Extracting Data Behind Web Forms

Stephen W. Liddle* David W. Embley* Del T. Scott Sai Ho Yau

Brigham Young University
Provo, UT 84602
USA
liddle@byu.edu, {yaus,embley@cs.byu.edu}, scottd@byu.edu

Abstract

A significant and ever-increasing amount of data is accessible only by filling out HTML forms to query an underlying Web data source. While this is most welcome from a user perspective (queries are relatively easy and precise) and from a data management perspective (static pages need not be maintained and databases can be accessed directly), automated agents must face the challenge of obtaining the data behind forms. In principle an agent can obtain all the data behind a form by multiple submissions of the form filled out in all possible ways, but efficiency concerns lead us to consider alternatives. We investigate these alternatives and show that we can estimate the amount of remaining data (if any) after a small number of submissions and that we can heuristically select a reasonably minimal number of submissions to maximize the coverage of the data. Experimental results show that these statistical predictions are appropriate and useful.

1 Introduction

To help consumers and providers manage the huge quantities of information on the World Wide Web, it is becoming increasingly common to use databases to generate Web pages dynamically. Often, dynamically generated pages are accessible only through an HTML form that invokes a Common Gateway Interaction (CGI) request to a Web server (e.g. Figure 1). Servers often convert CGI requests to database queries parameterized by information supplied by an end user through a supporting HTML form. Information available only through such CGI requests comprises a portion of what researchers sometimes call the "deep Web" [1] or the "hidden Web" [9, 21].

Figure 1: Typical Web Form for Finding Automobiles

Unlike ordinary Web pages mapped to standard URLs, information in the hidden Web is not accessible through regular HTTP GET requests by merely specifying a URL and receiving a referenced page in response. Perhaps the information requires client authentication by means of a user ID and password. Or maybe the information is hidden behind a firewall in an intranet only accessible from particular IP addresses. Other portions of the Web are "hidden" only in the sense that none of the major search engines index those pages [14]. Most commonly, however, data in the hidden Web is stored in a database and is accessible by issuing queries guided by HTML forms.

A commercial vendor, BrightPlanet.com, claims that the size of the deep Web is 500 times greater than the "shallow Web" [1]. Regardless of the actual relative size, it is clear that an enormous amount of data exists outside the so-called "indexable Web" [15]. Users want and need better access to this information.

Automated extraction of data behind form interfaces is desirable when we wish to have automated agents search for desired information, when we wish to wrap a site for higher level queries, and when we wish to extract and integrate information from different sites. How can we automatically access this information? As our contribution in this paper we answer this question by explaining how to (1) automate the filling in of forms, (2) statistically streamline the

*Supported in part by the National Science Foundation under grant IIS-0083127.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

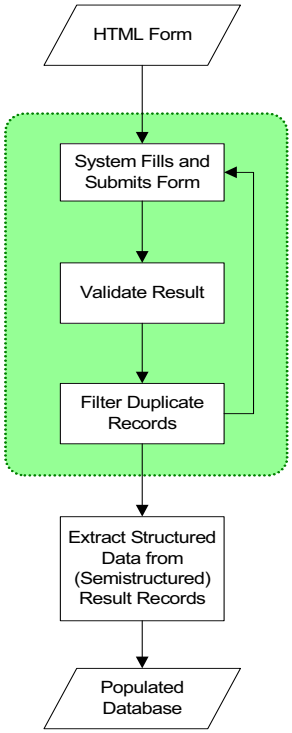

Figure 2: Flowchart of Our Approach

process to make the information gathering process efficient, and (3) analyze returned results to handle errors intelligently and to discard duplicate data. In the broader context of our project [11], information gathered during form processing will later be handed to a downstream data extraction process (see Figure 2), but an explanation of this activity has already been reported [8] is beyond the scope of this paper.

1.1 Issues in Automatic Form Filling

There are many ways to design Web forms, and dealing with all the possibilities is not easy. We can build Web forms from a variety of controls such as radio buttons, checkboxes, selection lists, text boxes, hidden controls, and even author-defined objects. However, we can simplify this list because information transmitted to the server in a CGI request is fundamentally just a list of (name, value) pairs, appropriately encoded. Thus, we can characterize a form with n controls as a tuple $F = \langle U, (N_1, V_1), (N_2, V_2), ..., (N_n, V_n) \rangle$, where U is the URL to which the encoded CGI request is sent, and the (N_i, V_i) are (name, value) pairs to be sent.

Unfortunately, this is an oversimplification. There is considerable information associated with other metadata in the HTML form specification. For example, controls can be labeled as well as named; both labels and names might suggest possible domains we could associate with the fields. Text boxes often have a maximum content length (thus limiting the domain). A server might only respond to one of the two possible access methods (GET or POST). Finite domains for certain controls might be nicely specified (e.g. hidden controls have static scalar values, and radio buttons, checkboxes, and selection groups each enumerate a relatively small set of possible values). There is a great deal we can learn from these HTML constraints.

Moreover, automated forms processing encounters a host of difficulties, including the following problems.

