Automatically Extracting Subsequent Response Pages from Web Search Sources

Dheerendranath Mundluru, Zonghuan Wu, Vijay Raghavan
Center for Advanced Computer Studies
Univ. of Louisiana at Lafayette, Lafayette, LA
{dnm8925, zwu, raghavan}@cacs.louisiana.edu

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

Abstract

Usually, when Web search sources such as search engines and deep Websites retrieve too many result records for a given query, they split them among several pages with, say, ten or twenty records on each page and return only the page that has the top ranked records. This page usually provides one or more hyperlinks or buttons pointing to one or more of the remaining response pages (called subsequent response pages), which inturn contain similar hyperlinks or buttons to allow users to navigate from one page to another. Information integration systems often need to access these subsequent response pages to extract the records contained in them. However, hyperlinks or buttons pointing to subsequent response pages are often displayed in different formats by different Web search sources. Due to this it becomes a challenging task to automatically identify these hyperlinks or buttons and extract the response pages referenced by them. In this paper, we propose a novel solution to automatically fetch any specified response page from autonomous and heterogeneous Web search sources for any given query. Our approach first identifies certain important hyperlinks present in the response page sampled from an input Web search source and then further analyzes them using four heuristics. Finally a wrapper is built to automatically extract any specified response page from the input source.

1. Introduction

Web-based information integration systems such as metasearch engines, online comparative shopping engines and deep Web crawlers often need to interact and extract information from several autonomous and heterogeneous Web search sources. For example, a metasearch engine provides a unified interface to several existing search engines (SE) by dispatching user queries to those SEs and subsequently merging and displaying the returned result records to the user [1]. Similarly, online comparative shopping engines [14] integrate information from multiple e-stores to provide value-added services such as price comparison for a certain product (whose name was submitted as a query to the engine and then dispatched to the e-stores). Deep Web crawlers interact with deep Websites by submitting queries to extract and index high quality content not indexed by SEs such as Google [2].

Usually, SEs, deep Websites and other Web search sources return a list of result records for a query submitted to them. When the list contains too many records to be shown in one page, these records are often split among several pages and only the *first response page* is returned to the user. Usually, first response page displays only ten or twenty records and provides one or more hyperlinks or buttons on which the user can click to fetch *subsequent response pages* to access more records. In this paper, we refer to such hyperlinks and buttons as Subsequent Page Pointers or SPPs. Often, different sources display these SPPs in different formats. Figure 1 displays few sample formats used by different sources. After surveying hundreds of sources, in this paper, we classify SPPs into three different types:

(a) ***multi_spp***: In this type, any response page has several SPPs in the form of hyperlinks pointing to several other pages to allow users to extract any desired page. Figure 1a is an example of this type.

(b) ***single_spp***: In this type, any response page has only one SPP in the form of a hyperlink pointing to a subsequent response page. Figure 1c is an example of this type.

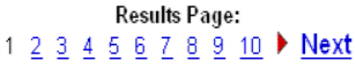
(c) ***submit_spp***: In this type, any response page has only one SPP in the form of a submit button pointing to a subsequent response page. Figure 1e is an example of this type.

Above SPP types can also be in the form of images. For example, Figure 1b displays multi_spp type SPPs that are in the form of images. Also, different Web sources may use different captions[1] or labels for the above SPP types. For example, Figure 1d is of the same type as Figure 1a, but uses different captions.

It is often important for information integration systems to extract information displayed not only in the first response page returned for a query, but also information present in a few, if not all, of the subsequent pages. For example, deep Web crawlers should index

[1] A caption is the label associated with an anchor tag. For e.g., in XYZ, XYZ is the caption.

information contained in several response pages returned by a Web search source for a given sample query. Similarly, for a metasearch engine to build a "representative" of an underlying SE, it is important to extract result records (returned for a sample query) contained in the subsequent response pages [1, 4]. Also, most commercial metasearch engines like Profusion (profusion.com) display records extracted from only the first response pages of different SEs, which may sometimes not meet a user's need. Hence, it is important to extract records contained in subsequent response pages. However, due to heterogeneities in displaying SPPs by different Web sources, it is not trivial to automatically identify the SPPs and to fetch the corresponding response pages. Despite its importance, to the best of our knowledge, there has not been any previously published work on techniques for identifying SPPs and extracting the search response pages referenced by them.



(a) Multiple SPPs as text

(b) Multiple SPPs as images

(c) Single SPP

(d) Multiple SPPs as text with new captions

(e) SPP in the form of a submit button

Figure 1. Different formats of SPPs.

In this paper, we provide a systematic, highly effective and efficient solution for automatically identifying SPPs and subsequently extracting any specified response page from the corresponding Web search sources. A prototype system called *PageExtractor* has also been implemented. The proposed method is based on a few observations, one of which is that *all response pages returned by a Web search source for a specific query are generated by the same program and hence they all possess certain common features.* Other observations will be discussed in section 5.1. Based on these observations, for a given Web search source, our PageExtractor system works in two phases: (1) In first phase, it initially identifies certain important hyperlinks called candidate SPPs present in the first response page returned by the input Web source for a given probe query. Candidate SPPs are further analyzed using four heuristics and finally a wrapper is constructed for the input Web source. (2) In second phase, the wrapper constructed in first phase is used to automatically and efficiently extract any specified search response page for any specified query from the input Web source.

In the rest of the paper, section 2 discusses related work; Section 3 presents the overview of the wrapper building process; Section 4 describes in detail the candidate SPP identification and analysis process that also includes the description of different heuristics; Section 5 describes what constitutes a wrapper and the procedure for building it using candidate SPPs; Section 6 reports our experimental results. We finally conclude in section 7.

