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 address the problem of designing a crawler capable of extracting content from this hidden Web. We introduce a generic operational model of a hidden Web crawler and describe how this model is realized in HiWE (Hidden Web Exposer), a prototype crawler built at Stanford. We introduce a new **L**ayout-based **I**nformation **E**xtraction **T**echnique (LITE) and demonstrate its use in automatically extracting semantic information from search forms and response pages. We also present results from experiments conducted to test and validate our techniques.

1 Introduction

Crawlers are programs that automatically traverse the Web graph, retrieving pages and building a local repository of the portion of the Web that they visit. Depending on the application at hand, the pages in the repository are either used to build search indexes, or are subjected to various forms of analysis (e.g., text mining). Traditionally, crawlers have only targeted a portion of the Web called the *publicly indexable Web (PIW)* [13]. This refers to the set of pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

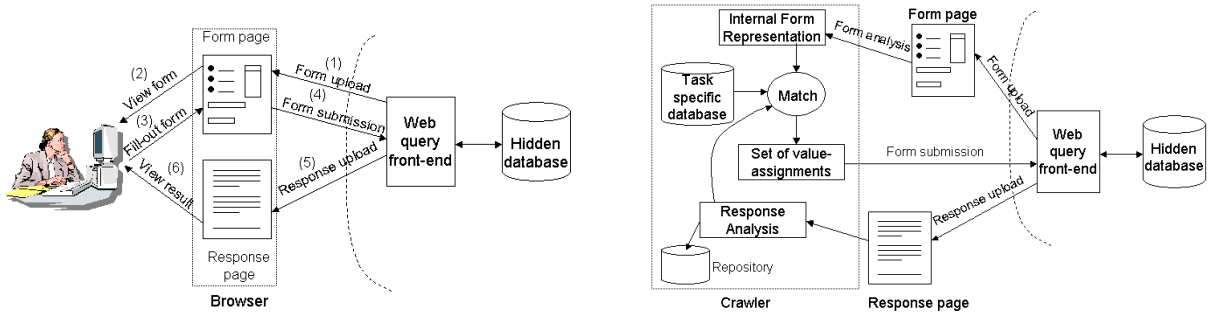
However, a number of recent studies [2, 13, 14] have observed that a significant fraction of Web content in fact lies outside the PIW. Specifically, large portions of the Web are 'hidden' behind search forms, in searchable structured and unstructured databases (called the *hidden Web* [8] or *deep Web* [2]). Pages in the hidden Web are *dynamically generated* in response to queries submitted via the search forms. The hidden Web continues to grow, as organizations with large amounts of *high-quality* information (e.g., the Census Bureau, Patents and Trademarks Office, news media companies) are placing their content online, providing Web-accessible search facilities over existing databases. For instance, the website `InvisibleWeb.com` lists over 10000 such databases ranging from archives of job listings to directories, news archives, and electronic catalogs. Recent estimates [2] place the size of the hidden Web (in terms of generated HTML pages) at around 500 times the size of the PIW.

In this paper, we address the problem of building a hidden Web crawler; one that can crawl and extract content from these hidden databases. Such a crawler will enable indexing, analysis, and mining of hidden Web content, akin to what is currently being achieved with the PIW. In addition, the content extracted by such crawlers can be used to categorize and classify the hidden databases.

Challenges. There are significant technical challenges in designing a hidden Web crawler. First, the crawler must be designed to automatically parse, process, and interact with form-based search interfaces that are designed primarily for human consumption. Second, unlike PIW crawlers which merely submit requests for URLs, hidden Web crawlers must also provide input in the form of search queries (i.e., "fill out forms"). This raises the issue of how best to equip crawlers with the necessary input values for use in constructing search queries.

To address these challenges, we adopt a *task-specific, human-assisted* approach to crawling the hidden Web.

Task-specificity: We aim to selectively crawl portions of the hidden Web, extracting content based on the requirements of a particular application or task. For example, consider a market analyst who is interested in building an archive of news articles, reports, press releases, and white papers pertaining to the semiconductor industry, and



(a) User form interaction

(b) Crawler form interaction

Figure 1: Interacting with forms

dated sometime in the last ten years. There are two steps in building this archive: *resource discovery*, wherein we identify sites and databases that are likely to be relevant to the task; and *content extraction*, where the crawler actually visits the identified sites to submit queries and extract hidden pages. In this paper, we do not directly address the resource discovery problem (see Section 6 for citations to relevant work). Rather, our work examines how best to automate content retrieval, given the results of the resource discovery step.