- Result pages might contain error messages. Some error messages are easy to recognize automatically, such as HTTP 404 error pages. Other error messages are more difficult to recognize automatically because messages may be embedded within a series of tables, frames, or other types of HTML divisions. Users can usually understand these embedded messages quite easily, but automated understanding is difficult.
- Sometimes we can retrieve all the data behind a form with a single submission. At other times, we must obtain the data piecemeal using multiple queries with different form control settings.
- When we obtain data piecemeal, we may retrieve duplicate information, which we should discard.
- Some forms lead to other, more specialized forms that require further interaction.
- A server may require a particular order of operation due to session tracking. A site, for example, might require a login or might use cookies to track a user's progress through a series of interactions.
- Client-side scripts may interact with forms in arbitrary ways to modify and constrain form behavior. A script, for example, might derive the value for a text box containing the total sales price in an order form. JavaScript may alter the behavior of forms. Unfortunately it is computationally hard to automatically analyze and understand arbitrary scripts.

All these issues present difficulties for automation. How can a system automatically fill in the fields of a form and submit it? How can a system deal with retrieved data, duplicate information, possible error messages or error notification pages, and embedded Web forms inside retrieved documents? Before describing our solution (which does not address and solve all the problems, but only the central problems of efficient form filling and submission, and duplicate result detection and elimination), we first discuss related work.

1.2 Related Work

Others have also studied the problem of automatically filling out Web forms. Most common are tools designed to make it easier for an end user to fill out a form. Commercial services exist, for example, to

provide information from a limited portfolio of user-specified information such as name, address, contact information, and credit card information [6, 7, 18]. These services, such as the Microsoft Passport and Wallet system, encrypt a user's personal information and then automatically fill in Web forms when fields can be recognized. Since many forms share common attributes (especially in the domain of e-commerce transactions), these tools can reliably assist users in entering personal information into Web forms.

One of the earliest efforts at automated form filling was the ShopBot project [5], which uses domain-specific heuristics to fill out forms for the purpose of comparison shopping. The ShopBot project, however, did not propose a general-purpose mechanism for filling in forms for non-shopping domains.

More recently, researchers have considered the problem of assisting users in complex information search tasks that may span many Web sites and multiple Web forms. Davulcu et al. report on an architecture for designing "webbases" that help users perform complex domain-specific searches using a guided, by-example tool [4]. Experts write underlying specifications declaratively because webbases are too difficult for end users to create themselves. The authors mention some heuristics, but give few details regarding the actual process of filling out forms. This project appears to be a good attempt to simplify the creation of domain-specific search services, but it does not try to be a general-purpose crawler for the hidden Web.

There are now commercial ventures providing access to portions of the hidden Web. For example, BrightPlanet.com's Complete Planet as of December 21, 2000 claims to have indexed 38,500 databases containing deep Web content [2]. Another service, InvisibleWeb.com, claims to be "a directory of over 10,000 databases, archives, and search engines" [13] containing information from the hidden Web. Both of these commercial services use semi-automated techniques for indexing the hidden Web, but they do not publicly discuss the details of their processes.

The most closely related work to our own is the Hidden Web Exposer (HiWE) project at Stanford [20, 21]. Raghavan and Garcia-Molina propose a way to extend crawlers beyond the publicly indexable Web by giving them the capability to fill out Web forms automatically. Because of the formidable challenges to a fully automatic process, HiWE assumes that crawls will be domain specific and human assisted (we also rely on human assistance at key points, but we do not use domain specific information in retrieving data from a particular site). Although HiWE must start with a user-provided description of the search task, HiWE learns from successfully extracted information and updates the task description database as it crawls. Besides an operational model of a hidden Web crawler, a significant contribution of this work is the label matching approach used to identify elements in a form based on layout position, rather than proximity within the underlying HTML code. These researchers also present the details of several ranking heuristics together with metrics and experimental results that help evaluate the quality of the proposed process. The independently-developed details of our approach are complementary to HiWE. For example, we consider the task of duplicate record elimination and we use a statistics-based sampling approach to efficiently decide when a particular source has been sufficiently extracted.

As a final related-work note, we observe that many researchers have considered the problem of planning the information gathering process in a data integration context. HTML forms can be thought of as query templates with "binding patterns" [22] that limit the "access paths" to the data of interest [10]. Our purpose is slightly different, in that we are not given source descriptions ahead of time — rather, in an effort to truly crawl the hidden Web, we attempt to extract all of the available data at a particular source, not just answer a specific query from the source.

1.3 Overview

In the remainder of the paper we describe the details of our contribution. We have created a prototype tool that automatically retrieves the data behind a particular HTML form. In Section 2, we discuss how we fill out and submit a form to a server. In Section 3 we explain how we classify and process a result page returned by a form submission. This includes, for example, error detection and following next-page links to gather all returned data. What we do with resulting data records and how we proceed with our execution plan critically depends on our ability to recognize duplicate data. We therefore next discuss in Section 4 how we recognize duplicate information and eliminate duplicate data records. Section 5 discusses our form-submission plan. In particular, we discuss the use of statistical procedures as a means to intelligently cover the search space and to determine how much of the available data has likely been retrieved and how close we are to completion. Section 6 presents the results of experiments we conducted to verify our approach to automatically extracting data behind Web forms. We summarize, report on our implementation status, and give plans for future work in Section 7

