Automatic Extraction of Dynamic Record Sections From Search Engine Result Pages

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

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

ABSTRACT

A search engine returned result page may contain search results that are organized into multiple dynamically generated sections in response to a user query. Furthermore, such a result page often also contains information irrelevant to the query, such as information related to the hosting site of the search engine. In this paper, we present a method to automatically generate wrappers for extracting search result records from all dynamic sections on result pages returned by search engines. This method has the following novel features: (1) it aims to explicitly identify all dynamic sections, including those that are not seen on sample result pages used to generate the wrapper, and (2) it addresses the issue of correctly differentiating sections and records. Experimental results indicate that this method is very promising. Automatic search result record extraction is critical for applications that need to interact with search engines such as automatic construction and maintenance of metasearch engines and deep Web crawling.

1. INTRODUCTION

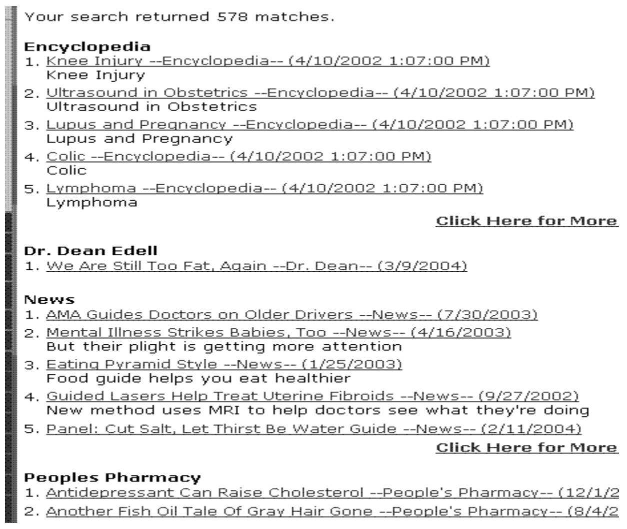
A recent survey reveals that there are hundreds of thousands of search engines on the Web [8]. Many web applications, such as metasearch engines [18, 26], deep web crawlers [20] and shopping agents, need to interact with search engines. Thus there is a demand to develop automated tools (wrappers) to extract *search result records* (**SRR**s) from the HTML result pages returned by search engines. Some search engines, like Google and Amazon, have web services interfaces, which make automated extraction easier. But a vast majority of search engines do not have web services interfaces and there is no incentive for them to develop such interfaces because they support B2C (business to customer) applications only. We also note that XML has been used to deliver web data in many applications. However, almost all search engines still present their search results in HTML format. Therefore, applications that need to harvest data from the search results of search engines must deal with the problem of extracting results presented in HTML files.

A typical search engine result page contains *static*, *semi-dynamic* and *dynamic* contents. In this paper, *static contents* refer to the portion that is query independent, i.e., they are identical on the result page of every query. Dynamic contents are the SRRs retrieved in response to a query. Each SRR is a semantically complete data unit corresponding to a retrieved entity (e.g., a book or a document). A SRR typically consists of a link to a retrieved Web page or database record (or further details about the SRR) and some pertinent information (snippet). If there is no ambiguity, the word "record" also refers to SRR in this paper. *Semi-dynamic contents* are those that may be affected by different queries but are generally independent of the content of any specific query. For example, in Figure 1, "Your search returned 578 matches" can be considered as semi-dynamic as its general format is independent of user queries and the dynamic component (the number of matches) is not directly related to the content of the query. As another example, section header "Encyclopedia" is semi-dynamic because it is common for all queries that retrieve at least one record from the Encyclopedia data repository of the search engine. If no result is retrieved from the Encyclopedia for a particular query, then this entire section, including the section header, will not be displayed. "Click here for more …" is also semi-dynamic as it appears only for sections that have more than five records.

Intuitively, a *dynamic section* on a search result page is a set of *all* SRRs that appear consecutively and have certain common features such as a common header and a common display format. Many search engines produce result pages with multiple dynamic sections. For example, some search engines categorize or cluster search results (Figure 1) and some search engines display regular search results and sponsored links in different dynamic sections. A significant percentage of the search engines return result pages with multiple dynamic sections. For example, 19 out of the 100 search engines from the dataset in [29] produce result pages with multiple dynamic sections.

In general, complete data extraction from web pages (including result pages returned from search engines) may consist of three tasks. The first is *section extraction*, i.e., extract all the sections from each page; the second is *record extraction*, i.e., extract the records within each section; and the third is *data annotation*, i.e., identify and annotate each data unit within each record. Existing work on data extraction (wrapper generation) has been mostly focused on record extraction (see the Related Work section) and some work on data annotation has also been reported (e.g., [24]). However, to the best of our knowledge, the section extraction problem as considered in this paper has *not* been explicitly studied before.

In this paper, we investigate how to automatically extract all dynamic sections as well as SRRs within each dynamic section from search result pages. The emphasis is on dynamic section extraction. Static and semi-dynamic contents are utilized to help identify the boundaries of different dynamic sections. For the rest of this paper, when there is no confusion, "*dynamic section*" and "*section*" will be used interchangeably. An important requirement for our method is to maintain the *section-record relationship*, i.e., the extracted SRRs should be grouped by section. This requires that the sections be explicitly extracted. The benefit of keeping the section-record relationship is to make it easier to use them later as different applications may be interested in the SRRs in different sections.



Your search returned 578 matches.

Encyclopedia
1. Knee Injury --Encyclopedia-- (4/10/2002 1:07:00 PM)
   Knee Injury
2. Ultrasound in Obstetrics --Encyclopedia-- (4/10/2002 1:07:00 PM)
   Ultrasound in Obstetrics
3. Lupus and Pregnancy --Encyclopedia-- (4/10/2002 1:07:00 PM)
   Lupus and Pregnancy
4. Colic --Encyclopedia-- (4/10/2002 1:07:00 PM)
   Colic
5. Lymphoma --Encyclopedia-- (4/10/2002 1:07:00 PM)
   Lymphoma

Click Here for More

Dr. Dean Edell
1. We Are Still Too Fat, Again --Dr. Dean-- (3/9/2004)

News
1. AMA Guides Doctors on Older Drivers --News-- (7/30/2003)
2. Mental Illness Strikes Babies, Too --News-- (4/16/2003)
   But their plight is getting more attention
3. Eating Pyramid Style --News-- (1/25/2003)
   Food guide helps you eat healthier
4. Guided Lasers Help Treat Uterine Fibroids --News-- (9/27/2002)
   New method uses MRI to help doctors see what they're doing
5. Panel: Cut Salt, Let Thirst Be Water Guide --News-- (2/11/2004)

Click Here for More

Peoples Pharmacy
1. Antidepressant Can Raise Cholesterol --People's Pharmacy-- (12/1/2
2. Another Fish Oil Tale Of Gray Hair Gone --People's Pharmacy-- (8/4/2

Figure 1. Part of a sample result page with multiple sections from healthcentral.com

The task of automatically constructing a wrapper to extract all dynamic sections from the result pages of a search engine is challenging because the following problems need to be solved:

- *Non-uniform section format problem*: Our observation indicates that even on the same result page, some dynamic sections may have the same format while other dynamic sections may have different formats. The lack of a general pattern for the sections on a result page makes it more difficult to extract the sections. Many researchers have studied the problem of automatic record extraction and the proposed techniques heavily depend on the fact that these records have similar patterns/formats [5, 15, 29]. Due to the non-uniformity of the section formats, the techniques proposed for extracting records cannot be directly applied to extracting sections.
- *Section-record granularity problem*: Some consecutive sections with the same format may be mistakenly extracted as records while some large records may be incorrectly extracted as sections. In this paper, we refer the problem of correctly differentiating sections and records as the *section-record granularity problem*.
- *Hidden section extraction problem*: Because dynamic sections may be query dependent, different result pages returned by the same search engine may contain different dynamic sections. A direct consequence of this phenomenon is that the sample/training pages that are used to generate the wrapper (extraction rules) may not contain all possible dynamic sections that the search engine may produce. In this paper, we call the problem of extracting sections that are unseen from training pages the *hidden section extraction problem*.