Human-assistance: Human-assistance is critical to ensure that the crawler issues queries that are relevant to the particular task. For instance, in the above example, the market analyst may provide the crawler (see Section 3.4 for details) with lists of companies or products that are of interest. This enables the crawler to use these values when filling out forms that require a company or product name to be provided. Furthermore, as we will see, the crawler will be able to gather additional potential company and product names as it visits and processes a number of pages.

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

- We develop a generic operational model of a hidden Web crawler and illustrate how this model was put to use in implementing HiWE. (Sections 2 and 3)
- We propose a new technique, called LITE (**L**ayout-based **I**nformation **E**xtraction **T**echnique), for information extraction from Web pages. We illustrate how LITE was employed in some parts of the HiWE design. (Section 4)
- Finally, we present some experiments to demonstrate the feasibility of hidden Web crawling and measure the effectiveness of our approach and techniques. (Section 5)

2 Hidden Web Crawlers

In this section, we first present a generic high-level operational model of a hidden Web crawler. Next, we propose

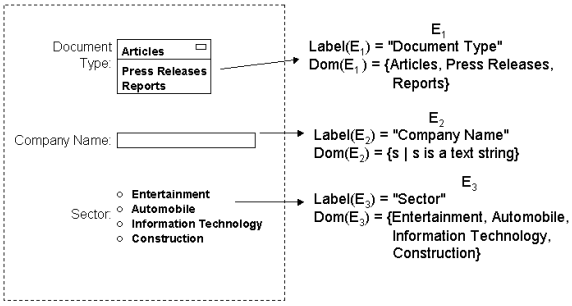


Figure 2: Sample labeled form

metrics for measuring the performance of such crawlers and justify the rationale behind our choices. Finally, we identify the key design issues in implementing the model.

2.1 Operational model

The fundamental difference between the actions of a hidden Web crawler, such as HiWE, and that of a traditional crawler [3, 6], is with respect to pages containing search forms. Figure 1(a) illustrates the sequence of steps (as indicated by the numbers above each arrow) that take place, when a *user* uses a search form to submit queries on a hidden database. Figure 1(b) illustrates the same interaction, with the *crawler* now playing the role of the human-browser combination.

Our model of a hidden Web crawler consists of the four components described below (see Figure 1(b)). We shall use the term *form page*, to denote the page containing a search form, and *response page*, to denote the page received in response to a form submission.

Internal Form Representation. On receiving a form page, a crawler first builds an internal representation of the search form. Abstractly, the internal representation of a form F includes the following pieces of information: $F = (\{E_1, E_2, \ldots, E_n\}, S, M)$, where $\{E_1, E_2, \ldots, E_n\}$ is a set of n form elements, S is the submission information associated with the form (e.g., submission URL, internal identifiers for each form element, etc.), and M is meta-information about the form (e.g., URL of the form

page, web-site hosting the form, set of pages pointing to this form page, other text on the page besides the form, etc.). A form element can be any one of the standard input elements: selection lists, text boxes, text areas, checkboxes, or radio buttons.[1] For example, Figure 2 shows a form with three elements (ignore $label(E_i)$ and $dom(E_i)$ for now). Details about the actual contents of M and the information associated with each E_i are specific to a particular crawler implementation.

Task-specific database. A crawler is equipped, at least conceptually, with a task-specific database D. This database contains all the information that is necessary for the crawler to formulate search queries relevant to the particular task. For example, in the 'market analyst' example introduced in Section 1, D could contain lists of semiconductor company and product names that are of interest. The actual format, structure, and organization of D are specific to a particular crawler implementation. For example, HiWE uses a set of labeled fuzzy sets (Section 3.2) to represent task-specific information. More complex representations are possible, depending on the kinds of information used by the matching function (see below).

Matching function. A crawler's matching algorithm, $Match$, takes as input, an internal form representation, and the current contents of the database D. It produces as output, a set of value assignments. Formally, $Match((\{E_1, \ldots, E_n\}, S, M), D) = \{[E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n]\}$.

A value assignment $[E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n]$ associates value v_i with form element E_i (e.g., if E_i is a textbox that takes a company name as input, v_i could be 'National Semicondutor Corp.'). The crawler uses each value assignment to 'fill-out' and submit the completed form. This process is repeated until either the set of value assignments is exhausted, or some other termination condition is satisfied.

Response Analysis. The response to a form submission is received by a response analysis module that stores the page in the crawler's repository. In addition, the response module analysis attempts to distinguish between pages containing search results and pages containing error messages. This feedback can be used to tune the matching function and update the set of value assignments (see Section 3).

Notice that the above model lends itself to a number of different implementations depending on the internal form representation, the organization of D, and the algorithm that underlies $Match$.

