Google's Deep-Web Crawl

Jayant Madhavan
Google Inc.
jayant@google.com

David Ko
Google Inc.
dko@google.com

Łucja Kot*
Cornell University
lucja@cs.cornell.edu

Vignesh Ganapathy
Google Inc.
vignesh@google.com

Alex Rasmussen*
University of California, San Diego
arasmuss@cs.ucsd.edu

Alon Halevy
Google Inc.
halevy@google.com

ABSTRACT

The Deep Web, i.e., content hidden behind HTML forms, has long been acknowledged as a significant gap in search engine coverage. Since it represents a large portion of the structured data on the Web, accessing Deep-Web content has been a long-standing challenge for the database community. This paper describes a system for *surfacing* Deep-Web content, i.e., pre-computing submissions for each HTML form and adding the resulting HTML pages into a search engine index. The results of our surfacing have been incorporated into the Google search engine and today drive more than a thousand queries per second to Deep-Web content.

Surfacing the Deep Web poses several challenges. First, our goal is to index the content behind many millions of HTML forms that span many languages and hundreds of domains. This necessitates an approach that is completely automatic, highly scalable, and very efficient. Second, a large number of forms have text inputs and require valid inputs values to be submitted. We present an algorithm for selecting input values for text search inputs that accept keywords and an algorithm for identifying inputs which accept only values of a specific type. Third, HTML forms often have more than one input and hence a naive strategy of enumerating the entire Cartesian product of all possible inputs can result in a very large number of URLs being generated. We present an algorithm that efficiently navigates the search space of possible input combinations to identify only those that generate URLs suitable for inclusion into our web search index. We present an extensive experimental evaluation validating the effectiveness of our algorithms.

1. INTRODUCTION

The *Deep Web* refers to content hidden behind HTML forms. In order to get to such content, a user has to perform a form submission with valid input values. The Deep Web has been acknowledged as a significant gap in the coverage of search engines because web crawlers employed by search engines rely on hyperlinks to discover new web pages and typically lack the ability to perform such form submissions. Various accounts have hypothesized that the Deep Web has an order of magnitude more data than the currently searchable World Wide Web [2, 9, 12]. Furthermore, the Deep Web has been a long-standing challenge for the database community [1, 9, 10, 13, 14, 18, 19] because it represents a large fraction of the structured data on the Web.

There are two common approaches to offering access to Deep-Web content. The first approach (essentially a data integration solution) is to create vertical search engines for specific domains (e.g. cars, books, or real estate). In this approach we could create a mediator form for each domain and semantic mappings between individual data sources and the mediator form. For a search engine, this approach suffers from several drawbacks. First, the cost of building and maintaining the mediator forms and the mappings is high. Second, identifying which queries are relevant to each domain is extremely challenging. Finally, and more fundamentally, data on the web is about everything and boundaries of domains are not clearly definable. Hence, creating a mediated schema for the web would be an epic challenge, and would need to be done in over 100 languages.

The second approach is *surfacing*, which pre-computes the most relevant form submissions for all interesting HTML forms. The URLs resulting from these submissions are generated off-line and indexed like any other HTML page. This approach enables leveraging the existing search engine infrastructure and hence the seamless inclusion of Deep-Web pages. User traffic is directed to Deep-Web content when a user clicks on such a search result, which he presumably already believes to be relevant based on its snippet. On clicking, the user is directed to the underlying web site and hence will see fresh content.

Our goal is to increase the accessibility of Deep-Web content for search engine users. We began our surfacing effort with a semi-automatic tool that employed state-of-the-art schema-matching techniques [6, 11]. The tool enabled a human annotator to quickly identify the most relevant inputs and input combinations for any form. The tool enabled the annotator to process a peak of around 100 forms a day but its efficacy was limited to that of the annotator, who soon grew tired or bored of annotating pages.

Our analysis [12] indicated that there are on the order of 10 million high-quality HTML forms. Hence, any approach that involved human effort was doomed to not scale. Instead, we needed an approach that was efficient and *fully automatic*, requiring absolutely no site-specific scripting or

*Work done while at Google Inc.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

analysis. This places restrictions on the techniques that we can employ and also sets us apart from prior work such as [1, 13, 18] that have proposed techniques with the objective of maximizing coverage on specific sites. In fact, we discovered that the potential impact of Deep-Web surfacing on Google.com queries was more dependent on the number of sites we surfaced, i.e., our coverage of the Deep Web at large, rather than our coverage on specific sites. We found that the top 10,000 forms (ordered by their measured impact on search engine queries) accounted for only 50% of Deep-Web results on Google.com, while even the top 100,000 forms only accounted for 85%. This observation justifies our focus on surfacing as many sites as possible.

This paper describes the algorithms we developed to surface the Deep Web at scale. The techniques we describe have surfaced several million forms that span 45 languages and hundreds of domains. Currently, the surfaced content appears in the first 10 results on Google.com for over 1000 queries per second. Our impact on search traffic provides a concrete validation of the potential of the Deep Web and crystallizes some future directions in this field.