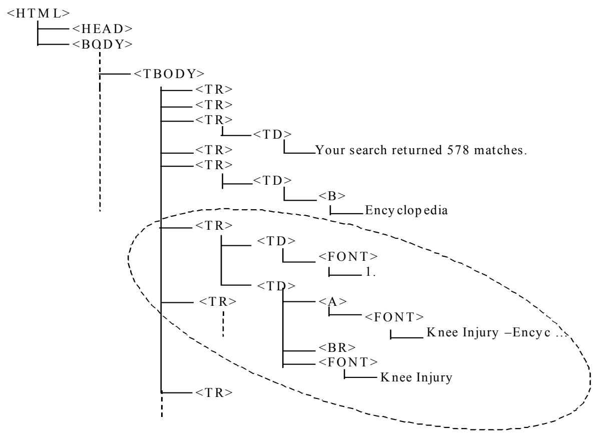


Figure 2. Part of a DOM tree for the page in Figure 1

The main contribution of this paper is the development and evaluation of a fully automatic and novel section extraction method that explicitly aims to tackle all of the above problems. The basic solution of our method first employs two independent techniques to identify potential dynamic sections and then merges these sections to obtain more accurate sections. To tackle the *non-uniform section format problem*, the techniques for identifying potential dynamic sections do not utilize the similarities between sections. To tackle the *section-record granularity problem*, we propose a novel technique based on analyzing the *inter-record distances* and *section cohesions*. To tackle the *hidden section extraction problem*, we introduce the concept of *section family*. Another feature of our method is that it utilizes both tag structure information and the visual content information of each result web page. In addition, we also consider the issue of extracting the records within each section. Our record extraction method has no constraint on the minimum number of SRRs that must be in a section for the SRRs to be extracted. In contrast, current techniques require at least two or more records in a single section [15, 29]. Our experimental results indicate that our solution is quite effective. Automatic extraction of SRRs from search engine returned result pages is a critical technique for crawling/mining data from the *deep web* as the data in the deep web are largely hidden behind the search interfaces of deep web search systems.

The rest of this paper is organized as follows. Section 2 presents a result page layout model. Section 3 provides an overview of our solution. Section 4 introduces the various content features on result web pages as well as various measures that are defined based on these features. These features and measures will be used by our extraction method. Section 5 discusses the details of our proposed solution to the section extraction problem. Section 6 reports the experimental results. Section 7 reviews related works. Section 8 concludes the paper.

2. RESULT PAGE LAYOUT MODEL

For a given search engine, search result pages are usually produced by a script program. The designer of the program has a layout plan for all the contents on the result pages. This layout plan is essentially the ***result page schema*** of the search engine, which can be represented as (D, S, SD, L), where D, S and SD are dynamic sections, static contents and semi-dynamic contents, respectively, and L is the layout relationship between these sections/contents. Let D = (S_1, …, S_m) be *all the possible* dynamic sections a result page may have based on the result page schema. Each S_i will be called a ***section schema***. An individual result page is an *instance* of the result page schema and a specific section is an instance of a section schema. It is possible that some section schemas have no instances on a particular result page (e.g., if no result is retrieved for a particular section).

While the contents on a result page are laid out in a two-dimensional space when the page is rendered on browsers, they can be represented in a one-dimensional space. DOM trees are widely used to represent web pages (see Figure 2 for an example). Non-tag contents, which are generally viewable contents on a browser, are leaves in DOM trees. A preorder traversal of a DOM tree of all non-tag contents will yield a sequence of the non-tag contents. Based on this view, the layout relationship L becomes a sequence of sections (Figure 3 provides an illustration, where the static sections form the *template*).

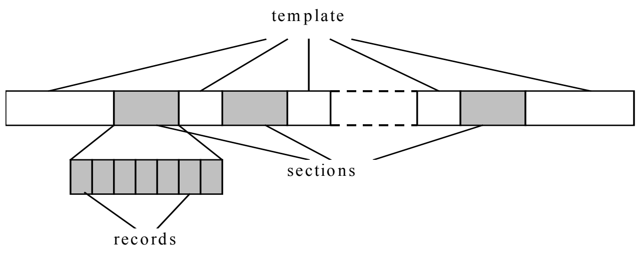


Figure 3. Sections, records, and template

We are interested in extracting all dynamic sections and the records within them only. If, for a given search engine, we can identify all sections with the correct order and construct a wrapper to extract the records for each section, then the list of wrappers with the same order as the sections will be a complete description of the rules for extracting sections as well as records from the result pages returned by the search engine.

To facilitate people locating information, search engine result pages often place special information at the boundaries of a section. We will call such information as **s**ection **b**oundary **m**arkers (SBM) in this paper. Based on the one-dimensional representation of web page content, a section may have a left boundary marker (LBM) and a right boundary marker (RBM). In Figure 1, “Encyclopedia” and “Click Here for More…” are the LBM and the RBM of the first section, respectively.

SBMs could help extract sections and records they contain if they can be correctly identified. In some cases, SBMs are a must for correct section extraction. Consider the sample page in Figure 1. The LBM of each section is the section header (e.g., “Encyclopedia” for the first section). Since all sections on this page have exactly the same tag structures, without considering the SBMs, correctly extracting these sections would be very difficult, if not impossible. Our investigation based on the result pages of 200 search engines shows that 96.9% of the sections have explicit boundary markers. How to accurately identify these boundary markers is an important problem we need to solve in this paper.

Our section extraction method tries to generate section wrappers by identifying SBMs first. Experiments shows that this strategy can achieve promising performance.

3. SOLUTION OVERVIEW

Figure 4 shows the system overview of our solution. Our wrapper generation algorithm will be called **MSE** (for **M**ultiple **S**ection **E**xtraction). The input to MSE is a set of *n* sample result pages from a search engine *SE*. These result pages are returned from *SE* in response to *n* different queries. The output of MSE is a wrapper (a set of rules) for extracting all dynamic sections (DSs) as well as all SRRs within them.

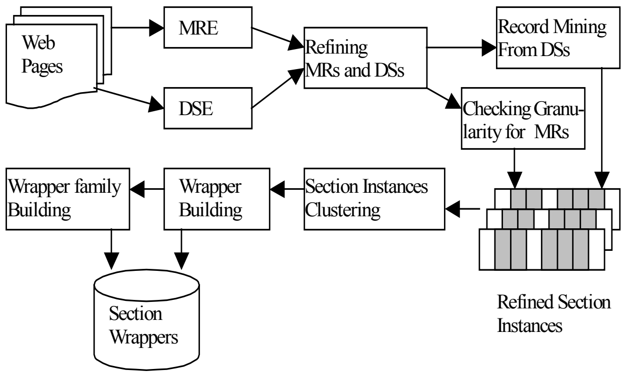


Figure 4. System overview

MSE consists of the following steps:

1. Render each result page and extract its content lines by a preorder-traversal of the DOM trees. We assign a line number (1, 2, …) to each content line.
2. Extract record sections that contain multiple records from each result page. These sections will be called *multi-record sections* and denoted as MRs and the algorithm for extracting MRs will be denoted as **MRE** (for MR Extraction). (Note: The MRs identified by algorithm MRE may contain MRs with static contents and MRs with incorrect boundaries. In addition, sections that contain less than three records are generally not identified by MRE.)
3. Identify dynamic sections (DS) by applying algorithm **DSE** (for DS Extraction). In order to perform this task, we need to identify candidate section boundary markers (CSBM) on each page. (Note: Some DSs identified by DSE in this step may be incorrect due to the difficulty to correctly identify all the SBMs.)
4. Refine MRs and DSs by analyzing their relationships. (By comparing MRs with DSs, MRs containing static contents can be identified and discarded, and some incorrect boundaries of MRs and DSs can be corrected. Note that, in order to deal with the *non-uniform section format problem*, neither MRE nor DSE assumes there is a common format/pattern among different sections when performing section extraction.)

