Fully Automatic Wrapper Generation For Search Engines

Hongkun Zhao, Weiyi Meng
Dept. of Computer Science
SUNY at Binghamton
Binghamton, NY 13902, USA
{hkzhao,meng}@cs.binghamton.edu

Zonghuan Wu, Vijay Raghavan
Center for Adv. Compu. Studies
Univ. of Louisiana at Lafayette
Lafayette, LA 70504, USA
{zwu, vijay}@cacs.louisiana.edu

Clement Yu
Dept. of Computer Science
University of Illinois at Chicago
Chicago, IL 60607, USA
yu@cs.uic.edu

ABSTRACT
When a query is submitted to a search engine, the search engine returns a dynamically generated result page containing the result records, each of which usually consists of a link to and/or snippet of a retrieved Web page. In addition, such a result page often also contains information irrelevant to the query, such as information related to the hosting site of the search engine and advertisements. In this paper, we present a technique for automatically producing wrappers that can be used to extract search result records from dynamically generated result pages returned by search engines. Automatic search result record extraction is very important for many applications that need to interact with search engines such as automatic construction and maintenance of metasearch engines and deep Web crawling. The novel aspect of the proposed technique is that it utilizes both the visual content features on the result page as displayed on a browser and the HTML tag structures of the HTML source file of the result page. Experimental results indicate that this technique can achieve very high extraction accuracy.

Categories and Subject Descriptors
H.3.5 [**Information Storage and Retrieval**]: Online Information Services – Commercial Services, Web-based Services.

General Terms
Algorithms, Performance, Design, Experimentation.

Keywords
Information extraction, wrapper generation, search engine.

1. INTRODUCTION
Search engines are very important tools for people to reach the vast information on the World Wide Web. Recent studies indicate that Web searching, behind email, is the second most popular activities on the Internet. Surveys indicate that there are hundreds of thousands of search engines on the Web (e.g., [4, 7]. Not only Web users interact with search engines, many Web applications also need to interact with search engines. For example, metasearch engines utilize existing search engines to perform search [22] and need to extract the search results from the result pages returned by the search engines used. As another example, deep web crawling is to crawl documents or data records from (deep web) search engines [24] and it too needs to extract the search results from the result pages returned by search engines.

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

This paper focuses on the issue of how to extract *search result records* (SRRs) from dynamically generated *result pages* returned by search engines in response to submitted queries. Each SRR typically consists of a link to a retrieved Web page and some pertinent information (snippet). A typical result page contains multiple SRRs plus some information irrelevant to the user query, such as information related to the hosting site of the search engine and advertisements. The objective is to extract SRRs and discard irrelevant information from a result page. For a given search engine, an experienced developer may manually write a program to extract the SRRs from the result pages returned by the search engine after manually analyzing some sample result pages. Manually generating SRR extraction programs (i.e., wrappers) is costly, time-consuming and impractical in many applications. For example, search engines frequently change their result display format and such changes will require manual maintenance of the extraction program. As another example, our WebScales project [11, 28] aims to connect to hundreds of thousands of search engines and it is not practical to manually construct a wrapper for each search engine. Therefore, we need an automated solution. For search engines that have a Web services interface like Google and Amazon.com, automated tools may be used to extract their SRRs because the result formats are clearly described in the WSDL file of the Web Services. However, our investigation indicates that very few search engines have Web services interfaces currently. One reason may be that Web services are designed to support B2B applications while most search engines are B2C applications. Therefore, we need to deal with search engines with no Web services interfaces and extract results that are presented in HTML files.

In this paper, we describe our solution to the problem of **automatically** extracting the SRRs from dynamically generated HTML *result pages* returned by search engines. Specifically, we present **ViNTs** (**V**isual **i**nformation a**N**d **T**ag **s**tructure based wrapper generator) – a tool for automatically producing the wrappers for any given search engines. Since the heart and soul of a search engine's wrapper is a set of SRR extraction rules, wrapper and SRR extraction rules will be used interchangeably in this paper. On the one hand, there are several reasons that make it very difficult to derive accurate wrappers entirely based on HTML tags [29]. First, HTML tags are designed to describe the presentation of data to facilitate browsing by human users. As such, the tags themselves convey very limited semantic information about the data. Second, HTML tags have been used in ways far beyond the imagination of the HTML tag designers. As a result, little convention can be relied upon. Third, HTML has a rather loose grammar and browsers typically do not enforce the grammar when displaying Web pages, i.e., ill-formed HTML pages can often be "perfectly" displayed. Fourth, not only the script program that produces result pages generates tags, the SRRs themselves may also contain tags. On the other hand, as Web pages are designed to facilitate human browsing, they contain rich

visual content features to help people locate and understand information. In fact, users typically rely entirely on visual content features to recognize SRRs. Therefore, it is natural to consider utilizing visual content features for SRR extraction.

The main contribution of this paper is the development and evaluation of a method that utilizes both the visual content features on the result page as displayed on a browser and the HTML tag structure of the HTML source file of the result page to derive SRR extraction rules. Unlike previous works [2, 5, 6, 8, 10, 12, 20, 26] in this field that exploit regularities in the HTML tag structure directly, ViNTs first utilizes the visual content (without HTML tags) to identify the regularities from content itself, and then combines them with the HTML tag structure regularities to generate wrappers. Our method is fully automated and the experimental results indicate that this technique can achieve considerably higher extraction accuracy than that of the state of art web information extraction systems like MDR [20], which utilize only the HTML tag structures on result pages.

The rest of this paper is organized as follows. Section 2 presents the architecture of ViNTs. Section 3 introduces the fundamentals of our method like the visual content features. Section 4 discusses how to find candidate SRRs. Section 5 presents our method for deriving wrappers. Section 6 reports the experimental results. Section 7 reviews related works. Section 8 concludes the paper.

2. SYSTEM ARCHITECTURE

Figure 1 shows the architecture of our automatic wrapper generation system. The input to the system is the URL of a search engine's interface page, which contains an HTML *form* used to accept user queries. The output of the system is a *wrapper* for the search engine. The *search engine form extractor* figures out how to connect to the search engine using the information available in the HTML *form*. Based on the extracted form information, the *query sender* component sends queries to the search engine and receives result pages returned by the search engine. Readers may refer to [28] for more details about these two components.

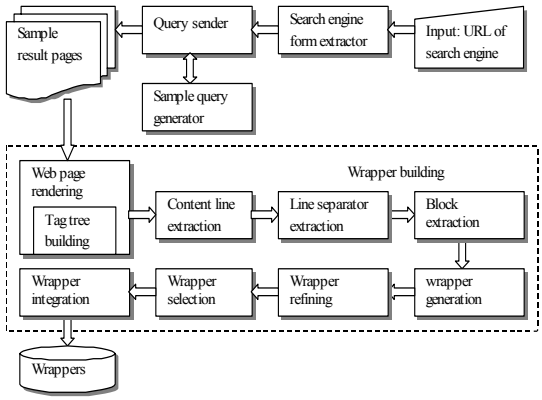

Figure 1. System Architecture of ViNTs

The *wrapper-building module*, shown in the dash-line box above, is the focus of this paper. The input to this module is a set of sample result pages produced by a search engine in response to automatically generated sample queries. The only requirement for a sample result page is that it contains a sufficient number of SRRs (at least four, and the more the better), to permit the regularities among the SRRs to be explored for wrapper building. The input to this module also contains a special result page called *no-result page*, which contains no SRRs. This page contains only information irrelevant to any user query and thus can be used to filter out useless information from other result pages. The sample query that yields the *no-result page* is called an *impossible query*. All sample queries are generated by the *sample query generator*. This component has been implemented in our system but it will not be discussed in this paper.

