Crawling the Hidden Web

Sriram Raghavan, Hector Garcia-Molina
Computer Science Department, Stanford University
Stanford, CA 94305, USA
{rsram, hector}@cs.stanford.edu

Abstract

Current-day crawlers retrieve content only from the publicly indexable Web, i.e., the set of web pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration. In particular, they ignore the tremendous amount of high quality content "hidden" behind search forms, in large searchable electronic databases. In this paper, we provide a framework for addressing the problem of extracting content from this hidden Web. At Stanford, we have built a task-specific hidden Web crawler called the Hidden Web Exposer (HiWE). We describe the architecture of HiWE and present a number of novel techniques that went into its design and implementation. We also present results from experiments we conducted to test and validate our techniques.

Keywords: Crawling, Hidden Web, Content extraction, HTML Forms

1 Introduction

A number of recent studies [4, 19, 20] have noted that a tremendous amount of content on the Web is *dynamic*. This dynamism takes a number of different forms (see Section 2). For instance, web pages can be dynamically generated, i.e., a server-side program creates a page *after* the request for the page is received from a client. Similarly, pages can be dynamic because they include code that executes on the client machine to retrieve content from remote servers (e.g., a page with an embedded applet that retrieves and displays the latest stock information).

Based on studies conducted in 1997, Lawrence and Giles [19] estimated that close to 80% of the content on the Web is dynamically generated, and that this number is continuing to increase. As major software vendors come up with new technologies [2, 17, 26] to make such dynamic page generation simpler and more efficient, this trend is likely to continue.

However, little of this dynamic content is being crawled and indexed. Current-day search and categorization services cover only a portion of the Web called the *publicly indexable Web* [19]. This refers to the set of web pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration.

In this paper, we address the problem of crawling a subset of the currently uncrawled dynamic Web content. In particular, we concentrate on extracting content from the portion of the Web that is hidden behind search forms in large searchable databases (the so called *Hidden Web*[1] [11]). The hidden Web is particularly important, as organizations with large amounts of **high-quality** information (e.g., the Census Bureau, Patents and Trademarks

[1]The term *Deep Web* has been used in reference [4] to refer to the same portion of the Web.

Office, News media companies) are placing their content online. This is typically achieved by building a Web query front-end to the database using standard HTML form elements [12]. As a result, the content from these databases is accessible only through dynamically generated pages, delivered in response to user queries.

Crawling the hidden Web is a very challenging problem for two fundamental reasons. First is the issue of scale; a recent study [4] estimates that the size of the content available through such searchable online databases is about 400 to 500 times larger than the size of the "static Web." As a result, it does not seem to be prudent to attempt *comprehensive* coverage of the hidden Web. Second, access to these databases is provided only through restricted search interfaces, intended for use by humans. Hence, "training" a crawler to use this restricted interface to extract relevant content, is a non-trivial problem.

To address these challenges, we propose a *task-specific, human-assisted* approach to crawling the hidden Web. Specifically, we aim to selectively crawl portions of the hidden Web, extracting content based on the requirements of a particular application, domain, or user profiles. In addition, we provide a framework that allows the human expert to customize and assist the crawler in its activity.

Task-specificity helps us counter the issue of scale. For example, a marketing analyst may be interested in news articles and press releases pertaining to the semiconductor industry. Similarly, a military analyst may be interested in political information about certain countries. The analysts can use existing search services to obtain URLs for sites likely to contain relevant information, and can then instruct the crawler to focus on those sites. In this paper we do not directly address this resource discovery problem per se; see Section 7 for citations to relevant work. Rather, our work addresses the issue of how best to automate content retrieval, given the location of potential sources.

Human-assistance is critical to enable the crawler to submit queries on the hidden Web that are relevant to the application/task. For example, the marketing analyst may provide lists of products and companies that are of interest, so that when the crawler encounters a form requiring that a "company" or a "product" be filled-in, the crawler can automatically fill in many such forms. Of course, the analyst could have filled out the forms manually, but this process would be very laborious. By encoding the analyst's knowledge for the crawler, we can speed up the process dramatically. Furthermore, as we will see, our crawler will be able to "learn" about other potential company and product names as it visits pages, so what the analyst provides is simply an initial seed set.

As the crawler submits forms and collects "hidden pages," it saves them in a repository (together with the queries that generated the pages). The repository also holds static pages crawled in a conventional fashion. An index can then be built on these pages. Searches on this index can now reveal both hidden and static content, at least for the targeted application. The repository can also be used as a cache. This use is especially important in military or intelligence applications, where direct Web access may not be desirable or possible. For instance, during a crisis we may want to hide our interest in a particular set of pages. Similarly, copies of the cache could be placed at sites that have intermittent net access, e.g., a submerged submarine. Thus, an analyst on the submarine could still access important "hidden" pages while access is cut off, without a need to submit queries to the original sources.

At Stanford, we have built a prototype hidden Web crawler called *HiWE* (**Hi**dden **W**eb **E**xposer). Using our experience in designing and implementing HiWE, we make the following contributions in this paper:

- We first present a systematic classification of dynamic content along two dimensions that are most relevant to crawling; the *type of dynamism* and the *generative mechanism*. This helps place our work in the overall context of crawling the Web. (Section 2)

- We propose model of forms and form fill-outs that succinctly captures the actions that the crawler must perform, to successfully extract content. This helps cast the content extraction problem as one of identifying the *domains* of form elements and gathering suitable *values* for these domains. (Section 3)

- We describe the architecture of the HiWE crawler and describe various strategies for building *(domain, list of values)* pairs. We also propose novel techniques to handle the actual mechanics of crawling the hidden Web (such as analyzing forms and deducing the domains of form elements). (Sections 4 and 5)

- Finally, we present proof-of-concept experiments to demonstrate the effectiveness of our approach and techniques. (Section 6)

Note that crawling dynamic pages from a database becomes significantly easier if the site hosting the database is *cooperative*. For instance, a crawler might be used by an organization to gather and index pages and databases on it's local intranet. In this case, the web servers running on the internal network can be configured to recognize requests from the crawler and in response, export the entire database in some predefined format. This approach is already employed by some e-commerce sites, which recognize requests from the crawlers of major search engine companies and in response, export their entire catalog/database for indexing.

In this paper, we consider the more general case of a crawler visiting sites on the public Internet where such cooperation does not exist. The big advantage is that no special agreements with visited sites are required. This advantage is especially important when a "competitor's" or a "unfriendly country's" sites are being studied. Of course, the drawback is that that the crawling process is *inherently imprecise*. That is, an automatic crawler may miss some pages or may fill our some forms incorrectly (as we will discuss). But in many cases, it will be better to index or cache a useful subset of hidden pages, rather than having nothing.

2 Classifying Dynamic Web Content

Before attempting to classify dynamic content, it is important to have a well-defined notion of a dynamic page. We shall adopt the following definition in this paper:

> A page P is said to be dynamic if some or all of its content is generated at run-time (i.e., *after* the request for P is received at the server) by a program executing either on the server or on the client. This is in contrast to a static page P', where the entire content of P' already exists on the server, ready to be transmitted to the client whenever a request is received.

Since our aim is to crawl and index dynamic content, our definition only encompasses dynamism in content, not dynamism in appearance or user interaction. For example, a page with static content, but containing *client-side* scripts and DHTML tags that dynamically modify the appearance and visibility of objects on the page, does not

satisfy our definition. Below, we categorize dynamic Web content along two important dimensions: the type of dynamism, and the mechanism used to implement the dynamism.