5. Mine SRRs from DSs that have no corresponding MRs. (These DSs include sections that contain less than three SRRs. As a result, even a single SRR could be extracted from a section.)
6. Check identified sections and records to see if they are correctly identified. (This is to tackle the *section-record granularity problem*.)
7. Group extracted section instances from all sample result pages into clusters such that each cluster corresponds to the same *section schema* of the *result page schema* of the search engine.
8. Generate the extraction wrapper for each section schema based on the section instances in the corresponding cluster.
9. Generate *section families*, each of which is a class of section schemas that share some common features. (Section families are introduced to tackle the *hidden section extraction problem*.)

Step 1 has already been discussed in our previous work [29] and will not be repeated in this paper. The details of the remaining steps will be provided in Section 5.

4. BASIC CONTENT FEATURES OF RESULT PAGES

In this section, we introduce the features that can be found from typical search result pages and are useful to our section extraction method. Among the tag structure features presented in Section 4.1, *tag paths* will be used in wrapper description to locate the sections, and are also used in DSE and section instance clustering to compare the contents on different web pages; the *tag tree edit distance* and *tag forest edit distance* are used for record mining. Section 4.2 presents basic visual features such as *content lines*, *block*, *shape*, etc, which are mainly used in MRE. We combine the tag structure features and visual features in Section 4.3 and Section 4.4 to define the *line distance*, *record distance*, *inter-record distance, record diversity* and *section cohesion*. They are used for record mining and for differentiating sections and records. In section 4.5, we introduce *section boundary markers*, which can precisely bound sections and will be used in DSE, section refining, section instance clustering and wrapper building.

4.1 Tag Structure Features

A DOM tree of an HTML web page is a rooted, ordered, and labeled tree. Figure 2 shows part of the DOM tree of the web page in Figure 1 (many tag nodes are omitted for simplicity).

All viewable content fragments on the rendered web page on a browser have a corresponding tag structure underneath. For each record, or section, we may extract its underneath tag structure, which normally is a tag forest. For example, the tag structure within the dotted ellipse in Figure 2 is the tag forest of the first record in Figure 1. For each section, there exists a minimum sub-tree *t* in the DOM tree of a result page such that all SRRs in the section are located in *t*. Each SRR corresponds to a sub-forest in *t*.

A node in a DOM tree can be located by following a path from the root to the node. Such a path is called a *tag path* in [29] (which is similar to an XPath). 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", called S node) or the first child of *pn* (indicated by "C", called C node). The tag path of the text "Your search returned 578 matches" in Figure 2 is "{HTML} C {HEAD} S {BODY}C{TABLE}S{TABLE}S{TABLE}C {TBO DY}C {TR} C{TD}S{TD}S{TD}S{TD}C{TABLE}S{TABLE}S{TABLE}C{ TBODY}C{TR}S {TR}S{TR}C{TD}C".

Clearly, any node *n* on a DOM tree can be located by following the tag path of *n*. *Compact tag path* was proposed in [29] to remove "noises" on the original path, making it more robust when matching paths from the DOM trees of different pages. Two compact tag paths are *compatible* if and only if they contain the same sequence of C nodes. Let $<c1_1, c1_2, \ldots c1_n>$ and $<c2_1, c2_2, \ldots c2_n>$ be the sequences of C nodes of two *compatible* tag paths tp_1 and tp_2, let $sn(c_i, c_j)$ denote the number of S nodes between C nodes c_i and c_j, we define the distance between tp_1 and tp_2 as:

$$Dtp(tp_1, tp_2) = \frac{\sum_{i=2}^{n} \left| sn(c1_i, c1_{i-1}) - sn(c2_i, c2_{i-1}) \right|}{\max(sn(c1_n, c1_1), sn(c2_n, c2_1))} \quad (1)$$

Since the underneath tag structure of any viewable content fragments on the rendered web page is (part of) a tag forest, we define a metric to measure the similarity between two tag forests. We use $Dtt(t_1,t_2)$ to denote the *tree edit distance* [9] between two tag trees t_1 and t_2 normalized by the size of the larger tree between t_1 and t_2. Each tag forest *tf* can be considered as a string (ordered list) of tag trees $<t_1, t_2, \ldots t_k>$. We use $Dtf(tf_1, tf_2)$ to denote *string edit distance* [24] between two tag forests tf_1 and tf_2 normalized by the length of the longer list between tf_1 and tf_2.

4.2 Visual Content Features

HTML tags convey rich semantics for web content presentation. Tag attributes and styles enrich web content further. Web data extraction techniques that use tag structures only will surely miss many important features of HTML. In this subsection, we introduce some visual content features that can be extracted from rendered web pages and used to improve data extraction performance.

In this paper, we follow the method in [29] and define *content lines* as the basic constructs to capture visual features. One big advantage of using lines instead of tokens (like in many other studies) as the basic constructs is that a line consisting of multiple tokens has more precise semantic meaning than individual tokens (analogous to the relationship between phrases and individual words). A content line *cl* is a group of characters that visually form a horizontal line in the same section on the rendered page. Eight content line types (e.g., text line, link line, HR-line, etc.), each with a *type code*, are defined in [29] to capture the basic appearances of content lines. Also the left-most *x* coordinate of a content line on the rendered page is called the *position code* of the content line. One or more consecutive content lines form a *block B*, which is an ordered list $<cl_1, cl_2, \ldots, cl_k>$ such that cl_i represents the *i*th content line in *B*. Any search result record on a rendered web page is a block. For each block, a *block shape* (the left contour of the block as defined by the position code sequence of its member content lines) and block *type code* (the sequence of type codes of the content lines) can also be defined to capture the appearance of the block [29].

Based on the above concepts, the similarity between two blocks can be measured in terms of *type distance*, *shape distance* and *position distance* (Please see [29] for details.)

In this paper, we introduce *text attribute* to capture more information about content lines. For a piece of text on a rendered web page, its text attribute represents the font (arial, times new roman, etc), size, style (plain, bold, and italic) and color (red, black, etc) of the text. Each text attribute *ta* is a quaternion *<f, w, s, c>*, where *f*, *w*, *s* and *c* represent font, size, style and color, respectively.

A content line *cl* may contain texts with different text attributes. A set {*ta*}, denoted *la*, is defined to represent line text attribute. Each member *ta* is a text attribute in *cl*. We define the *line text attribute distance* between the line text attributes la_1 and la_2 of two content lines as:

$$Dtal(la_1, la_2) = 1 - \frac{|la_1 \cap la_2|}{\max(|la_1|, |la_2|)} \qquad (2)$$

We use an ordered list < la_1, la_2, …, la_k>, denoted as *ba*, to further represent the text attribute of a block <cl_1, cl_2, …, cl_k>, where la_i is the text attribute set of cl_i. The *block text attribute distance Dbta* between the text attributes ba_1 and ba_2 of two blocks B_1 and B_2 is defined as the string edit distance between ba_1 and ba_2.

4.3 Line Distance, Record Distance, Inter-Record Distance

Consider two content lines cl_1 and cl_2, with type codes tc_1 and tc_2, position codes pc_1 and pc_2, and line text attributes la_1 and la_2, respectively. The type distance *Dtl* between cl_1 and cl_2 is a value between 0 to 1 based on tc_1 and tc_2. The position distance *Dpl* is defined as $K * \log(1+|pc_1 - pc_2|)$; currently *K* is set to 0.127, which will restrict *Dpl* to be between 0 to 1 in most cases. The line text attribute distance is *Dtal*(la_1, la_2) as defined in Formula 2 above. In this paper, we define the *line distance* between cl_1 and cl_2 as follows:

$$Dline(cl_1, cl_2) = u_1 \times Dtl + u_2 \times Dpl + u_3 \times Dtal \qquad (3)$$

where u_1, u_2 and u_3 are non-negative real numbers satisfying $u_1 + u_2 + u_3 = 1$.

Each record is a block. We have defined *tag forest distance Dtf*, *block type distance Dbt*, *block shape distance Dbs*, *block position distance Dbp* and *block text attribute distance Dbta*. We normalize block type distance in [29] to between 0 and 1, and we modified the block shape distance and block position distance definitions in [29] to normalize their values as well. Now we can define the *record distance* between two records r_1 and r_2 as follows:

$$\begin{aligned} Drec(r_1, r_2) = v_1 \times Dtf + v_2 \times Dbt + v_3 \times Dbs \\ + v_4 \times Dbp + v_5 \times Dbta \end{aligned} \qquad (4)$$

where v_1, v_2, v_3, v_4 and v_5 are non-negative real numbers satisfying $v_1 + v_2 + v_3 + v_4 + v_5 = 1$.

