Probing the Hidden Web
Research Internship Report

Avin Mittal

July 25, 2007

Abstract

Hidden databases on the web are an important source of information, which are not usually indexed by traditional search engines. Documents on the hidden web are not reachable by following the hyperlinked structure of the graph, and so, the need is being felt for a system which can automatically discover, understand and index hidden web documents. In this article, we present a system which analyzes the structure of HTML forms, annotates the various fields in the form with concepts from the knowledge domain, probes the fields with domain specific queries, and analyzes the response pages to find out whether or not they contain any results, to further confirm the annotations. The system also wraps the given HTML form into a web service described by a WSDL document, whose inputs are labeled by the domain concepts and which can be used to query the form to get response pages.

1 Introduction

The *Hidden Web* is defined as the set of webpages (which may or may not be dynamically generated) not accessible from the hyperlinked structure of the World Wide Web. It usually consists of structured information, mostly in the form of databases protected by query interfaces, and contains huge amounts of well organized and meaningful information. The hidden web is rapidly growing, and is currently estimated to contain at least 500 times [9] the amount of data present on the *Surface Web* (web pages which are reachable by following hyperlinks from other pages, and consequently can be discovered by search engines).

Given the huge volume of information contained in the Hidden Web, it is highly desirable to have a system for integrated querying of web services in order to get hidden web documents, so as to avoid querying each interface independently. For example, a casual user searching for the list of publications in a conference should be able to find it at a single source, rather than having to query every possible search interface on the web. Thus, a system which can automatically discover, understand and index the contents of the hidden web databases automatically is required. Due to the highly dynamic nature of the query interfaces as well as the contents of the hidden database, the system should be highly automatized, and avoid human supervision as much as possible.

The main purpose of the present work is to facilitate access to the hidden web and to answer high-level queries in a precise way using services. For example, let us consider a web user interested in movies by some particular actor. Traditional search engines would return a set of HTML pages giving information about the person himself, rather than the movies. Our goal is a system that would discover sources of information (such as IMDB [6]) ahead of time, and at query time, fill in the forms and get a precise answer. Building such an automatic system poses many challenges, some of which are outlined below:

- Discovering of query interfaces, and distinguishing them from other kinds of HTML forms such as login screens, complaint registration etc.
- Understanding the semantics of the query interfaces, such as the details of the input fields, so that the fields can be mapped to some concept.
- Forming meaningful queries that could be submitted to the form to extract the hidden data in the form of response pages
- Analyzing the response pages, to extract the necessary information from the database, in the form of records.

Many recent works have focused on various different aspects of hidden web analysis. [12] is a system which focuses on analyzing the web forms, automatically generating queries and extracting information from the response pages thus obtained. The authors in [12] propose a system which crawls the hidden web in order to find resources, and then use layout based methods for information extraction from the search forms and response pages. [1] is another example of a system which extracts hidden web data from keyword based interfaces by querying the interface with high coverage keywords. Their aim is to extract all the data from the database, and index it locally. Another system which works on multi-attribute forms is [17] which uses predicate mapping to convert the user given queries to specific form queries, and thus fetch the required response pages hidden behind the form interface. Another interesting work is [18], which focuses on wrapping search interfaces into a web service. The authors in [18] propose a mechanism for automatically extracting the information from the result page based on ViNTs (Visual information And Tag structure). They consider the tag paths present in the result page, and use the fact that the records would have almost identical paths to extract the records from the response page.

Our work focuses mainly on the syntactic and semantic understanding of multi-attribute web forms. Since the general problem of understanding HTML forms and giving a meaning to the input fields is quite difficult and arguably AI-complete, we focus our attention to a particular domain, viz. the domain of *scientific publications* and thus, restrict the number of concepts as well as the knowledge database that we deal with. The next section gives the basic overview of our system, and the consequent sections describe in detail the various modules of the system.

2 System Architecture

The entire system can be roughly divided into various modules as shown in Figure 1. The various modules help in keeping in the system portable, as well as easily maintainable. The different modules perform relatively different tasks, and thus each is capable of being examined independently. The XML database

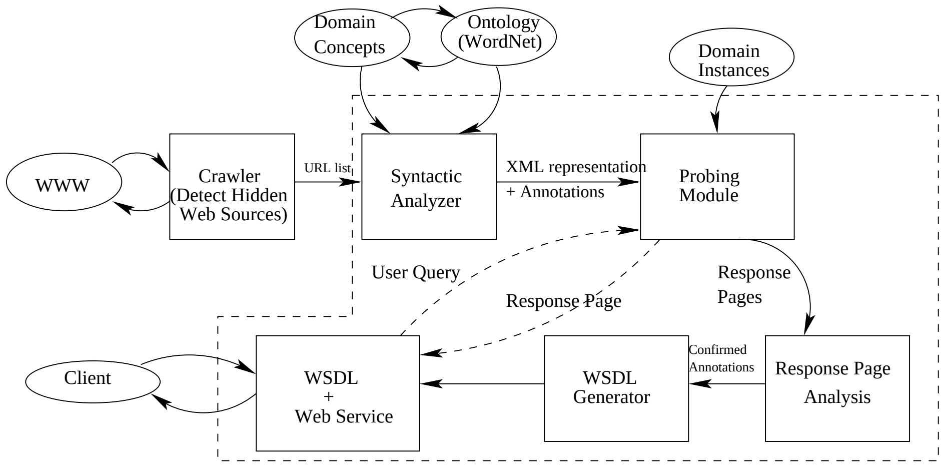

Figure 1: System Overview

that is used to represent the web forms, along with the domain specific knowledge is the glue that flows through the system, and is constantly exchanged by the various modules, thus binding the modules together.