2.1 Categorization based on type of dynamism

There are three common reasons for making Web content dynamic: time-sensitive information, user customization, and user input. This in turn, leads to the following three types of dynamism:

Temporal dynamism: A page containing time-sensitive dynamic content exhibits temporal dynamism. For example, a page displaying stock tickers or a list of the latest world news headlines might fall under this category. By definition, requests for a temporally dynamic page at two different points in time may return different content.[2]

Current-day crawlers do crawl temporally dynamic pages. The key issue in crawling such pages is *freshness* [7], i.e., a measure of how up to date the crawled collection is, when compared with the latest content on the web sites. The analyses and crawling strategies presented by Cho et. al. [6, 7], to maximize freshness, are applicable in this context.

Client-based dynamism: A page containing content that is custom generated for a particular client (or user) exhibits client-based dynamism. The most common use of client-based dynamism is for *personalization*. Web sites customize their pages (in terms of look, feel, behavior, and content) to suit a particular user or community of users. This entails generating pages on the fly, using information from client-side cookies or explicit logins, to identify a particular user.

Since pages with client-based dynamism have customized content, crawling such pages may not be useful for applications that target a heterogeneous user population (e.g., a crawler used by a generic Web search engine). However, for certain applications, a *restricted crawler*[3] can be equipped with the necessary cookies or login information (i.e., usernames and passwords) to allow it to crawl a fixed set of sites.

Input dynamism: Pages whose content depends on the input received from the user exhibit input dynamism. The prototypical example of such pages are the responses generated by a web server in response to form submissions. For example, a query on an online searchable database through a form generates one or more pages containing the search results. All these result pages fall under the category of input dynamism. In general, all pages in the *hidden Web* exhibit input dynamism. In this paper, our focus will be on crawling such pages.

Note that many dynamic pages exhibit a combination of the above three classes of dynamism. For instance, the welcome page on the Amazon web site [1] exhibits both client-based (e.g., book recommendations based on the user profile and interests) and temporal dynamism (e.g., latest bestseller list).

In addition, there are other miscellaneous sources of dynamism that do not fall into any of the above categories. For example, tools for web site creation and maintenance [10, 22] often allow the content to be stored on the server

[2]Note that simply modifying the content of a static page on the web server does not constitute temporal dynamism since our definition requires that a dynamic page be generated by a program at run-time.

[3]We use the term restricted crawler to refer to a crawler that limits it's crawling activity to a specific set of sites.

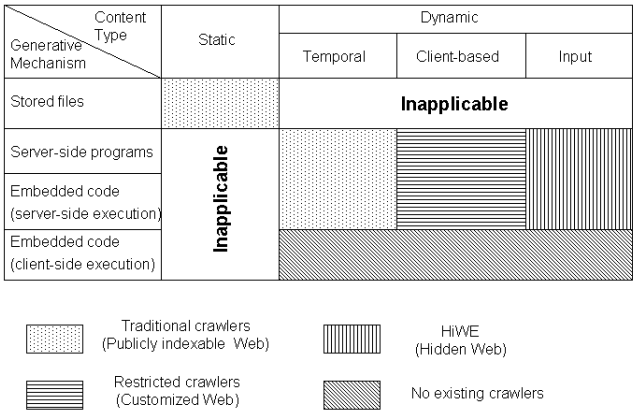

Figure 1: Classifying Web content based on impact on crawlers

in native databases and text files. These tools provide programs to generate HTML-formatted pages at run-time from the raw content, allowing for clean separation between content and presentation. In this scenario, even though pages are dynamically generated, the content is intrinsically static.

2.2 Categorization based on generative mechanism

There are a number of mechanisms and technologies that assist in the creation of dynamic Web content. These mechanisms can be divided into the following three categories:

- *Server-side programs:* In this technique, a program executes on the server to generate a complete HTML page which is then transmitted to the client. This is the oldest and most commonly used method for generating web pages on the fly. A variety of specifications are available (e.g., Common Gateway Interface (CGI), Java servlets [26]) to control the interactions between the web server and the program generating the page. Such server-side programs are most often used to process and generate responses to form submissions (i.e., to implement input dynamism).

- *Embedded code with server-side execution:* In this technique, dynamic web pages on the server contain both static HTML text and embedded code snippets. When a request for this page is received, the code snippets execute on the server and generate output that replaces the actual code in the page. Unlike server-side programs which produce a complete HTML page as output, these code snippets generate only portions of the page. Different scripting languages can be used to implement the code snippets [2, 17, 25].

- *Embedded code with client-side execution:* As in the previous case, web pages contain both HTML text and embedded code (or references to wherever the code is available). However, the code is now downloaded and executed on the client machine, typically in a controlled environment provided by the browser. Java applets and ActiveX controls are examples of technologies that support this mechanism.

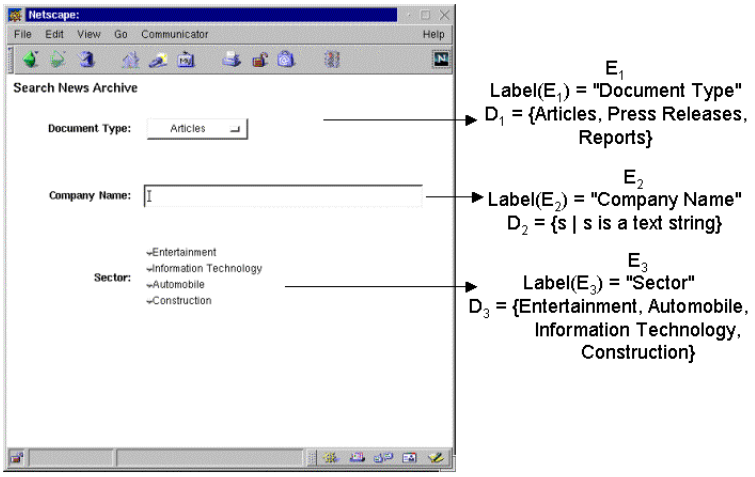

Figure 2: Sample labeled form

Pages that employ server-side programs or embedded code with server-side execution do not pose any special challenges to a crawler, once the page has been received. In both cases, the crawler merely receives HTML pages that it can process in the same way that it processes static content. However, pages that use client-side execution to pull in content from the server, require special environments (e.g., a Java virtual machine) in which to execute the embedded code. Equipping a crawler with the necessary environment(s) greatly complicates it's design and implementation. Since pages in the hidden Web are usually generated using the first two techniques, we do not address the third technique any further in this paper.

Figure 1 summarizes the classification that we have presented in this section. The vertical axis lists the different generative mechanisms and the horizontal axis, the different types of content. Various portions of the Web (and their corresponding crawlers) have been represented as regions in this 2-dimensional grid.

3 Modeling Forms and Form Submissions

The fundamental difference between the actions of a hidden Web crawler, such as HiWE, and a traditional crawler is in the way they treat pages containing forms. In this section, we describe how we model forms and form submissions. Later sections will describe how HiWE uses this model to extract hidden content.