2.2 Performance Metric

Traditional PIW crawlers use metrics such as crawling speed, scalability [10], page importance [6], and freshness [5], to measure the effectiveness of their crawling activity. Though all of these metrics are applicable and relevant to hidden Web crawlers, none of these capture the fundamental challenges in dealing with the Hidden Web, namely, automatic form processing and submission.

The choice of a good performance metric for hidden Web crawlers itself turns out to be an interesting issue. We considered a number of options. For instance, we considered a *coverage* metric that measures the ratio of the number of 'relevant' pages extracted by a crawler to the total number of 'relevant' pages present in the targeted hidden databases. Even though such a metric is conceptually appealing, there are two problems. First, without additional information about the hidden databases, it is very difficult to estimate how much of the their content is relevant to the task. Second, the metric is significantly dependent on the contents of D, the crawler's task-specific database. This in turn is determined by how well the crawler is configured for the task, by the human. All other things being equal, a crawler that has access to a more comprehensive task-specific database can extract more content and hence report better coverage. However, we seek a metric that can measure the effectiveness of the crawler's form representation and matching function, independent of the actual contents of D. Below, we define two versions of a metric that meet this requirement.

Submission Efficiency. Let N_{total} be the total number of forms that the crawler submits, during the course of its crawling activity. Let $N_{success}$ denote the number of submissions which result in a response page containing one or more search results.[2] Then, we define the *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. We also 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. For large experiments involving hundreds of form submissions, computing $SE_{lenient}$ becomes highly cumbersome.

Intuitively, the submission efficiency metrics estimate how much useful work a crawler accomplishes, in a given period of time. In particular, if two identically configured crawlers 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.

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

[2] 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.

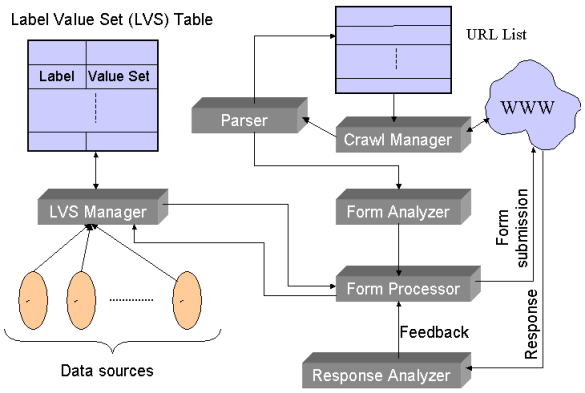


Figure 3: HiWE Architecture

2.3 Design Issues

Given the operational model and the performance metrics described in the previous two sections, the following questions arise:

- What information about each form element E_i, should the crawler collect? What meta-information about each form is likely to be useful in designing better matching functions?
- How should the task-specific database be organized, updated, and accessed?
- What is the algorithm for $Match$ that is most likely to maximize submission efficiency?
- Finally, how should the feedback from the response analysis module be used to tune $Match$?

In the following section, we shall describe how these issues are addressed in the HiWE prototype.

3 HiWE: Hidden Web Exposer

Based on the model outlined in Section 2, we have built a prototype hidden Web crawler called HiWE. The basic idea in HiWE is to extract some kind of descriptive information, or label, for each element of a form. In addition, the task-specific database is organized in terms of a finite number of concepts or categories, each of which is also associated with labels. The matching algorithms attempts to match form labels with database labels to compute a set of candidate value assignments.

Figure 3 shows the architecture of HiWE. The basic crawler data structure is the *URL List*. It contains all the URLs that the crawler has discovered so far. The *Crawl Manager* controls the entire crawling process. In our implementation, the crawler was configured to operate within a predetermined set of target sites provided to the Crawl Manager at startup. The *Parser* extracts hypertext links from the crawled pages and adds them to the URL List structure. Pages that do not contain forms are handled solely by the Parser and Crawl Manager modules. The *Form Analyzer*, *Form Processor*, and *Response Analyzer* modules, together implement the form processing and submission operations of the crawler. The *LVS table* is HiWE's implementation of the task-specific database described in Section 2.1. The *LVS Manager* manages additions and accesses to the LVS table.

3.1 Form Representation

Given a form $F = (\{E_1, E_2, \ldots, E_n\}, S, \phi\})$[3], for each element E_i, HiWE collects two pieces of information: a domain $Dom(E_i)$ and a label $label(E_i)$. 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, then $Dom(E_j)$ is the set of values that are contained in the list. Other elements with free-form input, such as text boxes, have *infinite domains* (e.g., set of all text strings).

The label of a form element is the descriptive information associated with that element, if any. Most forms are usually associated with some descriptive text to help the user understand the semantics of the element. If such descriptive information is not available, or cannot be extracted, the corresponding $label(E_i)$ is set to an empty string. Figure 2 shows a form with three elements and the corresponding representation using our notation.