For the remainder of our discussion here, we make several simplifying assumptions. (1) We assume that user authentication is not required (or will be performed by a user who guides the search process). (2) We assume that we have arrived at a form, which when filled out, yields information of interest (rather than some other form for further interaction). (3) We assume that we can process forms of interest repeatedly, without regard to a user's session state as maintained on the server. (4) We assume that client-side

```
<form name="searchform" action="..." method="GET">

  <input type="hidden" name="c"     value="remotecontrols">
  <input type="hidden" name="v"     value="1">
  <input type="hidden" name="_ttag" value="isrch.rslt">
  <input type="hidden" name="pg"    value="1">
  <input type="hidden" name="pgid"  value="shop">

  <select name="InputType" style="width: 150px;">
      <option value="" selected>Any
      <option value="button">Hard Button
      <option value="screen">Touchscreen </select>
  <select name="InputManufacturer" style="width: 150px;">
      <option value="" selected>Any
      <option value="denon">Denon
      <option value="general electric">General Electric
      <option value="harman kardon">Harman Kardon
      <option value="infocus">InFocus
      <option value="invoca">Invoca
      <option value="jensen">Jensen
      <option value="jvc">JVC
      <option value="one for all">One For All
      <option value="onkia">Onkia
      <option value="phillips">Phillips Magnavox
      <option value="rca">RCA
      <option value="sony">Sony
      <option value="universal">Universal
      <option value="zenith">Zenith </select>

  <input type="text" name="InputModel"    size="16" value="">
  <input type="text" name="InputKeyword"  size="16" value="">
  <input type="text" name="InputPriceMin" size="7"  value="">
  <input type="text" name="InputPriceMax" size="7"  value="">
</form>
```

Figure 3: Excerpt of the Source Code of a Web Form (www.mysimon.com, reformatted for clarity)

JavaScript embedded in HTML pages will not interfere with form submission. (5) We assume that forms of interest will respond to submission via the HTTP GET method. Our initial experience suggests anecdotally that these assumptions are reasonable in the sense that most forms with backend informational databases tend to satisfy these assumptions.

2 Automated Form Filling

The first step to automated form filling is to parse the HTML page and extract useful information from the form description. We begin by creating a parse tree for the given source page. We then look for the presence of an HTML form by searching for start and end tags, <form> and </form> respectively [12]. If a form is present, we extract its portion of the parse tree and, for the purposes of experimentation and repetitive automated processing, we store the parse tree in an easy-to-read form. Information of particular interest includes the source URL of the page, the action URL to which the form will be submitted, the number of fields, and details for each field, including field names, types (domain information such as the available options for a selection list), and default values. Figure 3 shows a cleaned up version of the HTML source for a Web form that retrieves information about television remote controls available for sale. Note that the empty string is the default value for the text fields in this form — in the forms we have surveyed, this is typical.

Given the parse tree for a form, we are ready to begin filling in field values. We start by assigning default values to each field. Once assignments are made, the form information needs to be encoded properly for the request method. There are two ways to submit a form for CGI processing. First, using the HTTP POST verb, forms can be submitted with (name, value) pairs encoded in the body of the request. Second, using the HTTP GET verb, forms can be submitted by supplying (name, value) pairs in the URL. A question mark (?) separates the base URL and action path from the encoded names and values. In the name-value list, an equality operator (=) separates a name from its assigned value, and an ampersand (&) separates one (name, value) pair from the next.

For the example form in Figure 3, the base URL is http://www.mysimon.com, and the action path is /isrch/index.jhtml. The form fields are c, v, pg, pgid, _ttag, InputType, InputManufacturer, InputModel, InputKeyword, InputPriceMin, and InputPriceMax. Observe that hidden fields are like constants — they cannot be modified by users. Using the default values for non-hidden fields from Figure 3, the value for InputType is blank (indicating in this case "Any" type of remote control). So the (name, value) assignment for InputType is: InputType=. Similarly, the rest of the form field settings are assigned their default values, and the query is thus constructed as:

http://www.mysimon.com/isrch/index.jhtml?
c=remotecontrols&v=1&_ttag=isrch.rslt&pg=1&
pgid=shop&InputType=&InputManufacturer=&
InputModel=&InputKeyword=&
InputPriceMin=&InputPriceMax=

We send this query directly to the Web site referenced by the base URL. This submission has the same effect as that of a user clicking the search button without selecting or typing anything on the Web form. Figure 4 shows a portion of the returned page for this query.

We can construct other queries by selecting various combinations of selection-list values, radio-button settings, and check-box selections. Usually it is not necessary to fill in text boxes automatically. Our system allows a user to provide values for text boxes, but does not require that values be provided. For forms with text boxes, our system only submits queries that have no entries for text boxes or that have user-supplied text-box values. The text boxes in the query that returned the results in Figure 4, for example, were all empty.

3 Processing a Response Page

Once a query is sent, the next step is to retrieve information from the target site. Several different results are possible. We discuss each in turn.

Data Returned Piecemeal. The most common response is that a Web site will return results a bit

Sort by Gómez Merchant Review	Sort by Brand	Sort by Model	Sort by Price	
★★★	Proton	**Proton 10-device Universal Learning Remote- SRC-2000** **Shipping:** See Site **Avail:** In Stock **Devices:** 10	$149.95	Buy
★★★	Sony	**Sony 5 Device Universal Learning Remote- RM-VL700** **Shipping:** See Site **Avail:** In Stock **Devices:** 5	$34.95	Buy
★★★	Recoton	**Recoton Infrared Remote Control Extender- DSC-IR100A** **Shipping:** See Site **Avail:** In Stock	$39.95	Buy
★★★	InVoca	**InVoca 24977 Voice Activated Universal Remote** **Shipping:** Currently, item can be shipped only within the U.S. **Avail:** Usually ships in 1 to 2 weeks	$59.99	Buy

Figure 4: Portion of a Retrieved Data Page (www.mysimon.com)

at a time, showing perhaps 10 or 20 results per page. Usually there is a link or a button to get to the next page until the last page is reached. For this case we treat all the consecutive next pages from the returned page as part of one single document by concatenating all the pages into one page. The system activates this process if the returned page contains a button or link indicating "next" or "more." In this way, the system constructs a logical page containing all the data for the query.