3.1 Modeling Forms

A form, F, is modeled as a set of $(element, domain)$ pairs; $F = \{(E_1, D_1), (E_2, D_2), \ldots (E_n, D_n)\}$ where the E_i's are the elements and the D_i's are the domains. A form element can be any one of the standard input objects: selection lists, text boxes, text areas, checkboxes, or radio buttons.[4] The domain of an element is the set of values which can be associated with the corresponding form element. Some elements have *finite domains*, where the set of valid values are already embedded in the page. For example, if E_j is a selection list (indicated by the <SELECT> HTML element), then D_j is the corresponding set of values that are contained in the list. Other elements such as

[4]Note that submit and reset buttons are not included, as they are only used to manipulate forms, not provide input.

```
<H3>Search News Archive</H3>
<FORM method="POST"  action="http://my.webserver.com/cgi-bin/form-process.pl">
<TABLE>
  <TR>
    <TD align="right" width="150"><B>Document Type:  </B></TD>
    <TD><SELECT NAME=what>
          <OPTION VALUE=art SELECTED>Articles
          <OPTION VALUE=rel>Press Releases
          <OPTION VALUE=rep>Reports
       </SELECT>
    </TD>
  </TR>
  <TR><TD><BR><BR></TD></TR>
  <TR>
    <TD align="right"><B>Company Name:  </B></TD>
    <TD><INPUT NAME=name size=45 maxlength=200 VALUE=""></TD>
  </TR>
  <TR><TD><BR><BR></TD></TR>
  <TR>
    <TD align="right"><B>Sector:  </B></TD>
    <TD>
        <INPUT TYPE="radio" NAME="sector" VALUE="ent">Entertainment<BR>
        <INPUT TYPE="radio" NAME="sector" VALUE="it">Information Technology<BR>
        <INPUT TYPE="radio" NAME="sector" VALUE="au">Automobile<BR>
        <INPUT TYPE="radio" NAME="sector" VALUE="constr">Construction<BR>
    </TD>
  </TR>
</TABLE>
</FORM>
```

Figure 3: HTML markup for the sample form of Figure 2

text boxes have *infinite domains* (e.g., set of all text strings) from which their values can be chosen. In addition, many form elements are usually associated with some descriptive text to help the user understand the semantics of the element. We shall refer to such descriptive information as *labels*, and shall use $label(E_i)$ to denote the label associated with the i^{th} form element. Figure 2 shows a form with three elements and the corresponding representation using our notation. Figure 3 is the piece of HTML markup that was used to generate this form.

We wish to emphasize that our notion of labels and domain values is quite distinct from the internal labels[5] and values[6] used within the form. For instance, referring to Figures 2 and 3, note that $label(E_1)$ is "Document Type", and not the internal label ("what") of the <SELECT> element. Similarly, D_1 is the set $\{Articles, PressArticles, Reports\}$ and not the set of internal identifiers $\{art, rel, rep\}$. These internal identifiers are used only during form submission. They are not visible when the form is displayed. As such, they are not meant for human consumption and are often very cryptic with very little indication of their true semantic meaning.

3.2 Modeling Value Sets

A user "fills out" a form by associating a value or piece of text with each element of the form. A crawler must perform a similar *value assignment* by selecting suitable values from the domain of each form element. The choice of a "suitable value" is dependent on the semantics of the form element, as well as on the application/task being performed by the crawler. For elements with small finite domains, one can potentially try one value after another exhaustively. For example, since domain D_1 in Figure 2 has only three elements, the crawler can first retrieve all relevant articles, then all relevant press releases, and finally all relevant reports. However, for infinite domain elements, the crawler must decide what values from the domain would be semantically meaningful and relevant to the particular application. For example, to fill out element E_2 in Figure 2, the crawler must somehow have access to a list of company names.

In general, we assume that each application/task requires the crawler to have access to a finite set of concepts or

[5]Specified using the NAME attribute of the <INPUT> or <SELECT> elements

[6]Specified using the <VALUE> attribute of the <INPUT> element

categories, with their associated values. In Section 5.5, we describe various data sources (including humans) from which the crawler can obtain such values. In addition, we also show how the crawler can use its own crawling experience to add to these lists of values. However, not all data sources are equally *reliable*. For instance, the crawler has more confidence in the usefulness of human-supplied values than it has, in the values it gathers based on it's crawling experience.

To model values and their confidence, inputs from all these sources are organized in a table called the *Label Value Set (LVS)* table. Each entry (or row) in the LVS table is of the form (L, V), where L is a label and V = $\{v_1, v_2, \ldots v_n\}$ is a *fuzzy/graded set* [14] of values belonging to that label. Fuzzy set V has an associated *membership function* M_V that assigns weights/grades, in the range $[0, 1]$, to each member of the set. Intuitively, each v_i represents a value that could potentially be assigned to an element E if $label(E)$ "matches" L. $M_V(v_i)$ represents the crawler's estimate of how useful/correct, the assignment of v_i to E, is likely to be.

The LVS table also supports the notion of *label aliasing*, i.e., two or more labels are allowed to share the same fuzzy value set. This helps us handle aliases and synonyms representing the same concept (e.g., "Company" and "Organization").

3.3 Generating Value Assignments

Given a form F = $\{(E_1, D_1), (E_2, D_2), \ldots (E_n, D_n)\}$, the crawler generates value assignments by textually matching element labels with labels in the LVS table. Specifically, for each E_i, we generate a fuzzy set of values V_i as follows:

- If E_i is an infinite domain element and (L, V) denotes the LVS entry whose label L most closely matches $label(E_i)$ (see Section 5.3 for details),[7] then V_i = V and M_{V_i} = M_V.
- If E_i is a finite domain element, then V_i = D_i and $M_{V_i}(x)$ = 1, $\forall\, x \,\epsilon\, V_i$.

Then, $ValAssign(F, LVS)$ = $V_1 \times V_2 \times \ldots V_n$ denotes the set of all possible value assignments for form F, given the current content of the LVS table.

Let S_{max} denote the maximum number of times a crawler is allowed to submit a given form.[8] By imposing an upper bound on the number of submissions per form, we ensure that the crawler does not spend all it's time at a single form and instead, extracts the most relevant content from all the databases that it visits. In particular, the crawler chooses the "best" $\min\{S_{max}, |V_1| \times \ldots |V_n|\}$ value assignments to generate form submissions. The notion of the "best" value assignment is based on ranking all the value assignments in $ValAssign(F, LVS)$. We experimented with three different ranking functions (below, ρ represents the ranking function and $\{E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n\}$ denotes a value assignment that associates value $v_i \epsilon V_i$ with element E_i):

Fuzzy Conjunction: The rank of a value assignment is the minimum of the weight/grade of each of the constituent values. This is equivalent to treating the value assignment as a standard Boolean conjunction of the individual fuzzy sets [14].

[7] With approximate matching, each element label could be matched with a set of labels in the LVS table. For now, we assume that only the "best" match is used.

[8] This value is a parameter that we pass to the crawler at startup.

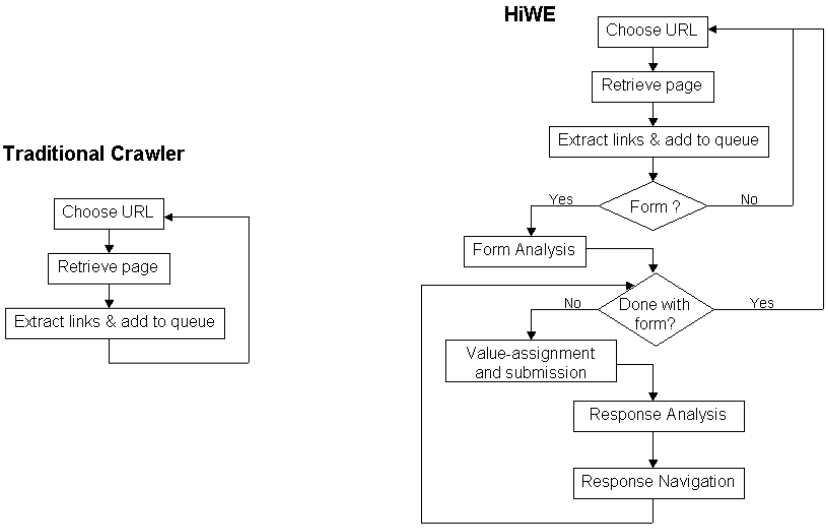

Figure 4: Comparing the basic execution loop of a traditional crawler and HiWE

$$\rho_{fuz}(\{E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n\}) = \min_{i=1\ldots n} M_{V_i}(v_i)$$

Average: The rank of a value assignment is the average of the weights of the constituent values.

$$\rho_{avg}\{E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n\}) = \frac{1}{n} \sum_{i=1\ldots n} M_{V_i}(v_i)$$

Probabilistic: This ranking function treats weights as probabilities. Hence $M_{V_i}(v_i)$ is the likelihood that the choice of v_i is useful and $1 - M_{V_i}(v_i)$ is the likelihood that it is not. Then, the likelihood of a value assignment being useful is computed as:

$$\rho_{prob}(\{E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n\}) = 1 - \prod_{i=1\ldots n} (1 - M_{V_i}(v_i))$$

Note that ρ_{fuz} is very conservative in assigning ranks. It assigns a high rank for a value assignment only if each individual weight is high. The average is less conservative, always assigning a rank which is at least as great as the rank of the fuzzy conjunction for the same value assignment. In contrast, the ρ_{prob} is more aggressive and assigns a low rank to a value assignment only if all individual weights are very low. Section 6 presents more detailed experiments comparing these ranking functions.