The surfacing problem consists of two main challenges: to decide which form inputs to fill when submitting queries to a form and to find appropriate values to fill in these inputs. HTML forms typically have more than one input and hence a naive strategy of enumerating the entire Cartesian product of all possible inputs can result in a very large number of URLs being generated. Crawling too many URLs will drain the resources of a web crawler preventing the good URLs from getting crawled, and posing an unreasonable load on web servers hosting the HTML forms. Furthermore, when the Cartesian product is very large, it is likely that a large number of the result pages are empty and hence useless from an indexing standpoint. As an example, a particular search form on cars.com has 5 inputs and a Cartesian product yields over 240 million URLs, though there are only 650,000 cars on sale [5].

Our first contribution is the *informativeness test* that we use to evaluate *query templates*, i.e., combinations of form inputs. For any template, we probe the form with different sets of values for the inputs in the template, and check whether the HTML pages we obtain are sufficiently distinct from each other. Templates that generate distinct pages are deemed good candidates for surfacing.

Our second contribution is an algorithm that efficiently traverses the space of query templates to identify those suitable for surfacing. The algorithm balances the trade-off between trying to generate fewer URLs and trying to achieve high coverage of the site's content. To the best of our knowledge, ours is the first work that addresses the specific problem of identifying input combinations for forms with multiple inputs. Most prior work has focused on single input forms or assumed naive strategies. We experimentally show that on average we surface only a few hundred URLs per form, while achieving good coverage.

Our third contribution is an algorithm for predicting appropriate input values for text boxes. First, we show how we can extend previous algorithms [1, 13] for selecting keywords for text inputs. Second, we observe that there are a small number of data types (e.g., zip codes, prices, dates) that are extremely common in web forms across many domains. Hence, investing special efforts to automatically recognize inputs of those types yields significant benefit. Here too, we show that we can leverage the informativeness test to recognize such typed boxes effectively. Our textbox-filling algorithm tries to identify one of these specific types and applies a generic keyword extraction algorithm if the type cannot be determined.

```
<form action="http://jobs.com/find" method="get">
  <input type="hidden" name="src" value="hp">
  Keywords:  <input type="text" name="kw">
  State:  <select name="st"> <option value="Any"/>
    <option value="AK"/> ...  </select>
  Sort By:  <select name="sort"> <option value="salary"/>
    <option value="startdate"/> ...  </select>
  <input type="submit" name="s" value="go">
</form>
```

Figure 1: HTML form to search for jobs by keywords and state code. The results can be sorted in different ways.

The rest of the paper is organized as follows: Section 2 describes the basics of HTML form processing, and Section 3 formally describes the problem of Deep-Web surfacing. In Section 4, we describe the informativeness test and how we identify the informative query templates within any form. In Section 5, we describe algorithms that predict values to fill into text inputs. We describe related work in Section 6 and conclude in Section 7.

2. HTML FORM PROCESSING

An HTML form is defined within a `form` tag (example in Figure 1). The `action` identifies the server that will perform the query processing in response to the form submission. Forms can have several input controls, each defined by an `input` tag. Input controls can be of a number of types, the prominent ones being text boxes, select menus (defined in a separate `select` tag), check boxes, radio buttons, and submit buttons. Each input has a `name` which is typically *not* the name that the user sees on the HTML page. Users select input values either by entering arbitrary keywords into text boxes or by selecting from pre-defined options in select menus, check boxes and radio buttons. In addition, there are hidden inputs whose values are fixed and are not visible to users interacting with the form. These are used to provide the server additional context about the form submission (e.g., the specific site from which it came). In this paper we focus on the select menus and text boxes in a form. Check boxes and radio buttons can be treated in the same way as select menus.

When a form is submitted, the web browser sends an HTTP request with the inputs and their values to the server using one of two methods: `get` or `post`. With `get`, the parameters are appended to the action and included as part of the URL in the HTTP request (e.g., `http://jobs.com/find?src=hp&kw=chef&st=Any&sort=salary&s=go` in Figure 1). With `post`, the parameters are sent in the body of the HTTP request and the URL is simply the action (e.g., `http://jobs.com/find` in Figure 1). Hence, the URLs obtained from forms that use `get` are unique (and dependent on submitted values), while the ones obtained with `post` are not.

Since search engines identify web pages based on their URLs, the result pages from a `post` are indistinguishable and hence not directly indexable. Further, as per the HTML specification, `post` forms are to be used whenever submission of the form results in state changes or side-effects (e.g. for shopping carts, travel reservations, and logins). For these

reasons, we restrict our attention in this paper to `get` forms that tend to produce content suitable for indexing.

We ignore forms that require any kind of personal information by filtering away forms with `password` inputs and any keywords typically associated with personal information, e.g., username, login, etc. We ignore `textarea` inputs that are typically used with feedback inputs, e.g., comment submission forms on forums.

Finally, we note that handling Javascript events is beyond the scope of this paper. Forms and inputs can have `onselect`, `onclick`, and `onsubmit` attributes where arbitrary Javascript code can be included. Handling these events involves simulating Javascript execution on all possible events. However, given an efficient Javascript simulator, our algorithms will be able to handle such forms as well.