The first step towards analyzing the hidden web is to automatically discover the multi input forms which are used as the query interfaces for hidden databases, through some kind of focused crawling. This step results in a list of URLs which can then be processed by our system.

The URLs are passed on to the "Syntactic Analyzer", which extracts the forms out of the webpage pointed to by the URL, and analyzes the structure of the form. In this step, the textual context for each input field in the form is matched with the concept names from the domain, and other related words (derived from the provided ontology). This step provides an XML representation of the form and its input fields. The XML representation also contains some initial annotations, assigned to the fields based on the concept names the textual context matches.

The form as described by the XML document along with the annotations is passed onto a "Form Probing" module, which probes the fields with instances of the corresponding annotation. In this phase, the form is submitted with different values assigned to the annotated input fields and the response pages are collected for further analysis. Note that, we are assuming that the form contains no compulsory fields and hence, we can submit the form filling the fields one at a time.

The response pages are analyzed by the "Response page analysis" module to classify the response as a result or a no result page. If the form submissions for a field being annotated as some concept yield some result pages, we can confirm the annotation for that field, and on the other hand, if no result pages are obtained indicating that the values were incorrect for that particular field, then the annotation can be discarded.

After getting the confirmed annotations for the input fields in the form, we convert the XML representation of the form to a web service which is described by a WSDL document. The web service has input parameters corresponding to each of the confirmed annotations in the form, which when invoked would submit the form with the field (labeled by the annotation) populated by the provided input value and return the response page that is obtained by submitting the form.

3 Finding Relevant Forms

There are many hidden databases on the web which can be queried to extract domain information. For example, the DBLP database [5] along with many other publication databases contain information about thousands of publications which can only be accessed through a query interface. The first step towards developing a system which can automatically query all such services is to discover and index these interfaces. This step demands some kind of crawling to be done on the web to automatically discover all such possible services.

Given the huge size of the entire web, we need to "focus" the crawl [3, 7], so as to discover such services without needing to index the entire Internet. The focused crawler can be used to build up a list of URLs of multi-input forms related to our target domain [12]. Care needs to be taken so as to filter out the forms with side effects such as booking services or mailing list management interfaces. Another approach could be to query traditional search engines for advanced search forms. For example, querying google for "advanced search publication databases" and following the links from the results obtained yields quite a large number of multi-input forms. Another source of such databases is the open directory [8], which contains an index of all online databases categorized by subjects.

The detection of sources is currently left as an open issue for subsequent works, since it focuses on orthogonal issues that are not relevant to the rest of the work. For our purposes, we prepare a list of URLs for testing the system using web search, and proceed with the rest of the phases taking this list of URLs as the input.

4 Structural Analysis of the HTML forms

After discovering the sources on the web which can be used to extract information from the hidden web, the primary goal is to extract the textual context and analyze the organization of the form elements, and thus identify the domain concept to which each field maps to. For this purpose, the forms contained in the obtained URLs have to be syntactically analyzed to identify the different fields, their names, ids, labels and the set of values which can be accepted by them. The obtained structure and the textual context for the input fields together with the domain concept names and a given ontology (for identifying the relationship between words) can be used to obtain possible annotations for each field in the form.

The LABEL tag in HTML (see Figure 4) is provided for identifying field labels, but is seldom used by most web developers. Thus, finding the correct label for a field may also require analyzing other details related to the field such name, id and also the textual information that is in spatial proximity to the input field when the HTML page is viewed in a web browser.

```
<label for="formID1">Form ID 1</label>
<input type="text" id="formID1" />
```

Figure 2: Example of LABEL tag

4.1 Extracting details about the input fields

For syntactic analysis, the obtained page is fed to a parser which breaks up the current page into its constituent HTML tags (along with all the attributes that might be present along with these tags). Since, we are only interested in the forms on the current page, all the content on the page that is outside the ⟨FORM⟩ and ⟨/FORM⟩ tags is ignored. Also, all the styling tags on the page such as ⟨FONT⟩, ⟨B⟩, ⟨I⟩ etc. are ignored as they are of no particular interest in understanding the structure of the form.

Once the form is identified on the page, all the possible information about the form is extracted from the HTML attributes, which includes action, method, encoding along with any other information that the attributes of the ⟨/FORM⟩ tag in the HTML page can provide. The form information is required at the time of form submission, and hence is required to be extracted at the time of syntactic analysis. Also, the list of all input fields in the form is constructed, along with all the possible information about the fields such as name, type, id, default value, content of an associated label tag, neighbouring text etc.