4 HiWE: Hidden Web Exposer

The basic actions of a hidden Web crawler, such as HiWE, are similar to those of other traditional crawlers [5, 8]. In Figure 4, the flowchart on the left indicates the typical crawler loop, consisting of URL selection, page retrieval, and page processing to extract links. Note that traditional crawlers do not distinguish between pages with and

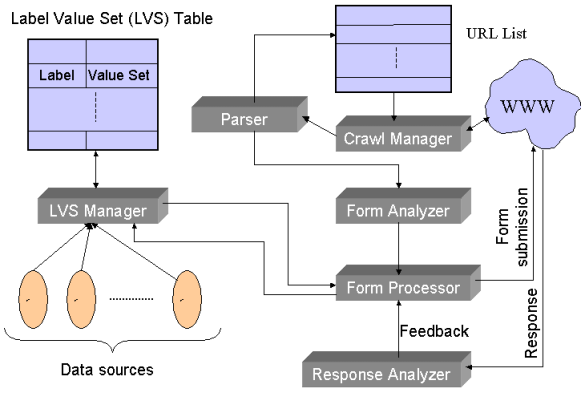

Figure 5: HiWE Architecture

without forms. However, as shown in the flowchart on the right, HiWE's execution sequence contains additional steps for pages on which forms are detected. Specifically, HiWE performs the following sequence of actions for each form on a page:

1. *Form Analysis:* Parse and process the form to build an internal representation, based on the model outlined in Section 3.1.

2. *Value assignment and submission:* Generate the best (untried) value assignment and submit a completed from using that assignment.

3. *Response Analysis:* Analyze the response page to check if the submission yielded valid search results or if there were no matches. This feedback could be used to tune the value assignments in step 2.

4. *Response Navigation:* If the response page contains hypertext links, these are followed immediately (except for links that have already been visited or added to the queue) and recursively, to some pre-specified depth. Note that we could as well have added the links in the response page to the URL queue. However, for ease of implementation, in HiWE, we chose to navigate the response pages immediately, and that too, only upto a depth of 1.

Steps 2, 3, and 4 are executed repeatedly, using different value assignments during each iteration. The sequence of value assignments are generated using the model described in Section 3.3.

4.1 HiWE Architecture

Figure 5 illustrates the complete architecture of the HiWE crawler. It includes six basic functional modules and two internal crawler data structures. The basic crawler data structure is the *URL List*. It contains all the URLs that the crawler has discovered so far. When starting up the crawler, the *URL List* is initialized to a seed set of URLs.

The *Crawl Manager* controls the entire crawling process. It decides which link to visit next, and makes the network connection to retrieve the page from the Web. In our implementation, the crawler was configured to stay

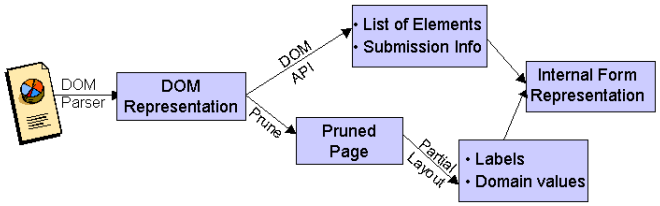

Figure 6: Actions of the Form Analysis Module

within a pre-determined set of target sites (provided to the Crawl Manager at startup), not following links that pointed to other sites.

The Crawl Manager hands the downloaded page over to the *Parser* module. In turn, the Parser extracts hypertext links from the page and adds them to the URL List structure. This sequence of operations is repeated until some termination condition (typically, after some number of hours have elapsed) is satisfied. We refer the reader to existing crawling literature [6, 8] for more details on the design of the Crawl Manager module.

To process forms and extract hidden content, HiWE employs four additional modules and the *LVS Table*. The *Form Analyzer*, *Form Processor*, and *Response Analyzer* modules, together implement the iterative sequence of steps outlined in the previous section.[9]

The *LVS Manager* is responsible for managing additions and accesses to the LVS table. It provides an interface for various application-specific data sources to supply new entries to the table. We shall discuss how this happens in Section 5.5.

5 Design Issues and Techniques

5.1 Form Analysis

HiWE's representation of a form includes the following information:

- A list of all the elements (e.g., selection lists, text boxes) in the form
- A label for each element
- For every element with a finite domain, a list of all the values that constitute that domain
- Submission information (such as the submission method (GET or POST) and the submission URL) to be used when submitting completed forms

To collect all this information, the Form Analyzer executes the sequence of steps indicated in Figure 6. It begins by constructing a logical tree representation of the structure of the HTML page, based on the Document Object Model (DOM) specification [9]. Next, it uses the DOM API to obtain the list of form elements as well as the necessary submission information. We refer the reader to the DOM specification [9] for details on how this can

[9]The Form Processor is responsible for the Response Navigation step.

be done. Then, the Form Analyzer uses the technique described in the remainder of this section to extract labels and domain values. Finally, it normalizes the extracted information (see Section 5.2) and integrates it with the information from the DOM API to produce the internal form representation.

Label and Domain value extraction: Accurately extracting labels and domain values proves to be a hard problem, since their nesting relationships with respect to the form elements is not fixed. For instance, in Figure 3, as is commonly the case, the entire form is laid out within a table. The pieces of text representing the labels (e.g., the word "Sector"), the domain values (e.g., the word "Automobile"), and the form control elements (e.g., the <INPUT> and <SELECT> elements) are interleaved arbitrarily with the tags used in the table markup. In this particular example, the layout is such that each label occurs in the first column and the actual form element widget appears in the second column of the table. However, for different forms, the nature and type of the layout markup will be different. In some cases, instead of tables, explicit spaces and line breaks may be used to control the alignment of labels and form widgets. As a result, the structural representation based on the DOM does not directly yield the labels and domain values.[10]

To address this problem, we adopted a *partial page layout* technique. The key to this technique is to realize that the only restriction on the relative locations of the labels, domain values, and form elements, is that when rendered by the browser, the relationships between these various entities must be obvious to the user. In other words, irrespective of how they are formatted, the phrase "Company Name" in Figure 2 must be **visually adjacent** to the textbox widget. Similarly, the word "Automobile" must be visually adjacent to the corresponding radio button widget.

Thus, we first lay out the form and it's associated labels, similar to the way a browser would lay out the page prior to physical rendering. Then, we use the following heuristic for identifying the label of a given form element (an analogous heuristic is used for domain values):

- Identify the pieces of text, if any, that are visually adjacent to the form element, in the horizontal and vertical directions. For this, based on the layout, we compute actual pixel distances between the centers of the form widgets and the centers of the text pieces. This step yields a list of at most four possible candidates.[11]
- If there are candidates to the left and/or above the element, then the candidates to the right and below are dropped.[12]
- If there are still two candidates remaining, ties are broken in favor of the one rendered in bold or using a larger font size.
- If the tie is still not resolved, then one of the two candidates is picked at random.

[10] Note that for selection lists alone, extracting domain values is straightforward, since these values have to be directly nested within the <SELECT> element.

[11] Note that text pieces containing more than a few words (6 in our default crawler configuration) are ignored, as most labels are either short words or short phrases.

[12] We observed that most forms place labels either to the left or above the form widget.