To utilize the visual content features, we render each sample result page during wrapper building. There are many objects such as *links* and *texts* on each result page (*Anchor text* associated with a URL is called a link in this paper). When a result page is rendered, for each object on the result page, a *rendering box* – a rectangle containing the object – is produced. We use a coordinate system based on the browser window to help describe the positions of the rendering boxes. Our wrapper generation method is sketched below. First, for each sample result page, we analyze the types (say link or text) and the positions of all the rendering boxes to identify some candidate result records (section 4). Based on these records and a hypothesis about the general format of the SRR wrappers, we build some initial wrappers (sections 5.1-5.4). These wrappers are refined to enable the detection of the boundaries separating different types of records (e.g., SRRs and non-SRRs) (section 5.5). Next, the most promising wrapper is selected for this result page from the refined wrappers using additional visual features (section 5.6). Some search engines may produce different irrelevant information on different result pages (e.g., the advertisements may be query dependent). As a result, different sample result pages may lead to slightly different wrappers. Our final step is to integrate the wrappers for all sample result pages of the search engine to produce the final wrapper for the search engine (section 5.7). The detail of our method will be presented in the next several sections.

3. METHOD FUNDAMENTALS

As mentioned earlier, existing techniques on web information extraction are based on the analysis of HTML tag structures. We believe that regularities in visual content (strings, images, etc. as shown on web pages) should also be utilized to achieve higher performance.

Many visual content features that are designed to help people locate and understand information on a web page can help information extraction. For example, the profile (or contour) of the left side of each SRR on the same result page tends to be very similar to each other, there are visual separators (e.g., blank lines) between consecutive SRRs, all SRRs tend to be arranged together in a special section on the result page, and this section occupies a large portion on the result page, and it also tends to be centrally located on the page. We describe some basic visual content features that are used in this study in the following sub-sections.

3.1 Content Line

A result page usually consists of multiple sections, each containing information in one category. For example, the result page in Figure 2 consists of two sections: the one on the left contains SRRs and the one on the right contains sponsored links. The section containing SRRs will be called the *SRR section*.

Definition 3.1 (Content line) *A content line is a group of characters that visually form a horizontal line in the same section on the rendered page.*

In Figure 2, "Category: Home > Personal Finance > Tax Preparation" forms a content line. Note that "Tax Info Center" forms a different content line even though it is visually in the same line as the line starting with "Category:" because they appear in different sections on the result page.

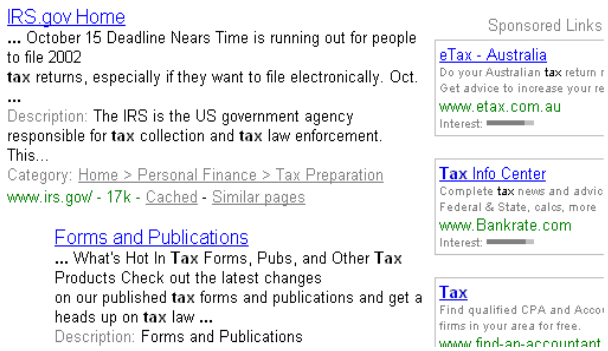

Figure 2. A result page by Google

Different types of content lines can be observed on typical result pages and their arrangements are useful for identifying records. In our approach, the following types of content lines are identified:

- LINK – more than 90% of the area of the rendering box of this line is link area. Only anchor text (i.e., clickable text) with an underlying URL is considered a link in this paper. Thus, a URL that is not an anchor text is not considered as a link, but as a text. [code: 1]
- TEXT – more than 90% of the area of the rendering box of this line is text area. [code: 2]
- LINK-TEXT – it contains both link and text, none of them occupies more than 90% of the area. [code 3]
- LINK-HEAD – link line but started with a number like 1, 2, 3, … [code: 4]
- TEXT-HEAD – text line but started with a number. [code: 5]
- LINK-TEXT-HEAD – link-text line but started with a number. [code: 6]
- HR-LINE – a visual line generated by HTML tag <HR>. [code: 7]
- BLANK – the blank line. [code: 8]

The record in Figure 3 contains 5 content lines. The first is a LINK line, followed by two TEXT lines, then another LINK line and the last (invisible) is a BLANK line. To facilitate computation, a *code* is assigned to each type of content line.

Each content line has a rendering box and the left x coordinate of the rendering box is called the *position code* of the content line. The position code of a blank line is set to be the position code of the visible line immediately before it. To summarize, each content line is represented as a (type code, position code) pair.

3.2 Shape of a Block

A record consists of one or more content lines, which together form a *block*. An observation about the records on a result page is that the left side profiles of all records in the same section tend to be very similar, and records from different sections tend to have different left side profiles. This observation is consistent with the fact that result pages are generated by computer programs and different sections are usually generated by different scripts. We define block shape to represent the left side profile of a block.

Definition 3.2 (Shape of a block) *Let c_1, …, c_k be the content lines in a block in top-down order and let pc_i be the position code of c_i, $i=1,…,k$. The shape of the block is an ordered list of the position codes of the member content lines of the block, namely (pc_1, …, pc_k). (pc_1, …, pc_k) is also called the shape code of the block.*

Consider the block in Figure 3. It has 5 content lines (the fifth is a blank line). Suppose the position codes of the 5 content lines from top to bottom are 8, 48, 48, 48 and 48, respectively. Then the shape of the block is represented as (8, 48, 48, 48, 48).

Figure 3. A record block and its shape

3.3 Block Similarity

Each block consists of three pieces of information: the ordered (from top to bottom) type codes of its content lines, the position codes of its content lines and the block shape. We define three metrics to measure the similarity between two blocks of content lines: *type distance*, *shape distance* and *position distance*.

Type distance. The type distance between two blocks is to capture the difference in their content line type sequences. The detail is described below. We define the *type code of a block* as a sequence of the type codes of the content lines of the block. Let TC_i be the type code of the ith content line in the block, then $TC_1…TC_n$ is the type code of the block, where n is the number of content lines in the block. Furthermore, multiple consecutive TEXT type codes are compressed to one occurrence based on the observation that texts in snippets of SRRs often vary in length significantly. Based on the above definition, the type code for the block in Figure 3 is *1 2 1 8* (one TEXT type code 2 is suppressed). In our implementation, type distance between two blocks a and b is the *edit distance* [27] between the type codes of the two blocks.

