience and adaptiveness”. By “resilience”, they mean the ability of the system to deal with “. . . changes in the pages to which they are targeted.” This is one of the goals of my system and it is able to deal with some changes to a page’s structure or contents. However, major changes to the page’s structure are likely to cause problems for TMS. “Adaptiveness“ refers to the ability where “. . . a wrapper built for the pages of a specific web source on a given application domain could work properly with pages from another source in the same application domain.”
TMS is focused on extraction by identifying structure, not content. For this reason, a given template will only work on a set of documents with similar structure.

TMS performed well on a variety of document sets and the ability for the system to automatically generate templates to handle complex mappings from multiple examples is very promising. However, it is disappointing that TMS could not handle the Hotmail™ [11] example fully. To correctly extract from the Hotmail™ page, TMS will need the ability to map nodes based on patterns with the text of the node. One solution would be to add an option for a user to mark text patterns within nodes. TMS could then use this information in its mapping calculations. With the addition of support for using other features of a document such as text patterns, it may be possible to expand the system to handle even more document sets.
CHAPTER V

CONCLUSION

Overview

A template based approach to the problem of data extraction is appealing because on a conceptual level it mirrors the way many structured documents are generated. Documents in set sharing a common structure are commonly created by filling out a common template with specific information; this can be done in an automated fashion using data from a database or by hand. With the proper template the process used to generate the document can be reversed to retrieve the data placed in it. However since we are not given the template for a document set, the system must use examples from the set to generate this template.

For structured document sets the approach presented here usually works well. However, it is not appropriate for groups of documents that do not share a common structure. Documents that have been created by hand and do not use any kind of template, even if they appear similar to a human reader, are unlikely to produce good results with this system. This approach is also dependant on the use of HTML elements to define the document structure. For purely text based structure in a document, the template would need to be defined in other ways. If a good system for representing the structure of non-HTML were developed, the template based approach could
have potential for these document types as well. However, this is beyond the scope of the current work. One advantage of this system is that it is very easy to create new templates. Most of the structural differences between documents in the set, and even between instances of a region within a document, can be detected automatically. For the most part, the user only needs to mark repeated regions and elements to extract. In some cases the user will also need to look through the document set to find special cases and ensure that they appear in examples, although in many cases the system is capable of handling unexpected structural differences.

Future Improvements

There are a number of modifications that could be made to the system to potentially improve results. Most of these relate to dealing with special cases that can cause problems for template creation in some circumstances. One such case, mentioned earlier, happens when multiple optional nodes occur next to each other. The order of these nodes is unclear and potentially arbitrary. Since the mapping algorithm requires that nodes be mapped in order this can cause problems, especially during template generation where presenting the example documents in different orders can result in these optional nodes being in the wrong order.

One solution to the problem of adjacent optional nodes would be to allow those nodes to be reordered in the sequencing step. This would, however, slow down the sequencing step. Potentially, this could be done only during generation to save time on extraction later. This solution would make the order of example documents unimportant, but if no example contains both optional nodes, and they can occur together in other
documents in the set, then the problem remains. Another issue is that it is unclear from
the template whether the order of adjacent optional nodes is significant. One option
would be to mark the template if the order has been determined by seeing the nodes
together in a single example document, and then prevent reordering. Or possibly a bonus
could be added to the similarity score if the template order is preserved. This would allow
the nodes to be reordered if needed, but still encourage sequences that preserve the
template order.

Another potential improvement could be to consider the text contents of the
nodes when mapping them. In the current version the topology of the document tree is the
most important consideration when mapping nodes, along with the HTML properties of
the element nodes in the tree. In some cases noted in chapter four there are nodes that
appear identical based on these measures, but could be distinguished based on the text
contained in them. There are a few different ways the text could be considered. One
option would be to simply determine if the two strings are identical. Another would be to
factor into the node similarity an additional component based on the edit distance
between the two strings, similar to the way that HTML properties are factored in now.

There are some drawbacks to either approach to using text in a similarity
measure. First, they are likely to slow down the mapping algorithm, especially if a
complex comparison function is used. Also in many cases the text of the nodes contain
document content. Some of this content is interesting to the user and some is not, but all
of it can be expected to vary from document to document. The system should not penalize
mappings between structural nodes containing different text strings if those strings are
really content. One option to resolve this would be to mark nodes with variable text in the
template, and not consider the text contents of those nodes when mapping them. Or the reverse could be done and nodes with identical text could be marked. These approaches are identical for extraction using a template, but they are different for generation, where initially it is not known whether text will vary or not. Potentially, it might be useful to vary the weighting of the text component of similarity between generation and extraction.

Another option that would reduce the number of mappings to be considered would be to ignore the children of any node in the template whose subtree does not contain any extraction targets. Instead of actually calculating the real similarity value for this node mapping an estimate could be used. This could be tuned by penetrating a certain depth down these trees, and then using estimates. For example if a node is found whose descendants contain no targets, then the system could still map that node’s children, but when mapping those children only use estimates of their similarity. This would be an acceptable estimate because the similarity values of a node’s descendents have reduced influence on its overall similarity as the algorithm progresses down the tree. Obviously this will only work for extraction, because in generation the user has not yet specified where the targets are.

Summary

The mapping algorithm presented in this work calculates the similarity between two nodes in a tree based on the properties of those nodes and the structure of the subtrees beneath them. This approach is especially well suited to comparing HTML documents, with their complex tree structures, to one another. Generating a template is reduced to merging documents and sections of documents, using the mapping algorithm, in order to locate a common structure. Once this common structure is stored in a template
extraction is simply a problem of matching the nodes in the document against the nodes in the template.
References