The possible values for some fields (such as SELECT, RADIO or CHECKBOX) can be explicit, while they can be totally implicit (through the use of JavaScript or server-side validation) for some text fields. For example, a text field labeled date accepts input in only some particular format and not any arbitrary string of characters. However, since the domain that we are dealing with involves search interfaces meant for casual web users, we assume for all further experiments that a text field can accept any string of characters, and that the domains of input for text fields are not curtailed using JavaScript.

4.2 Mapping input fields to domain concepts

After extracting the syntactic details about all the fields in the form, the next step is to map the field to the concepts from the domain knowledge. The domain concepts used for the publications domain are:

- Title
- Author
- Conference
- Journal

The procedure for annotating an input field with a domain concept is as follows:

- *Build a textual context for each field*: For each input field, all the text that is extracted from the label tag, name, id or the neighbouring text is used in getting the textual context for the field. The content of above fields is extracted and broken down into individual words. All the words which are thus extracted for an input field constitute the textual context for the field.

- *Pre-processing of the context*: After getting the context for each field, some pre-processing is done before comparison with the domain concepts:
  - *Stop Words Removal*: The frequently occurring words such as "and", "or", "the" along with digits are removed from the textual context, since they are not expected to be of much importance when it comes to determining the domain concept to which the field maps to.
  - *Stemming*: All the words obtained in the textual context are also subjected to Porter's stemming algorithm [11], in order to remove the differences because of linguistic variations. e.g a field labeled "Authored By" represents the same concept as the field labeled "Author". Such differences are removed by stemming the context terms, so that only the root of all the words remains. e.g. in the above case, both labels would be reduced to "Author" by the Porter's Stemming algorithm, and thus the variations because of linguistic differences are avoided.

  Apart from the above two actions, some other textual processing (removing underscores, removing numbers from the end, etc.) is also performed on the context.

- *Similarity to the concept name*: A list of related words for each concept from the domain is constructed using Wordnet [16]. The related words for each concept name can be derived by the following process:
  - Find the synset to which the concept name belongs
  - Extract all the words from this synset, and add all these to the related words list.
  - Find the connected synsets to the current one, by following relation paths such hyponymy, hypernymy etc.
  - Add the words obtained from the new synsets to the related words list
  - Repeat the last two steps, till no new words are added to the related words list

  After extracting the related words to the concept name, the textual context obtained above is compared to all the words from all the concepts. If any word from the context matches any related word, the corresponding field is assigned the corresponding concept with a confidence score that is obtained based on two factors:
  - *The source of the context word*: The annotation obtained is most reliable if the context is derived from the label tag for the same field. The remaining sources, such as name, id and neighbouring text are much less reliable in comparison to the text in the label tag, and hence the confidence assigned to the annotation is lower if the matching context word is derived from the name, id or the neighbouring text

```
<?xml version='1.0' encoding='utf-8'?>
<webpage url="http://scholar.google.com/advanced_scholar_search">
<form action="/scholar" name="f" method="get">

<input type="text" name="as_q" value="">
<annotation concept="conference" score="0.05">
</annotation>
</input>

... other fields

</form>
</webpage>
```

Figure 3: XML file obtained after structural analysis

- *Distance of the matched word to the concept name*: The confidence in the annotation decreases with the distance of the matched word to the concept name in the synset graph of wordnet.

Each URL is translated to an XML document at the end of this phase (see Figure 4.2) which describes all the details of the form(s) present at this URL. The information contained includes the form data such as the name, action, method, encoding etc. along with the details about the various input fields such as the name, type, default value, possible options(in case of multi-valued fields), and also the possible annotations for this particular field along with the confidence score for each annotation (obtained as described earlier).

5 Form Probing

The annotations obtained at the end of syntactic analysis can be inaccurate since they are derived from unreliable sources such as name of the field, nearby text etc. Also, the textual context may match more than one concept name (e.g. "Author Name" matches both "Author" as well as "Name") and consequently, we may have more than one annotation for some of the input fields. But, since an input field in a form can represent only one concept name, thus only one of the many annotations can be correct. Therefore, in order to confirm the possible annotations that are obtained from the previous phase, we need to submit (probe) the form with some input values, and analyze whether or not a result page was obtained. e.g. if a field is annotated as "Author", then we submit the form, filling in the field with some author names, and analyze the response pages obtained. If the response pages contain some records (or in other words, is a "Result Page"), then that particular annotation can be confirmed. The instances for the various concept names are derived from the domain knowledge (which in our case is the entire DBLP database, downloaded as an XML file).

One of the important aspects of this module is to be able to distinguish between match and no-match pages resulting from the submission of the form.

No-match pages indicate that some required field was not provided, the input value is incorrect for the corresponding field or that there are no records even though the input was correct. The distinction between match and no-match pages can be made using a number of heuristics (size of the no-match page is smaller, number of outgoing links is lesser, presence of keywords like "Error" or "No match", absence of keywords like "Next" or "More") or by comparison with response pages when deliberately absurd queries are formulated [1]. We further discuss the analysis of the response pages in Section 5.3

5.1 Probing Scheme

- *Identify a no-result page* by probing the form with a meaningless string, in order to obtain a no-result page. The no-result page represents the static part of the response page template in most cases, so it can help in extracting the dynamic portion of the result page (the actual data) by removing the common part between the result and no-result pages.