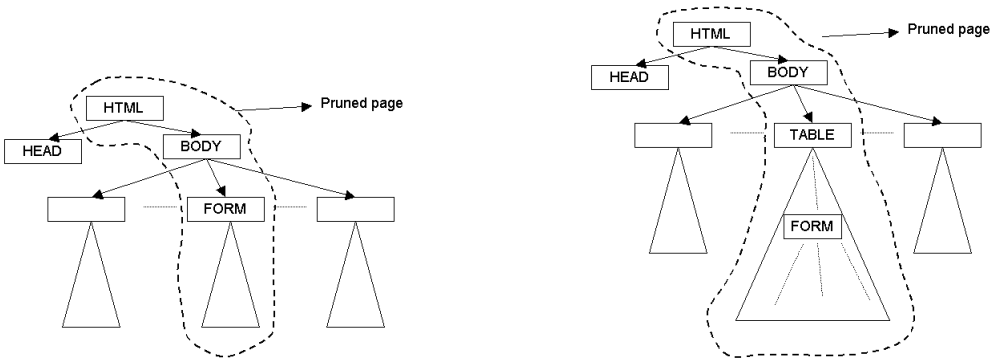

Figure 7: Pruning before partial layout

Note that to calculate visual adjacency, it is not necessary to completely layout the page. The location of the form with respect to the rest of the page is not relevant. Hence, we first prune the original page and isolate only those elements that directly influence the layout of the form elements and the labels. For instance, consider Figure 7, which shows the tree-structured representation of two different pages, one in which the FORM is directly embedded in the main body and another in which it is embedded within a table. The pruned tree is constructed by using only the subtree below the FORM element and the nodes on the path from the FORM to the root. In addition, the layout need not be 'perfect'; in fact, our implementation uses a simple custom layout engine that discards images, ignores font sizes, uses a default font, ignores styling information such as bold or italics (except to break up ties as mentioned above), and ignores any associated style sheets.

Our experiments in Section 6 indicate that visual adjacency is a very robust and effective strategy for extracting labels and domain values. Incidentally, in [18] we study and evaluate other techniques for matching labels to input elements. The techniques of [18] were developed in a different context than ours, for displaying forms on small hand-held devices.

5.2 Normalization

When generating a value assignment, pieces of text (i.e., the labels and domain values) extracted from a HTML page must be matched with other pieces of text stored in the LVS table. To ensure that spurious differences do not result in missed matches, all these text pieces are subjected to a normalization process. The Form Analyzer normalizes the extracted labels and values whereas the LVS manager normalizes entries in the LVS table. Normalization consists of the following sequence of steps:

- To counter possible errors during extraction, the extracted pieces of text are searched for HTML tags and HTML entity references. Any such tags and entity references are removed.
- Next, all characters other than alphanumeric characters are replaced by a space character.
- Uppercase characters, if any, are converted to their lower case equivalents.
- Stop words [13], if any, are removed.

- Finally, each word in the resulting text is stemmed, using the standard Potter suffix-stripping algorithm [13].

5.3 Form Processing

There are two main issues in the design of the *Form Processor* module in Figure 5: choosing an algorithm for matching element labels with labels in the LVS table, and deciding whether or not a form must be processed.

Matching labels: Recall that once a label is found in a form, we must obtain "reasonable" values to fill-in the corresponding input element, so that we can submit a completed form. For example, if we find a label "Enter State," we want to search our LVS table for some domain whose name is 'similar,' e.g., "State." Once we find a good domain in the LVS table, we can use the values associated with it (e.g., "Arizona", "California", etc.) to fill-in the element labeled "Enter State."

To match form labels to labels in the LVS table, we employ an approximate string matching algorithm. There is a large body of work in the design and analysis of such string matching algorithms. Our choice was based on the ability of the algorithm to account for two things: typing errors and word reorderings. Typing errors can be captured by the standard string matching notion of *edit distance*, which measures the minimum number of insertions, deletions, and character replacements required to transform one string to another (e.g., edit-distance(house, hose) = 1). However, word reorderings requires a new distance measure so that two labels such as "Company Type" and "Type of Company" (these become "company type" and "type company" after normalization) are identified as being very 'close' to each other.

The *block edit models* proposed in [21], succinctly represent both typing errors and word reorderings. These models define the concept of *block edit distance*; a generalization of the traditional notion of edit distances to handle block/word movements. We used one of a family of algorithms from [21] to implement our label matching system based on the block edit model.

We match a form element E to an LVS entry by minimizing the block edit distance between their labels, subject to a threshold. Specifically, let $ed_b(A, B)$ denote the block edit distance between strings A and B; let σ be a threshold block edit distance beyond which matches are discarded; and let $dist_{min} = \min_{(L,V)\,\epsilon\,LVS}\{ed_b(label(E), L)\}$ represent the minimum block edit distance. Then, $Match(E)$, the matching LVS entry is computed as follows:

$$\begin{aligned} Match(E) &= \text{nil, if } dist_{min} > \sigma \\ &= (L', V') \text{ such that } ed_b(label(E), L') = dist_{min}, \text{ otherwise.} \end{aligned}$$

Ignoring forms: As the crawler processes one page after another, it is likely to encounter forms which are not directly relevant to the task of extracting content from databases (e.g., a form for local site search). In addition, if the crawler's LVS table does not contain matching labels, even relevant forms may have to be ignored. HiWE uses the following policy to decide whether a form must be ignored or submitted:

A form $F = \{(E_1, D_1), (E_2, D_2), \ldots (E_n, D_n)\}$ is submitted iff

(i) $n \geq \alpha$, and

(ii) for each i such that D_i is infinite, $Match(E_i)$ is non-nil (i.e., there is a matching LVS entry).

Here, α is a configurable parameter that represents the smallest form size that the crawler will process. For instance, if $\alpha > 1$, the crawler will ignore single element forms (e.g., a form with just a simple search box).

Note that we ignore a form if we are unable to associate a matching LVS entry for *every* infinite domain element in the form. However, forms may not always require all inputs to be provided. For example, a form that searches a 'book catalog' may allow the user to enter either an author, a title, or both. We do not consider such partial form inputs in our model.

5.4 Response Analysis

The aim of response analysis is to automatically distinguish between a response page[13] that contains search results and one that contains an error message, reporting that no matches were found for the submitted query. The idea is to use this information to tune the crawler's value assignment strategy.

Response analysis turns out to be a very challenging problem, for a number of reasons:

- The absence of a 'standard' or commonly accepted format for reporting such errors means that each web site is free to use a custom error notification message (e.g., "No matching results," "0 results found," etc.).
- Error pages often include a variety of other textual content (such as site maps, titles, headers, footers, code snippets, etc.) besides the actual error message. Therefore, even if the text of the error message were known, simply searching the page for matching text could lead to false drops.
- Finally, many forms are often associated with multiple types of error messages, with the web server choosing between them based on some pre-programmed logic.

To tackle these challenges, HiWE's response module uses a technique based on identifying the 'significant portion' of the response page (i.e., the portion of the page obtained after discarding headers, footers, side bars, site maps, menus, etc.) To identify the significant portion, we use the heuristic that when the page is laid out, the significant portion will be visually in the middle of the page. Two cases arise:

- If the response page is formatted using frames, we use information about the frame sizes to layout the page and identify the center-most frame. Then, we retrieve the center-most frame's contents, and treat that content as the significant portion of the response page.
- If the response page does not use frames, we use our custom layout engine (Section 5.1) to first identify the HTML element E that is visually laid out at the center of the page. We also parse the page to construct it's DOM representation and locate E in the DOM tree. If E is present in the subtree of a <TABLE> element, we treat the entire table and its contents as the significant portion of the page. If E is not present within a table, then we treat the entire page content as being significant.

[13] Response page is the page received in response to a form submission.