Shape distance. This distance is to measure difference between the indention sequences of the shapes of two blocks. To focus on the shape and ignore where a block starts in the coordinate system, we subtract the smallest position code in a shape code from each position code. This will convert (8, 48, 48, 48, 48) to (0, 40, 40, 40, 40). To concentrate on indentions, multiple consecutive occurrences of the same position code are suppressed to one. Consequently, (0, 40, 40, 40, 40) is transformed to (0, 40), indicating that the shape has one indention with indent value 40. The final list will be called the *modified shape code* of a block. Let MSC(u) denote the modified shape code of block u. For the block shapes in Figure 4, if we assume the value of each indent is 10, then we have MSC(a) = (0), MSC(b) = (0, 10), MSC(c) = (0, 10), MSC(d) = (0, 10, 20), MSC(e) = (10, 0) and MSC(f) = (0, 10, 0). Note that blocks b and c have the same modified shape code while other blocks all have different modified shape codes.

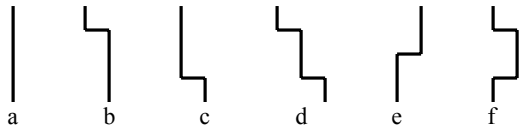

Figure 4. Some sample block shapes

In summary, the shape of a block is represented as a sequence of indentions in our method. The shape distance of two blocks a and b is defined as the maximum difference between the corresponding modified shape codes of the two blocks; if one modified shape code is longer than the other, we pad the shorter one with 0's at the end to make the lengths of the two shape codes the same before calculating their shape distance.

Position distance. This distance measures the difference between the closest points of the two blocks to the left boundary of the rendered result page. In other words, the position distance

between two blocks *a* and *b* is the difference between the smallest position code for any content line in *a* and that in *b*.

4. FINDING CANDIDATE SEARCH RESULT RECORDS

In this section, we discuss how to find some candidate SRRs from a given result page by exploiting regularities in visual content features. These candidate SRRs will be used to generate a wrapper for this result page in section 5.

For a given result page of a search engine, we first render it and extract content lines from it. Then we remove those content lines that also appear in the *no-result page* of the search engine to get rid of some useless content lines. Now we use a three-step method to find some candidate SRRs. Visually, different records are separated by a certain *separator*. Therefore, in the first step, we try to identify all *candidate content line separators* (CCLSs) (section 4.1). Each CCLS is then used to tentatively segment the content lines into blocks. In the second step, these blocks are clustered into different groups such that the blocks in the same group appear on the result page consecutively and are all visually similar (section 4.2). Intuitively, each such group corresponds to a section on the result page. While a separator may help identify records, it may not reliably separate records correctly. To solve this problem, in the third step, we present an algorithm to identify the first line of each candidate record (section 4.3). Clearly, if we can find the first lines of multiple consecutive records correctly, we can identify candidate SRRs easily.

4.1 Identifying Candidate Content Line Separators

This task is to identify content lines that can be used to segment the result page into blocks. Visually, different records are often separated by a blank line (e.g., the <p> tag) or a visual line (e.g., the <HR> tag). But different records could also be organized into different items of a list (e.g., the <li> tag within the <ol> or <ul> tags) or different rows (e.g., the <tr> tag) of a table, or special image lines. The above tags may also appear in different records. Furthermore, our observation is that a separator may consist of multiple tags, for example, a sequence of tags may collectively form a separator. Consequently, the problem of correctly identifying content line separators for arbitrary search result pages is very challenging. Our solution to this problem is to first identify all CCLSs on a result page and let other steps determine which yielded wrapper is correct (section 5). In this subsection, we discuss how to identify all CCLSs.

In our approach, a CCLS is a sequence of consecutive content lines, i.e., the pattern of the sequence of content lines is used to define a CCLS. Such a pattern is defined to be the sequence of (type code, position code) pairs of the content lines in the CCLS. When a CCLS is used to segment a result page into multiple blocks, the content lines in the CCLS are included in the blocks. In fact, the last content line in a CCLS is also the last content line of the block containing the CCLS. Since a sample result page is required to have at least four SRRs by our approach, the pattern that defines a CCLS must appear at least three times. To avoid missing any potentially correct content line separator, all content line patterns that appear at least three times on the result page are recognized as CCLSs. A suffix tree can be used to accomplish this step. To efficiently find all CCLSs, each distinct (type code, position code) pair is first represented as a special symbol. This will transform a result page of content lines to a string of symbols. Then a suffix tree is constructed for the symbol string with complexity $O(n)$ [25], where n is the number of symbols in the string. From the suffix tree, all sub-strings appearing three or more times can be found in linear time complexity.

4.2 Block Grouping

Using a CCLS, the result page can be segmented into multiple blocks of content lines. As mentioned before, a result page consists of multiple sections and only one of them is the *SRR section*. Therefore, we divide the blocks into groups. Blocks that are consecutive and *visually similar* are put into one group. Two blocks are visually similar if their *type distance*, *shape distance* and *position distance* are all below certain thresholds. At this point, we do not know which group may contain SRRs. As a result, all groups are used for further analysis in section 5.

4.3 Identifying the First Line of Record

The blocks in a group may not be equivalent to the records in the group as the CCLS used to obtain the blocks may be incorrect. A nice feature of our approach is that it does not require the identification and use of the correct CCLS to correctly extract SRRs. As far as three or more SRRs can be correctly obtained first, we can build a wrapper based on these SRRs to extract other SRRs on the same page. To this end, we attempt to identify the first line of each record. Clearly, if the first line of every record is correctly identified, then all records will be correctly identified. The blocks we obtained play an important role in identifying the first lines of records. That is, in each block, we aim to identify exactly one line as the first line of some record. If each block contains exactly one first line of a record and this line can be correctly identified for some consecutive blocks, our approach can still extract all records correctly, even when the blocks do not correspond to records exactly. This feature of our approach reduces the reliance on correctly identifying the content separator.

Based on our analysis of a large number of SRRs from different search engines, we developed a set of heuristic rules to identify the first line of a record from a given block. Some of these heuristic rules are: (1) the line following an HR-LINE is a first line; (2) if there is only one line starting with a number in a block, this line is a first line; (3) if only one line in a block has the smallest position code (i.e., the position codes of all other lines are strictly larger), this line is a first line; and (4) if there is only one BLANK line in a block, the line following the BLANK line is the first line. These heuristics are applied in certain order to reflect their priorities.

5. WRAPPER BUILDING

After the step described in section 4, for each result page, we have a set of block groups, each consisting of consecutive and visually similar blocks. We call these groups *candidate groups*, because they may contain SRRs. Each candidate group contains a number of *candidate records* obtained based on the first record lines of blocks (see section 4.3).

In this section, we describe how to build wrappers (SRR extraction rules) by exploiting regularities in both visual content features and in the HTML tag structures. Note that the wrappers of our approach are expressed based on HTML tag structures only. One advantage of such a wrapper is that it can be efficiently applied on result pages of user queries to extract SRRs, as the rendering of the result pages can be avoided.