Apart from the above restrictions, our work considers *all* forms found in the Google index. We note that the Deep Web spans all conceivable domains from product listing and review sites, non-profit, public-sector and government sites, community and social forums, to very eclectic hobby sites.

3. THE SURFACING PROBLEM

We define the problem of Deep-Web surfacing in terms of choosing a set of queries to submit to the form. We then discuss possible objective functions for the selecting the best surfacing strategy.

3.1 Surfacing as Query Evaluations

Form inputs: Formally, we model the content behind a web form as a database D with a single table of m attributes. We model the web form f_D that is used to query D as having n inputs, $X_1, \ldots, X_n$. A form submission takes values for each of the inputs and returns a subset of the records in D. As we explain below, some of the challenges in surfacing arise because some of the properties of the inputs are not known to the system a priori.

There are two kinds of inputs. First, there are *selection* inputs that impose selection conditions on the attributes in D, e.g., `kw`, `st` in Figure 1. The values for selection inputs can either be drawn from a pre-defined list (through a select menu) or entered into a text input. Text inputs may only accept values of a particular type, but in general that type is unknown to us. Selection inputs can often be assigned a *wild card* value that matches all the records in the database. For select menus, the wild card has to be one of the menu's options, e.g., the input `state` has the value `Any`. For text inputs, the wild card is the empty string.

Second, there are *presentation* inputs that only control presentation aspects of the results, such as the sort order or HTML layout, e.g. `sort` in Figure 1. Distinguishing between selection and presentation inputs is one of the challenges that we face.

Some inputs in the form may be required. Here too, we cannot assume a priori knowledge of the required inputs, because the web site might always return *some* HTML page.

Formally, the query posed by filling values for the inputs is `select * from D where P`, where P are the selection predicates expressed by the selection inputs.

Query templates and form submissions: The problem of surfacing is fundamentally a problem of selecting a good set of form submissions. In order to reason about collections of submissions, we define the notion of *query templates*, which are similar in spirit to binding patterns [15]. A query template designates a subset of the inputs of f_D as *binding* inputs and the rest as *free* inputs. Multiple form submissions can be generated by assigning different values to the binding inputs. Thinking in terms of SQL queries, the query template concisely represents all queries of the form `select * from D where` P_B, where P_B includes only the selection predicates imposed by the binding inputs in the form. The number of binding inputs is the *dimension* of a template.

Note that, in practice, values have to be assigned to the free inputs in a template in order to generate valid form submissions. Ideally, we would like these values not to add any additional selection condition to SQL queries for the template. For text inputs, we can assign the empty string, while for select menus, we assign the default value of the menu in the hope that it is a wild card value. We note that, in the interest of easing the interaction burden on their users, forms typically support wild card values for most, if not all, of their inputs.

We can now divide the problem of surfacing a Deep-Web site into two sub-problems:

1. Selecting an appropriate set of query templates, and
2. Selecting appropriate input values for the binding inputs, i.e. instantiating the query template with actual values. For a select menu, we use all values in the menu, but for a text input, the values have to be predicted and we cannot assume a priori knowledge of the domains of the values to be considered.

We assume that the set of values with which an input is instantiated is the same for all templates in which the input is binding. However, in practice some inputs may be correlated. For example, the values for one input (e.g., `cityName`) may be dependent on the value chosen for another input (e.g., `state`), or multiple inputs (e.g., `salaryMax` and `salaryMin`) might restrict the same underlying attribute. Identifying and leveraging such correlations is a subject of ongoing work and is beyond the scope of this paper.

3.2 Objective

The goal in most prior work has been to extract content from specific sites and hence the objective is to maximize the *coverage* of the underlying database, i.e. the total number of records retrieved, while bounding the total number of form submissions. Our goal is to drive traffic from a general purpose search engine to as much of the Deep Web as possible while limiting the load on both the target sites and our web crawler. We therefore share the goals of achieving good coverage on individual sites while limiting the number of submissions. However, we must address several other considerations as well.

We would like to cover as many distinct Deep-Web sites as possible. Specially, this means that we are willing to trade-off coverage of individual sites in the interest of coverage of the entire Deep Web. There are several practical reasons for doing so.

First, while the size of the main index of a search engine is quite large, it is still not nearly enough to store all the pages that can be possibly be extracted from the Deep Web. Of course, the Deep Web is only one of several feeds into the index; this further constrains the number of Deep-Web pages we can store. Since the over-arching goal of a search engine is to direct users to relevant web sites in response to their queries, we would much rather have diverse and important

content coming from many sites as opposed to optimizing the coverage of few individual ones.

Second, it is actually unnecessary for our surfacing of a web site to strive for complete coverage. It suffices to *seed* the web index with enough diverse content from the site. The regular web crawler will eventually crawl outgoing links from the seeds, such as links to more results for the same query or to results of related queries, thereby organically increasing the coverage for the site.

The web pages we surface include listings of results and these pages are inserted into our web index. While from a coverage standpoint we might prefer pages that have large numbers of results, such pages might not be good candidates for serving as search results to users; we will explain why this is so in the next section. We would also not like to index pages that are duplicative in their listings, since they might only add marginal benefits to the search engine index.