Using the significant portion of the response page, HiWE uses two techniques to identify error pages. The first technique searches the significant portion of the page for occurrences of any one of a pre-defined list of error messages (e.g., "No results", "No matches", etc.). The second technique is based on hashing the contents of the significant portion. After each form submission, the response analysis module computes the hash of the significant portion and maintains a list of such hashes for each form. If a particular hash value occurs very often (i.e., more than a specified threshold), we assume that all future response pages which generate the same hash value are error pages. Initial experiments using these techniques indicates that the response analysis module is reasonably successful in distinguishing between pages with search results and pages with error messages.

Tuning value assignment: We are currently investigating possible approaches for modifying the value assignment strategy of the crawler at run-time, based on feedback from the response analysis module. Specifically, if the response analysis module indicates that an error page was received in response to a particular value assignment, the crawler attempts to isolate the particular form element(s) whose input(s) were incorrect and therefore led to the error.

5.5 Populating the LVS Table

The HiWE crawler supports four mechanisms for populating the LVS table:

- *Explicit Initialization:* HiWE can be supplied with labels and associated value sets at startup time. These are loaded into the LVS table during crawler initialization. Explicit initialization is particularly useful to equip the crawler with values for the labels that the crawler is most likely to encounter. For example, when configuring HiWE for the 'semiconductor news' task described in Section 1, we supplied HiWE with a list of relevant company names and associated that list with labels such as "Company", "Company Name", "Organization", etc.

- *Built-in categories:* HiWE has built-in entries in the LVS table for certain commonly used categories, such as dates, times, names of months, days of the week, etc., which are likely to be useful across applications.

- *Wrapped data sources:* The LVS Manager (Figure 5) can communicate and receive entries for the LVS table by querying various data sources through a well-defined interface.[14] This interface includes two kinds of queries, one or both of which can be supported by a give data source:

  - **Type 1:** Given a label, return a fuzzy value set that can be associated with that label.
  - **Type 2:** Given a value, return other values that belong to the same value set.

  In Section 5.5.1, we describe how we built a wrapper program to use the online Yahoo directory [27] as a data source for the LVS table.

- *Crawling experience:* Form elements with finite domains are a useful source of $(label, value)$ pairs. When processing a form, the crawler can glean such pairs from a finite domain element and add them to the LVS

[14] If necessary, the data sources must be wrapped by programs to export this interface.

1	Set of sites to crawl
2	Explicit initialization entries for the LVS table
3	Set of data sources, wrapped if necessary
4	Label matching threshold (σ)
5	Maximum number of submissions per form (S_{max})
6	Minimum form size (α)
7	Value assignment ranking function to be used

Table 1: Configuring a crawler

table, so that they may be used when visiting a different form.[15] This is particularly useful if the same label is associated with a finite domain element in one form and with an infinite domain element in another. For example, we noticed that when experimenting with the crawling task described in Section 6, some forms contained a pre-defined set of subject categories (as a select list) dealing with semiconductor technology. Other forms had a text box with the label "Categories", expecting the user to come up with the category names on their own. By using the above technique, the crawler was able to use values from the first set of forms to more effectively fill out the second set of forms.

5.5.1 Directories and topic hierarchies as data sources

We discovered that online categorization services, such as the Yahoo directory [27] and the Open Directory Project [24], which structure their information as directories/topic hierarchies, could be effectively used as data sources to populate the LVS table. Specifically, these directories were very useful in expanding value sets, given one or more examples of values belonging to that set.

For example, consider a row in the LVS entry with a value set {"California", "Nevada", "Texas", "Utah"}. When the LVS manager presents this value set to the wrapper program associated with the Yahoo data source, the wrapper submits four separate search queries using each of the values in the set. For each query, the Yahoo directory returns lists of categories (in its hierarchy) pertaining to the query. The wrapper constructs the intersection of the four category lists and identifies *Regional::US States* as the name of the Yahoo category common to all four values.[16] Finally, the wrapper retrieves the list of all the entries that Yahoo lists under that particular category, which in this case, turns out to be a list of US states. If the list of entries in a category turns out to be too large, the wrapper returns just the top 50 entries.

Thus, starting with a small set of example values, the crawler, in conjunction with the wrapper, is able to use an existing topic hierarchy to expand the value set.

5.5.2 Integrating new values into the LVS table

Since value sets are modeled as fuzzy sets (Section 3.2), whenever a new value is added to the LVS table, it must be assigned a suitable weight. Typically, values obtained through explicit initialization and in-built categories have

[15] Note that we can even use $(label, value)$ pairs extracted from forms that are processed but not submitted (because they failed to satisfy the criteria listed in Section 5.3).

[16] If multiple categories result after the intersection, the wrapper chooses one randomly.

a weight of 1, representing maximum confidence (since these values are directly supplied by a human). Weights for values received from data sources are computed by the corresponding wrapper. However, the most interesting case is when computing weights for values gathered by the crawler.

Suppose a crawler encounters a form element E with an associated finite domain $D = \{v_1, \ldots, v_n\}$. Even though D is a (crisp) set, it can be treated as a fuzzy set with membership function M_D, such that $M_D(x) = 1$ if $x \epsilon \{v_1, \ldots, v_n\}$, and $M_D(x) = 0$, otherwise. The following cases arise, when incorporating D into the LVS table:

- *Case 1:* If the crawler successfully extracts $label(E)$ and computes $Match(E) = (L, V)$, we replace the (L, V) entry in the LVS table by the entry (L, V'), where $V' = V \cup D$. Here, $\cup$ is the standard fuzzy set union operator [14] which defines membership function $M_{V'}$ as $M_{V'}(x) = \max(M_V(x), M_D(x))$. Intuitively, D not only provides new elements to the value set but also 'boosts' the weights/confidence of existing elements.

- *Case 2:* If the crawler successfully extracts $label(E)$ but $Match(E)$ is nil, a new row/entry ($label(E)$, D) is created in the LVS table, with membership function M_D defined by $M_D(x) = 1$ if $x \epsilon \{v_1, \ldots, v_n\}$, and $M_D(x) = 0$, otherwise

- *Case 3:* In the final case, the crawler is unable to extract $label(E)$, either because the label is absent, or because there is a problem in label extraction. Therefore, we identify an entry in the LVS table whose value set already contains most of the values in D. Once such an entry is located, we shall add the values in D to the value set of that entry. Formally, for each entry (L, V) in the table, we compute a score,[17] defined by the expression $\frac{\sum_{x \epsilon D} M_V(x)}{|D|}$. Intuitively, the numerator of the score measures how much of D is already contained in V and the denominator normalizes the score by the size of D. Next, we identify the entry with the maximum score (say (L_i, V_i)) and also the value of the maximum score (say s_{max}). Let fuzzy set D' be derived from D by using the membership function $M_{D'}(x) = s_{max} M_D(x)$ (note: s_{max} is less than 1). We replace entry (L_i, V_i) by the entry $(L_i, V_i \cup D')$.

5.6 Configuring HiWE

In this section, we described different aspects of HiWE that require explicit customization or tuning to meet the needs of a particular application. In addition, we also introduced a few configurable parameters that control the actions of the crawler. Table 1 summarizes all the inputs that the user must provide to the crawler, before initiating the crawling activity.

6 Experiments

We performed a number of experiments to study and validate the overall architecture as well as the various techniques that we have employed. In this section, we summarize some of the more significant results from these experiments.

[17] In fuzzy set terminology, this score is the *degree of subsethood* of D in V, defined by $S(D, V) = \frac{|D \cap V|}{|D|}$.