2. Relevant work

Fetching Web data

Web data integrators need to fetch data from each of its sources distributed on the Web. In a friendly environment, an integrator may have special agreement with its sources on how to transfer data or utilizing XML and/or Web Services. If the number of sources are not too many and are stable, an integrator could have customized programs for different sources to fetch data. However, when an integrator needs to manage hundreds of sources or even more, especially when they are autonomous and heterogeneous and change their interfaces frequently in an unpredictable way; highly automated, adaptive, and robust methods are needed for the integrator to fetch data from its sources. In some cases, crawling-based approaches can be used when data is in the surface Web and can be accessed by navigating a sequence of URLs. However, when data needed is in the deep Web, which can be accessed through search interfaces at the deep Websites by sending proper queries, the integrator should have the capability to, 1) automatically discover such search interfaces and 2) automatically send queries and fetch the response pages. In [6], these two problems were studied in the context of large-scale metasearch engines. However, methods in [6] only discuss about fetching and processing the first response page returned in response to a query and does not mention about fetching subsequent response pages. Therefore, it is not clear how an integrator can fetch data on subsequent response pages.

In this paper, we assume that we are able to connect to a Web search source's interface, submit queries, and fetch the returned first response page. In our system, we use the SE connection component of SELEGO [6], a metasearch engine that we have developed.

Wrapper generation

Wrapper generation is a data extraction enabling process and has been studied extensively in the area of

Structured Data Extraction where the goal is to create wrappers, which consists of a set of extraction rules and the code for applying those rules, to extract structured information from Web pages for Web data harvesting, metasearching, and other Web mining tasks [4, 5, 6, 8, 9, 10, 11, 12, 14]. If we can identify SPPs and fetch the corresponding pages, wrappers can be generated to process these pages to extract the records contained in them. In the context of this paper, a wrapper (see section 5) is created to extract a specified response page from a Web search source rather than the result records contained in a particular response page.

3. Proposed technique

In this section, we present an overview of the proposed method. Figure 2 is a high-level flowchart to illustrate the steps involved in identifying SPPs for a given Web search source. Input to the system is the URL of a Web search source and output is a "wrapper", which essentially records discovered features of SPPs that can be utilized in the future for efficiently extracting any specified response page from the input Web search source.

Candidate SPP Identifier (CSI) component takes the URL of a Web search source as input and outputs a set of candidate SPPs. A candidate SPP in this case is a hyperlink although by definition it can be either a hyperlink or a submit button. Since SPPs encountered on the Web are mostly in the form of hyperlinks rather than submit buttons, most part of our algorithm is involved in analyzing hyperlinks to detect if SPPs are in the form of hyperlinks. It will be detected only in the final step (Wrapper Builder) if an SPP is not in the form of hyperlink, but is in the form of a submit button. Therefore, candidate SPPs correspond to hyperlinks in all steps before Wrapper Builder. CSI first generates a few sample queries and fetches the corresponding response pages. It then uses a link analysis method to identify a small set of important hyperlinks present in one of the response pages. The identified hyperlinks form the candidate SPPs, each of which is initialized with a score of 1.0. CSI will be discussed in section 4.1. Candidate SPPs are further analyzed by four independent heuristics (next four steps in Figure 2) that are invoked successively. Each heuristic may adjust the scores of the input candidate SPPs received from the previous step and outputs the same candidate SPPs along with their scores. Final score of each candidate SPP (after being processed by the four heuristics) indicates its likeliness to be the target SPP. We found that the four heuristics used are highly effective in identifying the target SPPs. The factor by which each heuristic may adjust the score of a candidate SPP depends on the accuracy of the corresponding heuristic. Section 6 discusses the details about determining the accuracy of each heuristic. Also, following the observation made in section 1, three of the four heuristics try to exploit the common features across the first response page and the subsequent response pages.

Label Matching (LM) heuristic uses a set of predetermined labels, which are widely used by various sources, to identify if any of the input candidate SPPs can be given a higher priority by adjusting their initial scores. LM will be discussed in section 4.2.

Static Links (SL) based heuristic uses a feature called *Static Links* to identify if the scores of any of the input candidate SPPs can be further adjusted. SL will be discussed in section 4.3.

Form Widget (FW) based heuristic uses the *Form Widgets* (e.g., text box etc.) as a feature to identify if the scores of any of the input candidate SPPs can be further adjusted. FW will be discussed in section 4.4.

Page Structure Similarity (PSS) uses tag strings (concatenation of all opening tags in a Web page) of Web pages as a feature to further adjust the scores of the input candidate SPPs. PSS will be discussed in section 4.5.

Finally, Wrapper Builder takes all the input candidate SPPs and their scores and generates a wrapper, which can be used for efficiently fetching any specified response page from the input Web search source for any specified query. Wrapper Builder will be discussed in section 5.