Finally, achieving maximum coverage of every individual site may actually decrease the diversity of Deep-Web coverage. In order to achieve maximum coverage for a site, previous works had to rely on customized scripts for each site that extracted and interpreted individual results on the surfaced pages to compute a running estimate of their achieved coverage. This is only realistically possible when dealing with a small number of hand-selected sites and cannot scale to the millions of sites we would like to index.

In summary, our objective is to select queries for millions of diverse forms such that we are able to achieve good (but perhaps incomplete) coverage through a small number of submissions per site and the surfaced pages are good candidates for selection into a search engine's index.

4. SELECTING QUERY TEMPLATES

There are two main challenges in selecting templates. First, we would like to select templates that do not contain any binding presentation inputs because these templates retrieve the same underlying records as the corresponding template without the presentation input. However, we do not know in advance whether an input is a presentation input or not.

Second, we have to select templates of the correct dimension. One strategy would be to choose templates with the largest possible dimension (i.e. with as many binding inputs as possible). Such a template would ensure maximum coverage by generating all possible queries. However, this method will increase crawling traffic and will likely produce many results with empty result sets.

Only choosing templates of smaller dimension has the advantage of generating a smaller number of form submissions. Further, if wild card values can be chosen for all free selection inputs, we might even achieve high coverage. However, such form submissions will each have a very large number of records. In practice, web sites limit the number of records retrieved per query - they might have `next` links for more results or simply truncate the result set. Further, we cannot assume that we can uniquely identify the `next` link for every form. Thus, we are less likely to get complete coverage.

In fact, we can also orthogonally explain the trade-off in template dimension in terms of form submissions being candidates for insertion into a search engine index. Specifically, we would like the surfaced web pages to better satisfy users' needs. Intuitively, if a user's query is a match to a record r in D, we want our index to retrieve for the user the web page with the set of records from D that are "related" to r and, by extension, likely to be relevant to their query.

Recall that D is a table of m attributes and therefore we can model records in D as points in m-dimensional space. Each form submission can be thought of as a rectangle, or more generally a hyperplane of dimension m, that contains a subset of the records from D. Hence, our goal is to create rectangles large enough to include a reasonable number of relevant records but small enough to contain only truly relevant records, i.e. records truly close to r in m-dimensional space. Thus, there is a trade-off in the size of result sets and hence in the choice of template dimension.

The precise choice of template dimensions is highly dependent on the specific database. Among other factors, very large databases are likely to have optimal templates with more binding inputs while smaller databases are likely to do better with templates with fewer binding inputs.

Section 4.1 introduces the *informativeness* test that enables us to select templates that satisfy our desiderata. Section 4.2 describes how to efficiently traverse the potentially large space of possible templates to select all desirable templates. Section 4.3 describes several practical refinements to our basic algorithm. Section 4.4 describes an extensive experimental evaluation of our algorithm.

4.1 Informative Query Templates

We evaluate a query template based on the *distinctness* of the web pages resulting from the form submissions it generates. We estimate the number of distinct web pages the template generates by clustering them based on the similarity of their content.

If the number of distinct web pages is small in comparison with the number of form submissions, it is very likely that either (1) the template includes a presentation input and hence multiple sets of pages essentially have the same records, (2) the template dimension is too high for the underlying database and hence there are a number of pages with no records, all of which resemble one another, or (3) there is a problem in the template (or the form) that leads to error pages that again resemble one another. If the template does not fall into one of the above categories but still generates pages with indistinct content, then it is only likely to be of marginal value to the search engine index and hence is unlikely to have any impact on search engine queries.

Since we cannot employ site-specific parsers to extract and compare the individual records retrieved by each form submission, our test approximates the distinctness by comparing the contents of the corresponding web pages.

We call a template *informative* if the generated pages are sufficiently distinct, and *uninformative* otherwise. Specifically, we compute a *signature* for the contents of the web page resulting from each form submission and deem templates to be uninformative if they compute much fewer signatures than the number of possible submissions. We define informativeness w.r.t. a threshold τ that we determine experimentally later.

Definition 1. **(Informative query template):** Let T be a query template and Sig be a function that computes signatures for HTML pages. Let G be the set of all possible form submissions generated by the template T and let S be the set $\{Sig(p) \mid p \in G\}$.

We say that T is an informative template if $|S|/|G| \geq \tau$. Otherwise we say that T is uninformative. The ratio $|S|/G|$

is called the distinctness fraction. □

We do not compute signatures for form submissions that return HTTP errors. Hence, if all form submissions result in errors, the distinctness fraction will be 0.