Default Query Retrieves All Data. For small databases, the initial default query often returns all the data in the database (usually piecemeal as described above). The system determines statistically that all the data might have been retrieved by sampling the database with a few additional queries. If these additional queries all return data that is equal to or subsumed by the data returned for the initial default query, we need not query with all combinations. Section 4 explains how we detect whether data is "equal" or "subsumed," and Section 5 gives the details about how many and which queries the system needs to execute to reach a user-specified confidence level (typically 95%) that the default query retrieves all data.

Default Query Does Not Retrieve All Data. Every set of returned data may be some particular subset of the overall database. When we sample the database and find data not already returned by the initial default query, we continue submitting queries until we have retrieved as much of the data as the user wishes (80%, 90%, 95%, 99%, ..., or all of it). Section 5 explains how we intelligently submit queries to cover the space and how we statistically determine how much of the data the system has retrieved.

Error: No-Record Notification or Required Field Missing. This situation occurs (a) when a query returns no data or (b) when text fields require an entry. The system could search for a message such as "No matching records found" for Case (a) and could search for a message such as "Required field missing" for Case (b). It is more reliable for both cases, however, to observe that the size of the information returned after removing duplicate miscelleneous header and footer information is normally very small — usually a constant small value for all queries that return no data. For Case (a) we simply continue, but for Case (b), we require user intervention. The user either aborts the operation or supplies values for required text fields, and the system continues with these given values.

Error: Unexpected Failure. A server might be down, a network connection may fail, or there may be HTTP errors. Our system maintains a timeout routine to terminate the operation if any abnormal delay occurs. In this case the system reports the possible error(s) and aborts the current operation.

4 Recognizing Duplicate Data

When we query a Web site, we gather retrieved data into a repository. As we retrieve data for multiple submissions of a form, we eliminate duplicate chunks of data (duplicate records) before placing them in the repository. To do this, we use the copy detection system [3], which is highly effective for finding duplicate sentences over a large set of textual documents. The system divides a document into sentences, computes hash values for each sentence, and locates duplicate sentences in each hash bucket. Instead of sentence boundaries, we wish to detect record boundaries, but otherwise the processing is the same.

We usually find records in data retrieved from behind Web forms displayed as paragraphs separated by the HTML paragraph tag <p>, as rows in a table separated by <tr> </tr> tags, or as blocks of data separated by the <hr> horizontal rule tag. In order to adapt the copy detection system for a collection of records, we devised a special tag called the sentence boundary separator tag denoted by <s.>. We then modified the copy detection system to acknowledge this special tag as the end of a sentence (i.e. the end of a record). During the duplicate detection process, the system inserts this tag into retrieved Web documents around certain HTML tags that most likely delimit records. The tags we have chosen for this treatment include </tr>, <hr>, <p>, </table>, </blockquote> and </html>. If none of the above tags except </html> appears in the document, the whole document is considered to be a single record.

With this modification, the copy detection system computes hash values for every record separated by <s.>. It then compares these new hash values with the hash values of all records retrieved previously and stored in the repository. When duplicate records are found, the system eliminates them, keeping only new, unique records and their hash values in the repository.

5 Form Submission Plan

Our goal is to retrieve all the data within the scope of a particular Web form. One way to do this is to fill in the form in all possible ways. Ignoring the issue of fields with unbounded domains (i.e., text boxes), there are still two problems with this strategy. First, the process may be time consuming. Second, we may have retrieved all (or at least a significant percentage) of the data before submitting all the queries. Many forms have a default query that obtains all data available from the site. If the default query does not yield all data, it is still likely that we can extract a sufficient percentage of the data without exhaustively trying all possible queries.

Our strategy involves several phases: (1) issue the default query, (2) sample the site to determine whether the default query response is likely to be comprehensive, and (3) exhaustively query until we reach a limiting threshold. In the exhaustive phase, we can often save considerable effort by using limiting thresholds. For example, we can estimate the size of the database behind a form, and then continue issuing queries until we have reached a certain percentage of completeness. The user can specify several thresholds:

- *Percentage of data retrieved*: What percentage of the estimated data has actually been retrieved so far? Typical values for this threshold might be 80%, 90%, 95%, or 99%. This threshold controls the quality of the crawl.

- *Number of queries issued*: How many total queries have been issued to this site? It may be prudent to limit the burden placed on individual sites by terminating a crawl after a particular number of queries. This threshold controls the burden placed on crawled sites.

- *Number of bytes retrieved*: How many bytes of unique data have been retrieved so far?

- *Amount of time spent*: How much total time has been spent crawling this site? This threshold and the previous one control the resources required on the crawler side to support the crawl.

- *Number of consecutive empty queries*: How many consecutive queries have returned no new data? The probability of encountering new data goes down significantly as the number of consecutive empty queries goes up.

Each of these thresholds constitutes a *sequential stopping rule* that can terminate the crawl before trying all possible queries (by "all" we mean all combinations of choices for fields with bounded domains — we exclude fields with unbounded domains in this study).