Total number of forms	70
Number of sites from which forms were picked	45
Total number of elements	315
Total number of finite domain elements	140
Average number of elements per form	4.5
Minimum number of elements per form	1
Maximum number of elements per form	12

Table 2: Statistics pertaining to the forms used in testing label and domain value extraction technique

6.1 Testing label and domain value extraction

Recall that as part of form analysis (Section 5.1), the crawler extracts labels (for all elements, whenever one is available) and domain values (for finite domain elements such as selection lists and checkboxes). Since this analysis is an important component of our crawling technique, we conducted extraction experiments on a set of 70 randomly chosen forms. Table 2 summarizes the relevant statistics of our test set. We ensured that a variety of different forms, ranging from the simplest search box to the most complex ones with 10 or more elements, were included in the test set. Each form was manually analyzed to derive the correct label and domain values for each element. Then, the extraction algorithms were executed on the same set of forms.

We observed that our extraction technique was able to achieve **91.5%** accuracy[18] in extracting labels and almost **98%** accuracy in extracting domain values. In manually inspecting the HTML markup for a random sample of these 70 forms, we noticed that the association between domain values and the corresponding form widgets is not as significantly affected by the complexity of the layout as the association between labels and form widgets. Therefore, we expect that simpler techniques, based on analyzing the HTML text, might perform reasonably well for domain value extraction. However, since we use the layout technique to extract labels, there is no overhead in using layout information for extracting domain values as well.

6.2 Crawler Performance Metric

Traditional crawlers, which deal with the publicly indexable Web, use metrics such as crawling speed, scalability [15], page importance [8], and freshness [7], to measure the effectiveness of their crawling activity. However, none of these metrics captures the fundamental challenge in dealing with the Hidden Web - namely processing and submitting forms.

We considered a number of options for measuring the performance of HiWE. *Coverage*, i.e., the ability to extract as much of the content from the databases as possible, is conceptually appealing but very difficult to estimate or compute without information about the databases themselves. Similarly, *relevancy*, i.e., the question of whether the extracted content is germane to the application/task being performed, is also useful but very difficult to compute without exhaustive manual inspection. We finally chose to use the *submission efficiency* metrics that we define below.

Let N_{total} be the total number of completed forms that the crawler submits, during the course of its crawling

[18] An extracted label/domain value is accurate if it matches the one obtained manually.

activity. Let $N_{success}$ denote the number of submissions which result in a response page containing one or more search results.[19] Then, we define *strict submission efficiency (SE_{strict})* metric as:

$$SE_{strict} = \frac{N_{success}}{N_{total}}$$

Note that this metric is 'strict', because it penalizes the crawler even for submissions which are intrinsically 'correct' but which did not yield any search results because the content in the database did not match the query parameters. One could define a *lenient submission efficiency ($SE_{lenient}$)* metric that penalizes a crawler only if a form submission is semantically incorrect (e.g., submitting a company name as input to a form element that was intended to receive names of company employees). Specifically, if N_{valid} denotes the number of semantically correct form submissions, then

$$SE_{lenient} = \frac{N_{valid}}{N_{total}}$$

$SE_{lenient}$ is more difficult to evaluate, since each form submission must be compared *manually* with the actual form, to decide whether it is a semantically correct submission. For large experiments involving hundreds of form submissions, computing $SE_{lenient}$ becomes highly cumbersome. Hence, in all our experiments, we used SE_{strict} to measure crawler performance.

Intuitively, the submission efficiency metrics estimate how much useful work a crawler accomplishes, in a given period of time. In particular, if two crawlers with different submission efficiency ratings are allowed to crawl for the same amount of time, the crawler with the higher rating is expected to retrieve more 'useful' content than the other.

Parameter	Value
Number of sites visited	50
Number of forms processed	218
Number of forms submitted	94
Label matching threshold (σ)	0.75
Maximum number of submissions per form (S_{max})	100
Minimum form size (α)	3
Value assignment ranking function(ρ)	Fuzzy conjunction
Minimum acceptable rank of a value assignment (ρ_{min})	0.6

Table 3: Parameter values for performance experiments

6.3 Proof-of-Concept Experiments

All the experiments we present in this section were based on configuring and executing the crawler for the following task, namely, looking for online news articles, reports, technical whitepapers, and press releases, pertaining to the semiconductor industry, released during the last 10 years. Table 3 lists the default values of α, σ, and S_{max} that we used in our experiments.

[19] In our experiments, to obtain a precise value for $N_{success}$, we used manual inspection of the pages, rather than using information from the crawler's response analysis module.

For the purposes of our experiments, in addition to restricting the number of submissions per form, we also imposed the condition that a value assignment X could be used to submit a form only if its rank, $\rho(X)$, was greater than or equal to a specified constant ρ_{min}(set to 0.6 in Table 3). Without this additional restriction, a form for which the set of possible value assignments is less than S_{max}, will always be submitted using the same set of assignments, irrespective of the ranking function being used. This additional restriction allows us to compare the three different value assignment ranking functions presented in Section 3.3.

Site Name	URL
Semiconductor Research Corporation	www.src.org
The Semiconductor Reference Site	www.semiref.com
Hoover Online Business Network	www.hoovers.com
Lycos Companies Online	companies.lycos.com

Table 4: Sources of information used to initialize the crawler

To configure the crawler, we required a target set of sites as well as some information to initialize the LVS table. Table 4 lists the online sources we used to generate some of the basic LVS entries required by the crawler. These entries included partial lists of names of semiconductor manufacturing companies as well as list of sub-sectors (or areas) within the semiconductor industry. The first two sources listed in Table 4 were (manually) used only once, to extract information for explicit initialization. The remaining two sources were wrapped by custom wrappers to interface with the LVS manager and automatically provide values at run-time. The Yahoo directory was also wrapped, as described in Section 5.5.1, to act as a run-time source of values.

The crawler was provided with a list of 50 sites that included both generic and semiconductor industry specific databases of press releases, reports, product announcements, product reviews, etc. Table 5 presents a sample of some of the sites that were targeted by the crawler. In all, the crawler encountered and processed 220 forms of which 126 were ignored as they did not satisfy the criteria described in Section 5.3. Most of the discarded forms were small intra-site search forms containing just a search box and a submit button.

Site Name	URL
IEEE Spectrum	www.spectrum.ieee.org
Semiconductor Online	www.semiconductoronline.com
Semiconductor Business News	www.semibiznews.com
Yahoo News	news.yahoo.com
Total News	www.totalnews.com
Semiconductor International	www.semiconductor-intl.com
Solid State Technology International Magazine	www.solid-state.com
CNN Financial News	www.cnnfn.com
TMCnet.com Technology News	www.tcmnet.com
SemiSeekNews	www.semiseeknews.com

Table 5: Sample of the target sites crawled

Effect of Value Assignment ranking function: To study the effect of the value assignment ranking function (Section 3.3), the crawler was executed three times, with the same parameters, same initialization values, and same

Ranking function	N_{total}	$N_{success}$	SE_{strict}
ρ_{fuz}	3214	2853	88.77
ρ_{avg}	3760	3126	83.14
ρ_{prob}	4316	2810	65.11

Table 6: Crawler performance with different ranking functions

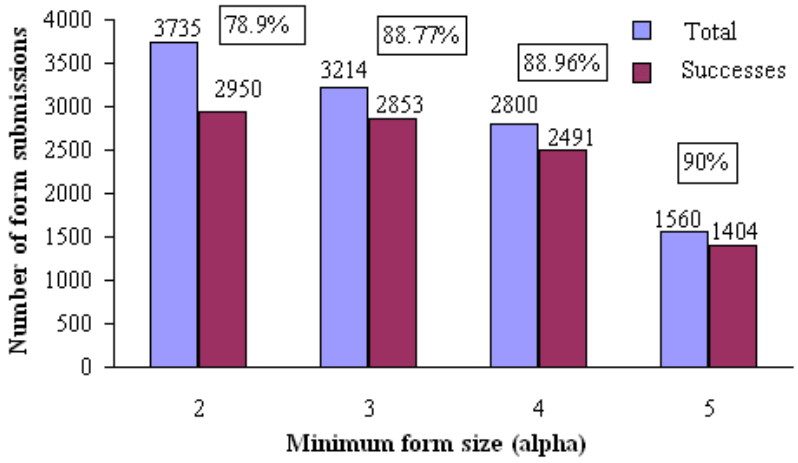

Figure 8: Variation of performance with α

set of data sources but using a different ranking function on each occasion. Table 6 shows the result of these executions. Ranking function ρ_{fuz} provides the best submission efficiency, but being conservative, causes less forms to be submitted than ρ_{avg}. The latter submits more forms but also generates more successful submissions without significantly compromising crawler efficiency. This indicates that if maximum content extraction, rather than crawler efficiency, was the primary criterion, at least for our application, ρ_{avg} might be a better choice. In comparison, ranking function ρ_{prob} performs poorly. Even though it submits 35% more forms than ρ_{fuz}, it achieves lesser number of successful form submissions, resulting in an overall success ratio of only 65%. This indicates that the ρ_{prob} function is not very good at identifying "good" value assignments.