In section 5.1, we first review the concepts of tag tree and tag paths, and then introduce how to obtain the *tag paths* of the candidate records in each group. The tag path of a record is the tag

path to its first line. Our observation indicates that even though SRRs on different result pages may be laid out differently, all SRRs on the same result page are usually arranged in the same sub-tree of the tag tree of the page and their tag paths follow certain pattern (section 5.2). Based on this observation, we propose a hypothesis about the general format of the wrappers for all search engines in the form of a regular expression (section 5.3). On the basis of this hypothesis, we build some initial wrappers using the tag paths for the candidate records in each candidate group (section 5.4). The initial wrappers are refined to detect the boundaries separating different candidate groups (section 5.5). Since there may be multiple candidate groups for a result page and multiple initial wrappers may be built for each candidate group, multiple refined wrappers may be generated. Next, the most promising wrapper is selected for the result page from the refined wrappers using additional visual content features (section 5.6). Our final step is to integrate the wrappers for all sample result pages of the same search engine to produce the final wrapper for the search engine (section 5.7).

5.1 Tag Paths of Records

A result page can be transformed into a tree representation based on the tags in its source HTML file, which is called a *tag tree*. The root of a tag tree is the <HTML> tag, and all content nodes (texts, images, etc.) are leaf nodes. Each internal node represents a pair of tags (the starting tag and the corresponding ending tag) if the tag has an ending tag, or just one tag if the tag has no ending tag (
, for example). The root tag and internal nodes are called *tag nodes*. A tag node and the sub-tree rooted at this tag represent the starting tag and its corresponding ending tag as well as all tags and elements in between. Figure 5 shows a sample result page and its partial tag tree. Note that many tag nodes such as <HEAD> and <CENTER> are not expanded.

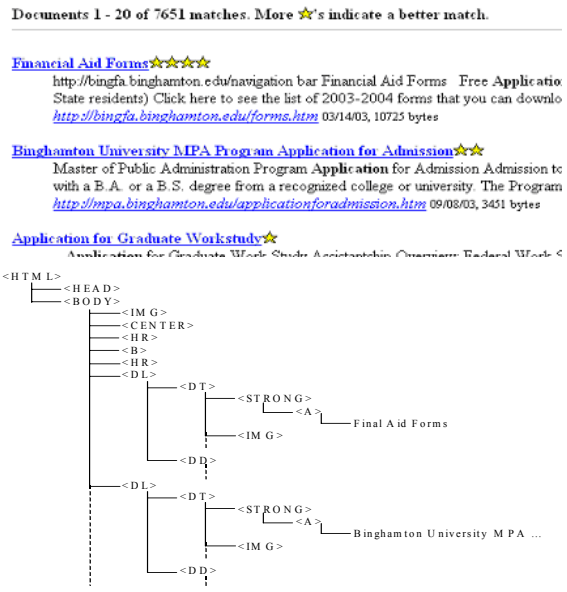

Figure 5. A sample result page and its tag tree

A node in a tag tree can be located by following a path from the root to the node. We call such a path a *tag path*. A tag path consists of a sequence of *path nodes*. Each path node *pn* consists of two components, the *tag name* (i.e., a tag node) and the *direction*, which indicates whether the next node following *pn* on the path is the next sibling of *pn* (indicated by "S") or the first child of *pn* (indicated by "C"). As an example, the tag path of the first <IMG> tag in Figure 5 is "<HTML>C<HEAD> S <BODY> C"; and the tag path of the first link (i.e., Financial Aid Forms; note <A> is part of the link) of the first SRR is "<HTML>C <HEAD>S<BODY>C<IMG>S<CENTER>S<HR>S<B>S<HR> S<DL>C<DT>C <STRONG>C".

Once we know the first line of a record, we can search the tag tree in reverse order from the first node of this line to the root of the tag tree to find the tag path of the record. From a given candidate group of size *n*, we get *n* tag paths. Table 1 lists some tag paths for the sample result page in Figure 5.

It is not difficult to see that a regular expression exists for the tag paths in Table 1. But the problem of automatic regular expression grammar inference is known to be difficult and we generally cannot obtain a regular expression grammar using only positive samples [13], like in our case. Our approach is to provide a hypothesis about the general format of wrappers, and then try to build the wrappers based on the hypothesized format.

Table 1. Tag paths extracted from result page in Figure 2

R#	Tag path
1	<HTML>C<HEAD>S<BODY>C<IMG>S<CENTER>S<HR>S<B>S<HR>S<DL>C<DT>C<STRONG>C
2	<HTML>C<HEAD>S<BODY>C<IMG>S<CENTER>S<HR>S<B>S<HR>S<DL>S<DL>C<DT>C<STRONG>C
3	<HTML>C<HEAD>S<BODY>C<IMG>S<CENTER>S<HR>S<B>S<HR>S<DL>S<DL>S<DL>C<DT>C<STRONG>C
4	<HTML>C<HEAD>S<BODY>C<IMG>S<CENTER>S<HR>S<B>S<HR>S<DL>S<DL>S<DL>S<DL>C<DT>C<STRONG>C

5.2 Structure of the Minimal Sub-tree That Contains SRRs

There exists a minimal sub-tree *t* of the tag tree of a result page such that all SRRs are located in *t*. The minimal here means that no *proper* sub-tree of *t* contains all SRRs. Each SRR corresponds to a sub-forest of *t* (see the dotted circles in Figure 6).

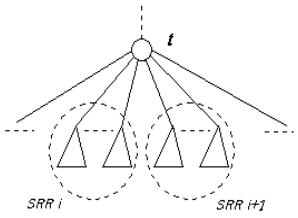

Figure 6. Structure of the minimal sub-tree containing SRRs

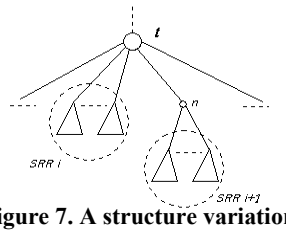

Figure 7. A structure variation of the minimal sub-tree that contains SRRs

Search engine result pages are dynamically generated by a computer program. Since a result page typically has multiple SRRs, it is reasonable to think that there is a loop in the program that wraps up the data extracted from the underlying database when producing the result page. As a result, the corresponding sub-forests of SRRs all have identical or similar tag structures (the parent-child relationships of tags), and the roots of all sub-forests of SRRs should be siblings. Thus we can identify SRRs by identifying their corresponding sub-forests. In other words, we can divide the descendants of *t* into a set of sub-forests, and hopefully, each SRR corresponds a sub-forest. A separator can be used to perform the segmentation. A valid separator satisfies the following conditions: (1) it is a common subset of the sub-forests of all SRRs (i.e., it appears in the sub-forest of each SRR and

itself is a sub-forest); (2) it appears in the sub-forest of each SRR exactly one (otherwise it would incorrectly divide a single SRR into multiple records); and (3) it contains the rightmost subtrees of the sub-forest of each SRR (i.e., the separator that separates SRR_i and SRR_{i+1} is part of SRR_i). Note that the separator here is different from the content line separator introduced in section 4.1. Here it is a tag structure (tag forest) while the separator in section 4.1 is a set of content lines.