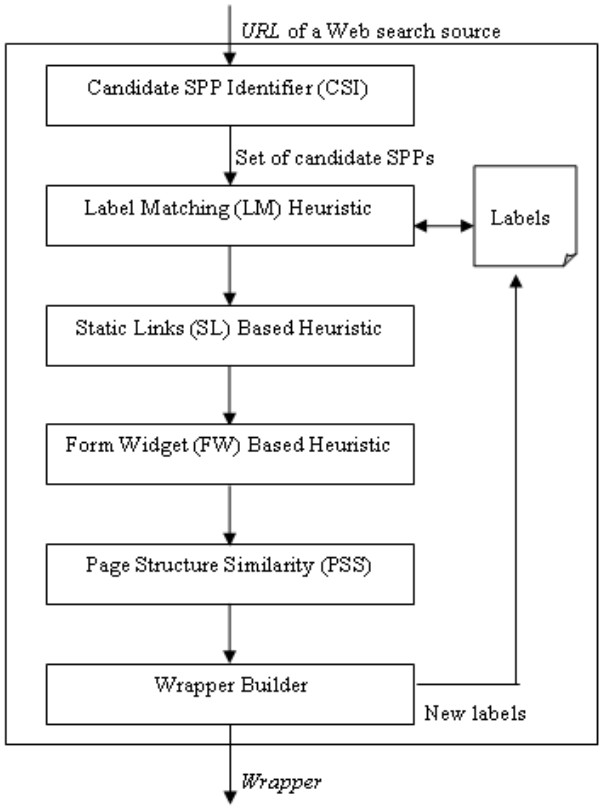


Figure 2. SPP discovery process.

4. Candidate SPP identification and analysis

In this section, we discuss in detail the first five steps in Figure 2, which are responsible for identifying and analyzing the candidate SPPs.

4.1. Candidate SPP identifier (CSI)

We first define few terms that we use for the purpose of simplifying the description of our approach:

Definition 1: *Invalid Query* and *Ideal Query*: When a query q is submitted to a Web search source S, δ result records split into K pages are returned with each page containing atmost T result records. q is an *Invalid Query* to S when $\delta = 0$ and $K = 0$. q is an *Ideal Query* to S when $\delta > T$ and $K > 1$.

Definition 2: *Invalid Response Page* and *Valid Response Page*: The response page returned for an invalid query is an *Invalid Response Page* (*IRP*). The response page returned for an ideal query is a *Valid Response Page* (*VRP*).

CSI takes URL of a Web search source as input and outputs a set of candidate SPPs, which form a small sub-set of all hyperlinks found in a VRP sampled from the input Web source. Each candidate SPP is initialized with a score of 1.0. We use example pages in Figure 3 as a running example to explain the following steps in CSI:

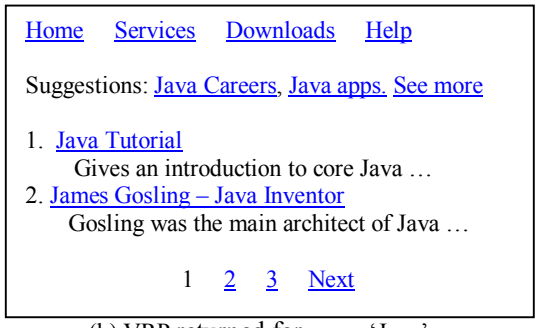
(1) **Probe query generation**. First, we need to automatically generate three probe queries corresponding to the input Web source. These probe queries include one invalid query (q_i) and two distinct ideal queries (q_{id1}, q_{id2}). In our running example, 'AnI248mpossibleq73uery', 'Java', and 'XML' are the three probe queries corresponding to q_i, q_{id1}, and q_{id2}. Automatic generation of q_i is explained in [3]. In this step, we present Ideal Query Generator (IQG), a module for automatically generating q_{id1} and q_{id2}.

IQG selects the best queries that can produce largest response pages from a set of candidate queries. For general purpose SEs like Google, every non-stopword can produce reasonably good response pages. Thus every non-stopword can be considered as a candidate query. But this does not work for many specialized SEs (e.g., local SE at ieee.org). We only consider one term queries as the candidate queries. The initial candidate query terms are non-stopword terms coming from three sources: cover page of Web source, title of cover page (text between tags <TITLE> and </TITLE>), and URL of the cover page. Rationale behind this is that cover page and title often contain several descriptions (e.g., company's name and its products/services) that are present in several documents indexed by the local SE, while URL contains very few highly important terms that are also present in several documents indexed by the local SE.

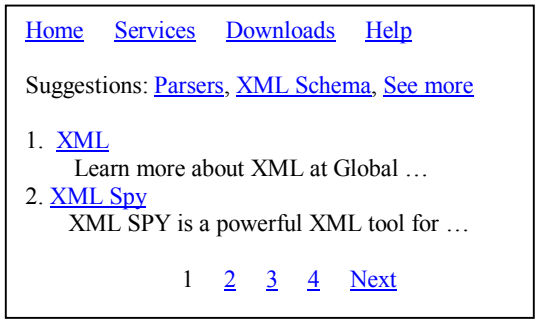
The cover page may contain too many terms, not all of them can produce a good response page. Also checking them all can be inefficient. Thus we keep only those terms that are most likely to appear in a document. A word frequency list from [1] is used to measure the popularity of terms thereby deleting the less frequent terms.

We need adequate candidate queries to generate the required ideal queries. We set the size of the candidate query set as two times the number of required ideal queries. If the cover page of a source is too simple, such that enough candidate queries cannot be generated, we add terms with highest frequencies from the word frequency list [1] to ensure that adequate candidate queries are available. We then send the candidate queries to the source; those candidate queries with the largest response page sizes are kept as ideal queries.

(a) IRP returned for query 'AnI248mpossibleq73uery'



(b) VRP returned for query 'Java'



(c) VRP returned for query 'XML'

Figure 3. Three example response pages.

(2) **SE connection**. Let IRP, VRP_1, and VRP_2 be the response pages returned for the probe queries q_i, q_{id1}, and q_{id2}. As mentioned in section 2, to fetch these response pages, we use the SE connection component of SELEGO

[6]. In our example, response pages in Figures 3a, 3b, and 3c correspond to IRP, VRP_1, and VRP_2.