- *Identify the field with the maximum confidence annotation.* The field having the highest confidence annotation is expected to be most accurately annotated.

- *Probe the highest confidence annotation field* with words from the corresponding concept. The words to be chosen for probing are drawn with a probability that is proportional to their frequency of occurrence in the DBLP database. This way of choosing the words helps satisfy the following properties:

  - The probability of obtaining a result page provided the annotation was correct is high
  - At the same time, probing with words from all frequencies gives a chance to fetch all possible kinds of response pages that the service can generate.

  Also note that, in this scheme of probing we are assuming that all the fields in the form are optional, and we can actually probe the fields one at a time. This assumption holds quite well on the actual web, since in most of the query interfaces, all the fields are optional. Detection of compulsory fields requires certain heuristics such as presence of an asterisk (*) sign after the field label or some other change in the appearance of the label. Another way of detecting compulsory fields is to check the JavaScript validation code associated with the form (using software testing or programming language semantics techniques), and analyze it to extract more information about the fields.

- *Analyze the obtained response pages* (see Section 5.3), and if the probing results in some result pages confirm (increase the confidence value of) the annotation.

  The confidence value associated with the annotation is a number between 0 and 1, which is obtained during structural analysis of the form (see Section 4.2). But, the meaning that should be assigned to the confidence values on the real web is not clear at this stage, and is left as an open issue for future research. Currently, we just set the confidence value to 1 if there were any result pages, indicating that the annotation is confirmed.

- Repeat the above procedure for other fields or other annotations of the same field.

Note that, in the above probing scheme more than one annotation for a particular field can get confirmed. If such a case arises, it is highly probable that the field is a keyword search field, and thus would match a large number of domain concepts. The multiple annotation can also a arise due to ambiguity in the domain instance that we probe the field with. For e.g. the keyword "Hall" is present in both titles as well as author names in the DBLP database. A field can be classified as a keyword field by probing it with unannotated concepts and checking whether it yields a result page. Another approach could be to probe with keywords which are specific to the concept, and then perform some kind of a sampling over multiple probing instances to see whether the current field has been annotated with multiple concepts in many cases. So, detecting a field to be keyword search field is a non-trivial issue, and is currently left as a research issue for future works.

5.2 Choosing the number of probes

The number of values with which each field is to be probed is a tunable parameter of the system. There is an obvious trade-off between the time taken (in sending the request and fetching the response page), and the accuracy of response page analysis. Also, some web servers may block the Java agent, if they find that the agent is sending more requests than they can handle. All these factors have to be kept in mind before deciding with how many values to probe each field of the form with. Currently the system probes each field with 12 different values for each annotation drawn from the domain knowledge with a probability that is proportional to the frequency of occurrence in the DBLP database.

5.3 Response Page Analysis

The probing phase results in many response pages along with a no-result page that is obtained by intentionally probing the form with a meaningless string. These pages have to be analyzed in order to find out whether or not there are any results in the response pages, and hence, whether or not the annotation suggested by the syntactic analysis was correct. The analysis of the result pages can be done in purely heuristic ways such as counting the number of hyperlinks in the page as compared to the number of links in the no-result page, size of the page (number of tags in the HTML representation) etc. A more general approach is to cluster the pages to separate the result pages from the no-result page [2]. This is the approach that we use in our system, albeit the construction of the feature vector and the clustering metric used is significantly different. In [2], the authors construct the feature vector for a page by extracting the tags from the HTML code, and using the tf-idf measure, assign the similarity measure based on cosine similarity. In practice (see Section 7), we found out that the tag signature based clustering didn't work very well in comparison to our scheme of clustering based on the terminal paths in the DOM tree.

5.3.1 DOM tree representation

The Document Object Model (DOM) defines the logical structure of the XML and well formed HTML documents, and the way a document is accessed and manipulated. In the DOM, documents have a logical structure which is very much like a tree. DOM trees represent the structure of an HTML document, and can be very useful in distinguishing between different looking pages, while at the same time documents generated from the same template have more or less identical DOM trees.

