On the Automatic Extraction of Data from the Hidden Web

Stephen W. Liddle[1], Sai Ho Yau, and David W. Embley[1]

Information Systems Group and Computer Science Department
Brigham Young University
Provo, UT 84602, USA
liddle@byu.edu, {yaus, embley}@cs.byu.edu

Abstract. An increasing amount of Web data is accessible only by filling out HTML forms to query an underlying data source. While this is most welcome from a user perspective (queries are easy and precise) and from a data management perspective (static pages need not be maintained; databases can be accessed directly), automated agents have greater difficulty accessing data behind forms. In this paper we present a method for automatically filling in forms to retrieve the associated dynamically generated pages. Using our approach automated agents can begin to systematically access portions of the "hidden Web."

1 Introduction

The World Wide Web is not only vast, but it is growing at an enormous rate. Dealing with such huge quantities of information has been a challenge for both consumers and providers. On the one hand, services such as search engines, directories, portals, and price-comparison sites have become crucial tools for end users. In an attempt to better meet user needs on the scale of the Web, researchers and product developers have also been working on agent-oriented technologies that can use more intelligence to assist users in their online activities by automating as many tasks as possible. On the other hand, information and service providers have had to apply advanced techniques to manage issues of scale. Among other issues, typically this implies using a database to generate Web pages dynamically.

Often, dynamically generated pages are accessible only through the use of an HTML form that invokes a Common Gateway Interaction (CGI) request to the Web server. CGI requests are often converted to database queries parameterized by information supplied by the end user through the 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" [8, 19]. Unlike ordinary Web pages mapped to standard URL's, information in the hidden Web is not accessible through regular HTTP GET requests by merely specifying a URL and receiving the desired 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"

[1] Supported in part by the National Science Foundation under grant IIS-0083127.

only in the sense that none of the major search engines indexes those pages [12]. Most commonly, however, data in the hidden Web is stored in a database and is accessible by issuing queries guided by the aforementioned HTML forms.

A commercial vendor, BrightPlanet.com, claims the size of the deep Web is 500 times greater than the "shallow Web" [1]. Regardless of the actual relative sizes, it is clear that an enormous amount of data exists outside the so-called "indexable Web" [13]. Users want and need better access to this information.

From the information provider's point of view, this paradigm of using HTML forms to act as an interface to an underlying database provides better information management and a greater variety of ways to display information. From the user point of view, this paradigm increases flexibility and control over what data is retrieved and displayed in response to a specific query. From a Web crawler's point of view, however, this paradigm makes it difficult to extract the data behind the form interface automatically. Such automation is desirable (1) when we wish to have automated agents search for particular items of information, (2) when we wish to wrap a site for higher level queries, and (3) when we wish to extract and integrate information from different sites.

We begin with a reference model of the information search task:

1. Formulate a query or task description.
2. Find sources that pertain to the task.
3. For each potentially useful source:
   - Fill in the source's search form.
   - Analyze the results.
   - Gather any useful information that aids in the task.
   - Refine the query criteria and repeat if necessary.

Step 1 is often an informal step, but we will assume that the task is described clearly. Step 2, source discovery, typically begins either with a keyword search on one of the major search engines or a query to one of the directory services. Sometimes the user already knows where to begin or they have a social network that can quickly give a good recommendation [11]. For this study, we assume a potential source has already been discovered, and so limit our focus to step 3. In particular we seek to automate the filling in of forms, the analysis of results (including intelligent handling of errors), and the gathering of relevant data from the results. In the broader context of our system, information gathered during form processing will later be handed to a downstream data extraction process [7]; but data extraction is also beyond the scope of the current study.