(3) **Initial candidate SPP list generation**. Initial list of candidate SPPs (l_c) are generated by extracting only those hyperlinks in VRP_1 whose captions match with the captions of hyperlinks in VRP_2. In our example, by matching the captions of hyperlinks in Figures 3b and 3c, we get l_c = {Home, Services, Downloads, Help, See more, 2, 3, Next}. The main reason for using captions and not URLs as a feature for extracting candidate SPPs is that in almost all the Web search sources that we encountered, we observed that URLs of the SPPs are mostly query dependent, i.e. query is included in the URL of the SPP. Hence, URLs of the SPPs in two different VRPs returned for two different queries will not match and will not be included in the list of candidate SPPs. Therefore, this simple technique of using captions as a feature to identify the candidate SPPs allows us to include some or all of the actual target SPPs as part of the candidate SPPs. This step also gets rid of all other hyperlinks such as result links, advertisement links, suggestions links, etc., which are mostly query dependent and usually form the majority of hyperlinks in a Web page. Therefore, this step usually generates only few hyperlinks as the candidate SPPs, which are further pruned in the next step.

(4) **Final candidate SPP list generation**. Static links (l_s) are those hyperlinks that appear in both IRP and VRP. We observed that in most Web sources, Website navigation links and other miscellaneous links such as Terms of Use form the static links, which can be used for further pruning candidate SPPs. In our example, by extracting common hyperlinks in Figures 3a and 3b, we get l_s = {Home, Services, Downloads, Help}. It can be seen that l_s forms a subset of l_c. Since it is intuitive that l_s can never contain the actual target SPPs, we remove them from l_c to get the final list of candidate SPPs i.e., $l_c = l_c - l_s$. In our example, l_c = {Home, Services, Downloads, Help, See more, 2, 3, Next} - {Home, Services, Downloads, Help} = {See more, 2, 3, Next}. We next initialize each candidate SPP in the final list with an initial score (s_c) of 1.0, which may be further adjusted by the subsequent steps. Final score of a candidate SPP defines its measure of likeliness to be the actual target SPP.

4.2. Label matching heuristic (LM)

LM takes the list of candidate SPPs and their corresponding scores as input and outputs the same list with scores that may have been adjusted. LM uses a predetermined list of SPP labels collected from two hundred Web sources. LM matches the caption of each candidate SPP with the collected labels. If the caption of any candidate SPP matches *exactly* with a label, then we decrease its score by a constant parameter α ($\alpha \in [0, 1]$). This way, candidate SPPs whose scores are decreased are considered to be more likely to be the target SPPs. Sometimes, if a specific Web source uses entirely new captions for the SPPs, then none of the captions of the candidate SPPs match with the predetermined labels. In this case, the scores remain same. However, subsequent steps can still identify target SPPs and when a target SPP is identified its caption is extracted and is added to the predetermined list of labels. This way, newly *learned* labels may be useful when a new Web source uses similar captions for its SPPs.

4.3. Static links based heuristic (SL)

SL takes the list of candidate SPPs and their scores as input and outputs the same list with scores that may have been further adjusted. We use static links (l_s) found in section 4.1 as a feature to further adjust the scores of the candidate SPPs. For each candidate SPP, we first download its corresponding page and save it in a repository to avoid downloading it again in subsequent steps. We then check if the downloaded page contains all the static links (l_s). If this condition is true, we update the score of that candidate SPP by a constant parameter β_1 ($\beta_1 \in [0, 1]$). Otherwise, the score is unchanged. Motivation behind this heuristic is that all static links that appeared in first response page (VRP_1) also appear in subsequent response pages and hence the scores of candidate SPPs containing all the static links are adjusted.

4.4. Form widget based heuristic (FW)

Another feature common among the different response pages returned for a particular query by a Web search source is the form widget feature. We use form widgets (type of form widget) and their names (value of attribute 'name' in the HTML tag of form widget) as a feature in further adjusting the scores of the input candidate SPPs. Form widgets that our algorithm uses are text box, text area, check box, radio button, submit button, image button, drop-down box, and hidden tag (input tag of type hidden). Figure 4 illustrates FW functionality.

In Figure 4, we first find all the form widgets in VRP, represented as wl_{VRP} (line 1). If VRP does not have any form widgets, then we check if at least one candidate SPP page (in the repository) has at least one form widget (lines 2 – 3). If this condition is true, we adjust the scores ($s_{c,i}$) of each of the candidate SPPs not having any form widgets by a constant parameter β_2 ($\beta_2 \in [0, 1]$) (lines 4 – 6), whereas scores of all candidate SPPs having at least one form widget are unchanged. Motivation behind this is that if VRP does not have any widgets, then it is highly unlikely that subsequent pages will have any new widgets. Therefore, candidate SPPs not having any widgets have a higher chance of being the target SPPs (hence their scores are reduced) than those having at least

one widget. Note that if VRP does not have any form widgets and if none of the candidate SPPs has at least one form widget, then the scores of all candidate SPPs are unchanged. However, if VRP does have some form widgets, then for each candidate SPP page, we check if all form widgets in VRP also appear in the candidate SPP page with the same name. If this condition is true, we adjust the score of the candidate SPP by a constant parameter β_3 ($\beta_3 \in [0, 1]$) indicating that it is highly likely to be the target SPP (lines 10 – 12). If all form widgets in VRP do not appear in a candidate SPP page, then the score of that candidate SPP is unchanged. Motivation behind this step is that all form widgets that appeared in first response page (VRP_1) also appear in subsequent response pages. Section 6 explains how parameters α, β_1, β_2, and β_3 are set.