For example, consider the code in Figure 4 describing two rows of a table in HTML. The code generates the DOM tree as shown in Figure 5.

```
<TABLE><TBODY>
<TR>
<TD>Shady Grove</TD><TD>Aeolian</TD>
</TR>
<TR>
<TD>Over the River, Charlie</TD><TD>Dorian</TD>
</TR>
</TBODY></TABLE>
```

Figure 4: HTML code for describing two rows of a table

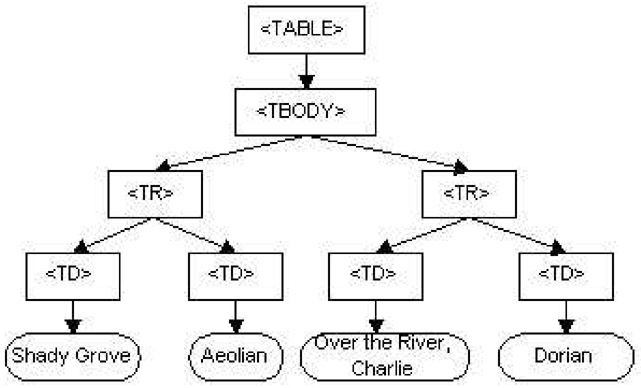

Figure 5: DOM tree for the example table

The DOM tree is used in our clustering algorithm, since it is observed that the response pages for most web services are automatically generated, and hence follow a particular template. As can be seen through the example above, the DOM tree structure remains almost the same if we change the content of the table, and thus for response pages generated using a common template, the DOM trees are expected to be quite similar.

5.3.2 Feature Vector Construction

For each response page, its DOM tree is constructed and the set of all terminal paths is extracted from the tree. Since all the pages are generated by the same service, they are expected to belong to the same template. So, the pages are mapped to the feature space formed by the terminal paths present in the DOM tree. The features are assigned weights based on their tf-idf score in the collection. All the result pages which are structurally similar have more or less identical feature vectors. So, distinguishing between result and no-result pages or even between different kinds of result pages becomes quite convenient, once we represent the response pages in this manner.

For example, the DOM tree shown in Figure 6 has the feature vector: (html-head-meta, html-head-title, html-body-table-tr-td).

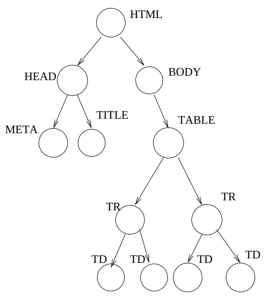

Figure 6: DOM tree

A number of hidden web services are capable of generating more than one kinds of result pages and thus, simple distance calculation from the no-result page is not sufficient for response page classification. For e.g. DBLP has a disambiguating page when it is queried for a popular author name. So, once the feature vector is constructed for each page in the collection, the vectors are subjected to a clustering algorithm based on the proximity as given by the cosine similarity. Since the number of types of response pages is not known a priori, the clustering algorithm used is an incremental clustering algorithm.

Note that, while classifying a page as a result page or a no-result page (see Section 5.3.5), we assume the fact that there is only one kind of result page, and that has been extracted by probing the form with the meaningless string (Section 5.1). In practice, this may not be the case, and the service may actually be capable of producing multiple types of no-result pages, which would be assigned to different clusters through our clustering algorithm. Such case have to be detected through other heuristics (Section 5.3.5).

5.3.3 Clustering Algorithm

Starting from the no-result page as the only cluster, we add one page at a time to our clustering. The cosine similarity of the current page with the centroids of each of the clusters is computed. If the maximum cosine similarity of the new page and the centroid of some existing cluster is greater than a *similarity threshold* (a tunable parameter of the system, currently using 0.9), then the page is assigned to that cluster and the new centroid is computed. In case the maximum similarity is less than the similarity threshold, the page is assigned to a new cluster and hence, itself becomes the centroid of the new cluster.

For example, consider the example shown in Figure 7. The new vector has maximum cosine similarity to the cluster number 3, and this similarity is greater

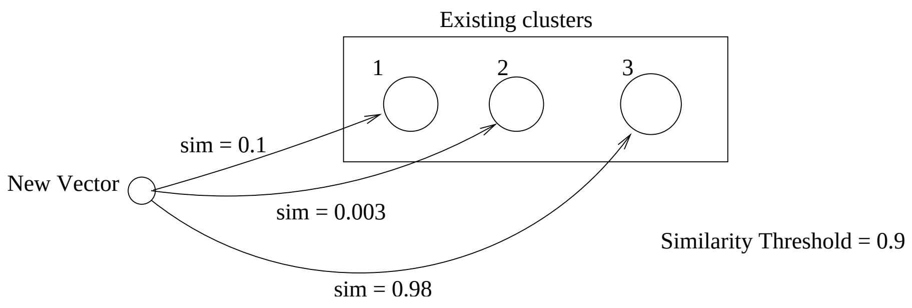

Figure 7: Clustering Example - The new vector is assigned to cluster 3

than the similarity threshold. Consequently, the vector gets added to cluster number 3 and the centroid information is updated. On the other hand, had the maximum similarity been less than the similarity threshold, then the new vector would have started a new cluster and thus, it would have become cluster number 4.

We use an incremental clustering algorithm in order to cluster the response pages. The algorithm is incremental since it considers the feature vectors one at a time and assigns it to the most suitable cluster, or creates a new cluster with the current vector if no suitable cluster can be found. Though this algorithm is of lesser complexity than the orthodox hierarchical agglomerative clustering algorithm, it is known to be suboptimal [4] viz. it can produce clusterings whose diameter is not minimum. But we use this algorithm since the result pages are quite different in practice from the no-result pages and the algorithm works quite well in this case, despite of having just one iteration over the data.