It is possible that more than one separator is needed for some search engine. This happens when a search engine arranges some of its SRRs in a way that is different from other SRRs, for example, an SRR may be arranged indented relative to the SRR preceding it. Figure 2 in section 3.1 is an example. Figure 7 shows the tag tree structure of such a case; the parent node *n* of the sub-forest of SRR_{i+1} is at the same level as the sub-forest of SRR_i. Note that SRR_{i+1} still has the similar tag structure as SRR_i. In general, if the SRRs of a search engine are arranged in *k* different ways, then *k* different kinds of separators need to be identified. In practice, however, *k* is usually 1 and occasionally 2.

5.3 Wrapper Format Hypothesis

A wrapper defined over a tag tree needs to specify two things: (a) the location of the minimal sub-tree *t* that contains all SRRs, and (b) the separator set. The minimal sub-tree of *t* can be determined by a tag path from the root of the tag tree to the root of *t*. Within *t*, SRRs are separated by possibly different kinds of separators and each separator is also (the ending) part of a record. In addition, a search engine may display only certain number of SRRs on a result page. Based on the above analysis, we hypothesize that a wrapper can be represented as the following regular expression:

prefix (X (separator1 | separator2 | …))[min, max] (1)

where *X* is a wild card for sub-forests of the tag tree, *prefix* is a tag path, separators are also sub-forests of the tag tree, "|" is the alternation operator, the concatenation of *X* and a separator corresponds to a record, min and max are used to select records from a list of records. For example, if the wrapper without the [min, max] restriction extracts a list of *n* records, then only the records between the min-th and the max-th records are extracted. In general, min ≥ 0 and max can be infinite (some search engines do not limit the number of results that can be displayed on a result page). The *prefix* determines the minimal sub-tree *t* that contains all SRRs in the result page. The *separators* are used to segment all descendants of *t* into SRRs.

Once such a wrapper is generated for a search engine, extracting SRRs from a result page of the search engine is straightforward. First we parse the result page and build the tag tree. Next, we follow the *prefix* of the wrapper to locate the root of the minimal sub-tree *t* that contains all SRRs. Then we find all existing occurrences of the separators in the descendants of *t*, and arrange them in the order of their appearances in the sub-tree *t*. We extract the *i*th SRR from the descendant nodes of *t* located within the *i*th and *(i+1)*th occurrences of separators (the nodes representing the *(i+1)*th occurrence of separators are part of the *i*th SRR). Finally, we extract the SRRs whose serial numbers are within the range [min, max].

5.4 Initial Wrapper Building

For a give candidate record group, we form sub-groups of consecutive records of size *k* (*k* = 3 is used in our experiment). The first *k* records form one sub-group, the second to the (*k*+1)th records form the next sub-group, and so on. With the tag paths of the records in each sub-group and the hypothesis about the format of the wrapper (expression (1)), we try to build an initial wrapper for the records in each sub-group. It is possible that different initial wrappers will be generated for different sub-groups. It is also possible that no initial wrapper can be generated for some sub-groups. All produced wrappers will be passed to the refinement step of our method.

We now discuss how to generate an initial wrapper for a sub-group G. In this step, we focus on identifying the *prefix* and the *separator*(*s*) in expression (1). Parameters *min* and *max* will be determined later in a refinement step (section 5.5). Suppose the records in G appear in order r_1, …, r_k. Let path(r_i) denote the tag path of r_i. The tag paths for the second, third and fourth records in Table 1 will be used as a running example to explain our method. The main ideas of our method are as follows.

1. We find the maximum common prefix PRE of all input tag paths (i.e., those for records in G). For our running example, we have PRE = <HTML>C<HEAD>S <BODY> C<IMG> S <CENTER>S<HR>S<B>S<HR>S<DL>S. Note that this PRE may be different from the *prefix* needed by expression (1). The reason is that the first record of the group (the one with the shortest tag path) may not be in G. In general, the correct prefix is contained in PRE but PRE may contain extra path nodes at the end. To identify the extra path nodes, we first remove PRE from each tag path (let p_i = path(r_i) – PRE) and then compute $Diff_i = p_{i+1} - p_i$ (p_i is a suffix of p_{i+1}). If all Diff's are the same, then it is a *separator* for expression (1). In our running example, Diff = <DL>S is the separator. We now remove all occurrences of Diff from the end of PRE. Let PRE1 be the new PRE and E be the last node of PRE1. At this point, an effort is made to identify additional separators based on whether the tag path of Diff is identical to the tag pattern composed of the child node(s) of E and whether Diff also appears immediately before E. When both conditions are satisfied, the path node of E is identified as a new separator and the occurrences of all separators (including previously identified ones) are removed from the end of PRE1. This process is repeated until no new separator can be identified and the remaining tag path (of PRE1) is the prefix for expression (1). For our running example, only one separator is identified and the correct prefix is <HTML> C <HEAD> S <BODY> C <IMG> S <CENTER> S<HR>S <B>S<HR>S.
2. If Diffs are different, three cases are identified. Case 1: A common suffix of the Diffs does not exist. In this case, the wrapper generating process fails and the process is terminated. Case 2: A common suffix exists and it does not have multiple occurrences in any Diff. In this case, this suffix is a *separator*. We subtract from PRE any suffix that is identical to any of the Diffs until no further subtraction is possible and the remaining PRE is the *prefix* for expression (1). Case 3: All common suffixes have multiple occurrences in some Diffs. In this case, an attempt is made to expand each Diff by taking the structure of the child nodes (or even deeper descendant nodes) of the nodes in the Diff into consideration (structures of child nodes help differentiate different nodes in the Diff and therefore help to find a separator that does not have multiple occurrences in Diffs). The expanded Diffs are then used to identify separators as in the second case. If the separator still cannot be found, the wrapper building process fails.
3. Based on the *prefix* and *separator*(*s*) identified in the last two steps, an initial wrapper is generated for G by assuming *min* = 0 and *max* = ∞. For example, the initial wrapper generated for the running example is <HTML>C<HEAD>S<BODY>C

<IMG>S<CENTER>S<HR>S<B>S<HR>S(*X*<DL>S)[0, ∞], where *X* is a wild card. The initial wrapper is then used to extract all matching records from the result page to see if all records in G can be correctly extracted in consecutive order. If this is true, the wrapper is accepted for further evaluation (refinement in section 5.5). If this is not true, a possible reason is that the separator used is incorrect. Therefore, an attempt is made to expand the nodes in the separator by their child (descendant) nodes as in Step 2 to see if a new separator can be found. If it can be found, it is used to revise the initial wrapper and repeat the above process. If the new wrapper cannot be accepted or a new separator cannot be found, the wrapper building process fails for G.

5.5 Wrapper Refining

This task is to determine the values of the parameters *min* and *max* of a wrapper (see expression (1)). The input to the wrapper refining process includes an initial wrapper (generated in section 5.4) and a list of consecutive records extracted by applying the wrapper. Let these records be numbered from 1 to *n*, and let Rm be the record in the middle. The wrapper refining process works as follows. We start from Rm and move towards the two ends of the list. Let's consider the process of moving towards the beginning of the list. When the next record is encountered, if it does not contain a link or its block is not visually similar to the block of Rm, the serial number of the record plus 1 becomes *min*. Similarly, *max* can be determined when we consider the process of moving towards the end of the list.