While the exact details of *Sig* are less important, we enumerate the important properties we want from such a function. First, the signature should be agnostic to HTML formatting, since presentation inputs often simply change the layout of the web page. Second, the signature must be agnostic of term ordering, since result re-ordering is a common presentation operation. Third, the signature must be tolerant to minor differences in page content. A common source of differences are advertisements, especially on commercial sites. These advertisements are typically displayed on page margins. They contribute to the text on the page but do not reflect the content of the retrieved records and hence have to be filtered away. Lastly, the signature should not include the input values themselves. A used car search site that has no red Honda Civic cars for sale in the zip code 94107, is likely to have an error message "`No search results for Red Honda Civic in 94107!`" Likewise, the result page for a large fraction of the *Color Make Model Zip* queries will be "`No search results for` *Color Make Model* `in` *Zip*". The only difference between these pages are the search terms themselves and a signature that does not exclude the search terms is likely to deem them different and hence deem the corresponding template informative.

4.2 Searching for Informative Templates

To avoid testing each of the $2^n - 1$ possible templates, we develop a strategy that explores the space of candidate templates and tests only those likely to be informative.

Our strategy is to search through the space of templates in a bottom-up fashion, beginning from templates with single binding input. The main intuition leading to this strategy is that the informativeness of a template is very likely to be dependent on templates that it *extends*, i.e., has one additional binding input. If template T has dimension k and none of the k templates that it extends is informative, then T is unlikely to be informative as well.

The `ISIT` algorithm is shown in Figure 2. We start with candidate templates of dimension 1. The `GetCandidateInputs` method chooses the inputs that are to be considered, i.e., the select menus and the text boxes (if input values are known). The other inputs are set to default values. The `CheckInformative` method tests each of the candidate templates for informativeness as per Definition 1. If any template of dimension 1 is deemed informative, the `Augment` method constructs candidates of dimension 2 that have a super-set of it's binding inputs. Thus, the candidate templates are such that at least one of the templates they extend is known to be informative (but not necessarily both). Each of the new candidate templates is then tested to determine if it is informative. From the informative ones of dimension 2, we continue in a similar fashion to build candidate templates of dimension 3, and so on. We terminate when there are no informative templates of a given dimension.

We note that all candidates inputs are considered while augmenting a template. We could choose a more aggressive strategy where we only consider informative inputs, i.e. their corresponding template of dimension 1 was deemed informative. However, we believe that in practice such a strategy will (erroneously) omit some informative templates. It

```
GetInformativeQueryTemplates (W: WebForm)
  I:  Set of Input = GetCandidateInputs(W)
  candidates:  Set of Template =
        { T: Template | T.binding = {I}, I∈ I}
  informative:  Set of Template = φ
  while (candidates ≠ φ)
    newcands:  Set of Template = φ
    foreach (T: Template in candidates)
      if ( CheckInformative(T, W) )
        informative = informative ∪ { T }
        newcands = newcands ∪ Augment(T, I)
    candidates = newcands
  return informative

Augment (T: Template, I:  Set of Input)
  return { T' | T'.binding = T.binding ∪ {I},
          I∈ P, I ∉ T.binding }
```

Figure 2: Algorithm: Incremental Search for Informative Query Templates (`ISIT`)

is not uncommon to have forms with one primary input that is required to retrieve any results and other inputs that are essentially refinements. For example, a form with `make` and `color` can be such that the default value for `make` ensures that no records are returned by simply selecting a `color`. Hence, the template with binding input `color` is uninformative, while the one with `make` and `color` is informative.

We note that there can be cases where our bottom-up search might falter. For example, a form may require *both* `make` and `zip`, and neither of these inputs in isolation is informative. In practice, we can address this problem by testing templates of dimension 2 when none of dimension 1 are deemed informative.

`ISIT` enables us to incrementally explore the space of templates by testing relatively few. For example, as we show in Section 4.4, for forms with 6 inputs, a naive strategy tests 63 templates per form, while our algorithm, on average, tests only 17.43 templates (a 72% reduction) and finds 7.17 to be informative. Further, we generate only a small subset of the Cartesian product. For example, even for forms with 5 inputs, while the Cartesian product averages close to a trillion URLs per form, we generate only 10,000 per form.

Once our search terminates, we can add the URLs generated by all the informative templates to the search engine index. Note that, to avoid duplicated content, we only need to add those with distinct signatures.

4.3 Refinements to the basic algorithm

Fielding our algorithm on a large scale required that we refine it in several ways.

Limiting the search space: As already noted, templates that generate a very large number of URLs are unlikely to be informative. Hence, as a practical refinement, we set a threshold for the maximum number of URLs that can be generated from a template and do not consider those that generate more than the threshold. For our experiment, we set the threshold as 10,000. Likewise, we found that templates with dimension of more than 3 are almost never informative. Hence, in practice, we restrict the dimension of our candidate templates to 3. Note that the above restrictions are consistent with our desiderata for templates. In Section 4.4, we found that simply applying these restrictions (without the informativeness test) does reduce the number of URLs generated, but the informativeness test results in a further order of magnitude reduction.

Table 1: Distribution of HTML Forms by the number of inputs.