Procedure: fwHeuristic()
1: let wl_{VRP} be the form widgets in VRP
2: **if** $|wl_{VRP}| == 0$ **then**
3: **if** atleast 1 $l_{c,i}$ has atleast 1 widget **then**
4: **for** each $l_{c,i}$ having no widgets **do**
5: $s_{c,i} = s_{c,i} * \beta_2$, where $\beta_2 \in [0, 1]$
6: **end for**
7: **end if**
8: **else**
9: **for** each $l_{c,i}$ **do**
10: **if** wl_{VRP} appear in $wl_{c,i}$ with the same name **then**
11: $s_{c,i} = s_{c,i} * \beta_3$, where $\beta_3 \in [0, 1]$
12: **end if**
13: **end for**
14: **end if**

Figure 4. Form widget heuristic.

4.5. Page structure similarity (PSS)

Another important feature common among different response pages returned for a particular query by a Web search source is the page structure of the response pages itself. Page structure of a response page is reflected in its tag string, which is the concatenation of all opening tags in the page. According to observation in section 1, since all response pages returned by a Web search source are generated by the same program, it is intuitive that the program just wraps content in each response page using similar/same HTML tags. Therefore tag strings of different response pages are considered to be similar/same.

To capture the similarity between the tag string of VRP and the tag string of each of the candidate SPPs, we use a popular approximate string matching algorithm called Levenshtein Distance (LD) [7]. LD in our algorithm is defined as the smallest number of insertions, deletions, and substitutions of tags needed to convert one tag string into another. If <table><tr><td><a> and <table><tr><td><b><a> are two tag strings, then LD between them is 1. For current work, we use normalized LD (NLD), which was also used in other works such as [8]. NLD between two tag strings t_1 and t_2 is defined as,

$$NLD(t_1, t_2) = \frac{LD(t_1, t_2)}{(length(t_1)+length(t_2))/2}$$

where $LD(t_1, t_2)$ gives the LD between t_1 and t_2 and length function returns the number of tags in the input tag string. Therefore, in our algorithm we calculate NLD between the tag string of VRP and the tag string of each of the candidate SPP pages. NLD value obtained between a VRP tag string and a candidate SPP tag string is added to the current score of that candidate SPP to get its final score. We found that NLD between VRP and the candidate SPPs that form the actual target SPPs is almost always very low compared to NLD between VRP and the candidate SPPs that are not the actual target SPPs.

5. Wrapper construction

This section discusses Wrapper Builder component in detail. Wrapper Builder takes candidate SPPs and their scores, identified by different steps in section 4, as input and outputs a single wrapper that can be used to extract any specified response page from the input Web search source for any specified query. In section 5.1, we discuss the features that constitute an SPP wrapper and in section 5.2 we discuss how an SPP wrapper is constructed when the target SPPs are of any of the types seen in section 1.

5.1. Wrapper format description

As noted earlier, SPPs are mostly in the form of hyperlinks and are represented by the corresponding URLs. Even when SPPs are of submit_spp type, they are represented by the URLs constructed as described in section 5.2. URLs representing SPPs almost always contain two parts: (1) a host part and (2) a query part. If www.google.com/search?q=xml&p=11 is an example SPP, then www.google.com/search is the host part while q=xml&p=11 is the query part. A delimiter '?' separates the host and query parts. Similarly, query part consists of one or more parameters which are also separated by delimiters. A parameter is a <name, value> pair and is usually represented as 'name=value'. '&' is used as the parameter delimiter. An SPP described above can be represented as the following regular expression:

hostpart(D(parameter(d)))

where hostpart is the host part of the SPP, D is the delimiter separating the host and query parts, parameter represents the parameters in the query part while d is the delimiter separating those parameters.

Observations. Structured data extraction algorithms [4, 5, 6, 8, 9, 10, 11, 12, 14] are based on the assumption that the data to be extracted from a Web page follows few regularities in the way they are displayed (e.g., similar tag strings, each record starting in a new line, etc.) for making it user comprehensible. Similarly, we observed that URLs of SPPs generated by a Web source in response to a query also follow few regularities, most likely to have a simple and straightforward logic in the server side programs that generate response pages:

(1) One feature is that the query submitted to any Web search source is included as a parameter value in the query part of all the SPPs generated for that query. We refer to such a parameter which has the query as its value as a *query parameter*. In the above example SPP, q=xml is a query parameter, if the query submitted is xml.

(2) Another feature is that in most cases, one or at the most two parameters are used to *uniquely* identify each SPP returned for a particular query by a Web search source. We refer to such a parameter(s) which uniquely identifies an SPP as a *page parameter(s)*.

If we consider two different SPPs generated for a particular query by a Web search source, then the two SPPs only differ in the page parameter(s) i.e. the two SPPs are identical if we ignore values in the page parameter(s). For example, consider the following three example SPPs to fetch second, third, and fourth response pages returned for the query 'xml' by Google. The three SPPs are identical, if the page parameters values 11, 21, and 31 in the SPPs are ignored. A page parameter in an SPP maintains information about the actual number of the corresponding response page.

www.google.com/search?q=xml&p=11&sl=1
www.google.com/search?q=xml&p=21&sl=1
www.google.com/search?q=xml&p=31&sl=1

By plugging a new value in the query parameter and an appropriate value in the page parameter, we can fetch the desired response page for the new query. Therefore, goal of Wrapper Builder is to identify both query and page parameters in at least two candidate SPPs. Once identified, we *generalize* the corresponding SPPs by replacing query and page parameter values with two place holders ($query and $start respectively) thereby obtaining a single *generalized SPP*. For the above three SPPs, www.google.com/search?q=$query&p=$start&sl=1 is a generalized SPP. By replacing $query and $start with appropriate values, any desired response page can be fetched. However, to replace $start with an appropriate value; we have to first know what the appropriate value is. For this we discover two more values called *initial value* and *incremental value* and are defined as follows:

Definition 1: An *initial value* for a generalized SPP G is the *minimum* of all page parameter values identified in the candidate SPPs that generated G.