Table 1: Two-Way Layout with Random Sampling

		Factor A						
		a_1	a_2	a_3	a_4	a_5	a_6	a_7
Factor B	b_1	x	x		x			
	b_2		x		x			
	b_3							
	b_4			x		x		

5.1 Sampling Phase

Earlier we described the method for determining and issuing the default query. We now discuss the sampling procedure used to determine whether the default query is likely to have returned all the data behind a particular form.

There are several parameters of interest to us. First, we characterize each form field as a "factor" in our search space. Let $f_1, f_2, ...f_n$ be the n factors corresponding to fields with bounded domains, and let $|f_i|$ represent the number of choices for the i^{th} factor. Then the total number of possible combinations N for this form is:

$$N = \prod_{i=1}^{n} |f_i|.$$

We are also interested in the cardinality c of the largest factor: $c = \max(|f_1|, |f_2|, ..., |f_n|)$.

Next, we define C to be the size of a sampling batch. We want each sampling batch to be large enough to cover the margins of our sample space — that is, we want to have fair coverage over all the factors. So we let $C = \max(c, \lceil \log_2 N \rceil)$. This accounts for the case where there are many factors of small cardinality. For example, if there were 16 factors each of cardinality 2, then $N = 2^{16} = 65536$, but $c = 2$. We want c to be representative of the size of our search space, so we require that it be at least $\log_2 N$, which is a statistically reasonable number to use when sampling populations of known size (consider, for example, the 2^k factorial experiments method [23, 17]).

Our decision rule for determining whether the default query returned all the data available from a particular site is based on a sample of C queries. If all C queries return no additional data (i.e. after duplicates have been eliminated), then we assume the default query did indeed retrieve all available data.

However, we need to be careful about how we choose the C queries. Suppose the form of interest contains two bounded fields with choices of 7 and 4 possibilities respectively. Then we have $N = 7 \times 4$ and $C = \max(\max(4, 7), \lceil \log_2 28 \rceil) = \max(7, 5) = 7$. If we simply choose C random queries, we might end up with a sample set like the one shown in Table 1. This table shows a two-way layout [23, 16, 17, 19] that helps us use a two-factor method [23, 17, 19] to choose a query sample. Notice that a_6, a_7, and b_3 were not considered in any of the sample queries, while b_1 is oversampled.

One solution is to keep track of how many times we have sampled each factor, and spread the samples

Table 2: Regular Sampling with Maximal Coverage

		Factor A						
		a_1	a_2	a_3	a_4	a_5	a_6	a_7
Factor B	b_1	x				x		
	b_2		x				x	
	b_3			x				x
	b_4				x			

Table 3: Random Sampling with Maximal Coverage

		Factor A						
		a_1	a_2	a_3	a_4	a_5	a_6	a_7
Factor B	b_1				x			x
	b_2	x					x	
	b_3					x		
	b_4		x	x				

evenly, as Table 2 shows. This approach for constructing a sampling search pattern yields "maximal coverage."

In Table 2 we use a regular pattern to cover both factors as broadly as possible. The sample consists of the sequence (a_1, b_1), (a_2, b_2), (a_3, b_3), (a_4, b_4), (a_5, b_1), (a_6, b_2), (a_7, b_3). If we were to continue, the next query we would choose would be (a_1, b_4). The algorithm chooses a next sample that is as far away from all previous samples as possible. Since we have categorical, not quantitative, data each of the a_i choices for factor A is equally distant from all others. Thus, our distance function simply measures the number of coordinates that are different. For example, the distance between (a_1, b_1) and (a_2, b_2) is 2, while the distance between (a_1, b_1) and (a_7, b_1) is 1.

To ensure that a regular pattern does not bias our results, we introduced a stochastic element by randomly choosing next samples from the list of all those that are equally furthest from the set selected so far. This yields a layout like the one in Table 3. Note that our technique is general for n dimensions, $n \geq 2$.

After issuing C queries stochastically in a maximally covering fashion, if we have received no new data we judge the default query to be sufficient and halt, claiming that we have successfully retrieved all data. In practice, this rule has been highly successful (as judged by human operators manually verifying the decision). Although this rule never reported that all data had been retrieved when it had not, we did encounter sites where our rule was too strict. (The copy detection system reported new bytes in subsequent queries, but there were not really any new records; this can be the result of personalized advertizing information or other dynamic variations in Web pages).

Our decision rule for determining whether a query yields "new" data is based on a size heuristic. First we strip a returned page of its HTML tags, leaving on the special sentence boundary tags (<s.>) described in Section 4. Then we use the copy detection system to remove duplicate sentences, and count the number of bytes U remaining. If U is at least 1000, we assume we have received new data. If U is less than 1000, we use order analysis to prune small results. We first find the minimum size M returned by the previous C queries. If U is greater than M then we assume we have received new data. We have also tested a percentage threshold, comparing U to the total number of bytes returned in the page (excluding HTML tags). Both heuristics perform reasonably well. Since we are using a statistical approach, some noise in the system is acceptable, and indeed some even cancels itself out.

Another way to test for uniqueness of data would be to call on our downstream record extraction process to actually extract and structure the records from this page. Then we could perform database-level record comparisons to determine when we have found unique records. However, this makes the Web-form retrieval less general by tying it to a specific domain ontology. In the future we will investigate more fully the ramifications of this alternate approach.

5.2 Exhaustive Phase

If the C sample queries yield new data, we proceed by sampling additional batches of C queries at a time, until we reach one of the user-specified thresholds or we exhaust all the possible combinations. Sampling proceeds according to the maximal covering algorithm discussed earlier — each new query is guaranteed to be as far away from previous queries as possible, thus maximizing our coverage of the factors.