For a section *S* with *n* records <r_1, r_2, …, r_n>, we compute the average distance between the records in *S* to measure the *inter-record distance* of *S*, denoted as *Dinr*(*S*). This distance will be used in Section 4.4 to introduce an important measure for our method.

$$Dinr(S) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Drec(r_i, r_j) \Bigg/ \binom{n}{2} \qquad (5)$$

4.4 Record Diversity and Section Cohesion

A section *S* can be considered as a list of records or a list of content lines. After the content lines are obtained, we face the question of how to correctly partition/group these content lines into records. Previous works [5, 11, 15, 29] use DOM tree structures to find a tag structure as a separator to create a partition. There are two potential problems with these methods. First, consecutive records may be mistakenly combined into a big record. Second, a correct record may be wrongly split into several small false-records. In this paper, we solve the content line partition problem differently by using a measure called *section cohesion*, such that the higher the cohesion of a partition is, the more likely the partition is correct.

Our cohesion definition is based on the following observations: the records within a section tend to be similar to each other, while the lines within a record tend to be dissimilar to each other. To measure the degree of dissimilarity of the lines in a record, we define *record diversity* based on the *line distance* defined in Formula 3. More specifically, for a given record *r* with content lines <l_1, l_2, … l_m>, its record diversity *Div*(*r*) is defined as:

$$Div(r) = \sum_{i=1}^{m-1} \sum_{j=i+1}^{m} Dline(li, lj) \Bigg/ \binom{m}{2} \qquad (6)$$

The inter-record distance defined in Formula 5 measures the overall record similarity among all records of a section. Thus we have the following definition for the *cohesion* of a section *S* with records <r_1, r_2, …, r_n>:

$$Cohs(S) = \frac{\sum_{i=1}^{n} Div(r_i) \Big/ n}{1 + Dinr(S)} \qquad (7)$$

In summary, a good partition of a section should have high record diversity and low inter-record distance. With the concept of section cohesion, a new approach for partitioning content lines into records can now be used, i.e., by finding the partition with the highest cohesion. This method does not need to identify separators based on the tag structures. Alternatively, if different candidate separators exist, this approach can be used to determine which separator is most likely to be correct, i.e., the one that leads to the partition with the highest section cohesion.

4.5 Section Boundary Marker

We define the section boundary markers (SBMs) of a section *S* as content lines that are not members of any sections, and are located closest to *S* on the result page. More specifically, we define the left (right) section boundary marker LBM (RBM) of *S* as the content line that is not a member of any sections, and is located closest to *S* on the beginning (ending) side. SBMs are important for identifying dynamic sections. Our method for finding SBMs will be discussed in Section 5.2.

5. SECTION WRAPPER BUILDING

In this section, we provide the details of Steps 2-9 (see Section 3) of our dynamic section extraction algorithm.

5.1 MR Extraction with MRE

The MRE algorithm is revised from the ViNTs algorithm in [29] and is briefly reviewed here for the convenience of the readers. For each result page, it identifies consecutive content line patterns that occur more than two times. The pattern here refers to the sequence of content line types and positions. Then the list of content lines is partitioned into blocks by an identified pattern, such that each block contains the pattern and the pattern is located at the ending part of the block. There will be *n* different partitions if there are *n* patterns. For each partition, the extracted blocks are grouped by putting consecutive and *visually similar* [29] blocks into the same group. In this way, we obtain a set of groups, which are candidate sections, whose member blocks are candidate records.

Candidate records may be real records, but they may also be blocks containing content lines from different records, or even false records. ViNTs algorithm [29] then identifies the first line of record within the content lines of each candidate record. Clearly, if the first lines of some consecutive records can be correctly identified, these records can also be correctly identified. Then the tag path to each identified first record line is used to represent the corresponding candidate record. Next, tentative wrappers are built from each set of three tag paths of every three consecutive candidate records. Each tentative wrapper goes through a verification process to see if it should be kept. Finally, verified wrappers are refined by finding appropriate section boundaries. The wrapper building, refining and verification steps of MRE are the same as in ViNTs [29].

We call the sections generated by applying tentative wrappers *tentative multi-record sections* (MRs), and they normally contain 4 or more records. Next, we merge tentative MRs: if two MRs overlap considerably, we merge them into one group. We then apply a wrapper selection algorithm similar to the one in ViNTs to find the best MR for each MR group. Those best MRs are the extracted multi-record sections.

The main difference between MRE and ViNTs is that ViNTs assumes there is only one (major) MR to be extracted, while the goal of MRE is to extract all MRs on a web page. Thus ViNTs compares all tentative MRs to find the best one as the major MR, while MRE groups tentative MRs by the screen areas they occupy, and then find out the best MR for each tentative MR group.

Using only MRE to extract MRs has four potential problems. First, the boundary problem, i.e., some records near the two boundaries of an MR may be incorrectly extracted. Second, sections with less than three records will not be extracted. Third, some extracted sections may contain static contents with repeating patterns. Fourth, some extracted MRs may mistakenly take consecutive sections with the same format as records, and some large records may be incorrectly extracted as sections. In this paper these problems will be dealt with in subsequent steps to be described in the following subsections.

5.2 Identifying DSs with DSE

In this section, we present the DSE algorithm (Figure 5) for identifying DSs. This algorithm consists of two main steps: the first step (lines 1-11) identifies CSBMs (candidate SBMs) and the second step (lines 12-13) identifies DSs based on the CSBMs.

DSE works on a pair of rendered sample result pages $<p_1, p_2>$ at a time. Let L_1 and L_2 represent the content line sets of p_1 and p_2. We first clean semi-dynamic content lines by removing dynamic component(s) (lines 1-2). In order to identify SBMs that are semi-dynamic in nature but have dynamic components, we need to remove these dynamic components from all content lines. For example, content line "Your search returned 578 matches" in Figure 1 has a dynamic component "578" (it is query dependent) and it needs to be removed in order to match a possible content line, say "Your search returned 89 matches", on another result page. To achieve this, we remove all numbers and query terms (which were used to retrieve the result page) from all content lines. From now on, we consider only content lines with dynamic components removed.

```
Algorithm DSE($L_1$, $L_2$)
1 for each line $l \in L_1$ or $L_2$
2    clean_line($l$);    /* remove dynamic components */
/* Identify CSBMs */
3 for each line $l \in L_1$
4    $mc(l)$ = find_most_compatible_line($l$, $L_2$);
5 for each line $l \in L_2$
6    $mc(l)$ = find_most_compatible_line($l$, $L_1$);
7 for each line $l \in L_1$ or $L_2$
8    if ($l = mc(mc(l))$) then
9       mark_as_CSBM($l$);
10 filter_CSBMs($L_1$);
11 filter_CSBMs($L_2$);
/* Identify DSs based on found CSBMs */
12 identify_DSs($L_1$);
13 identify_DSs($L_2$);
```

Figure 5. Algorithm DSE

Next we identify tentative CSBMs (lines 3-9). We begin by identifying matching content lines from L_1 and L_2 such that they are likely to be SBMs. SBMs are typically static contents or semi-dynamic contents on result pages. As a result, these content lines are likely to appear in both L_1 and L_2. Because the SBMs on the result page of a search engine are part of the template of the result page, the corresponding SBMs from two result pages of the same search engine usually have compatible tag paths. Thus procedure find_most_compatible_line(l, L) first finds the content lines in L that have the same text content as l and their tag paths are also compatible to l's tag path. As l might match multiple content lines in L, the procedure returns the line with the smallest tag path distance (see Formula 1) to l as the most compatible line of l, denoted $mc(l)$. To reduce false match, a content line l is recognized as a tentative CSBM if the most compatible line of $mc(l)$ is l itself (lines 7-9). In other words, we only recognize content line pairs (l_1, l_2), $l_1 \in L_1$ and $l_2 \in L_2$, that satisfy $mc(l_1) = l_2$ and $mc(l_2) = l_1$, as tentative CSBMs.