3.2 Task-specific Database

In HiWE, task-specific information is organized in terms of a finite set of concepts or categories. Each concept has one or more labels and an associated set of values. For example, the label 'Company Name' could be associated with the set of values {'IBM', 'Microsoft', 'HP', ... }. The concepts 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), L is a label and $V = \{v_1, \ldots v_n\}$ is a *fuzzy/graded set* [23] of values. 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. $M_V(v_i)$ is a measure of the crawler's confidence that the assignment of v_i to E is in fact a semantically meaningful assignment. Labels can be *aliased*, which means that two or more labels can share the same fuzzy value set. Section 3.4 describes how the LVS table is populated and Section 3.5 describes how weights are computed.

3.3 Matching Function

For a form element with a finite domain, the set of possible values that can be assigned to the element is fixed, and can be exhaustively enumerated. For example, since domain $Dom(E_1)$ 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. For in-

[3]The current implementation of HiWE does not collect any meta-information about a search form. Therefore, the third component of F is an empty set.

finite domain elements, HiWE textually matches the labels of these elements with labels in the LVS table. For example, if a textbox element has the label "Enter state" which best matches an LVS entry with the label "State", the values associated with that LVS entry (e.g., "California" or "New York") can be used to fill out the textbox.

Label Matching. There are two steps in matching form labels with LVS labels. First, all labels are normalized; this includes, among other things, conversion to a common case and standard IR-style stemming and stop-word removal [9] (see [20] for details). Next, an approximate string matching algorithm is used to compute minimum edit distances, taking into account not just typing errors but also word reorderings (e.g, we require that two labels 'Company Type' and 'Type of Company', which become "company type" and "type company" after normalization, be identified as being very similar, separated by a very small edit distance). HiWE employs a string matching algorithm from [15] that meets these requirements. Given element E_i, let $LabelMatch(E_i)$ denote the entry in the LVS table whose label has the minimum edit distance to $label(E_i)$, subject to a threshold σ. If all entries in the LVS table are more than σ edit operations away from $label(E_i)$, $LabelMatch(E_i)$ is set to nil.

Given a form $F = (\{E_1, \ldots, E_n\}, S, \phi)$, HiWE's matching function computes, for each element E_i, a fuzzy set V_i denoting the set of values that the crawler intends to assign to E_i. Specifically, if E_i is an infinite domain element and $(L, V) = LabelMatch(E_i)$ is the closest matching LVS entry, then $V_i = V$ and $M_{V_i} = M_V$. However, if E_i is a finite domain element, then $V_i = Dom(E_i)$ and $M_{V_i}(x) = 1, \forall\, x \,\epsilon\, V_i$.

The set of value assignments is computed as the product of all the V_i's; i.e., $Match(F, LVS) = \{[E_1 \leftarrow v_1, \ldots, E_n \leftarrow v_n] : v_i \,\epsilon\, V_i, i = 1 \ldots n\}$

Ranking value assignments. HiWE employs an aggregation function to compute a rank for each value assignment, using the weights of the individual values in the assignment. In addition, HiWE accepts, as a configurable parameter, a minimum acceptable value assignment rank (ρ_{min}). The intent is to improve submission efficiency by only using relatively 'high-quality' value assignments. Hence, to generate submissions, HiWE uses only value assignments whose rank is at least ρ_{min}. We experimented with the following aggregation functions:

1. Fuzzy Conjunction The rank of a value assignment is the minimum of the weights of all the constituent values. This is equivalent to treating the value assignment as a standard Boolean conjunction of the individual fuzzy sets [23].

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

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

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

3. 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. Hence, 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, ρ_{prob} is more aggressive and assigns a low rank only if all the individual weights are very low.

3.4 Populating the LVS Table

HiWE supports a variety of mechanisms for adding entries to 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 task described in Section 1, we supplied HiWE with a list of relevant company names from the semiconductor industry and associated that list with labels such as "Company", "Company Name", "Organization", etc.

Built-in entries. 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 for a variety of tasks.

Wrapped data sources. The LVS Manager (Figure 3) can communicate and receive entries for the LVS table by querying various data sources (on the Web or elsewhere), through a well-defined interface. These data sources can either be task-specific (for example, Table 4 lists some of the task-specific Web sources that we used for the task outlined in Section 1), or correspond to relevant portions of generic directories, such as the Yahoo directory [22] and the Open Directory [18]. Each data source must be 'wrapped' by a program to export an interface that supports one or both of the following two kinds of queries:

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

Type1 queries are used to add new entries to the LVS table whereas Type2 queries are used to expand existing entries. In [20], we describe in some detail, how the Yahoo directory was wrapped to export the above interface.