1.1 Issues in Automatic Form Filling

There are many ways to design Web forms, and dealing with all the possibilities is not easy. Web forms can be built 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

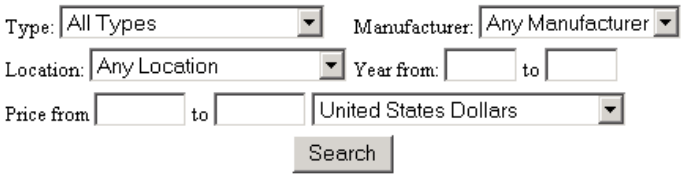


Fig. 1. Typical Web Form from an Automobile Search Site

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 oversimplifies things with respect to the task of choosing the values to submit. 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), and 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 the constraints in the HTML, so it is very helpful when initially analyzing a Web form to consider the whole Document Object Model (DOM) representation of the form.

Some forms lead to other, more specialized forms that require further interaction. And it is often the case that a result page will include both retrieved data and a form for further processing. For example, when you search Google, there is a text box above the results that allows you to refine your search. Result pages might also contain error messages. Some error messages are easy to recognize automatically, such as an HTTP 404 error (page not found), which is defined as part of the HTTP protocol. Other error messages are more difficult to recognize automatically because the message is 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 more difficult.

Sometimes all the data behind a form can be retrieved with a single query. At other times, the data must be obtained piecemeal using multiple queries with different form control settings. When data is obtained piecemeal, we may retrieve duplicate information.

It might even be the case that interaction with the server requires a particular order of operation due to session tracking. For example, a login might be required, or cookies might be initialized and used to track a user's progress through a series of interactions with the server.

Furthermore, client-side scripts may interact with forms in arbitrary ways to modify and constrain form behavior. For instance, a text box control containing the total sales price in an order form might be automatically derived from the values in other text boxes by JavaScript whenever the form changes or is submitted to the server. JavaScript is often used to alter the behavior of

forms. Unfortunately it is computationally hard to automatically analyze and understand arbitrary scripts.

All these issues present difficulties for automation. There are already parsers to convert HTML into a useful structured representation (i.e., DOM). But 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? In this paper, we present our particular approach to these issues. Before describing our solution, we first discuss the 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. For example, commercial services exist to provide information from a limited portfolio of user-specified information such as name, address, contact information, and credit card information [5, 6, 16]. 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 [4], which used domain-specific heuristics to fill out forms for the purpose of comparison shopping. Given the focus of this project, ShopBot did not propose a general-purpose mechanism for filling in forms for non-shopping domains.

More recently, researchers have considered the problem of assisting the user in a complex information search task 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 [3]. Underlying specifications are written declaratively by experts because webbases are probably too difficult for end users to create themselves. Some heuristics are mentioned, but few details are given 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" [10] containing information from the hidden Web. Both of these commercial services claim to 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 [18, 19]. Raghavan and Garcia-Molina propose a way to ex-

tend 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 crawls will be domain specific and human assisted (we also rely on human assistance at key points, but we do not currently 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, not 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 stratified sampling approach to decide when a particular source has been fully extracted.

1.3 Outline

In the remainder of the paper we describe the details of our approach. We have created a prototype tool that allows users to semi-automatically and synergistically retrieve the data behind a particular source. In Section 2, we discuss how we fill out and submit a given form to a server with the object of retrieving all the data behind the form. Section 3 discusses our query execution plan, and the use of stratified sampling as a means to determine how much of the available data has likely been retrieved and how close we are to completion. Then in Section 4 we describe how we handle the issue of classifying the kind of result page returned by a query. Section 5 describes how we eliminate duplicate records from the accumulated results. We report on our implementation status, experimental results, and future work in Section 6

Recognizing that this project is a first step, we make several simplifying assumptions. First, we assume that user authentication is not required (or will be performed by the user who synergistically guides the search process), and forms of interest can be processed repeatedly, without regard to a user's session state as maintained on the server. We also assume that client-side JavaScript embedded in HTML pages will not interfere with the form submission. In the event that a page contains multiple forms, we also assume that the first form is the one of interest. Finally, 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, but we will test this assertion quantitatively and more carefully in the coming months.