5.3.4 Extracting the Next links

For most of the web forms, the number of results for a given query is much larger than are displayed on a single page. Different web sites may have different approaches towards handling this situation. For example, DBLP [15] curtails the result page to display only the top 100 results while some others may choose to put all the results on a single page. An interesting case arises when the results are segmented into many pages, with links to the next result page from the current page. For example, in the case of Google Scholar [13], the result page contains links to the next, previous and the neighbouring page numbers (see Figure 8). In this case, we extract the "Next" (which can be identified through the anchor text, alt text for an image etc.) link from the result page, and follow the hyperlink in order to get another result page. The presence of such links is another strong indication that the page obtained is a result page.

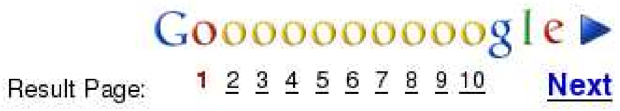

Figure 8: Next links on Google Scholar

5.3.5 Classification of response pages

Several metrics are used to finally classify whether an obtained page contains some results or is a no-result page:

- The page belongs to a different cluster than the one to which the no-result page belongs.
- The size of the page is quite larger than the no-result page.
- The number of outgoing links present in the page is significantly greater than the number of links in the no-result page
- The page contains links with anchor text very similar to "Next", "More" etc. indicating the presence of more results for the current query.

The classification of response pages into a result or a no-result page enables us to modify the confidence score of the annotations. If probing with instances from some concept results in some result pages, it implies that it is more likely that the concept is the correct annotation for the corresponding field, and hence we can increase the confidence of the annotation. Similarly, if there are no result pages for probing with instances with a domain, then it is likely that the concept is not the right annotation for the current field, and the score for the annotation can be decreased.

At the end of the probing module, the modified XML file contains the changed scores for annotations, and the annotations for which there were no results are removed. For each annotation we also keep the centroids of the clusters, in order to classify the pages obtained when we wrap the current form as a web service described by a WSDL document(See Section 6). A snippet of the XML file obtained after this phase is shown in Figure 9.

The feature vectors within the ⟨CLUSTER⟩ and ⟨/CLUSTER⟩ represent the centroids of the clusters obtained. Usually, there would be 2 or more centroids for each annotation, one for the no-result page and other(s) for the response pages that contain some results. More than 2 clusters are formed when the service is capable of returning multiple kinds of result pages (as in the case of the disambiguation page of DBLP [15]). The XML file is now passed on to the wrapping module, which produces a web service based on the annotations which are present in the form, and can be used by the client to query the form that has been to produce the WSDL document.

6 Wrapping the form into a Web Service

The XML file produced by the probing module contains all the confirmed annotations for each field of the form. In order to meet our original goal of integrating all the data of the hidden databases so that it can be queried from a single source, we need to wrap each HTML form analyzed, for querying the form using the domain concept names instead of trying to understand the semantics of the form. The main idea here is to convert the form to an interface, the semantics of which are uniform and can be easily understood by an automatic agent. For example, in a form if we have three fields named "A" , "B", "C" which get annotated as "Title", "Author" and "Date" respectively after the probing phase,

```
<?xml version='1.0' encoding='utf-8'?>
<webpage url="http://scholar.google.com/advanced_scholar_search">
<form action="/scholar" name="f" method="get">

<input type="text" name="as_sauthors" value="">
<annotation concept="author" score="0.6">
<cluster>
{html-body-center-font-a=0.0517, html-head-style=0.0172, ...other features}
</cluster>
<cluster>
{html-body-table-tr-td=0.0091, html-body-center-br=0.0091, ...other features}
</cluster>
</annotation>
</input>

... other input fields

</form>
</webpage>
```

Figure 9: XML file after the probing phase

then the web service generated would have three input parameters viz. "Title", "Author" and "Date". The output of the web service is the same as the output of the HTML form, i.e. the response page.

6.1 XML to WSDL

WSDL (Web Service Description Language) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. The wrapping module of our system wraps the XML file obtained through the probing phase into a web service described by a WSDL document. The input message contains parts corresponding to each confirmed annotation present in the XML file, and the output is the HTML document returned by the HTML form. For example, the WSDL document corresponding to "Google Scholar" is shown in Figure 10.

The confirmed annotations for Google Scholar were "author", "conference" and "journal", and hence the input request has three (optional) parts, corresponding to each of the concepts. The NumberOfPages part of the request indicates the number of pages that should be returned by following the "Next" links on the result pages.