Crawling experience. Finite domain form elements are a useful source of labels and associated value sets. Whenever HiWE encounters a finite domain form element, it extracts the label and domain values of that element and add the information to the LVS table. As we demonstrate in Section 5, this technique is particularly effective if the

same/similar label is associated with a finite domain element in one form and with an infinite domain element in another. For example, we observed that when experimenting with the crawling task described in Section 1, some forms contained a predefined 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.

3.5 Computing weights

Since value sets in the LVS table 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 built-in categories have fixed predefined weights that do not vary with time (usually the weight is 1, representing maximum confidence in these human-supplied values). Values obtained either from external data sources or through the crawler's own activity, are assigned weights that vary with time. The weight of a value gets a positive (negative) boost ever time it is used in a successful (unsuccessful) form submission. The success or otherwise, of a form submission, is reported by the response analysis module. In [20], we describe how feedback from the response analysis module is used to tune the weights.

The initial weights for values obtained from external data sources are usually computed by the respective wrappers. However, for values directly gathered by the crawler, the following strategy is used:

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

Case 1. *Crawler successfully extracts* $label(E)$ *and computes* $LabelMatch(E) = (L, V)$. We replace the (L, V) entry in the LVS table by the entry $(L, V \cup Dom(E))$. Here, $\cup$ is the standard fuzzy set union operator [23] which defines the new membership function as $M_{V \cup Dom(E)}(x) = \max(M_V(x), M_{Dom(E)}(x))$. Intuitively, $Dom(E)$ not only provides new elements to the value set but also 'boosts' the weights/confidence of existing elements.

Case 2. *Crawler successfully extracts* $label(E)$ *but* $LabelMatch(E)$ = *nil.* A new row/entry ($label(E)$, $Dom(E)$) is created in the LVS table.

Case 3. *Crawler cannot extract* $label(E)$. This can happen either because the label is absent, or because there is a problem in label extraction. We identify an entry in the LVS table whose value set most closely resembles $Dom(E)$. Once such an entry is located, we shall add the values in $Dom(E)$ to the value set of that entry. Formally, for each entry (L, V) in the table, we compute a score,[4] defined by the expression $\frac{\sum_{x \in Dom(E)} M_V(x)}{|Dom(E)|}$. Intuitively, the numerator of the score measures how much of $Dom(E)$ is already contained in V and the denominator normalizes the score by the size of $Dom(E)$. Next, we identify the entry with the maximum score (L_{max}, V_{max}) and also the value of the maximum score s_{max}. We derive a new fuzzy set D' from $Dom(E)$ by using the membership function $M_{D'}(x) = s_{max} M_{Dom(E)}(x)$. We replace entry (L_{max}, V_{max}) by the new entry $(L_{max}, V_{max} \cup D')$.

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	Minimum acceptable value assignment rank (ρ_{min})
6	Minimum form size (α)
7	Value assignment aggregation function

Table 1: Configuring a crawler

3.6 Configuring HiWE

In the previous sections, we described different aspects of HiWE that require explicit customization or tuning to meet the needs of a particular task. 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, before initiating the crawling activity.

4 LITE

Recall that as part of its operations, HiWE must extract various pieces of information out of forms and response pages. A number of other Web applications are also faced with the same problem of 'scraping' information from pages. For example, Web-based information integration applications such as online comparison shopping engines or process automation systems use *wrappers* [19] to provide structured interfaces to Web sites. As one of its functions, a wrapper for a website is required to scrape the Web pages on that site to extract data elements (e.g., names, addresses, zip-codes, prices, etc.) of interest. Traditionally, wrappers scrape pages by using a suite of (regular expression) patterns that are constructed using a variety of automatic and semi-automatic techniques [19, 21]. However, such techniques operate purely on the underlying HTML text of Web pages.

In this section, we introduce a new technique called LITE (Layout-based Information Extraction), where, in addition to the text, the *physical layout* of a page is also used to aid in extraction. LITE is based on the observation that the physical layout of different elements of a Web page contains significant semantic information. For example, a piece of text that is physically adjacent to a table or a form

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

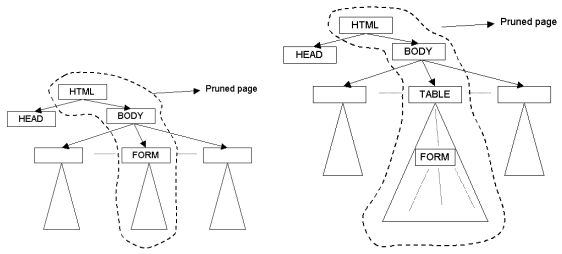


Figure 4: Pruning before partial layout

widget (such as a text box) is very likely a description of the contents of that table or the purpose of that form widget.