The algorithm next proceeds to filter out false tentative CSBMs. Some strings (patterns) that appear in many records in a section may lead to false SBMs. For example, in the result pages returned by Amazon.com, "Buy new: $XXX.XX" occurs frequently and it would be recognized as a tentative CSBM by the above process. The procedure filter_CSBMs(*L*) checks tentative CSBMs of *L* against the MR set extracted from the page of *L*. If a tentative CSBM appears in all member SRRs in any MR, it is removed.

The second step (lines 12-13) of the DSE algorithm is to identify DSs based on the CSBMs found in the first step. The procedure identify_DSs(*L*) works as follows. First, the list of content lines of the page is partitioned into segments, such that each segment *G* satisfies the following conditions: (1) *G* contains consecutive content lines that are either all CSBMs or all non-CSBMs; and (2) *G* is not a proper sub-segment of any possible segment satisfying condition 1. The segments that contain consecutive content lines but are not CSBMs are recognized as (candidate) DSs. Each DS has a LBM (left boundary marker) and a RBM (right boundary marker), which are CSBMs and are not part of the DS.

5.3 Refining MRs and DSs

Now we have a set of MRs and a set of DSs. Each MR has the following properties: it contains three or more candidate records, which may not necessarily be correct (especially for records near the boundaries); the tag structures (tag forests) of member records are siblings in a common sub-tree of the DOM tree; the tag structures of all member records are similar; and, the content of records may be static or dynamic. As for each DS, its contents are always dynamic but its records have not been identified; its candidate LBM and RBM have been identified, but some of them may be incorrect.

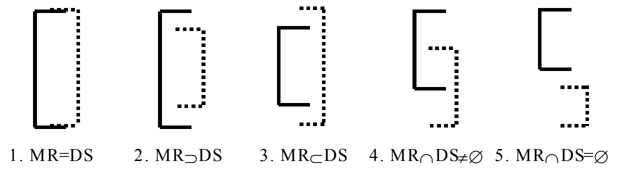


Figure 6. Five cases of relationships between MRs and DSs

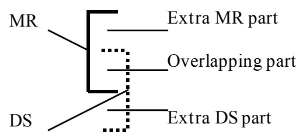


Figure 7. Case 4 in details

Recall that our goal is to extract records from DSs. If a DS contains at least three records, then it should match an extracted MR. Note that MRs and DSs are obtained independently using different techniques. The main idea of the refining process is to utilize matching MRs and DSs to help obtain the correct boundaries for both MRs and DSs. We analyze the relationships between MRs and DSs based on the screen areas they occupy on the rendered page. Five cases can be identified as shown in Figure 6: (1) an MR matches a DS exactly; (2) an MR contains some DSs; (3) a DS contains some MRs; (4) an MR intersects with a DS, and (5) an MR has no overlap with any DS, or a DS has no overlap with any MR.

Case 1 reflects a perfect match. In this case, we have high confidence that the MR and DS both have correct boundaries. We keep such MRs and their records become the records in the corresponding DSs. For other cases, further work is needed. We discuss Case 4 and Case 5 only here as Cases 2 and 3 can be similarly handled as Case 4.

For Case 4 (see Figure 7), three parts are identified: the extra MR part (*EM*), the overlapping part (*OL*) and the extra DS part (*ED*). Since the records in *OL* are verified by both MR and DS, we have high confidence that they are correct. The basic idea of our solution for finding the correct boundaries for DS and MR is based on the assumption that the records within the same section (either MR or DS) are similar and should have small record distances. More specifically, we achieve our goal by comparing the records in *OL* with the records in *EM* and *ED*. The algorithm for Case 4 is sketched in Figure 8.

```
Algorithm Refine_MR_DS_4($MR$, $DS$)
1 partition $MR \cup DS$ into three disjoint sets {$EM$, $OL$, $ED$};
  /* Dealing with EM part. */
2 while $\exists$ $br \in EM$ and LBM $\in br$ /*LBM is the LBM of DS */
3     if $Davgrs(br, OL) > W * Dinr(OL)$ /* LBM is verified */
4         adjust $MR$ by discarding $EM$; break;
5     else /* LBM is false */
6         find a new LBM in $EM$;
  /* Dealing with ED part. */
7 while $ED \neq \Phi$
8     form tentative records $T$: {$rt_1, rt_2, \ldots, rt_k$} from $ED$;
9     find $rt^* \in T$ with smallest $Davgrs(rt^*, OL)$;
10    if $Davgrs(rt^*, OL) \leq W \times Dinr(OL)$
11        add $rt^*$ to $MR$ and truncate $ED$;
12    else break;
13 if $ED \neq \Phi$
14    take $ED$ as a new $DS$;
```

Figure 8. Refining MR and DS: Case 4

Since the LBM of the DS is the content line immediately before the first content line of the DS, there is a record *br* in *EM* that contains the LBM of the DS. To determine if the current LBM is correct, we compute the average record distance between *br* and all the records in *OL*, denoted as *Davgrs*(*br*, *OL*), based on the record distance *Drec* (Formula 4). If *Davgrs*(*br*, *OL*) > *W* * *Dinr*(*OL*), where *Dinr*(*OL*) is the inter-record distance between the records in *OL* (see Formula 5) and *W* is a parameter (1.8 is currently used), we consider *br* to be a record not matching the records already in the DS. As a result, we confirm the correctness of the LBM. And, accordingly, we truncate MR by discarding *EM*. Otherwise the LBM is considered to be incorrect. In this case, we find a new LBM for the DS. The new LBM is the CSBM immediately before the discarded LBM. We then adjust the *EM* (it becomes smaller) based on the new LBM. This process is repeated until a LBM is verified or the *EM* becomes empty.

The *ED* contains a list of content lines $<l_1, l_2, \ldots, l_k>$. We try to form a tentative record by using l_1. Then another tentative record by using l_1 and l_2 together, then another tentative record by using l_1, l_2 and l_3 together, etc. By repeating this process, we will have k tentative records $\{rt_1, rt_2, \ldots, rt_k\}$, where rt_i consists of $<l_1, l_2, \ldots, l_i>$. For each rt_i, we compute $Davgrs(rt_i, OL)$, the average record distance between rt_i and all records in *OL*. Let rt^* be the tentative record with the smallest *Davgrs*. If $Davgrs(rt^*, OL) \leq W \times Dinr(OL)$, we accept rt^* as a record in MR and adjust the *ED* accordingly, and then check the new *ED* in the same manner. If $Davgrs(rt^*, OL) > W \times Dinr(OL)$, we stop right here and if the remaining *ED* is not empty, we form a new DS based on remaining *ED* for further processing (see Section 5.4).

As for Case 5, we discard the MRs that do not overlap with any DS because they are static items with repeating patterns. We keep all DSs (including the newly formed DSs) that do not overlap with any MRs because they are dynamic content for sure. These DSs usually have less than three records and cannot be detected by the MRE algorithm described in Section 5.2.

In Section 5.4, we consider how to extract records from these DSs.

5.4 Mining Records from DSs

A DS consists of consecutive content lines with none of them being CSBMs. We introduce an algorithm to mine records from a DS. An advantage of the algorithm is that it does not require the input DS to contain multiple records. It has the potential to identify even a single record in a DS.

We use tag forest separators (follow the approach in [29]) to partition the content (a tag forest) in a DS into records. The algorithm first tries to identify all possible tag forest separators, and then select the best partition as output.

Suppose there are *m* partitions corresponding to *m* tentative tag forest separators. We need to select the best partition to turn the DS into an MR with a set of records. The main idea of our selection criteria is based on the definition of *section cohesion* in Section 4.4 (Formula 7). We compute section cohesions for all partitions and select the partition with the highest cohesion as the record mining result.

Based on Formula 7, if a DS has just one record, the partition that produces the entire DS as a single record is likely to be selected. Thus the algorithm has the ability to find the only record in a DS.

At the end of this step, every DS is turned into an MR (i.e., the records within the DS have been identified) and some MRs may have less than 3 records, and the SBMs of the DSs become the SBMs of the corresponding MRs.