However, before proceeding, we first estimate and report the maximum possible space needed for storing the results and the maximum remaining time needed to finish the process. We also give the user the choice of specifying the various thresholds for completeness of retrieved data, maximum number of queries to be issued, maximum storage space to use, and maximum time to take. Additionally, the user can decide to stop and use only the information already obtained. (All this information can also be specified ahead of time so the process can run unattended.)

We estimate the maximum space requirement S by multiplying the total number of queries N by the average of the space needed for data retrieved from n sample queries:

$$S = \left(\frac{N}{n}\right) \sum_{i=1}^{n} b_i$$

where b_i is the size in bytes of the i^{th} sample query, and typically $n = C$.

We estimate the remaining time required T similarly:

$$T = \left(\frac{N}{n}\right) \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} t_i = \left(\frac{N-n}{n}\right) \sum_{i=1}^{n} t_i$$

where t_i is the total duration of the i^{th} sample query. Note that we subtract the time already spent in the initial sampling phase. Also observe that since we follow "next" links, what we are calling a "query" could

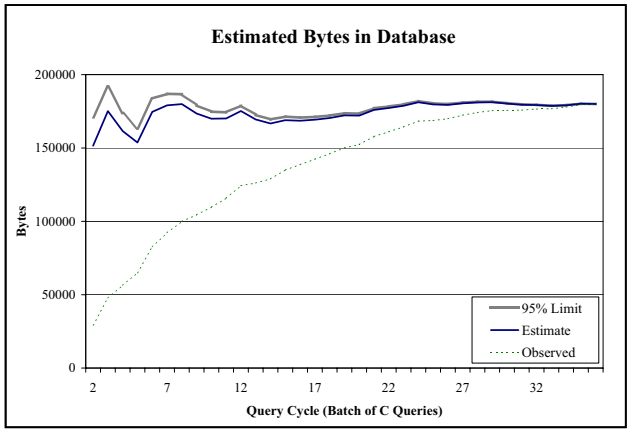

Figure 5: Completeness Measures for Automobile Ads Web Site (www.slc-classifieds.com, Form 1)

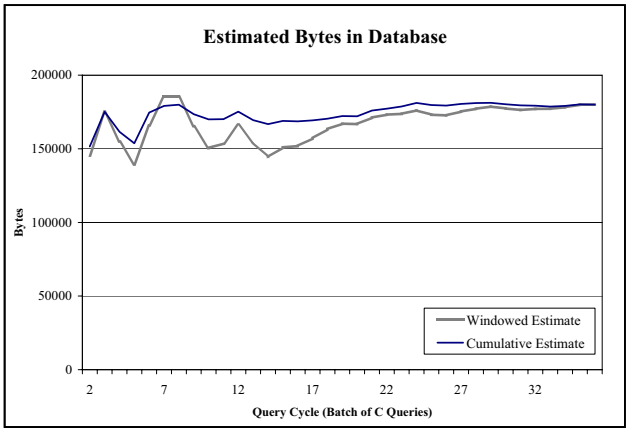

Figure 6: Windowed vs. Cumulative Estimates

actually involve a fair number of HTTP GET requests. For larger sites, this can take a significant amount of time.

After establishing the user's preferences, we proceed to process additional batches of C query samples. At the end of each sample batch, we test the thresholds to see if it would be productive to continue processing. Note that each batch provides maximal coverage of the various factors, using unique combinations that have not yet been tried.

Figure 5 illustrates how we measure our progress with respect to the percentage of information retrieved so far. The "Observed" line indicates how many actual unique bytes we have seen after each query cycle. The "Estimate" line shows our estimate of the number of unique bytes we would encounter if we were to exhaustively crawl this Web site (trying all possible combinations of queries). The "95% Limit" line uses the standard deviation of our prediction estimate for the probability of finding additional data to determine the level at which we can claim 95% confidence about our estimate. That is, based on what we have seen so far, we are 95% confident that the real number is less than the "95% Limit" number. In this example, we cross the 80% completeness threshold in cycle 14 (out of 36 total). After 22 cycles we estimate that we are 90% complete. We reach 95% in cycle 25, and 99% in cycle 31. Only after all 36 cycles have been exhaustively attempted would we determine that we are 100% complete. In this example, $N = 2124$ and $C = 59$. Thus, if 80% completeness is sufficient for the user, we prune 1297 out of 2124 queries (61%) from the list.

The formula for estimating the database size D_i after i queries is shown in Equation 1:

$$D_i = O_i \left(1 + \frac{N-i}{i} p_i\right) \tag{1}$$

where O_i is the number of unique bytes observed after i queries, and p_i is the estimate of the probability of finding new data in query $i+1$. The only random variable in Equation 1 is p_i, which is defined as the number of queries that returned new data divided by i. Thus, p_i is a cumulative probability estimate reflecting the ratio of successful queries to total queries. We compute D_i by predicting that in the remaining $N-i$ queries the proportion that return new data will be approximately p_i. Further, we estimate that on average successful queries will return approximately $\frac{O_i}{i}$ new bytes. We can compute $\hat{D}_i$, that is, D_i with 95% confidence by including in p_i a measure of the standard deviation of p_i over the previous two query cycles, σ_i, as shown in Equation 2:

$$\hat{D}_i = O_i \left(1 + \frac{N-i}{i}(p_i + 1.645\sigma_i)\right) \tag{2}$$

Rather than using a cumulative estimate of database size, we could instead use a window to ignore older data points. For example, Figure 6 compares a cumulative estimate of D_i with a windowed estimate, where the probability p_i is only computed using the last two query cycles. One might think that a windowed probability estimate should be more accurate because it reflects more recent experience. But in practice the more stable cumulative estimate appears to give better performance among the sites we have examined. Both converge relatively quickly and show nicely declining variance over time.