Definition 2: An *incremental value* for a generalized SPP G is the *minimum* of all pair-wise absolute differences between all the page parameter values identified in the candidate SPPs that generated G.

For the above three example SPPs, initial value is 11 i.e. min(11, 21, 31) and incremental value is 10 i.e. min(abs(11-21), abs(21-31), abs(31-11)). If we assume that the first SPP in the first response page points to k^{th} (k = 1 or 2) response page, then to fetch the n^{th} response page, we replace $start with,

initial value + incremental value * (n – k)

For example, if first example SPP shown above is pointing to the second response page, then to fetch the fifth response page for a query "health", we replace $query in above generalized SPP with "health" and $start with 41.

It should be noted that in all the response pages we encountered, initial and incremental values are always constant for any query submitted to a Web source. Also, for cases where there are two page parameters in an SPP, we use $end as a place holder for the second page parameter value and its initial and incremental values are found in the same way as described above. We consider such SPPs that differ only in one or two page parameters to be following *regular query patterns*. Rarely, we may encounter SPPs not following a regular query pattern and we consider such SPPs to be following *irregular query patterns*. Currently, our system reports that a generalized SPP could not be created when it encounters SPPs following irregular query patterns.

Once a generalized SPP and the corresponding initial and incremental values are discovered, we record them in an XML file, which forms our final SPP wrapper.

5.2. Constructing SPP wrapper

In this section, we explain how an SPP wrapper (discussed in previous section) is constructed when target SPPs are of any of the types described in section 1. We first check if the number of input candidate SPPs is greater than one or equal to one or equal to zero. If the number of candidate SPPs is greater than one, procedure checkmulti_spp (Figure 5) is invoked, to check if SPPs are of multi_spp type.

Input to checkmulti_spp is the list of candidate SPPs (SPPlist) and their corresponding scores (Scorelist). In line 1, procedure buildwrapper is invoked, which tries to build a wrapper as described in section 5.1. Main functionality of buildwrapper includes building a generalized SPP by comparing two candidate SPPs at a time and identifying its initial and incremental values. Due to space constraint, pseudocode for buildwrapper is not provided. buildwrapper returns true if it can successfully build a wrapper implying that SPPs are of multip_spp type. Otherwise, it returns false in which case

we invoke procedure checksingle_spp (Figure 6) to check if SPPs are of single_spp type (line 3). If checksingle_spp returns true, it implies wrapper is constructed successfully and SPPs are of single_spp type (line 12). Otherwise, procedure checksubmit_spp (Figure 7) is invoked to check if SPPs are of submit_spp type (line 5). If checksubmit_spp returns true, it implies wrapper is constructed successfully and SPPs are of submit_spp type (line 9). Otherwise, it implies that algorithm is unable to build a wrapper (lines 6 – 7).

Multiple generalized SPP case. In some cases, more than one generalized SPP may be identified for the same source. This happens when different groups of candidate SPPs were following a regular query pattern. In such cases, system should choose only one of the generalized SPPs as the target generalized SPP. Randomly choosing one of them might lead to selecting a *false positive*. To decrease the likelihood of selecting a false positive, system first determines a *generalized SPP score* for each generalized SPP, which is defined as,

$$\frac{\sum_{l_{c,i} \in l_c.G} l_{c,i}.\text{score}}{|l_c.G|}$$

where G is a generalized SPP, $l_c.G$ is the list of candidate SPPs that generated G, $l_{c,i}$.score is score of a candidate SPP that was involved in generating G, and $|l_c.G|$ is the total number of candidate SPPs that generated G. System then chooses the generalized SPP having the lowest generalized SPP score as the target generalized SPP. This functionality for choosing a single generalized SPP is also part of the procedure buildwrapper.

```
Procedure: checkmulti_spp(SPPlist, Scorelist)
 1: wflag = buildwrapper(SPPlist, Scorelist)
 2: if wflag is false then
 3:    wflag = checksingle_spp(SPPlist, Scorelist)
 4:    if wflag is false then
 5:       wflag = checksubmit_spp()
 6:       if wflag is false then
 7:          wrapper could not be created
 8:       else
 9:          wrapper created successfully
10:       end if
11:    else
12:       wrapper created successfully
13:    end if
14: else
15:    wrapper created successfully
16: end if
```

Figure 5. Checking for multi_spp type.