6.2 Web Service described by the WSDL document

The WSDL document is just a means of describing the web service to which the HTML form is wrapped. The service, which is at the backend, and the address of which has to be specified in the "soap:address location=..." field,

```
<wsdl:types>
<xsd:schema targetNamespace="urn:google"><xsd:complexType name="concepts0"><xsd:all>
<xsd:element name="author" minOccurs="0" maxOccurs="1" type="xsd:string"/>
<xsd:element name="journal" minOccurs="0" maxOccurs="1" type="xsd:string"/>
<xsd:element name="conference" minOccurs="0" maxOccurs="1" type="xsd:string"/>
</xsd:all></xsd:complexType></xsd:schema>
</wsdl:types>

<wsdl:message name="MessageIn">
<wsdl:part name="Request" type="tns:concepts0"/>
<wsdl:part name="NumberOfPages" type="xsd:integer"/>
</wsdl:message>

<wsdl:message name="MessageOut">
<wsdl:part name="Webpage" type="xsd:string"/>
</wsdl:message>

<wsdl:portType name="googlePort">...</wsdl:portType>

<wsdl:binding name="googleBinding" type="tns:googlePort">...</wsdl:binding>

<wsdl:service name="googleService">
<wsdl:port name="googlePort" binding="tns:googleBinding">
<soap:address location="..." />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Figure 10: WSDL file generated for Google Scholar

is responsible for understanding the query that is sent by the client and translating it to a query that can be sent to the actual form, in order to get the response page(s). Again, the XML description of the form is used for this purpose. The query that arrives from the client assigns some value(s) to the concept name(s). The service refers to the XML description, and for each concept in the query, extracts the text field in the HTML form which is annotated with the current concept (If there are multiple fields annotated with the same concept, then the one which has the maximum confidence for the current concept is chosen). After translating the query in this way from "*concept* = *value*" format to "$FieldName$ = *value*" format, the form is submitted with these value assignments, and the response page thus obtained is returned to the client.

7 Experimental Results

The system is implemented completely in the Java programming language. The WordNet database [16] for creating the ontology is downloaded and plugged in externally to the system. The ontology is external to the system, and the system only uses a java function call to get the related words. So, it is quite easy to fit any other ontology, and make the system work for any other linguistic model. The domain knowledge is extracted from the DBLP records present in XML format [5].

7.1 Syntactic Analysis and Probing

We tested the syntactic analyzer and the probing module on a list of about 15 URLs and the results obtained are shown in Table 1.

As can be seen from Table 1, the syntactic analysis module performs excellently for well annotated forms such as as DBLP [15], where as the results show a drop in performance when many extraneous fields such as number of results, series, ISBN are present in the form. It is quite expected too, since the syntactic analyzer labels the fields based on the neighbouring text and the names, and if none of these is remotely similar to any concept name (as per the ontology), then the field is unannotated. Also, the number of confirmations is less than the number of fields annotated, since during the probing phase we fill out only those fields which have been annotated in the syntactic analysis phase.

The possible way to annotate unannotated fields at the end of syntactic analysis is to probe the unannotated fields with some instances of each concept, and check if we obtain any results for some concept. Though the method can work if we get a very less fraction of fields annotated, but as found by experiments, in most cases the probing does not yield many concept confirmations for initially unannotated fields. So, we ignore the unannotated fields at present and do not probe them with any values.

7.2 Clustering

7.2.1 Similarity Threshold

The most critical factor in clustering is the similarity threshold, which has to be kept at an optimal value, so as to distinguish between the result and no result pages, while at the same time taking care to see that it is not very high so as to classify two result pages differently due to some minor difference in the page structure. In our experiments we found the similarity threshold of 0.9 to work relatively well, and thus we used this value for all our experiments.

7.2.2 Feature Vector Construction

The feature vector that we use for clustering is the set of terminal paths in the DOM tree corresponding to the document and the weights are assigned based on the tf-idf measure. The DOM tree captures the structure of the document perfectly, and works particularly well for our experiments. The cosine similarities between the result pages from Google Scholar ([13]) are up at around 0.99, where as the similarity between result and no-result page is of the order of 0.01.

To show that DOM tree model is the best choice, we also experimented with the feature vector based simply on the occurrence of HTML tags in the document [2]. We simply consider all the tags that occur in the document, compute the tf-idf score based on the occurrence of tags in the collection and use the cosine similarity between these vectors for clustering. It was found that this approach assigns a rather high degree of similarity between result and no-result pages. The cosine similarity between the result and no-result page for

URL	Text fields	Annotated	Confirmed	Recall	Precision
`www.informatik.uni-trier.de/~ley/db/indices/query.html`	12	8	7	100%	100%
`pubs.er.usgs.gov/usgspubs/index.jsp?view=adv`	5	3	3	100%	100%
`nrelpubs.nrel.gov/Webtop/ws/nich/www/public/SearchForm`	6	5	3	100%	66.6%
`eprints.ecs.soton.ac.uk/perl/search/advanced`	11	6	2	66.6%	100%
`eprints.aktors.org/perl/search/advanced`	9	6	2	66.6&	100%
`eprints.unifi.it/perl/search/advanced`	9	6	2	66.6%	100%
`caltechlib.library.caltech.edu/perl/search/advanced`	9	5	2	100%	100%
`dlist.sir.arizona.edu/perl/search/advanced`	9	6	3	66.6%	66.6%
`www.diva-portal.se/`	7	2	1	33.3%	100%