5.5 Solving Section-Record Granularity Problem

The MRs generated by MRE and the refining step may have the section-record granularity problem. There are two cases. First, some consecutive sections with the same format may be mistakenly considered as records and grouped together to form an MR, or some consecutive records in one section may be mistakenly combined as one large single record. We refer to such a problem as *oversized-record problem*. Second, some large records may be incorrectly extracted as sections. This is referred to as the *splitting-record problem*. The splitting-record problem also refers to the situation where some record in an MR is incorrectly split into two or more smaller records while the section MR itself is correctly identified.

To deal with the *oversized-record problem*, for a given MR, we check its largest record LR (having largest number of content lines) first. We apply the records mining algorithm described in Section 5.4 in an attempt to mine records from LR. If only one record can be found in LR, we assume there is no *oversized-record problem* for this MR. Otherwise, we keep the newly mined small records and keep checking other large records in this MR. Still, we need to differentiate the case of sections being considered as records with the case of consecutive records being combined into a single record while the section is correctly identified. For the former case, we need to partition the original MR and recognize each record as a section. While for the latter case, we only need to adjust the record partition in the original MR.

Let $<R_1, R_2, \ldots, R_n>$ represent the original MR. We check each pair of consecutive records. Without lose of generality, let R_1 and R_2 represent the two consecutive records in question. Suppose after record mining, R_1 consists of small records $<r_{11}, r_{12}, \ldots, r_{1u}>$ while R_2 consists of small records $<r_{21}, r_{22}, \ldots, r_{2v}>$. If R_1 and R_2 are actually sections, there should be some kind of visual representation, such as SBMs, a space, etc., to let the user separate them visually. While normally the case involving SBMs can be taken care of by the refining process described in Section 5.3, it is reasonable to think that there should be a special tag structure between R_1 and R_2 to separate them in all cases. Since R_1 and R_2 are consecutive, such a special tag structure must be either part of the last small record r_{1u} in R_1, or part of the first small record r_{21} in R_2, which will make r_{1u} or r_{21} special. To detect this property, we compute the inter-record distance between the small records in R_1 and R_2, referred to as $Dinr(R_1)$ and $Dinr(R_2)$ respectively. Then we compute the average record distance between r_{21} and all the records in R_1, denoted as $Davgrs(r_{21}, R_1)$, and the average record distance between r_{1u} and all the records in R_2, denoted as $Davgrs(r_{1u}, R_2)$. If $Davgrs(r_{21}, R_1) > W \times Dinr(R_1)$ or $Davgrs(r_{1u}, R_2) > W \times Dinr(R_2)$, where W is a parameter (1.8 is used currently), we recognize R_1 and R_2 as actually sections, and we adjust the original section MR accordingly, i.e., R_1 and R_2 are removed from MR. Otherwise we use the mined small records to replace the original big record in the MR.

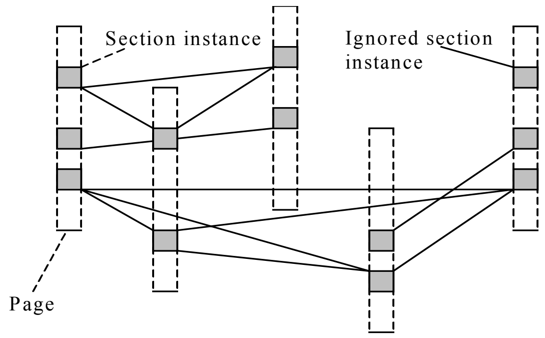
Now we deal with the *splitting-record problem*. There are also two sub-cases: the sub-case where some large records are extracted as sections, and the sub-case where a record is split into smaller records while the section is correctly identified. We present how to check the existence of the latter case for each MR first.

Let $<r_1, r_2, \ldots, r_n>$ be the records in the MR that contains a possible large record L in question. We consider all the records in MR together rather than only those in L. Note that $<r_1, r_2, \ldots, r_n>$ represent one possible way of partitioning MR into records. We tentatively combine each pair of two consecutive records as one possible large record: $<r_1, r_2>$ as the first record, $<r_3, r_4>$ as the second record, etc. By doing that, we have a new record partition of MR. We do this for each three, four, … and n consecutive records to get a series of new partitions of MR. Then we compute the cohesion of each partition using Formula 6. If a partition *P* other than $<r_1, r_2, \ldots, r_n>$ has the highest cohesion, we would take *P* as the partition of MR.

Note that records in all MRs have been checked by the above step. Then we check the existence of the case where large records are extracted as sections. If there exists a set of consecutive MRs that are siblings under the same sub-tree of the DOM tree, and all MRs in the set consist of only one record, then these sections are likely to be large records mistakenly recognized as sections. In this case, we form a new section with each original section in the set as a record and remove the original sections.

5.6 Grouping Section Instances of the Same Section Schema

At this stage, we have a set of refined MRs for each sample page. To improve reliability, an MR on one sample page is certified only if it matches with an MR in at least another sample page. All matching MRs from different sample pages are section instances of the same *section schema*. We apply the *stable marriage algorithm* [16] here to find out the matching MRs, with a minor modification to allow no match. A matching score is computed between two MRs from two pages based on their tag path similarity, SBM similarity and tag forest similarity. If the matching score of two



MRs is below a threshold they will not be matched even when they have the highest matching score among all the MR pairs from the two sample pages.

Figure 9. An example graph of section instances

Consider an undirected graph SG = (V, E), where each vertex $v \in$ V corresponds to an MR on a page and an edge $e \in$ E exists between two vertices v_i and v_j if the MR corresponding to v_i on one page matches the MR corresponding to v_j on another page. Figure 9 shows an example.

A clique C (Vc, Ec) is a sub graph of SG (V, E) such that Vc $\subset$ V and Ec $\subset$ E and for every $v_i \in$ Vc and $v_j \in$ Vc, $i \neq j$, there is an $e \in$ Ec between v_i and v_j. Each maximum clique in SG of size 2 or greater is a *section instance group* of the same section schema. We apply the Bron-Kerbosch algorithm [4] to find all maximum cliques in SG of size 2 or greater and take these cliques as the section instance groups for wrapper building. By doing this, we ignore dangling section instances (MRs) that have no matches in any other sample page.

5.7 Constructing Section Wrappers from Section Instance Groups

A section wrapper is a set of rules that can be applied to the tag tree of a result page to extract a section and the records it contains. In this paper, a section wrapper is a quaternion <*pref, seps, LBMs, RBMs*>, where *pref* represents the tag path that leads to the minimum sub-tree *t* that contains all records in this section, *seps* is the separator set used to partition the sub-forest of *t* into records, LBMs and RBMs are the sets of left and right boundary markers of the section. The same techniques for identifying *pref* and *seps* as proposed in [29] are adopted in our approach.

For each group of matching section instances (MRs), we combine the tag paths of member sections that lead to the minimum sub-trees containing all the records to generate *pref*, then combine their separators, LBMs and RBMs to generate *seps*, *LBMs*, and *RBMs*, respectively, for the corresponding section schema. Combining tag paths is carried out by converting each tag path into a *compact tag path* (see Section 4.1) and merging the compact tag paths [29]. While combining LBMs and RBMs is based on simple majority vote.

5.8 Section Family

Because we build section wrappers based on sample pages, due to the step described in Section 5.6, only those section schemas with instances occurred in at least one pair of sample pages could be detected and certified. The current set of section wrappers will miss so-called *hidden sections* (i.e., section instances of those section schemas that do not occur in any sample page) as well as section instances that occur on only one sample page.

We propose the concept of *section family* to deal with the *hidden section extraction problem*. This method can also handle the section instances that occur on only one sample page. A section family represents a class of section schemas that share some common features. The following are two types of section families that are among those considered by our approach:

1. All member section schemas have the same *pref* and *seps*, and their *LBMs* (*RBMs*) share the same line text attribute, which is different from the line text attribute of any content line in any record. Figure 10 shows the tag structure of an example. The *pref* for both sections is <HTML>C<BODY> C …<TBODY>C, and the *seps* for both sections contains only one sub-tree rooted at a <TR>. For this example, we assume that their *LBMs* (*RBMs*) share the same line text attribute.
2. All member section schemas have the same *seps*, and their *pref*s have the same prefix as well as the same suffix, and their *LBMs* (*RBMs*) share the same line text attribute, which is different from the line text attribute of any content line in any record. Figure 11 shows the tag structure of an example. We assume the *seps*, *LBMs* and *RBMs* satisify the condition as Type 1 section family. Note that the *pref* of section 1 is: <HTML>C<BODY>C…<TBODY>C<TR>S<TR>S<TR>C<TD>C, while the *pref* of section 2 is: <HTML>C<BODY>C…<TBODY>C<TR>S<TR>S<TR>S<TR>S<TR>C<TD>C. They share the same prefix <HTML>C<BODY>C … <TBODY>C<TR>S <TR>S and the same suffix <TR>C<TD>C.