5.6 Wrapper Selection for One Sample Page

At this step, we have a set of wrappers and record groups extracted by applying those wrappers. Among these wrappers, four cases may occur. First, some wrappers can correctly extract all SRRs and nothing else. Second, some may extract some but not all correct SRRs. Third, some may retrieval all SRRs but also some non-SRR records. Fourth, some may be suitable for other neatly arranged information on the result page such as ads and host information. The wrapper selection step is to determine the wrapper that mostly likely belongs to the first case.

Our approach uses content features (both visual and non-visual) to help find the correct SRR group hence the correct wrapper. It is not difficult to observe that on the rendered result page, the correct SRR group likely (1) occupies a large area, (2) is centrally located, (3) contains many characters, (4) has a large number of records. To utilize these content features, we define the following four weights:

1. *Rendering area weight* (RAW). A group's RAW is defined as the relative rendering area of this record group over the largest rendering area of all record groups.
2. *Center distance weight* (CDW). CDW is based on the distance between the center of a group's rendering box and the center of the rendering box of the whole result page; this distance is called the *center distance* of the group. Let p_0 be the center of the whole page's rendering box. If the rendering box of a group contains p_0, its center distance is defined to be 0; otherwise we use the Euclid distance between the center of the group's rendering box and p_0. We define CDW as the relative center distance over the smallest center distance.
3. *Number of records weight* (NRW). A group's NRW is defined as the number of records of the group divided by the number of records of the largest group.
4. *Average number of characters weight* (ACNW). The average number of characters of a group is the average number of characters in each block in this group. A group's ACNW is defined as the relative average number of characters of the group over the largest average number of characters in all groups.

We combine the above four weights by weighted summation, pick the group with the highest combined weight as the search result group and output its corresponding wrapper as the correct wrapper for the input sample page.

5.7 Wrapper Integration

After wrapper selection, we are able to build a wrapper for each sample result page. The wrappers built from different sample result pages of the same search engine may be different even though they are all correct with respect to their corresponding sample result pages. The reason is that frequently search engines may include information on a result page that is query dependent or changes from time to time. As a result, the tag paths (*prefixes* in wrapper expressions) of the minimal sub-tree that contains all SRRs as well as the separator sets may vary. Thus, a wrapper built from one sample page may not correctly extract SRRs when applied to another page. Wrapper integration is to integrate the wrappers built from multiple sample result pages of the same search engine into a single robust wrapper for the search engine.

The integration involves three parts: separator integration, prefix integration and [min, max] integration. During the integration process, two wrappers are considered at a time. If both their separator sets and their prefixes can be integrated, the two wrappers are integrated. An integrated wrapper may then be integrated with another (possibly integrated) wrapper. At the end of this process, there may be multiple integrated wrappers and no integration between them can be carried out. At this time, the integrated wrapper with the largest support (i.e., it is integrated from the largest number of input wrappers) will be selected as the final wrapper for the search engine. In our current implementation, tie is broken arbitrarily. In the following, we outline how two wrappers are integrated.

Separator integration:

For two separator sets, if one set is a subset of the other set, we take the larger set as the integrated separator set. The case that the two separator sets are identical is a special case of the above case. If none of the sets is a subset of the other, the integration of these two separator sets fails.

Prefix integration:

Prefix integration is carried out by converting each prefix into a *compact form* through the removal of unimportant path nodes. This would remove "noises" from the prefixes.

The conversion is based on the following rules:

(1) Keep all path nodes with direction code of "C".
(2) Keep path nodes with direction code of "S" only if their tag name is the same as that of the closest future path node with direction code of "C".
(3) For any path node after the last node with direction code "C", if its tag name is identical to the tag name of the first node of any separator in the wrapper, it is kept; otherwise it is deleted.

For example, the prefix of the wrapper in Section 5.4 (item 3) becomes <HTML>C<BODY>C. It is obvious that the compact prefix will find the same minimal sub-tree as the original *prefix*.

Prefix integration can be carried out reasonably easily after prefixes are converted into compact forms.

[min, max] integration:
Let the two input [min, max]'s be [min1, max1] and [min2, max2], respectively. Let [min3, max3] be the integration result. Then min3 = min {min1, min2} and max3 = max {max1, max2}.

6. EXPERIMENTS

We have built an operational wrapper generation prototype system (ViNTs) based on our method. Result page rendering and tag tree construction are performed by a commercial tool ICEbrowser [16]. On a Pentium 4 1.7GH PC, the current ViNTs can build a wrapper for a search engine with 5 sample result pages and 1 *no-result page* in 3 to 7 seconds. Once a wrapper is built for a search engine, SRRs from a new result page of the search engine can be extracted in a small fraction of a second (about 100 milliseconds). Thus, the wrappers generated by ViNTs are practically useful in real-time web applications. In fact, ViNTs has been used in the development of a commercial news metasearch engine (www.allinonenews.com). The ViNTs prototype system and the data sets used to evaluate it can be accessed at http://www.data.binghamton.edu/vints.html**.**

6.1 Data Sets

Three data sets are used to test ViNTs and they are described below.

Data set 1 contains 100 search engines in 4 categories: *education*, *government*, *medical*, and *general*. Search engines in the education category are randomly selected from the Yahoo search engine using query "American universities". Search engines in government and medical categories are collected from search.com. The general category contains some general-purpose search engines like Google, AltaVista, Yahoo, etc. This data set is used as a training set for learning optimal performance parameters (e.g., the weights in section 5.6). Almost all search engines in this data set are document search engines, i.e., they search text documents.

Data set 2 contains 100 search engines collected from profusion.com and none of them is included in Data set 1. These search engines are not exposed to ViNTs until they are tested. 20 of them are non-document search engines (they are for jobs, e-commerce, and entertainment).

We should mention that the above data sets do not contain search engines that return multiple sections of results since our current method is designed to extract records from just the major section of a search engine result page. Search engines whose result pages cannot be rendered by ICEbrowser are also excluded (there are very few such cases).

For each search engine in the above two data sets, 10 queries are submitted and the 10 first result pages are manually collected. In addition, a *no-result page* is also collected for each search engine by submitting a non-existent term as a query to the search engine. Most search engines limit the number of records displayed on each result page, say 10. But a few search engines display all results (could be hundreds) on their result page. To avoid the bias that may be caused by these search engines on the overall performance, only the first 25 records are used if a result page contains more than 25 records.

Data set 3 is obtained from the Omini [5] testbed (available at http://sourceforge.net/projects/omini/). Omini testbed consists of more than 2,000 web pages collected from 50 websites (many of them are e-commerce search engines). Since the number of web pages per web sites is highly uneven, from 1 to several dozens, we decide to take one random page per website. Thus, data set 3 consists of 50 web pages from 50 websites, one page per site.