Num. Inputs	Num. Forms	Num. Inputs	Num. Forms
1	429271	6	1684
2	43776	7	1071
3	12987	8	418
4	7378	9	185
5	3101	10	129

Monotonic inputs: The informativeness test as described often fails on a particular class of presentation inputs. Specifically, forms can have inputs that restrict the number of results per page (e.g., 10, 20, or 30 job lists per page). The web pages for different input values can be very distinct from each other, but we would do not want to generate URLs for each of them (and select only one instead). Observe that increasing the number of results per page leads to monotonically increasing page length. To identify such inputs, we include the length of the generated web page in its signature. In addition to distinctness, we perform a monotonicity test for page lengths. Once we detect monotonicity, we select a single value for the input. Note that the smallest value might return too few records, while the largest value might return too many.

Sampling for informativeness: For the informativeness test, when the number of URLs that can be generated from a template is very large, we instead generate and analyze only a sample. We currently set the sample size at a fixed number, 200, based on our experiments.

Form-wide distinctness: While a template T_1 might result in web pages with distinct content, it is possible that the content is identical to those generated by a different template T_2. Such templates are of no additional value to the search index. To prune them, we consider form-wide distinctness in the informativeness test. Specifically, when counting the number of distinct signatures for a template, we do not include signatures generated by previously tested templates. We observed a significant effect when considering templates with more than one binding input – with $\tau = 0.2$, 36% of templates of dimension 2 that are deemed informative by simple distinctness are deemed uninformative by form-wide distinctness.

4.4 Experimental Results

In this section we describe a set of experiments that validate the claim that our algorithm is effective at predicting the desirable templates for an HTML form. To focus just on templates (and not value selection for text inputs), for now we only consider forms that have select menus. By manually inspecting a number of forms, we observed that select menus with fewer than 5 options are to a large extent presentation inputs. Hence, we do not consider select menus with less than 5 values as potential binding inputs.

We now compare our algorithm (`ISIT`) against other strategies. First, we show that we do not generate too many URLs. Second, we show that we efficiently traverse the search space of query templates. Third, we show that we do not generate too few URLs, i.e., we generate more than a simple conservative strategy, but the additional URLs generated are useful to the search engine. Fourth, we show that, in practice, we achieve good coverage of the underlying database. Finally, we try to understand the conditions when we are unable to find any informative templates.

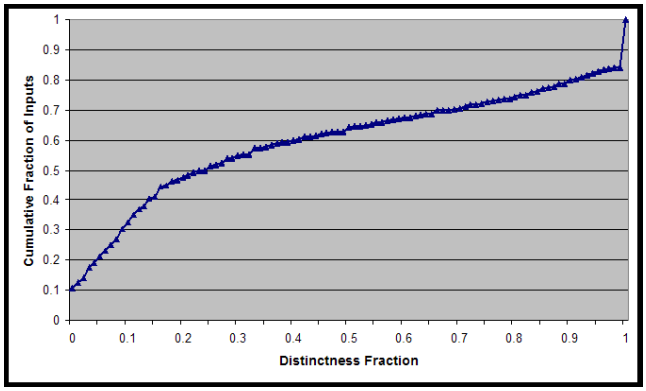

Figure 3: Cumulative Distribution of Distinctness Fraction. The line plots for each value of distinctness fraction d, the fraction of inputs with a distinctness fraction less than d.

Dataset: In the rest of this section, we consider a sample of 500,000 HTML forms for which our algorithm was able to successfully generate URLs, i.e., find at least one informative template. Table 1 summarizes the distribution of forms by their number of select menus. To the best of our knowledge, this is the first analysis of HTML forms on such a large scale.

Distinctness threshold τ: Definition 1 depends on the value of threshold τ. In order to determine an appropriate value, we considered the cumulative distribution of the distinctness fraction of all templates of dimension 1 (see Figure 3). We found that about 10% have a distinctness fraction of 0, i.e., all the URLs we generate result in HTTP errors. About 15% have a distinctness fraction of 1, i.e., each generated URL has potentially distinct content. There appear to be two distinct linear ranges in the rest of the distribution. For distinctness fraction below 0.2 there is a steep drop-off in the percentage of inputs (slope = 2), but above 0.2, there is a gentler increase (slope = 0.5). We believe that templates in the range below 0.2 are clearly uninformative – a large number of them generate only one distinct signature. Since we only consider inputs with 5 or more values, their distinctness fraction is less than 0.2. The threshold τ can be chosen to be any value above 0.2 with the specific choice determining the aggressiveness of our URL generation algorithm. In our experiments, we used a distinctness threshold of 0.25 and a form-wide distinctness threshold of 0.2 (both of which must be satisfied).

Scalable URL Generation: We compare the number of URLs generated per form by `ISIT` against three other algorithms. The first is the `CP` algorithm, which generates a URL for each entry in the Cartesian product of all input values. This will clearly create a huge number of URLs and will be impractical for larger forms. To perform a more realistic comparison for larger forms, we consider two other alternatives:

- the `TP` (Triple Product) algorithm: for forms with more than 3 inputs, `TP` generates URLs from all possible templates of dimension 3, i.e., for a form with n inputs, it with generate URLs from all $\binom{n}{3}$ templates.
- the `TPL` (Limited Triple Product) algorithm: `TPL` is a restriction of `TP` that only considers URLs from templates that generate fewer than 100,000 URLs.