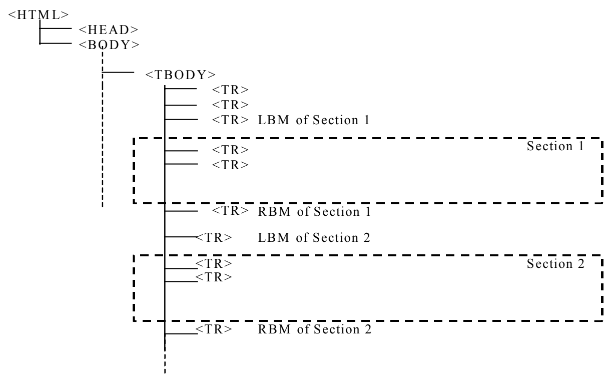


Figure 10. Tag structure illustrating a Type 1 section family

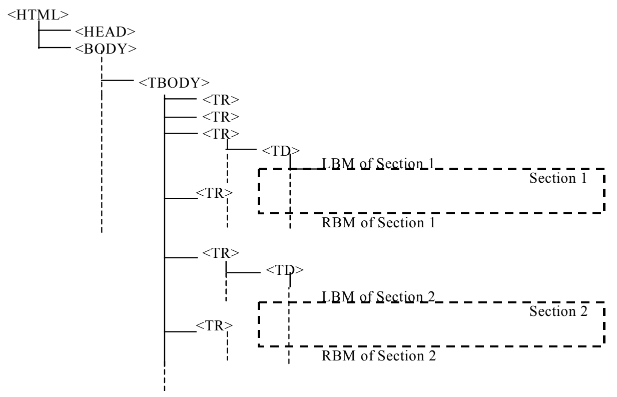


Figure 11. Tag structure illustrating a Type 2 section family

In a Type 1 section family, all records in all member section schemas are siblings right under the sub-tree denoted by *pref* on the DOM tree. Thus the Type 1 section family wrapper can be represented as a quaternion <*pref, seps, aLBMs, aRBMs*>, where *aLBMs* and *aRBMs* represent the text attributes of the left and right boundary markers, respectively. In a Type 2 section family, all member section schemas are siblings right under the sub-tree denoted by the common prefix of *pref*s, while the common suffix of *pref*s are the relative tag paths that lead to the sub-trees containing the records in each member section schema. Thus we can use a quintuple <*ppref, spref, seps, aLBMs, aRBMs*> to represent a Type 2 section family wrapper. Here *ppref* is the common prefix of the *prefs* of all member section schemas, while *spref* is the common suffix of the *prefs* of all member section schemas.

We check all section wrappers produced in Section 5.7 to see if some of them can be combined to form Type 1 or Type 2 section wrapper families. If so, the corresponding section wrapper family is constructed and the original section wrappers from which the section wrapper family is built are deleted.

6. EXPERIMENTS

We built a prototype system to test the MSE algorithm. On a laptop with a Pentium M 1.3G processor, the system can construct section wrappers for a search engine with 5 sample pages in 20 to 50 seconds. Once the wrappers are built, the section and record extraction from a new result page can be done in a small fraction of a second.

The test bed consists of 100 search engines from the ViNTs test bed dataset 2 [29], plus 19 additional search engines that organize their SRRs into multiple sections. Among the 100 search engines from the ViNTs test bed dataset 2, 19 return result pages with more than one dynamic section. Therefore, 38 search engines in the test bed return multiple dynamic sections, while 81 search engines return only one dynamic section.

For each search engine, 10 result pages are collected by manually submitting 10 different queries. In order to test the robustness of the extracted wrappers, we partition the 10 pages into two subsets: 5 sample/training pages and 5 test pages. Only the sample pages are used for wrapper construction and parameter/threshold tuning. Then the wrappers are tested on both parts. We use the *recall* and *precision* measures (which are widely used to evaluate information retrieval systems) to evaluate the performance of our system for extracting dynamic sections as well as records they contain. Recall is the percentage of the correct sections (records) that are extracted while precision is the percentage of the extracted sections (records) that are correct.

The test results are shown in a number of tables below. Rows labeled as "S pgs" give the results on sample pages, while rows labeled as "T pgs" give the results on test pages. Rows labeled as "Total" give the results based on all pages. Table 1 shows the section extraction results on all 119 search engines, totally 1,190 pages are tested. Table 2 shows the section extraction results on the 38 search engines whose result pages have multiple dynamic sections, totally 380 pages are tested. The columns labeled as "#Actual", "#Extracted", "#Perfect" and "#Partially Correct" give the numbers of sections that are actually present, extracted, perfectly extracted and partially correctly extracted, respectively. A section is perfectly extracted if all records in that section are extracted, and, no incorrect records are extracted. If more than 60% of the records in a section are extracted but some records are not extracted, we consider the section to be partially correctly extracted. The columns under "Perfect" give the performance (recall and precision) based on perfectly extracted sections, while the columns under "Total" give the performance if the partially correctly extracted sections are also acceptable.

The slight performance drop from the results for sample pages to those for test pages indicates the constructed wrappers are quite robust. We can see that the total recall and precision on all search engines for perfect extraction are 84.3% and 80.6%, respectively. But if we take partially correct sections into consideration, the numbers rise to 97.6% and 93.2%, respectively. A close examination of those partially correctly extracted sections reveals that the majority of the problems is caused by missing some records or falsely extracting some records. This is verified by Table 3, which shows the record extraction performance on all perfectly and partially correctly extracted sections. We can see that 98.7% of all records are extracted, with a precision of 98.8%.

Table 1. Section extraction results on all 119 search engines

	#Actual	#Extracted	#Perfect	#Partially Correct	Recall %		Precision %	
					Perfect	Total	Perfect	Total
S pgs	1057	1106	899	136	85.0	97.9	81.3	93.6
T pgs	981	1028	820	134	83.6	97.2	79.8	92.8
Total	2038	2134	1719	270	84.3	97.6	80.6	93.2

Table 2. Section extraction results on 38 search engines whose result pages have multiple dynamic sections

	#Actual	#Extracted	#Perfect	#Partially Correct	Recall %		Precision %	
					Perfect	Total	Perfect	Total
S pgs	652	670	538	92	82.5	96.6	80.2	94.0
T pgs	590	611	468	95	79.3	95.4	76.6	92.1
Total	1242	1281	1006	187	81.0	96.1	78.5	93.1

Table 3. Record extraction results on all perfectly and partially correctly extracted sections.

	#Actual	#Extracted	#Correct	Recall %	Precision %
S pgs	9615	9597	9490	98.7	98.9
T pgs	8248	8245	8139	98.7	98.7
Total	17863	17842	17628	98.7	98.8

Table 1 shows that the total precision (93.1%) of section extraction is lower than the total recall (96.1%), which indicates that for the current implementation, the problem of extracting incorrect sections is more serious than that of missing correct sections. The main reason for false extraction is that the system takes the missing records from some sections to form new sections. While the major reasons for missing records are: incorrect identification of SBMs and the existence of problematic DOM tree structures of some sections — our wrapper design requires that the tag structures of all records of a section be siblings under a common sub-tree, but some sections may contain records whose tag structures are not siblings.

7. RELATED WORKS

There have been a lot of researches on web data extraction recently. Readers may refer to [14] for a survey on major earlier works. Earlier wrapper generation techniques [1, 3, 12, 13, 16, 19, 28] are mostly semi-automatic or even manual, relying on training and human assistance to different extents. These techniques are becoming impractical as more and more large-scale web applications are emerging, such as building large-scale metasearch engines or building metasearch engines on-demand [18, 26]. Consequently, more recent works try to reduce or even totally eliminate human assistance.