2 Automated Form Filling

The first step to automatically fill in and submit a chosen form is to parse the HTML page and extract useful information from the form description. So we

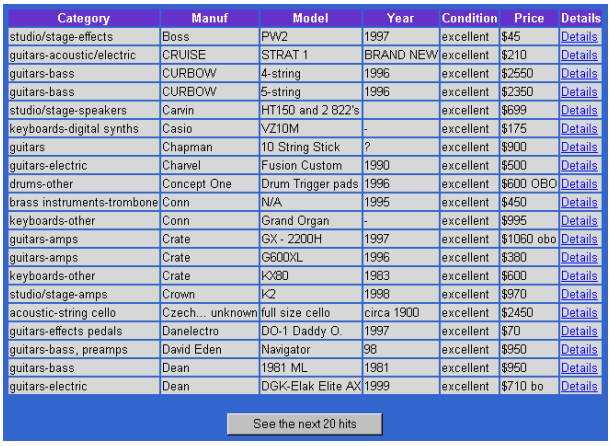
```
(1)  <form action="/cgi-bin/umg/search.cgi" method="POST"
(2)        enctype="x-www-form-encoded">
(3)  <b>Category:</b>
(4)    <select name="category" size="1">
(5)      <option selected value="">all categories</option>
(6)      <option value="accessories">accessories</option>
(7)      <option value="acoustic">acoustic (&quot;violins&quot;)</option>
(8)      <option value="drums">drums</option>
(9)      <option value="guitars">guitars</option>
(10)     <option value="keyboards">keyboards</option>
(11)     <option value="studio/stage">studio/stage</option>
(12)   </select> choose one<br>
(13) <b>Manufacturer:</b>
(14)   <input type="text" size="30" name="manuf"> example Gibson<br>
(15) <b>Model:</b>
(16)   <input type="text" size="30" name="model"><br>
(17) <b>Year:</b>
(18)   <input type="text" size="30" name="year"><br>
(19) <b>Condition:</b>
(20)   <input type="text" size="30" name="condition"><br>
(21) <b>Sort results in alphabetical order of:</b>
(22)   <select name="sort_by" size="1">
(23)     <option selected value="1">Category</option>
(24)     <option value="2">Manufacturer</option>
(25)     <option value="3">Model</option>
(26)     <option value="4">Condition</option>
(27)   </select><br>
(28)   <input type="reset" value="Clear Form">
(29)   <input type="submit" name="submit_search" value="SEARCH">
(30) </form>
```

Fig. 2. Excerpt of the Source Code of a Web Form (www.usedmusicgear.com)

begin by creating a parse tree for the given source page. An HTML form is indicated by the presence of start and end tags, <form> and </form> respectively [9]. 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 on disk in a 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. These details include field names, types (domain information including, e.g., the available options for a selection list), and default values. Figure 2 shows a cleaned up version of the HTML source for a Web form that retrieves information about used musical instruments. Note that for all the fields except *sort_by*, the default value is the empty string.

Given the parse tree for a form, we are ready to begin actually filling in values for the form. 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 the (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.



Category	Manuf	Model	Year	Condition	Price	Details
studio/stage-effects	Boss	PW2	1997	excellent	$45	Details
guitars-acoustic/electric	CRUISE	STRAT 1	BRAND NEW	excellent	$210	Details
guitars-bass	CURBOW	4-string	1996	excellent	$2550	Details
guitars-bass	CURBOW	5-string	1996	excellent	$2350	Details
studio/stage-speakers	Carvin	HT150 and 2 822's		excellent	$699	Details
keyboards-digital synths	Casio	VZ10M	-	excellent	$175	Details
guitars	Chapman	10 String Stick	?	excellent	$900	Details
guitars-electric	Charvel	Fusion Custom	1990	excellent	$500	Details
drums-other	Concept One	Drum Trigger pads	1996	excellent	$600 OBO	Details
brass instruments-trombone	Conn	N/A	1995	excellent	$450	Details
keyboards-other	Conn	Grand Organ	-	excellent	$995	Details
guitars-amps	Crate	GX - 2200H	1997	excellent	$1060 obo	Details
guitars-amps	Crate	G600XL	1996	excellent	$380	Details
keyboards-other	Crate	KX80	1983	excellent	$600	Details
studio/stage-amps	Crown	K2	1998	excellent	$970	Details
acoustic-string cello	Czech... unknown	full size cello	circa 1900	excellent	$2450	Details
guitars-effects pedals	Danelectro	DO-1 Daddy O.	1997	excellent	$70	Details
guitars-bass, preamps	David Eden	Navigator	98	excellent	$950	Details
guitars-bass	Dean	1981 ML	1981	excellent	$950	Details
guitars-electric	Dean	DGK-Elak Elite AX	1999	excellent	$710 bo	Details