Effect of α: Figure 8 illustrates the effect of changing α, the minimum form size. For each value of α, the figure indicates the values of both N_{total} and $N_{success}$. The percentage figure represents the corresponding value of SE_{strict}. Note that in general, the crawler performs better on larger forms. Smaller forms tend to have less descriptive labels, often consisting merely of an unlabeled text box with an associated "Search" button. Expectedly, the crawler either ignores such forms or is unable to find matches for element labels. On the other hand, larger and more complicated forms tend to have more descriptive labels as well as a significant number of finite domain elements, both of which contribute to improved performance.

Effect of crawler input to LVS table: In Section 5.5, we described the process by which a crawler can contribute entries to the LVS table. Figure 9 studies the effect of this technique on the performance of the crawler. To generate the data for Figure 9, the crawler was executed twice, once with the crawler contributing LVS entries, and another

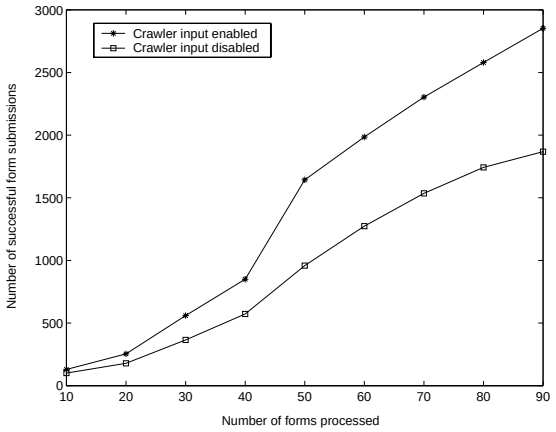

Figure 9: Effect of crawler input to LVS on performance

time, with such contributions disabled. Figure 9 shows that in the initial stages of the crawl, the presence or absence of the crawler contributions do not have a significant impact on performance. However, as more forms are processed, the crawler encounters a number of different finite domain elements and is able contribute new entries to the LVS table. In addition, the LVS manager uses these new entries to retrieve additional values from the data sources (as described in Section 5.5.1). As a result, by the end of the crawl, contributions from the crawler are, directly or indirectly, responsible for almost a 1000 additional successful form submissions.

7 Related Work

In recent years, the growth of the Web has stimulated significant interest in the study of Web crawlers. These studies have addressed various issues, such as performance, scalability, freshness, extensibility, and parallelism, in the design and implementation of crawlers [5, 6, 8, 15, 23]. However, all of this work has focused solely on the publicly indexable portion of the Web (see Figure 1). To the best of our knowledge, there has not been any previous report (at least, none that is publicly available) on techniques and architectures for crawling the hidden Web.

Our task-driven approach to crawling is similar to the approach adopted in the work on *focused crawling*[5]. Specifically, a focused crawler can be tuned to seek out and retrieve pages relevant to a predefined set of topics. However, the focused crawling approach presented in [5] is applicable only to pages in the publicly indexable Web.

In [18], we have addressed the issue of matching form elements to descriptive text labels, in the context of enabling HTML form support on small devices, such as PDAs. In [18], we present a variety of text-based techniques for extracting labels and conduct extensive performance experiments. The techniques in [18] are based on a detailed study of the most common ways in which forms are laid out on web pages.

The online service InvisibleWeb.com [16] provides easy access to thousands of online databases, by organizing pointers to these databases in a searchable topic hierarchy. Their web page indicates that a ‘combination of automated intelligent agents along with human experts’ are responsible for creating and maintaining this hierarchy. Similarly, the online service BrightPlanet.com [3] claims to automatically ‘identify, classify, and categorize’ con-

tent stored in the hidden Web. In both cases, the techniques are proprietary and details are not publicly available.

8 Conclusion

In this paper, we addressed the problem of crawling and extracting content from the "hidden Web", the portion of the Web hidden behind searchable HTML forms. We proposed an application/task specific approach to hidden Web crawling. We argued that because of the tremendous size of the hidden Web, comprehensive coverage is very difficult, and possibly less useful, than task-specific crawling. This specificity is also useful in designing a configurable crawler that can benefit from knowledge of the particular application domain.

We presented a simple model of a search form and the process of filling out forms. Based on this model, we described an architecture for a task-configurable hidden Web crawler and illustrated how it could be used to automate content extraction from the hidden Web. We also presented various novel techniques that went into the design and implementation of our HiWE crawler. Finally, we presented some results from experiments that we conducted to validate our techniques. Our results show that human-assisted crawling of the hidden Web is feasible, and that relatively few forms are filled-in incorrectly.

References

[1] Amazon.com web site. http://www.amazon.com.

[2] Active Server Pages Technology. http://msdn.microsoft.com/workshop/server/asp/aspfeat.asp.

[3] BrightPlanet.com. http://www.brightplanet.com.

[4] The Deep Web: Surfacing Hidden Value. http://www.completeplanet.com/Tutorials/DeepWeb/.

[5] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proceedings of the Eighth International World-Wide Web Conference*, 1999.

[6] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0129.

[7] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the International Conference on Management of Data*, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0116.

[8] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the Seventh International World-Wide Web Conference*, 1998. Available at http://www-diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1999-0103.

[9] Document Object Model Level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1/.

[10] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. STRUDEL: A web-site management system. In *Proceedings of the International Conference on Management of Data*, pages 549–552, 1997.

[11] 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.

[12] Forms in HTML Documents – W3C HTML 4.01 Recommendation. http://www.w3.org/TR/html401/interact/forms.html.

[13] W. B. Frakes and R. Baeza-Yates. *Information Retrieval Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, N.J., 1992.

[14] H.-J.Zimmermann. *Fuzzy Set Theory*. Kluwer Academic Publishers, 1996.

[15] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, December 1999.

[16] InvisibleWeb.com. http://www.invisibleweb.com.

[17] JavaServer Pages (JSP™) Technology. http://java.sun.com/products/jsp/.

[18] Oliver Kaljuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient web form entry on pdas. In *Submitted for publication*, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-2000-0145.

[19] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98, 1998.

[20] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[21] Daniel Lopresti and Andrew Tomkins. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, July 1997.

[22] Giansalvatore Mecca, Paolo Atzeni, Alessandro Masci, Paolo Merialdo, and Giuseppe Sindoni. The ARANEUS web-base management system. In *Proceedings of the International Conference on Management of Data*, pages 544–546, 1998.

[23] Robert C. Miller and Krishna Bharat. Sphinx: a framework for creating personal, site-specific web crawlers. In *Proceedings of the Seventh International World-Wide Web Conference*, 1998.

[24] Open directory. http://www.dmoz.org.

[25] PHP Hypertext Processor. http://www.php.net.

[26] Java Servlet™Technology. http://java.sun.com/products/servlet/.

[27] Yahoo incorporated. http://www.yahoo.com.