Unfortunately, this semantic association between elements is not always directly reflected in the underlying HTML markup of the Web page. There are two reasons for this. First, elements of a page that are visually very close to each other when displayed on a screen, may in fact be separated arbitrarily, in the actual text of the page. Second, even when the HTML specification provides a facility for semantic relationships to be reflected in the markup, such facilities are not used in a majority of Web pages. For example, many Web pages do not use the `CAPTION` element to specify the title of a table, relying instead on the physical placement of the title text relative to the table, to convey the same information. Similarly, recent HTML standards provide a `LABEL` element to associate descriptive information with individual form elements. However, almost none of the Web pages that HiWE visited during its experimental runs used this facility.

Note that accurate page layout is a relatively complex process, since it must take into account factors such as zip codes, font metrics, images, etc. However, for the purposes of information extraction, our experience (see succeeding sections) has been that even a crude and approximate layout of portions of a page, can yield very useful semantic information.

LITE is used in HiWE to extract information from both form and response pages. In the next section, we briefly describe how LITE is used for form analysis and refer the reader to [20] for a similar description of response analysis.

4.1 Form Analysis in HiWE

Recall that the aim of form analysis is to process a form page and extract all the information necessary to build the internal representation (Section 3.1) of the form. For HiWE, the main challenge in form analysis is the accurate extraction of the labels and domains of form elements.

Label extraction is a hard problem, since the nesting relationship between forms and labels in the HTML markup is not fixed. For example, some pages layout form elements and labels within the cells of a table whereas others control alignment through explicit spaces and line breaks. To achieve high-accuracy label extraction, in HiWE, we employ the following LITE-based heuristic:

- Prune the form page and isolate only those elements that directly influence the layout of the form elements and the labels. For instance, consider Figure 4, which shows the tree-structured representation of two different Web 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.
- Approximately layout the pruned page using a custom layout engine that discards images, and ignores styling information such as font sizes, font styles, and style sheets.
- Using the layout engine, identify the pieces of text, if any, that are physically closest to the form element, in the horizontal and vertical directions. These pieces of text are the *candidates*.[5]
- Rank each candidate using a variety of measures that take into account the its position, font size, font style, number of words, etc. (see [20] for details).
- Choose the highest ranked candidate as the label associated with the form element. Perform any post-processing on the label as necessary (e.g., removing stop words and non alphanumeric characters, stemming, etc.)

Reference [20] describes a similar heuristic for extracting the domains of form elements.

5 Experiments

We conducted a number of experiments to study and measure the performance of HiWE. In this section, we report on some of the more significant results from these experiments.

Parameter	Value
Number of sites visited	50
Number of forms encountered	218
Number of forms chosen for submission	94
Label matching threshold (σ)	0.75
Minimum form size (α)	3
Value assignment ranking function	ρ_{fuz}
Minimum acceptable value assignment rank (ρ_{min})	0.6

Table 3: Parameter values for Task 1

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: Sample data sources for Task 1

Table 2 describes the three tasks that we undertook to accomplish using HiWE. Due to space constraints, we provide configuration details and other related information only for Task 1. Table 3 lists the default values of some of the parameters that we used for experiments involving Task 1. The parameter α represents the minimum size of a

[5] For form elements involving groups of items, such as a set of checkboxes, distances are measured relative to the 'center' of the group.

No.	Task Description - Collect Web pages containing:
1	News articles, reports, press releases, and white papers relating to the semiconductor industry, dated sometime in the last ten years
2	Reviews, synopses, articles, and historical information about movies directed by Oscar-winning directors in the last 30 years
3	Database technical reports from 30 CS departments, published in the last 5 years

Table 2: Description of the three experimental tasks

form (in terms of number of elements) that HiWE will attempt to process. Since α was set to 3, all forms containing less than 3 elements were ignored by HiWE. This helped to eliminate most of the forms that dealt with simple keyword searches within a site ('local site-search'), not relevant to extracting content from hidden databases. As indicated in Table 3, the crawler encountered 218 forms when crawling the 50 sites, of which 124 were ignored, either because they were too small (less than 3 elements) or because HiWE was unable to generate valid value assignments for them.

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

Table 5: Sample target sites crawled for Task 1

Table 4 lists some of the online sources we used to generate LVS entries for Task 1. 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 in Table 4, as well as the Yahoo [22] and Open [18] directories, were wrapped by custom wrappers to interface with the LVS manager and provide values at run-time. Table 5 presents a sample of some of the 50 sites that were targeted by HiWE for Task 1.