See the next 20 hits

Fig. 3. Returned Page Containing Retrieved Data

For the example form in Figure 2, the base URL is http://www.usedmusicgear.com, and the action path is /cgi-bin/umg/search.cgi. The form fields are *category*, *manuf*, *model*, *year*, *condition*, *sort_by*, *reset*, and *submit_search*. Using the default values from Figure 2, the value for *category* is blank (indicating in this case "all categories"). So the (name, value) assignment for *category* is: category=. Similarly, the rest of the form field settings are assigned their default values, and the query is thus constructed as:

http://www.usedmusicgear.com/cgi-bin/umg/search.cgi?category=&manuf=&model=&year=&condition=&sort_by=1&submit_search=SEARCH

This query is then sent directly to the Web site of interest. It has the same effect as that of a user clicking the search button without selecting or typing anything on the Web form. Figure 3 shows 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 3, for example, were all empty.

3 Query 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 the data before submitting all the queries. The latter case is often due to a default query that obtains all available data in the first place.

If one query or a small number of queries can retrieve all the data, it is desirable to quit before exhaustively processing all the combinations of the form

Table 1. A Two-Way Layout with Random Sampling

		Factor A: category						
		all categories	accessories	acoustic	drums	guitars	keyboards	studio/stage
Factor B: sort by	Category	x	x		x			
	Manufacturer		x					
	Model							
	Condition					x		

Table 2. Stratified Sampling

		Factor A: category (Regular Pattern)						
		all categories	accessories	acoustic	drums	guitars	keyboards	studio/stage
Factor B: sort by	Category	x				x		
	Manufacturer		x					
	Model			x				
	Condition				x			

		Factor A: category (Stratified Random Pattern)						
		all categories	accessories	acoustic	drums	guitars	keyboards	studio/stage
Factor B: sort by	Category	x					x	
	Manufacturer			x				
	Model							x
	Condition					x		

fields in order to get all the information. We cannot be sure, however, that one query will always obtain all the information. Hence, we send a sampling of queries and determine statistically whether we have obtained all available data. One way is to randomly select the sampling of queries among all possible queries. Unfortunately, randomly selected queries might not thoroughly and evenly cover all the fields in a form. Some fields may be underused while others are overused. The size of the sampling is another issue. How many samples should be good enough to determine whether we have retrieved all available data? In order to have better coverage and an adequate number of representative sample queries, we adopt a stratified sampling method [21, 22].

3.1 Stratified Sampling Phase

The particular approach we take is to adopt the factorial and fractional techniques [21, 15, 17] of the stratified sampling method. The essence of the method is to avoid the probability of uneven selection of queries that might be biased toward certain fields. Stratified sampling seeks to use all form fields evenly to give maximum likely coverage of the underlying data.

The form in Figure 2 can be used to illustrate the formation of a stratified sampling pattern. Of the 8 controls in this form, we ignore the reset button and assume the submit button will always be selected. Further, 4 of the remaining controls are text boxes which we will ignore (if necessary, the user may choose values at runtime for the text boxes). Remaining are two selection lists, *sort_by* with 4 choices and *category* with 7 choices, that determine the set of queries we can attempt. Table 1 shows a two-way layout [21, 14, 15, 17] that helps us use a two-factor method [21, 15, 17] to choose a query sample. In Table 1 we show a random sampling that is not stratified.