Data set 3 is used to compare our method with MDR [20], which is a state of the art web information extraction system based on HTML tag structure analysis only and can be downloaded at http://www.cs.uic.edu/~liub/MDR/MDR-download.html. There is currently no standard testbed for web information extraction. As a result, researchers always report the performance of their systems based on their own testbed. This can easily cause biased results. By using a data set from a third party, fairer comparison can be made.

6.2 Performance Measures

We use the *recall* and *precision* measures (which are widely used to evaluate information retrieval system) to evaluate the performance of our system for extracting SRRs. Recall and precision are defined below:

$$recall = \frac{Ec}{Nt} \text{ and } precision = \frac{Ec}{Et}$$

where *Ec* is the total number of correctly extracted SRRs, *Nt* is the total number of SRRs on all result pages used, and *Et* is total number of records extracted.

6.3 Experimental Results on Data Sets 1 and 2

To determine the impact of using visual content features on our wrapper generation approach, we also implemented a version of ViNTs that uses no visual content features. In this version, content lines are identified by HTML tags (e.g., <p>,
 and <tr>), block similarity is based on type distance only, and wrapper selection does not use visual features such as rendering area.

For each search engine in data sets 1&2, we use 5 result pages and the *no-result page* to build the wrapper, which is then applied to extract SRRs for all the 10 result pages. Table 2 shows the results when the wrapper is applied to the 5 pages that are used to build the wrapper, and Table 3 shows the results when the wrapper is applied to the 5 pages that are not used to build the wrapper. The columns headed by VW are the results of the regular ViNTs, and the columns headed by NV are the results when the visual content features are not utilized.

Table 2. Results on samples used to build the wrapper

	Data set 1		Data set 2	
	VW	NV	VW	NV
#SRRs	6919	6919	6905	6905
#Extracted SRRs	6905	6833	6872	6465
#Correct SRRs	6901	6722	6740	6283
Recall	99.7%	97.2%	97.6%	91.0%
Precision	99.9%	98.4%	98.1%	97.2%

Table 3. Results on samples not used to build the wrapper

	Data set 1		Data set 2	
	VW	NV	VW	NV
#SRRs	6219	6219	5822	5822
#Extracted SRRs	6169	6111	5801	5525
#Correct SRRs	6164	6001	5673	5390
Recall	99.1%	96.5%	97.4%	92.6%
Precision	99.9%	98.2%	97.8%	97.6%

As it can be seen from Tables 2 and 3, ViNTs can generate very high quality wrappers, with both recall and precision close to

100% on data set 1 and close to 98% on data set 2. The small 2% decrease in performance from using data set 2 to using data set 1 strongly indicates that our approach is very robust, considering the facts that data set 2 is completely new and 20 of its search engines are non-document search engines (our system is trained using only document search engines). The main reason for the above 2% decrease in performance is due to the failure of ViNTs on 2 search engines in data set 2. One failure is caused by the wrapper selection step (a wrong wrapper is selected by ViNTs) and the reason for the other failure is not immediately clear.

By comparing the results under columns VW and NV, we can see that utilizing visual content features has moderately increased the precision but significantly increased the recall, especially for data set 2. Note that even though the increases are not large in absolute terms, they are highly significant because they are increases beyond the 90^{+}% base performance and the last several percentage points are usually the most difficult to achieve. Since result pages for non-document search engines are usually more complex than those from document search engines and data set 2 has 20 non-document search engines, it seems that utilizing visual content features is more effective for complex result pages. We plan to carry out more experiments in the future to verify this observation.

6.4 Comparison with MDR

MDR extracts from a single page at a time. To compare with MDR, we configure ViNTs to build a wrapper from a single page, and then apply the wrapper to extract SRRs from the page. ViNTs returns only the SRRs in the major section of a web page, while MDR reports all identified sections. Only the major section is considered if there are multiple sections of SRRs. MDR has a *similarity threshold*, which is set at 60% in our test, based on the suggestion of the authors of MDR.

MDR could not produce any output for 8 web pages in data set 3 because the MDR program terminated abnormally, while ViNTs worked on all 50 pages. These 8 pages are not used in our comparison as the reason of the abnormal termination of the MDR program on these pages during our test was not clear. Misaligned SRRs, such as an extracted SRR consisting of part of an actual SRR and part of the next SRR, are counted as error. Table 4 shows the summary of the test results using the 42 web pages MDR produced results. The detailed test result can be accessed at our ViNTs demo site.

As we can see from Table 4, the performance of ViNTs is considerably better than that of MDR. Our test also found out that MDR is much better at extracting records from HTML tables (with recall 73.7% and precision 87.2% on data set 3) than from non-tables (with recall 7.7% and precision 100%), while our method performs well in both situations (recall and precision both at 99.1% for tables and both at 98% for non-tables).

Table 4. Comparison results with MDR

	ViNTs	MDR
#SRRs	795	795
#Extracted SRRs	795	479
#Correct SRRs	785	420
Recall	98.7%	52.8%
Precision	98.7%	87.7%

7. RELATED WORKS

The problem of extracting search results from search engine result pages is an information extraction (IE) problem. IE has received a lot of attention in recent years. A good survey about current works on IE can be found in [19]. Earlier works are mainly semi-automatic or even manual [1, 3, 9, 15, 18, 21, 23]. They rely on training and human assistance to generate extraction rules for web pages. Many new applications such as building large-scale metasearch engines or building metasearch engines on-demand [28] require fully automated wrapper generation techniques. Several automated or nearly automated IE methods have been proposed recently and the most representative ones are Omini [5], the method in [12], IEPAD [6], MDR [20], RoadRunner [10], EXALG [2], DeLa [26], PickUp [8].

RoadRunner extracts template by analyzing a pair of Web pages of the same class at a time. It uses one page to derive an initial template and then tries to match the second page with the template. Mismatches are used to modify the template to eventually fit all input pages to the template. EXALG works on a set of Web pages of the same class. Equivalence classes that are sets of tokens having the same frequency of occurrence on all input pages are computed, and large and frequently occurring equivalence classes (LFEQs) are extracted for template generation. Dela employs a multi-level pattern extraction algorithm to build regular expression to represent the nested schema of data on the web page. PickUp identifies table structures in web pages by mining repeated patterns in HTML tag sequence. RoadRunner, EXALG, Dela and PickUp all support complicated data types and are more general than works developed primarily for flat record extraction. However, they only reported experimental results when data at a level lower than search result records are extracted. Consequently, their results are not directly comparable with ours. While it is possible to use these systems to extract data at record level, we are not able to implement their complicated algorithms as a lot of details are left out in the published papers. As a result, we are not able to compare them with our method experimentally.