Both RoadRunner [10] and EXALG [2] model the creating of web pages as enwrapping data by tokens following a template. RoadRunner compares a pair of web pages at a time to induce the template while EXALG works on a set of Web pages of the same class at the same time. IEPAD [7] assumes that repeating token patterns contain data to be extracted. PAT trees and some heuristics are applied to extract candidate patterns, from which a human user selects the best one. DeLa [24] extends the pattern idea by introducing a multi-level pattern extraction algorithm to build regular expressions to represent the nested schema of data on the web page automatically.

While RoadRunner, EXALG, IEPAD and DeLa consider the web page as a token string, many other researches consider the web page as a tag tree. Omini [5] and the method in [11] assume that there is a minimum data-rich sub-tree, which is detected by applying some heuristics. A separator, which is an HTML tag that can segment the sub-tree into data objects, is determined by another set of heuristics. Omini improves [11] by removing the domain specific ontology and uses a different set of heuristics to achieve better performance. MDR [15, 27] extracts the data-rich sub-tree indirectly by detecting the existence of multiple similar generalized-nodes, which is a collection of child nodes of the sub-tree. Then each generalized-node is checked to extract records. ViPER [23] extends the technique in [15] by utilizing some visual features. The work in [20] explicitly studies the tree edit distance problem by introducing a tree mapping algorithm and applies it to domain based page clustering and news extraction. ViNTs [29] utilizes both visual content features as well as tag tree structures. It assumes that the data records are located in a minimum data-rich sub-tree and are separable by separators of tag forests.

A somewhat related research problem is Web page segmentation (e.g., [6, 22]) whose goal is to partition web pages into semantic fragments (called blocks or pagelets by different researchers). The focus of this type of work is to differentiate fragments with different characteristics such as different styles and different refresh cycles. The fragments here are in general different from the dynamic sections discussed in this paper. First, fragments may include static blocks. Second, multiple continuous dynamic sections are likely to be treated as a single fragment. In addition, Web page segmentation does not consider identifying records within each fragment because supporting data extraction from web pages is not one of its intended applications.

None of the above works, with the exception of MDR [15], pays attention to the *section extraction problem*, which is the main focus of this paper. The models presented by RoadRunner, EXALG, and DeLa do not accommodate the existence of data sections, while IEPAD, Omini, and ViNTs simply assume that there exists only one section to be extracted. MDR has the ability to output multiple sections but it does not differentiate dynamic sections from static contents. It also does not address the *non-uniform format problem* and the *section-record granularity problem*. (The *hidden section extraction problem* does not occur for MDR as it does not generate wrapper, which can lead to other problems such as lower efficiency.) Furthermore, MDR works well only for table and form enwrapped records while our method does not have this limitation. Our experimental results reported in [29] show that ViNTs is significantly more accurate than MDR in extracting SRRs from the main (largest) section. Another weakness of MDR is that it can only detect sections with at least two records whereas our method does not have this problem.

Finally, we would like to point out the relationships and differences between this work and our previous work reported in [29]. First, the work in [29] was focused on *record extraction* from a *single section* (the main section) whereas our current work is focused on *section extraction* while also extracting records within each section. Second, the multi-record extraction (MRE) algorithm described in Section 5.1 in this paper is essentially from [29] with slight extension. MRE is just one component among many in our solution to the section extraction problem. Other than this component, the overall techniques in [29] and the current paper are completely different. Issues such as the *non-uniform format problem*, the *section-record granularity problem*, and the *hidden section extraction problem* are not addressed in [29]. Third, the techniques developed in this paper can help record extraction of ViNTs in two aspects: (1) the requirement that a section must contain at least three records is no longer needed; and (2) section boundaries can be more accurately identified. As a result, the accuracy of record extraction can be improved (the results are not reported here).

8. CONCLUSIONS

In this paper, we presented an algorithm (MSE) to tackle the problem of automatically extracting dynamic sections as well as search result records within these sections. We believe this is the first work that explicitly aims at extracting all dynamic sections from Web pages. We identified several new challenges for providing a comprehensive solution to the dynamic section extraction problem, i.e., the *non-uniform section format problem*, the *section-record granularity problem* and the *hidden section extraction problem*. MSE attempted to address these challenges with promising results. Our solution can potentially be very useful for all Web applications that need to interact with web-based search systems, including regular search engines and e-commerce search engines. By being able to extract search result records from all dynamic sections and maintaining the section-record relationships, MSE allows an application to select the desired sections for data extraction.

Preliminary experimental results indicate that the proposed technique is very promising but with room for improvement. We plan to investigate how to further improve the accuracy of identifying section boundary markers and to carry out additional experiments to evaluate the effectiveness of each component of the MSE solution.

9. ACKNOWLEDGMENTS

This work is supported in part by the following NSF grants: IIS-0414981, IIS-0414939 and CNS-0454298.

10. REFERENCES

[1] B. Adelberg. NoDoSE – A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents. SIGMOD Conference, 1998.

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

[3] R. Baumgartner, S. Flesca and G. Gottlob. Visual Web Information Extraction with Lixto. VLDB Conference, 2001.

[4] C. Bron, J. Kerbosch, Algorithm 457: Finding All Cliques of an Undirected Graph. Commu. of the ACM, Sep 1973.

[5] D. Buttler, L. Liu, C. Pu. A Fully Automated Object Extraction System for the World Wide Web. ICDCS 2001.

[6] D. Cai, S. Yu, J. Wen, W. Ma. Block-based Web Search. SIGIR Conference, 2004.

[7] C. Chang, S. Lui. IEPAD: Information Extraction based on Pattern Discovery. WWW 2001.

[8] K. Chang, B. He, C. Li, M. Patel, and Z. Zhang, Structured Databases on the Web: Observations and Implications. SIGMOD Record, 33(3), September 2004.

[9] S. Chawathe. Comparing Hierarchical Data in External Memory. VLDB Conference, 1999.

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

[11] D. Embley, Y. Jiang, Y. Ng. Record-Boundary Discovery in Web Documents. SIGMOD Conference, 1999.

[12] C. Hsu and M. Dung. Generating Finite-State Transducers for Semi-structured Data Extraction from the Web. Information Systems. 23(8): 521-538, 1998.

[13] N. Kushmerick, D. Weld, R. Doorenbos. Wrapper Induction for Information Extraction. IJCAI, 1997.

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

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

[16] L. Liu, C. Pu and W. Han. XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources. ICDE, 2000.

[17] D. McVitie, L. Wilson. The Stable Marriage Problem. Comm. of the ACM, 14(7), July 1971.

[18] W. Meng, C. Yu, K. Liu. Building Efficient and Effective Metasearch Engines. ACM Compu. Surv., 34(1), 2002.

[19] I. Muslea, S. Minton, C. Knoblock. A Hierarchical Approach to Wrapper Induction. Int'l Conf. on Auton. Agents, 1999.

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

[21] D. Reis, P. Golgher, A. Silva, A. Laender. Automatic Web News Extraction Using Tree Edit Distance. WWW 2004.

[22] L. Ramaswamy, A. Iyengar, L. Liu, F. Douglis. Automatic Detection of Fragments in Dynamically Generated Web Pages. WWW 2004.

[23] K. Simon, G. Lausen. ViPER: Augmenting Automatic Information Extraction with Visual Perceptions. CIKM 2005.

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

[25] S. Wu, U. Manber. Fast Text Searching Allowing Errors. Comm. of the ACM, 35(10), 1992.

[26] Z. Wu, V. Raghavan et al. Towards Automatic Incorporation of Search Engines into a Large-Scale Metasearch Engine. IEEE/WIC WI-2003 Conf., 2003.

[27] Y. Zhai, B. Liu. Web Data Extraction Based on Partial Tree Alignment. WWW 2005.

[28] Y. Zhai, B. Liu. Extracting Web Data Using Instance-Based Learning. WISE 2005.

[29] H. Zhao, W. Meng, Z. Wu, V. Raghavan, C. Yu. Fully Automatic Wrapper Generation for Search Engines. WWW 2005.