From Table 1, letting $p = 7$ and $g = 4$, we can observe that there are $p \times g = 28$ combinations to consider. In order to estimate how many sample queries should

```
n              := number of sample queries from Equation 1
factor_A_list  := list of field names for factor A
length_A       := total number of field names for factor A
factor_B_list  := list of field names for factor B
length_B       := total number of field names for factor B
query_string   := a pre-constructed query with base URL, action path,
                  and criteria (like hidden fields) required for all queries.
While (n > 0) do:
    Randomly choose a field name from factor_A_list.
    Append the chosen field name as a criterion to query_string.
    length_A := length_A − 1
    If (length_A = 0) Then
        Reset factor_A_list to the list of all field names for factor A.
        Reset length_A to the number of fields for factor A.
    End if
    Randomly choose a field name from factor_B_list.
    Append the chosen field name as a criterion to query_string.
    length_B := length_B − 1
    If (length_B = 0) Then
        Reset factor_B_list to the list of all field names for factor B.
        Reset length_B to the number of fields for factor B.
    End if
    Send query_string to the target site and retrieve information.
    Reset query_string to its initial value.
    n := n − 1
End while
```

Fig. 4. Pseudocode for Stratified Sampling Method

be used, we adapt the 2^k factorial experiments method [21, 15]. Here k refers to the number of factors needed (*category* and *sort_by* in this example). We want $2^k = N$ where N is the total number of possible observations. Thus the desired sample size, n is defined by Equation 1:

$$n = \lceil log_2 N \rceil \tag{1}$$

For the example in Figure 2, since $N = 28$ we have $n = \lceil log_2 28 \rceil = \lceil 4.8 \rceil = 5$. Thus, our system first issues the default query ("all categories" from factor A and sort by category from factor B), and then an additional $n - 1 = 4$ queries.

Notice how the sample chosen in Table 1 does not cover three levels of factor A and one level of factor B. At the same time, "sort by category" and "accessories" are oversampled. Stratified sampling reduces this problem. Table 2 shows two stratified samples, in a regular pattern and a random pattern.

The regular pattern in Table 2 covers every level for factor B and most of the levels of Factor A. However, if the pattern is always chosen in this regular manner, a certain degree of bias is inevitably introduced since the left side of the table is always considered whereas the right side is not. On the other hand, the random pattern exhibits the advantage of thorough coverage and yet evenly distributes the selection randomly. Pseudocode of the algorithm for constructing the stratified sampling is shown in Figure 4. It randomly selects factors and levels, while also guaranteeing maximum coverage.

For three or more factors, the two-factor method can be generalized by replicating portions of factors over the columns and rows of the layout matrix [21]. The extension of the algorithm is straightforward so we do not present it here.

The number n obtained from Equation 1 is the upper bound of the stratified sampling size for determining the condition that if there is no new discovered

information, the system should quit early and declare that all information has been found instead of exhaustively submitting all N possible queries. During the submission of the n sample queries, the system keeps track of returned Web pages and whether the retrieved data has been seen before. In general, the data may be a subset of the first returned Web document resulting from the default query, it may be the same as the default, or it may be different. If no new information has been seen within n sample queries, it is assumed that the default query retrieved all information. If one of the returned Web pages has information that has never been seen before, then it is assumed that the default query has not retrieved all the information. In this case, the system enters to the exhaustive phase.

3.2 Exhaustive Phase

Before exhaustively executing all queries, we first estimate and report the maximum possible space needed for storing the results and the maximum remaining time needed to finish the process. The user can decide to continue the exhaustive query processing or stop and use only the information already obtained.

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 the n sample queries, as shown in Equation 2:

$$S = \left(\frac{N}{n}\right) \sum_{i=1}^{n} b_i \tag{2}$$

where b_i is the size in bytes of the i^{th} sample query.