Observe that `TP` and `TPL` correspond to the two restrictions

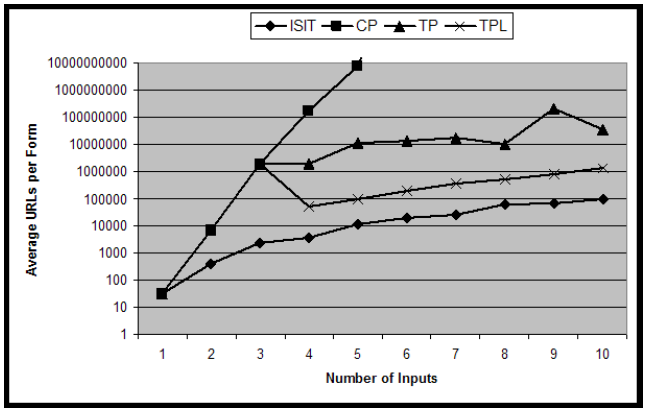

Figure 4: Comparison of the average number of URLs generated per form by the ISIT, CP, TP, and TPL algorithms. ISIT generates an order of magnitude fewer URLs per form than TPL, the best strategy that does not test for informativeness.

to the search space of templates described in Section 4.3, but applied to the CP algorithm. Hence, by comparing ISIT to TP and TPL, we are trying to quantify the effects of the informativeness test against simply choosing arbitrary templates of a given size. Note that ISIT and TPL only restrict the number of URLs *per template* to 100,000 – the total number generated from a form can be much larger.

Figure 4 compares the performance of the four algorithms for different numbers of form inputs. Each algorithm considers the same set of candidate binding inputs for each form, i.e., the filters that are used to restrict the binding inputs chosen by ISIT are also applied to the other two. All the algorithms are identical for forms with only one input, while CP, TP and TPL by definition generate the same number of URLs for forms with three or fewer inputs. TPL generates fewer URLs per form for $n = 4$ that $n = 3$ since for larger forms, it does not consider all templates of dimension 3 for $n \geq 4$ (but does so for $n = 3$).

We can draw the following conclusions. CP is impractical since it generates more than a billion URLs per form with 4 inputs and more than a million per form with only 3 inputs. TP and TPL generate fewer URLs by only considering triples, but even TPL generates more than 100,000 URLs per form with 5 inputs. ISIT is able to consistently generate an order of magnitude fewer URLs than TPL and even for forms with 10 inputs, the number of URLs generated is only 100,000.

Observe that the real benefit of ISIT is greater than that demonstrated in Figure 4, since our dataset does not include the forms for which ISIT found no template to be informative. The CP, TP, and TPL algorithms would naively (and most often in error) generate URLs for those forms too.

Figure 4 groups forms based on their number of inputs. However, the number of URLs generated by forms with the same number of inputs can vary greatly if the number of values that can be fed into the inputs differs significantly. To compensate for this variability, Table 2 groups forms based on the number of URLs generated by algorithm TPL. It can be clearly seen from Table 2 that ISIT generates far fewer URLs. Even for very large forms where TPL generates between 100,000 to 1 million URLs, ISIT only generates 18,799 URLs on average. In fact, the fraction of URLs generated halves with every subsequent bucket. Note that TPL fails

Table 2: Comparison of the average number of URLs generated per form by ISIT against TPL. Forms are grouped by the number of URLs generated by TPL.

TPL Gen. Bucket	Num. Forms	Avg. Gen. ISIT	Avg. Gen TPL	Fraction ISIT/TPL
1 - 9	95637	7.5	7.5	1.00
10 - 99	337613	26.4	28.9	0.91
100 - 999	36368	165	278	0.59
1000 - 9999	12706	930	3674	0.25
10000 - 99999	11332	4824	37605	0.12
100000 - 999999	5546	18799	280625	0.06
0	211	17858	fails	–

Table 3: Average number of templates tested and URLs analyzed per form. As the number of inputs increase, the number of possible templates increases exponentially, but the number tested only increases linearly, as does the number found to be informative.

Num. Inputs	Max Templates [dimension ≤ 3]	Average Templates Tested	Average Templates Informative	Average URLs Analyzed
1	1 [1]	1	1	23
2	3 [3]	2.82	1.76	136
3	7 [7]	5.38	3.04	446
4	15 [14]	8.80	3.75	891
5	31 [25]	12.66	5.93	1323
6	63 [41]	17.43	7.17	1925
7	127 [63]	24.81	10.62	3256
8	255 [92]	29.46	13.45	3919
9	511 [129]	35.98	15.80	4239
10	1023 [175]	41.46	17.71	5083

to generate any URLs for forms that have select menus so large that all templates of dimension 3 generate more than 100,000 URLs. ISIT is able to generate on average 17,858 URLs in these cases.

Efficient traversal of search space: Table 3 shows the average number of templates we test per HTML form and the number of templates that are found to be informative. If an algorithm were to naively consider all possible templates, then for a form with n inputs, we would have to test $2^n - 1$ templates. If we restrict the templates to be only those with dimension less than equal to 3, we would still have to test $\binom{n}{1} + \binom{n}{2} + \binom{n}{3}$ templates. However as Table 3 shows, ISIT exhibits an almost a linear rate of increase. Thus, we are in a position to test arbitrarily large forms.