6 Experimental Results

A considerable amount of our work was done by simulation, crawling a site once and then playing with different sampling scenarios to understand the ramifications of various approaches. However, we also tried our tool on 13 different Web sites from several different application domains to evaluate its performance empirically. In all cases, we manually verified the decision reported by the system regarding whether the default query retrieved all data. In five of the cases, the default query did indeed return all the data.

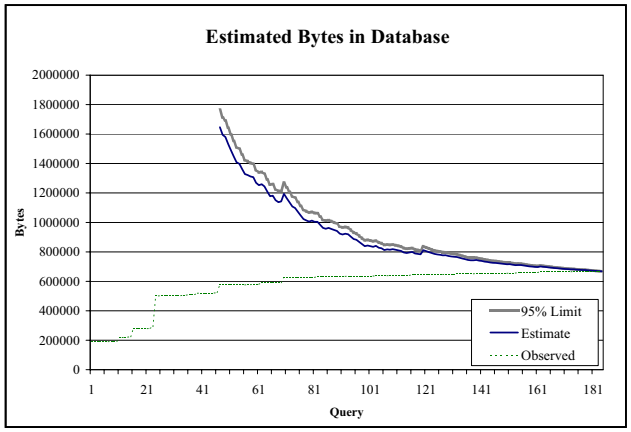

Figure 7: A Contrasting Automobile Search (www.slc-classifieds.com, Form 2)

In our testing, processing a single HTTP request took anywhere from 2 to 25 seconds on average, depending on the Web site. A single query, which includes following "next" links, averaged between 5 seconds and 14 minutes. Some pages had no "next" links, but others had as many as 140 such links! Thus, the sampling phase could take anywhere from a few dozen seconds to several hours. Storage requirements were modest by modern standards, requiring anywhere from several megabytes to hundreds of megabytes per tested site.

We encountered two typical data patterns. First is the relatively sparse behavior exhibited by the site crawled in Figure 5. The second is fairly dense, as typified by Figure 7. Both of these data sets come from the same Web site, but through different forms to different sub-portions of the site. In this second example, there are only 8 query cycles of 23 queries each (8 choices for one bounded field, 23 for another), for a total of $N = 184$. However, it is not until the end of cycle 6 (query 138) that we estimate we have retrieved 80% of the available data. And even then, there is still an estimated 42% probability of obtaining additional data with another query. Thus, we can only save a small portion of the 184 queries (25% or fewer). Even in such cases, however, the 15-25% savings can be significant.

7 Conclusion

In this paper we have described our domain-independent approach for automatically retrieving the data behind a given Web form. We have prototyped a synergistic tool that brings the user into the process when an automatic decision is hard to make. We use a two-phase approach to gathering data: first we sample the responses from the Web site of interest, and then, if necessary, we methodically try all possible queries (until either we believe we have arrived at a fixpoint of retrieved data, or we have reached some other stopping threshold, or we have exhausted all possible queries).

We have created a prototype (mostly in Java, but also using JavaScript, PHP, and Perl) to test our ideas, and the initial results are encouraging. We have been successful with a number of Web sites, and we are continuing to study ways to improve our tool.