Apart from being invoked from checkmulti_spp, procedure checksingle_spp (Figure 6) will also be invoked when the number of input candidate SPPs is equal to one. Input to checksingle_spp is the list of candidate SPPs (SPPlist) and their scores (Scorelist). In line 1, we sort SPPlist in ascending order of the scores of the candidate SPPs. In line 2, we start processing each of the first K (K = 5 in our experiments) candidate SPPs in the sorted SPPlist having query q_{id1} in the query part of their URLs. In line 3, we check if caption of any of the hyperlinks (say l_{temp}) present in page $p_{c,i}$ (corresponding to candidate SPP $l_{c,i}$) matches with caption of $l_{c,i}$. If this condition is true, we invoke procedure buildwrapper, which tries to build a wrapper using parameters $l_{c,i}$ and l_{temp}. If buildwrapper returns false, we repeat the process with the next candidate SPP. Otherwise, checksingle_spp returns true indicating that a wrapper has been constructed successfully (line 6). If all K candidate SPPs failed to build a wrapper, checksingle_spp returns false indicating that target SPPs are not of single_spp type. In this case, procedure checksubmit_spp is invoked to check if SPPs are in the form of submit_spp type. Note that parameter K is used only for efficiency purpose as in practice it is highly unlikely to find a wrapper if it was not found after processing the first K candidate SPPs having the lowest scores. Also, sorting input candidate SPPs in line 1 decreases the likelihood of finding a generalized SPP that is a false positive as we return the first identified generalized SPP. Main motivation behind finding a hyperlink (in page $p_{c,i}$) with a caption similar to the caption of $l_{c,i}$ is that in most cases of single_spp type SPPs, the caption of the target SPP is similar in all response pages returned by a Web search source. Due to space constraint, we do not discuss the functionality that handles the case when SPPs are of single_spp type, but their captions are different in different response pages returned by a Web search source.

```
Procedure: checksingle_spp(SPPlist, Scorelist)
 1: sort SPPlist in ascending order of SPP scores
 2: for each l_{c,i} having query q_{id1} in SPPlist until K do
 3:    if l_{c,i}.caption matches a caption of any link (l_temp) in
          p_{c,i} then
 4:       wflag = buildwrapper(l_{c,i}, l_temp)
 5:       if wlag is true then
 6:          return true
 7:       end if
 8:    end if
 9: end for
10: return wflag
```

Figure 6. Checking for single_spp type.

We now describe procedure checksubmit_spp (Figure 7), which may also be invoked when the total number of

input candidate SPPs is equal to zero. In line 1, we build a tag tree [13] of VRP_1. A tag tree data model for Web pages consists of tag nodes forming a nested tree structure. A start tag and its optional end tag in a Web page are represented as a single tag node in the tag tree. Any content (text or other tags) between the start and the end tag in the Web page is reflected in the sub-tree of the corresponding tag node in the tag tree. Root of the tag tree is HTML and each tag node can be located by following a path from the root to the node. In line 2, we extract all the form objects i.e., tag nodes corresponding to the form tags along with their sub-trees. In lines 3 – 8, we process each form object having at least one hidden tag. In line 4, we construct the URL by extracting <name, value> pairs from each hidden tag i.e., by extracting the values of the attributes *name* and *value*. If the constructed URL contains q_{id1}, we add it to list1 (lines 5-6). Finally, if list1 has at least one element, procedure processHiddenURL is invoked with list1 as the parameter (lines 9-10). Otherwise, we return false indicating that a wrapper could not be constructed (line 12). processHiddenURL takes a list of hidden URLs (URLs constructed from hidden tags) as input and processes each URL separately. For each hidden URL (h_c), it first downloads the page referenced by h_c and extracts all hidden URLs in that page as described above. It then tries to build a wrapper by invoking procedure buildwrapper using h_c and each of the newly extracted hidden URLs (say h_{c1}) as the parameters. If buildwrapper returns true, then processHiddenURL also returns true indicating that a wrapper has been constructed successfully. Otherwise, buildwrapper is invoked using a new combination of h_c and h_{c1}.

Procedure: checksubmit_spp()
1: build a tag tree of VRP_1
2: extract all form objects in the tag tree
3: **for** each form object having hidden tags **do**
4: form a URL by extracting <name, value> pairs from hidden tags
5: **if** URL contains query q_{id1} **then**
6: add URL in list1
7: **end if**
8: **end for**
9: **if** |list1| >= 1 **then**
10: **return** processHiddenURL(list1)
11: **else**
12: **return** false
13: **end if**

Figure 7. Checking for submit_spp type.

6. Experiments

We initially surveyed several Web sources to determine the important heuristics that can be used in our algorithm. After identifying the heuristics, we developed the system by setting some initial (intuitive) values for parameters α, β_1, β_2, and β_3. We then conducted our initial experiments to determine the final values of α, β_1, β_2, and β_3 that reflect the effectiveness of LM, SL, and FW heuristics in identifying SPPs. We then used the final parameter values to perform our final experiments to evaluate our system by testing it on 85 new Web sources. Below, we first explain the results of our initial experiments followed by our final experiments.

Table 1. Initial experiments summary.

	LM	SL	FW(β_2)	FW(β_3)
A	42	36	2	43
B	1	13	1	14
Accuracy	97%	73%	66%	75%

Initial Experiments: Our initial experiments included a total of 50 Web search sources taken from diverse categories. We measured the accuracy of each of the heuristics, which indicates their effectiveness in identifying target SPPs. Let A be the number of sources for which the target SPP is identified by a heuristic H and B be the number of sources for which at least one candidate SPP that is not the target SPP is identified by H. Then, accuracy of H is defined as A/A+B. It should be noted that apart from A and B there may be some sources among the 50 Web sources for which H may not identify even a single candidate SPP as the target SPP.

Table 1, which presents summary of our initial experiments, shows that LM is the best heuristic with an accuracy of 97%. FW for the case when β_3 is used is the next best with an accuracy of 75%. SL is third best with an accuracy of 73%. Though, FW for the case when β_2 is used was encountered only 2 times, it could identify target SPP on both occasions and hence we consider it to be an important feature. From these observations, we decided to use 0.1 for α, 0.4 for β_3, 0.5 for β_1, and 0.6 for β_2.