We estimate the remaining time required T similarly, as shown in Equation 3:

$$T = \left(\frac{N}{n}\right) \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} t_i = \left(\frac{N-n}{n}\right) \sum_{i=1}^{n} t_i \tag{3}$$

where t_i is the total duration of the i^{th} sample query. Note that we subtract the time already spent in the sample phase.

Finally we note that the user can specify a cutoff threshold to throttle query processing during the exhaustive phase if a certain number of consecutive queries have returned no new records. By default this threshold is the cardinality of the largest factor considered in the stratified sampling phase.

4 Processing a Query Response Page

Once a query is sent, the next step is to retrieve information from the target site. Seven different results are possible:

1. The response page may contain all the data behind the form.
2. The response page may contain data, but not show all the data for the query in a single page. Instead, there may be a "next" button or hyperlink leading to another page of data, such as the "See the next 20 hits" button in Figure 3.

3. The query might return data, but only part of the data behind the form because the query is just one of many possible combinations of the form fields.
4. The query may return a page that not only contains data, but also contains the original form.
5. The query may return a page that has another different form to fill in.
6. The query might return an error message or an error notification page, stating that certain text fields are required to be filled in, or simply a message stating that there is no record found for that submitted query.
7. Some other error cases might involve a server being down, an unexpected failure of a network connection, or other HTTP errors.

Our system maintains a timeout routine to terminate the operation if the case 7 or any abnormal delay occurs. In such a case the system reports the possible error(s) and aborts the current operation. When Web pages are being successfully retrieved within the timeout period, they are first saved as files in a temporary directory. The content of each returned page is then analyzed for the other 6 cases listed above. We now discuss each case in turn.

Response Page Contains All Data. This case occurs when the first default page already contains all the data specified within the scope of all possible combinations of form fields. The system determines that all the data might have been retrieved when consecutive returned pages from the sampling queries are either equal to or subsumed by the data on the first default page. Section 5 describes our approach to duplicate record detection. If all records in a response page are duplicates of data previously retrieved, we say the response page is subsumed. If all the sample queries return subsumed response pages, we terminate the query execution and decide that the first, default page did indeed contain all available data. Otherwise we treat the pages as instances of case 3.

Response Page Includes Next Link. Perhaps the most common case is that a Web site will return results a bit 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. This logical page can then be treated as one of cases 1, 3, or 4.

Returned Page Contains Only Part of Site's Data. This case is the common situation that every returned page may be some particular subset of the overall database. In this case, we can assume that even after all the sample queries have been processed, new information would still be available from the database behind the form. As explained previously, the system prompts the user asking whether to enter the exhaustive phase or discontinue the operation after the retrieval of the data from the sample queries. In this case, each response page is added to the repository we are building of retrieved data. Duplicate record elimination is described in Section 5.

Returned Data and Original Form Together. The system looks for a form structure within the response page. If found, the system compares the form in the returned page with the original form to see whether they are the same. If so, the system disregards the form in the returned page so that it will not recursively process the form indefinitely. Data in the response page is filtered for duplicates and added to the repository of results.

One Form Leads to Another. After all the sample queries have been processed, it may be the case that all the response pages contain another form to be filled in, but no data. We discover this case by comparing the structure of a response form with the original form. If they are equal or the original subsumes the response form, there is likely an error, so we treat it as case 6. Otherwise the system prompts the user for further action. One possibility is for the user to choose to process the new form, treating it as a new information source.

No-Record Notification or Required Field Missing. This situation occurs (A) when all the sample queries return no data or (B) required fields needed to be supplied by the user. In case (A), all the response pages would most likely contain the same message, e.g. "No matching records found". In case (B), it is likely that all the response pages would contain the same "required field missing" message. Usually the original form is embedded in the error response page, perhaps with default values changed to reflect the query that failed. At this point our system prompts the user for intervention. The user either aborts the operation or supplies values for certain fields and the system repeats the sampling phase with the new values.