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 set of data sources, but using a different ranking function on each occasion. Table 6 shows the result of these executions, for all three tasks. Notice that when using ρ_{fuz} and ρ_{avg}, the crawler's submission efficiency is mostly above 80%, even reaching 90% on one occasion. This indicates that the label extraction and matching algorithms used in HiWE are highly effective in automating form processing and submission. Table 6 also illustrates an interesting trade-off between ρ_{fuz} and ρ_{avg}. Ranking function ρ_{fuz} consistently provides the best submission efficiency, but being conservative, causes less forms to be submitted, when compared with ρ_{avg}. The latter submits more forms but also generates more successful submissions without significantly compromising crawler efficiency (at least for Tasks 1 and 2). This

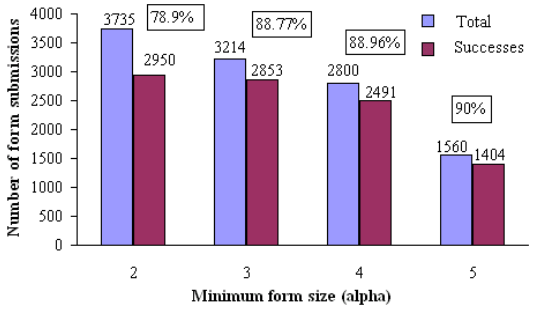


Figure 5: Variation of performance with α, for Task 1

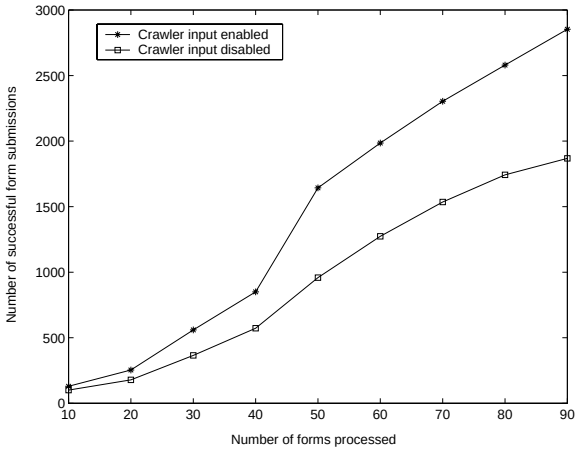


Figure 6: Effect of crawler input to LVS table, for Task 1

indicates that if maximum content extraction with 'reasonable' crawler efficiency were to be the goal, at least for our tasks, ρ_{avg} might be a better choice. In comparison, ranking function ρ_{prob} performs poorly. For instance, in the case of Task 1, even though ρ_{prob} causes 35% more forms to be submitted when compared with ρ_{fuz}, it still achieves lesser number of successful form submissions, resulting in an overall success ratio of only 65%.

Effect of α. Figure 5 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 $N_{success}$, for Task 1. 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. As expected, 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.

Ranking function	Task 1			Task 2			Task 3		
	N_{total}	$N_{success}$	SE_{strict}	N_{total}	$N_{success}$	SE_{strict}	N_{total}	$N_{success}$	SE_{strict}
ρ_{fuz}	3214	2853	88.8	2615	2354	90.0	1018	886	87.0
ρ_{avg}	3760	3126	83.1	3404	2622	77.0	1467	1243	84.7
ρ_{prob}	4316	2810	65.1	3648	2240	61.4	1789	1128	63.0

Table 6: Performance with different ranking functions

Effect of crawler input to LVS table. In Section 3.4, we described the process by which a crawler can contribute entries to the LVS table. Figure 6 studies the effect of this technique on the performance of the crawler. To generate the data for Figure 6, the crawler was executed twice, once with the crawler contributing LVS entries, and another time, with such contributions disabled. Figure 6 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 to 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 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. We observed similar trends for Tasks 2 and 3 (see [20] for corresponding plots).

5.1 Label extraction

We conducted a separate set of experiments to measure the performance of our LITE-based heuristic for label extraction. Table 7 summarizes the relevant statistics of our test set of forms. In choosing the test set, we ensured that a variety of forms were included, ranging from the simplest single element search box to more complex ones with 10 or more elements. Each form in the test set was manually analyzed to derive the correct label for each form element.

In addition to evaluating the LITE-based heuristic on this set of forms, we also tested other label extraction methods [12] that we developed in the context of enabling form support on small devices, such as PDAs. In [12], we describe two classes of label extraction heuristics; one class based purely on textual analysis, and another based on extensive manual observations of the most common ways in which forms are laid out. For comparison, we ran two of the more effective heuristics from [12], one from each class, on the same test set.