Final Experiments: Our final experiments included 85 new Web sources (completely different from those used in initial experiments) that include several general purpose SEs (e.g., Google), e-stores and other specialized sources taken from categories such as health (cancer research), science & technology, newspapers, etc. Such diverse sources were chosen to validate our method's effectiveness comprehensively. It should be noted that sources whose SPPs follow irregular query patterns were not included in our final experiments. Below, we summarize our test results.

Accuracy: Our system achieved an accuracy of 95.2% i.e. out of 85 sources it failed on only 4 sources. Among the 4 failed cases, one source failed as its IRP always displayed SPPs though 0 results were returned as it always returns several advertisement links no matter what invalid query is submitted. Due to this, SPPs are always included in the

static links and hence will be removed from the final candidate SPP list returned by CSI. Two other sources having a single_spp type SPP failed as the system wrongly found that SPPs are of multi_spp type as few candidate SPPs followed regular query patterns. Last source, which had a single_spp type SPP, failed as a wrong generalized SPP was chosen since the corresponding candidate SPP had a score less than the score of the candidate SPP that was the actual target SPP.

Effectiveness of CSI module: Total number of unique hyperlinks identified across all 85 Web pages was 4228, while total number of candidate SPPs identified by CSI was only 734. Therefore, on average 49.74 unique links were identified in a Web page, while on average only 8.63 candidate SPPs were identified. This shows that CSI plays a critical role in keeping the system efficient as subsequent steps have to process only few candidate SPPs.

SPP types on the Web: We found that multi_spp and single_spp type SPPs were employed by most Web sources. multi_spp type SPPs were used by 47 sources, while single_spp type SPPs were used by 31 sources. SPPs of submit_spp type were used by only 2 sources and 5 sources did not use any SPPs at all i.e. result records were always displayed in the first response page.

Execution time: Average time taken for our system to build a wrapper was 20.5 seconds. Once a wrapper is created for a Web source, any specified response page can be fetched from that source in a fraction of a second. All experiments were conducted on a Pentium 4 3.1GHz laptop with a 512 MB RAM and T-1 Internet access. We consider that the proposed method is efficient enough to be practically used.

7. Conclusions and future work

This paper proposed an effective and efficient solution for automatically fetching any specified response page from Web search sources. This is an important task for information integration systems as most often Web search sources split their results among several response pages and return only the first response page. The proposed approach first identifies certain important hyperlinks present in the response page sampled from an input Web search source and then further analyzes them using four independent heuristics. Finally a wrapper is built to automatically extract any specified response page from the input Web search source. Experimental results showed that the proposed method is highly effective (95.2% accuracy) and efficient. In immediate future, we plan to:

(1) Handle cases where SPPs follow irregular query patterns. For this, we will be using a machine learning approach, where we use a feature vector for each candidate SPP. We will use the output obtained from each heuristic, after processing the candidate SPP, as a feature in the feature vector.

(2) Handle cases where SPPs are Java Script enabled (e.g., preventcancer.org), i.e. clicking an SPP will invoke a Java Script method before sending request to the server.

(3) Use visual position of SPPs to identify them. For example, in some cases SPPs appear both at the top and bottom of the Web page and such visual information can be utilized for identifying them.

(4) Perform large-scale experiments.

References

[1] W. Meng, C. Yu, and K. Liu. Building Efficient and Effective Metasearch Engines. *ACM Computing Surveys*, 34(1), 48-89, March 2002.

[2] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. *VLDB*, 129–138, September 2001.

[3] Z. Wu, D. Mundluru, and V. Raghavan. Automatically Detecting Boolean Operations Supported by Search Engines, towards Search Engine Query Language Discovery. *Workshop on Web-based Support Systems*, 171-178, September 2004.

[4] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu. Fully Automatic Wrapper Generation for Search Engines. *WWW*, 66-75, May 2005.

[5] L. Chen, H. Jamil, and N. Wang. Automatic Composite Wrapper Generation for Semi-Structured Biological Data Based on Table Structure Identification. *SIGMOD Record*, 33(2), 58-64, 2004.

[6] Z. Wu, V. Raghavan, H. Qian, V. Rama, W. Meng, H. He, and C. Yu. Towards Automatic Incorporation of Search Engines into a Large Scale Metasearch Engine. *Web Intelligence*, 658-661, October 2003.

[7] P. Hall and G. Dowling. Approximate String Matching. *ACM Computing Surveys*, 12(4): 381-402, 1980.

[8] B. Liu, R. Grossman, and Y. Zhai. Mining Data Records in Web Pages. *SIGKDD*, 601-606, August 2003.

[9] B. Liu and K. Chang. Editorial: special issue on web content mining. *SIGKDD Explorations*, 6(2), 1-4, December 2004.

[10] I. Muslea, S. Minton, and C. Knoblock. A Hierarchical Approach to Wrapper Induction. *Agents*, 190-197, 1999

[11] L. Buttler, L. Liu, and C. Pu. A fully automated object extraction system for the World Wide Web. *ICDCS*, 361-370, 2001.

[12] C. Chang and S. Liu. IEPAD: Information Extraction Based on Pattern Discovery. *WWW*, 681-688, 2001.

[13] S. Chakrabarti. Mining the Web: Discovering Knowledge from Hypertext Data. Morgan Kaufman, San Francisco, CA, 2002.

[14] R. Doorenbos, O. Etzioni, and D. Weld. A Scalable Comparison-Shopping Agent for the World-Wide Web. *Agents*, 39-48, 1997.