5 Filtering Duplicate Records

As we query a Web site, we gather retrieved records into a repository for later downstream data extraction. To avoid wasting space and to help us know when we have retrieved all the data behind a form, we eliminate duplicate records before placing them in the repository. To do this, we use the copy detection system [20], which detects sentence boundaries and computes hash values for each sentence. The copy detection system is highly effective for finding duplicate sentences over a very large set of textual documents. Instead of sentence boundaries, we wish to detect record boundaries, but it turns out that the two problems are very similar.

Records in response pages are usually displayed as paragraphs separated by the HTML paragraph tag <p>, rows in a table separated by <tr> </tr> tags, or 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 filtering process, this tag is inserted into the retrieved Web documents around certain HTML tags which can most likely be discerned as the end of a record. 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 is invoked. The detection system computes hash values for every record separated by <s.>. These hash values are compared with the hash values of all the records retrieved previously and stored in the repository. If duplicate records are found, they are eliminated. Unique records and their hash values are added to the repository.

6 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 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 preliminary results are encouraging. We have been successful with a number of Web sites (e.g., www.autotrader.com, www.deseretnews.com, www.usedmusicgear.com), and we are now preparing a more comprehensive experiment. We will test 50-100 sites, gathering data on estimated and actual time to process, estimated and actual storage space required, total number of records retrieved, number of duplicate records reported, and manually verified actual number of duplicate records. We also plan to gather aggegrate data indicating the number of sites visited, the number where our approach was successful, and how many required only the sampling phase. For those sampled, we will manually verify whether the result was complete. We will also measure the metrics suggested for HiWE, strict and lenient forms of submission efficiency. These are respectively ratios of successful (SE_{strict}) or valid ($SE_{lenient}$) queries to the total number of queries submitted [19].

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 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. Michael 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. Hasan Davulcu, Juliana Freire, Michael Kifer, and I.V. Ramakrishnan. A layered architecture for querying dynamic Web content. In *SIGMOD '99 Proceedings*, pages 491–502, Philadelphia, PA, May 1999.
4. Robert B. Doorenbos, Oren Etzioni, and Daniel 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, CA, February 1997.
5. Patil systems home page. http://www.patils.com. Describes LiveFORM and ebCARD services. Checked August 10, 2001.
6. eCode.com home page. http://www.eCode.com. Checked August 10, 2001.
7. 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.
8. Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.
9. HTML 4.01 specification. http://www.w3.org/TR/html4, December 1999. Checked August 10, 2001.
10. InvisibleWeb.com home page. http://www.invisibleweb.com. Checked August 10, 2001.
11. Henry Kautz, Bart Selman, and Mehul Shah. The hidden web. *AI Magazine*, 18(2):27–36, Summer 1997.
12. Steve Lawrence and C. Lee Giles. Accessibility of information on the Web. *Nature*, 400:107–109, 1999.
13. Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280:98–100, April 1999.
14. Thomas Leonard. *A Course In Categorical Data Analysis*. Chapman & Hall/CRC, New York, 2000.
15. Robert A. McLean and Virgil L. Anderson. *Applied Factorial and Fractional Designs*. Marcel Dekker, Inc., New York, 1984.
16. Microsoft Passport and Wallet services. http://memberservices.passport.com. Checked August 10, 2001.
17. R.L. Plackett. *The Analysis of Categorical Data,* 2^{nd} *Edition*. Charles Griffin & Company Ltd., London, 1981.
18. Sriram Raghavan and Hector 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.
19. Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden Web. In *VLDB 2001 Proceedings*, Rome, Italy, September 2001. To appear.
20. Randy D. Smith. Copy detection system for digital documents. Master's thesis, Computer Science Department, Brigham Young University, 2000.
21. Ajit C. Tamhane and Dorothy D. Dunlop. *Statistics and Data Analysis: From Elementary to Intermediate*. Prentice-Hall, New Jersey, 2000.
22. Peter Tryfos. *Sampling Methods For Applied Research: Text and Cases*. Wiley, New York, 1996.