We treated an extracted label as accurate, if it matched the one obtained through manual inspection. We observed that the LITE-based heuristic consistently outperformed the other two heuristics, achieving an overall accuracy of **93%**, compared to **72%** and **83%** respectively, for the other two heuristics. In particular, we noted that the LITE-based heuristic avoids two of the three common failure reasons identified in [12], and also performs significantly better on more complex forms. We believe that an effective label extraction technique was an important factor in HiWE's high submission efficiency, as reported in Table 6.

Total number of forms	100
Number of sites from which forms were picked	52
Total number of elements	460
Total number of finite domain elements	140
Average number of elements per form	4.6
Minimum number of elements per form	1
Maximum number of elements per form	12

Table 7: Forms used to test label extraction techniques

6 Related Work

In recent years, there has been 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 [3, 4, 6, 10, 17]. However, all of this work has focused solely on the PIW. 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.

The work on *focused crawling* [3, 7, 16] addresses the resource discovery problem, (i.e., identifying sites and pages relevant to a specific task or topic) and describes the design of topic-specific PIW crawlers. This work is complementary to ours, since these resource discovery techniques can be used to identify target sites for a hidden Web crawler.

The online service InvisibleWeb.com [11] 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 [1] claims to automatically 'identify, classify, and categorize' content stored in the hidden Web. In both cases, the techniques are proprietary and details are not publicly available.

7 Conclusion

Current-day crawlers are used to build repositories of Web pages that provide the input for systems that index, mine, and otherwise analyze pages (e.g., a Web search engine). However, these crawlers are restricted to the set of pages in the publicly indexable portion of the Web. In this paper, we addressed the problem of extending current-day crawlers to build repositories that include pages from the "hidden Web", the portion of the Web behind searchable HTML forms.

We proposed an application/task specific approach to hidden Web crawling. We argued that as with the PIW, the tremendous size and heterogeneity of the hidden Web

makes comprehensive coverage very difficult, and possibly less useful, than task-specific crawling. A narrow application focus is also useful in designing a crawler that can benefit from knowledge of the particular application domain.

We presented a simple operational model of a hidden Web crawler that succinctly describes the steps that a crawler must take, to process and submit forms. We described the architecture and design techniques used in HiWE, a prototype crawler implementation based on this model. The promising experimental results using HiWE demonstrate the feasibility of hidden Web crawling and the effectiveness of our form processing and matching techniques. We believe that our operational model sets the stage for designing a variety of hidden Web crawlers, ranging in complexity from the simple label matching approach of HiWE, to the use of sophisticated natural language and knowledge representation techniques.

For the immediate future, we plan to address two limitations of the HiWE design that if rectified, can significantly improve HiWE's performance. The first limitation is HiWE's inability to recognize and respond to simple dependencies between form elements (e.g., given two form elements corresponding to states and cities, the values assigned to the 'city' element must be cities that are located in the state assigned to the 'state' element). The second limitation is HiWE's lack of support for partially filling out forms; i.e., providing values only for some of the elements in a form.

References

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

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

[3] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of the* 8^{th} *Intl. WWW Conf.* , 1999.

[4] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of the* 26^{th} *Intl. Conf. on Very Large Databases*, 2000.

[5] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2000.

[6] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *Proc. of the* 7^{th} *Intl. WWW Conf.*, 1998.

[7] M. Diligenti, F. Coetzee, S. Lawrence, C. L. Giles, and M. Gori. Focused crawling using context graphs. In *Proc. of the* 26^{th} *Intl. Conf. on Very Large Databases*, pages 527–534, Sept. 2000.

[8] D. Florescu, A. Y. Levy, and A. O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

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

[10] A. Heydon and M. Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, Dec. 1999.

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

[12] O. Kaljuvee, O. Buyukkokten, H. Garcia-Molina, and A. Paepcke. Efficient web form entry on pdas. *Proc. of the* 10^{th} *Intl. WWW Conf.*, May 2001.

[13] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98, 1998.

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

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

[16] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Building domain-specific search engines with machine learning techniques. In *Proc. of the AAAI Spring Symposium on Intelligent Agents in Cyberspace*, 1999.

[17] R. C. Miller and K. Bharat. Sphinx: a framework for creating personal, site-specific web crawlers. In *Proc. of the* 7^{th} *Intl. WWW Conf.*, 1998.

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

[19] Y. Papakonstantinou, H. Garcia-Molina, A. Gupta, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the* 4^{th} *Intl. Conf. on Deductive and Object-Oriented Databases*, pages 161–186, National University of Singapore(NUS), Singapore, 1995.

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

[21] Whizbang! labs. http://www.whizbanglabs.com.

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

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