The methods of Omini, IEPAD, MDR and that in [12] are the most relevant to our method because they all extract data at the record level. IEPAD first symbolizes all HTML tokens of a parsed web page into a string, and then uses a PAT tree and some heuristics to find candidate patterns. Finally, a **human user selects** the best pattern among the candidate patterns. As a result, IEPAD is not fully automated. Omini and the method in [12] build a tag tree for an input web page, then apply some heuristics to extract a sub-tree that contains data objects of interest. Then another set of heuristics is applied to find out a separator, which is a tag that can segment the minimum object-rich sub-tree into data objects. In addition, the method in [12] uses match heuristics based on a manually constructed ontology; thus it is not truly automatic. Omini does not use any ontology and uses a different set of heuristics to achieve better performance. The idea of separator in our method is similar to [12] and Omini but our separators are more general. Their separators contain only one HTML tag, which is insufficient in some samples we tested, while our separator is a tag forest, which can be expanded to desired depth if necessary. We also support optional separators. In [5] (for Omini) and [12], only the experimental results on separator identification were reported but no results on wrapper generation were reported. The results reported in [20] indicate that Omini has low effectiveness (recall 39% and precision 56%) for extracting data records. MDR identifies data regions by finding the existence of multiple similar generalized-nodes of a tag node, while a generalized-node is a collection of child nodes of that tag node. Then each generalized-node is checked to decide if it contains multiple records or only one. Near perfect results (with recall 99.8% and precision 100%) are reported for MDR in [20].

However, when a third party data set is used, MDR has a much lower performance than our method (see section 6). A major reason for the discrepancy is probably due to the fact that MDR is primarily designed to handle tables only. In addition, the definition of record is MDR seems to be slightly different from that of SRR as in this paper. MDR does not identify the correct section for search result records and extracts records from all sections. This will likely extract some advertisement records. Furthermore, MDR does not generate wrapper and needs to perform the complex and time-consuming extraction for each result page. This is not practical when dealing with a large number of result pages for each user query as in a metasearch engine context.

The main difference between our method and existing techniques is that our method is the only one that utilizes both visual content features and HTML tag structure regularities, while existing techniques use only HTML tag structures. This makes our method less sensitive to the misuse of HTML tags. Our method also utilizes HTML tag structure differently. For example, our separators are more general and our wrapper generation process (i.e., generating multiple candidate wrappers for each page, selecting the best wrapper for each page and integrate the wrappers for multiple pages) is also different. Our method is fully automated and highly accurate (the average recall and precision for the three data sets are 98.3% and 98.9%, respectively). We should point out that, among similar studies, our experiments used the largest number of search engines.

There are some works on using visual information to process web pages in other applications (e.g., [14, 17, 29]) but none of them are directly related to search engine result extraction. These methods and our method also use different sets of visual features.

8. CONCLUSIONS

In this paper, we presented a fully automated technique to generate wrappers for extracting search result records from result pages dynamically generated by search engines. Our technique utilizes both the visual content features on the result page as displayed on a browser and the HTML tag structures of the HTML source file of the result page. This differentiates our technique from other competing techniques for similar applications. Our experimental results indicate that our technique can achieve high extraction accuracy. In the future, we plan to utilize additional visual features (such as font type and color) to further reduce the reliance on HTML tag structure.

9. ACKNOWLEDGMENTS

This work is supported in part by the following grants from NSF: IIS-0208574 and IIS-0208434.

10. REFERENCES

[1] B. Adelberg. NoDoSE – A tool for semi-automatically extracting structured and semistructured data from text documents. ACM SIGMOD Conference, 1998.

[2] A. Arasu, H. Garcia-Molina. Extracting Structured Data from Web Pages. ACM SIGMOD Conference, June 2003.

[3] R. Baumgartner, S. Flesca and G. Gottlob. Visual web information extraction with Lixto. VLDB Conference, 2001.

[4] M. Bergman. The Deep Web: Surfacing Hidden Value. White Paper, BrightPlanet, 2000 (www.completeplanet.com/Tutorials/DeepWeb/index.asp)

[5] D. Buttler, L. Liu, C. Pu. A Fully Automated Object Extraction System for the World Wide Web. International Conference on Distributed Computing Systems (ICDCS 2001), 2001.

[6] C. Chang, S. Lui. IEPAD: Information Extraction based on Pattern Discovery. World Wide Web Conference, 2001.

[7] K. Chang, B. He, C. Li, M. P, Z. Zhang. Structured Databases on the Web: Observations and Implications. Technical Report, UIUCDCS-R-2003-2321, UIUC, 2003.

[8] L. Chen, H. Jamil, N. Wang. Automatic Composite Wrapper Generation for Semi-Structured Biological Data Based on Table Structure Identification. SIGMOD Record, June 2004.

[9] B. Chidlowskii, J. Ragetli, M. de Rijke. Automatic Wrapper Generation for Web Search Engines. WAIM Conf., 2000.

[10] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. VLDB Conference, pp. 109-118, 2001.

[11] www.cs.binghamton.edu/~meng/metasearch.html.

[12] D. Embley, Y. Jiang, and Y. -K. Ng. Record-boundary discovery in Web documents. ACM SIGMOD Conf., 1999.

[13] E. Gold. Language Identification in the Limit. Information and Control, 10(5), 1967.

[14] X. Gu, J. Chen, W. Ma, G. Chen. Visual based Content Understanding towards Web Adaptation. Int'l Conf. on Adaptive Hypermedia & Adaptive Web-based Systems, pp.164-173, 2002.

[15] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the Web. Information Systems. 23(8): 521-538, 1998.

[16] http://www.icesoft.com

[17] M. Kovacevic, M. Diligenti, M. Gori, M. Maggini, V. Milutinovic. Recognition of Common Areas in a Web Page Using Visual Information: A Possible Application in a Page Classification. ICDM Conference, 2002.

[18] N. Kushmerick, D. Weld, R. Doorenbos. Wrapper Induction for Information Extraction. Int'l Joint Conf. on AI, 1997.

[19] A. Laender, B. Ribeiro-Neto, A. da Silva, and J. Teixeira. A Brief Survey of Web Data Extraction Tools. ACM SIGMOD Record, 31(2), 2002.

[20] B. Liu, R. Grossman and Y. Zhai. Mining Data Records in Web Pages. SIGKDD'03, 2003.

[21] L. Liu, C. Pu and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. Int'l Conf. on Data Engineering, 2000.

[22] W. Meng, C. Yu, K. Liu. Building Efficient and Effective Metasearch Engines. ACM Computing Surveys, 34(1), March 2002, pp.48-84.

[23] I. Muslea, S. Minton and C. Knoblock. A hierarchical approach to wrapper induction. Int'l Conf. on Autonomous Agents, 190-197, 1999.

[24] S. Raghavan, H. Garcia-Molina. Crawling the Hidden Web. VLDB Conference, Italy, 2001.

[25] E. Ukkonen. On-line Construction of Suffix Trees. Algorithmica, 14:249-260, 1995.

[26] J. Wang, F. H. Lochovsky. Data Extraction and Label Assignment for Web Databases. WWW Conference, 2003.

[27] S. Wu and U. Manber. Fast Text Searching Allowing Errors. Communications of the ACM, 35(10):83-91, 1992.

[28] Z. Wu, W. Meng, V. Raghavan, C. Yu, H. He, H. Qian, R. Vuyyuru. Towards Automatic Incorporation of Search Engines into a Large-Scale Metasearch Engine. IEEE/WIC WI-2003 Conference, October 2003.

[29] Y. Yang, H. Zhang. HTML Page Analysis based on Visual Cues. 6th International Conference on Document Analysis and Recognition, 2001.