`eprints.rclis.org/perl/search/advanced`	11	6	2	66.6%	100%
`highwire.stanford.edu/cgi/search`	7	4	3	100%	66.6%
`www.ingentaconnect.com/search/advanced`	5	4	2	66.6%	100%
`eprints.cs.vt.edu/perl/search/advanced`	9	6	2	66.6%	100%
`scholar.google.com/advanced_scholar_search`	8	5	4	100%	50%
`archives.cs.iastate.edu/perl/advsearch`	11	5	2	50%	100%

Table 1: Experimental Results

Google scholar in this model was of the order of 0.5-0.6, which is rather high. The possible reason for this is that the HTML tags are quite abundant, i.e. almost all tags appear in all pages and hence the idf measure is wasted. Also, length normalization leads to the increase in similarity of the vectors between result and no result pages, since the structural details such as paths, ordering of tags etc. are lost.

8 Conclusions/Future Work

The system described in this article provides the basis for understanding and indexing the Hidden Web resources. Understanding the structure of the HTML forms and determining the concepts which are represented by the various fields in the form is crucial to the building of an automated and integrated system, which can query the hidden web resources on the web and fetch the results. Our system uses probing of the query interfaces with the domain instances in order to annotate the text fields in the form with domain concepts. The form is then wrapped into a web service described by a WSDL document, so that any client can access the form and fetch the response pages hidden behind the form without having to understand the form semantics.

The system that we have developed is quite generic, i.e. that it can be used for forms from any other language or any other domain quite easily. The external entities to the system such as domain concepts, domain instances and the ontology are the only factors that decide the application domain and language for it is to be used.

Though the system solves some of the problems associated with the complete understanding of hidden web resources and indexing them, there is a lot of work still to be completed in order to have a system which will automatically fetch, understand, index and query such interfaces in order to provide all the results that a user desires at a single location. Some of the future directions for research are outlined below:

- *Automatic discovery of Hidden Web resources*: Currently the hidden web crawler is not a part of our system. An automatic agent which would crawl the web, following the hyperlinks in order to discover hidden web resources or perhaps querying traditional search engines and analyzing the results, is highly desirable.

- *Detecting compulsory fields*: Currently, the system assumes that all fields in the HTML form are optional and we can probe the fields one at a time. However, for some resources on the web this may not be the case, and we need to look at heuristics and JavaScript analysis in order to detect the presence of compulsory fields.

- *Semantics of the query interface*: The query interface may have more complex semantics than "text beside an input field", e.g. in [14], the label for the text field is decided by the chosen option in the drop-down list beside the text field. Such cases can be detected possibly by the analysis of drop-down fields, and if the options inside the drop-down box are similar to the concept names, we can conclude that the drop-down field serves as the dynamic label for the adjoining text field.

- *Confidence of annotations*: The confidence scores that have been assigned to the annotations have to be given some logical meaning. The scheme for increasing/decreasing the confidences based on the probing results is also not clear yet. Perhaps the confidences can be used in some kind of a ranking scheme among the HTML forms when there is a single source that queries all of them.

- *Extracting results from the response pages*: The result snippets from the response pages have to be extracted, analyzed and indexed in order to have a unified response set when all the sources are queried from a single web service. Much research is already going on this direction (See [10]). Thus, the output of the web service can also be wrapped to provide the results in our own format, rather than returning the HTML pages generated by the forms.

References

[1] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. In Sérgio Lifschitz, editor, *SBBD*, pages 309–321. UnB, 2004.

[2] James Caverlee, Ling Liu, and David Buttler. Probe, cluster, and discover: Focused extraction of qa-pagelets from the deep web. In *ICDE*, pages 103–115. IEEE Computer Society, 2004.

[3] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[4] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SIAM J. Comput.*, 33(6):1417–1440, 2004.

[5] DBLP database. `http://dblp.uni-tier.de/xml/`.

[6] Internet Movie Database. `http://www.imdb.com`.

[7] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused crawling using context graphs. In *VLDB*, pages 527–534. Morgan Kaufmann, 2000.

[8] Open Directory. `http://dmoz.org/Science/Publications/Archives/Free_Access_Online_Archiv%es`.

[9] B. LLC. The deep web: Surfacing hidden value. 2000.

[10] Daniel Muschik. Unsupervised learning of XML tree annotations. Master's thesis, Universite de Technology de Lille and Graz University of Technology, 2007.

[11] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, July 1980.

[12] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 129–138. Morgan Kaufmann, 2001.

[13] Google Scholar. `http://scholar.google.com/advanced_scholar_search`.

[14] Virginia Tech. Library Search. `http://scholar.lib.vt.edu:8765/index.html?ql=a`.

[15] DBLP search interface. `http://www.informatik.uni-trier.de/~ley/db/indices/query.html`.

[16] WordNet. `http://wordnet.princeton.edu/`.

[17] Zhen Zhang, Bin He, and Kevin Chen-Chuan Chang. Light-weight domain-based form assistant: Querying web databases on the fly. In *VLDB*, pages 97–108. ACM, 2005.

[18] Hongkun Zhao, Weiyi Meng, Zonghuan Wu, Vijay Raghavan, and Clement T. Yu. Fully automatic wrapper generation for search engines. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 66–75. ACM, 2005.