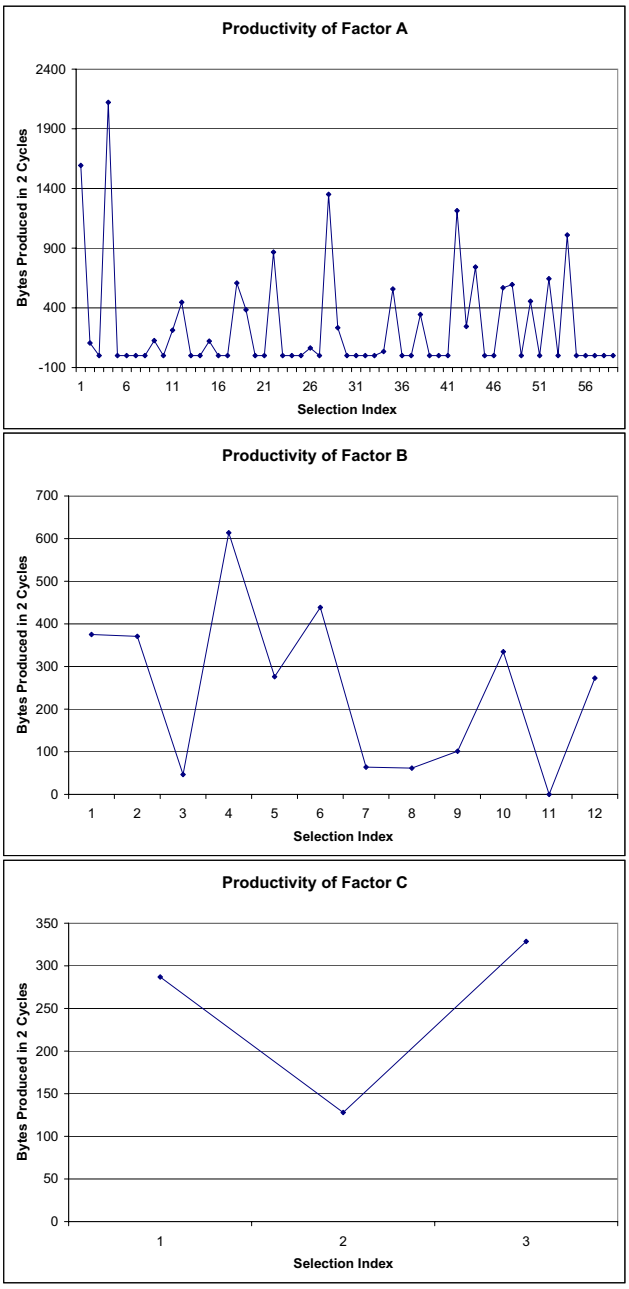

Figure 8: Relative Productivity of Various Factors

One improvement that seems particularly promising is to perform an analysis of the productivity of various factors in order to emphasize those that yield more data earlier in the search process. For example, Figure 8 shows that choices 1, 4, and 28 of Factor A have been significantly more productive than average in the first two query cycles. Similarly, choices 4 and 6 of Factor B, and choices 1 and 3 of Factor C

are highly productive. A straightforward two-way interaction analysis indicates which pairs of choices for various factors are most productive. There are challenging issues with determining how best to partition the search space, but we are studying how a directed search algorithm might perform.

Our research group is generally working in the broader context of ontology-based data extraction and information integration. If we combine the hidden Web retrieval problem with the tool of domain-specific ontologies, we could automatically fill in text boxes with values from the ontologies. While this makes the retrieval process task-specific, it also increases the likelihood of being able to extract just the relevant subset of data at a particular Web site. (Our assumption in this paper has been that the user wants to retrieve all or most of the data at a given site.) Also, it is likely that we could avoid a fair amount of the manual intervention in our current process if we use ontologies.

Users need and want better access to the hidden Web. We believe this will be an increasingly important and fertile area to explore. This paper represents a step in that direction, but a great deal remains to be done. We look forward to continuing this promising line of research.

References

[1] M.K. Bergman. *The Deep Web: Surfacing Hidden Value.* BrightPlanet.com, July 2000. Downloadable from http://www.brightplanet.com/deep_content/-deepwebwhitepaper.pdf, checked August 10, 2001.

[2] Completeplanet.com home page. http://www.completeplanet.com. Checked August 10, 2001.

[3] D.M. Campbell, W.R. Chen, and R.D. Smith. Copy detection system for digital documents. In *Proceedings of the IEEE Advances in Digital Libraries (ADL 2000)*, pages 78–88, Washington, DC, May 2000.

[4] H. Davulcu, J. Freire, M. Kifer, and I.V. Ramakrishnan. A layered architecture for querying dynamic Web content. In *SIGMOD '99 Proceedings*, pages 491–502, Philadelphia, Pennsylvania, May 1999.

[5] R.B. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proceedings of the First International Confence on Autonomous Agents*, pages 39–48, Marina del Rey, California, February 1997.

[6] Patil systems home page. http://www.patils.com. Describes LiveFORM and ebCARD services. Checked August 10, 2001.

[7] eCode.com home page. http://www.eCode.com. Checked August 10, 2001.

[8] D.W. Embley, D.M. Campbell, Y.S. Jiang, S.W. Liddle, D.W. Lonsdale, Y.-K. Ng, and R.D. Smith. Conceptual-model-based data extraction from multiple-record Web pages. *Data and Knowledge Engineering*, 31:227–251, 1999.

[9] D. Florescu, A.Y. Levy, and A.O. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[10] A.Y. Halevy. Answering queries using views: A survey. *VLDB Journal (online, to appear)*, 2001.

[11] Home Page for BYU Data Extraction Group, 2000. URL: http://www.deg.byu.edu.

[12] HTML 4.01 specification. http://www.w3.org/TR/html4, December 1999. Checked August 10, 2001.

[13] InvisibleWeb.com home page. http://www.invisibleweb.com. Checked August 10, 2001.

[14] S. Lawrence and C.L. Giles. Accessibility of information on the Web. *Nature*, 400:107–109, 1999.

[15] S. Lawrence and C.L. Giles. Searching the World Wide Web. *Science*, 280:98–100, April 1999.

[16] T. Leonard. *A Course In Categorical Data Analysis.* Chapman & Hall/CRC, New York, 2000.

[17] R.A. McLean and V.L. Anderson. *Applied Factorial and Fractional Designs.* Marcel Dekker, Inc., New York, 1984.

[18] Microsoft Passport and Wallet services. http://memberservices.passport.com. Checked August 10, 2001.

[19] R.L. Plackett. *The Analysis of Categorical Data, 2^{nd} Edition.* Charles Griffin & Company Ltd., London, 1981.

[20] S. Raghavan and H. Garcia-Molina. Crawling the hidden Web. Technical Report 2000-36, Computer Science Department, Stanford University, December 2000. Available at http://dbpubs.stanford.edu/pub/2000-36.

[21] S. Raghavan and H. Garcia-Molina. Crawling the hidden Web. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, Rome, Italy, September 2001.

[22] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *PODS '95 Proceedings*, pages 105–112, San Jose, California, 1995.

[23] A.C. Tamhane and D.D. Dunlop. *Statistics and Data Analysis: From Elementary to Intermediate*. Prentice-Hall, New Jersey, 2000.