Table 3 also shows the average number of URLs analyzed per HTML form. Recall that, to determine informativeness, a sample of the URLs that can be generated by template are downloaded and their contents analyzed. On average only 84 URLs are fetched per form site, and even for forms with 10 inputs the number is only 5083. Further, the web pages are fetched typically over a week thus not placing an undue burden on the form site.

Adequate URL generation: We compared ISIT with 1T that only generates URLs from templates of dimension 1. Clearly, such a strategy generates far fewer URLs. However, we found that there is a corresponding loss in accessibility to content. In what follows, we analyze the impact of the generated URLs on the Google.com query stream to compare the utility of URLs generated by templates of dimension 2 and 3 against those generated by 1T. We measure the impact by counting the number of times one of the URLs appears in the first 10 results for some query on the search engine.

To perform a fair comparison, we only considered forms for which URLs were generated from at least one informative template each of dimension 1, 2, and 3 (there were 3856 such forms in our sample dataset). Of the search results contributed by URLs generated from these forms, we found that templates of dimension 1 accounted for only 0.06 of the results, while those of dimensions 2 and 3 accounted for 0.37 and 0.57 respectively. Likewise, if we instead considered forms with informative templates of dimensions 1 and 2, but none of dimension 3 (22,872 forms in our sample), we found that the former accounted for 0.30 of the search results as compared to the latter which accounted for the remaining 0.70. Thus the URLs generated from larger templates do contribute significantly to web search results and these URLs should not be ignored.

Database Coverage: To get a rough idea for the coverage we obtain on web sites, we considered a set of sites for which we are able to determine the number of records they store (either by a specific query or where the number is published by the site). Table 4 shows how well `ISIT` performs on these sites. In all examples, we generate far fewer URLs than the Cartesian product, and in most of them we are able to retrieve close to all the records. In some cases (1 and 7), the manual inspection found the unrestricted queries to retrieve all records with a single query. However, in order to perform a fair comparison the corresponding URL was excluded from the ones `ISIT` generated.

It is easy to see from the table that the Cartesian product of the inputs is not correlated with the size of the underlying database. On the contrary, the number of URLs generated by `ISIT` is correlated with the database size. We are able to retrieve more than 50% of the records in all but one example (example 10 has far too big a database). Observe that in example 5, we retrieve more records than those published on the home page of the website (which is likely to be outdated).

In some examples, we generate more URLs than retrieved records. While this might seem excessive, observe that we can prune the generated URLs by eliminating those with duplicate page signatures. In fact in example 10, the largest template has a distinctness fraction of only 0.27. Hence, the number of URLs eventually selected into the search engine index is dramatically lesser.

Applicability of Informativeness: As a final question, we measure the applicability of the informativeness test across *all* forms. Hence, we applied our algorithm on all forms on the Google index that had at least one select menu with at least 5 values and no text inputs. We were able to find at least one informative template in 48% of the forms.

In order to determine the reasons for not finding any informative templates, we took a closer look at forms with only one select menu, i.e. only one template. We found that for 18% of the failing forms, the generated URLs all result in HTTP errors, and for a further 54%, the generated URLs all result in web pages that not only have equivalent signatures, but are also byte-wise equivalent. An inspection of the forms in these two categories indicates Javascript to be the predominant reason for failure. For these forms, the form processing is entirely performed by Javascript and hence the `get` URL (as described in Section 2) is ignored. For 1.8%, the generated URLs could not be crawled due to robots.txt entries that blocked our crawler. For 8.3%, there is only one distinct signature generated, which implies that the input is likely to be a presentation input. For 1.7%, the input was found to be monotonic (and hence a presentation input). Finally, for the remaining 15%, there is more than one signature, but not enough for the template to be informative.

The current inability to handle Javascript is not a fundamental limitation of our algorithm and the forms can be processed using a Javascript simulator. If we were to exclude the Javascript affected forms and the uncrawlable forms, we are in fact able to generate URLs for 80% of all forms with one select menu.

5. GENERATING INPUT VALUES

A large number of HTML forms have text boxes. In addition, some forms with select menus require valid values in their text boxes before any results can be retrieved.

Text boxes are used in two different ways. In the first, the words entered in the text box are used to retrieve all documents in a backend text database that contain those words. Common examples of this case are searching for books by title or author. In the second, the text box serves as a selection predicate on a specific attribute in the `where` clause of a SQL query over the backend database. The values either belong to a well-defined finite set, e.g., zip codes or state names in the US, or can be an instance of a continuous data type, e.g., positive numbers for prices or integer triples for dates. For our purposes, this underlying distinction induces a division of text boxes into two types: *generic* and *typed*. Invalid entries in typed text boxes generally lead to error pages and hence it is important to identify the correct data type. Badly chosen keywords in generic text boxes can still return some results and hence the challenge lies in identifying a finite set of words that extracts a diverse set of result pages.

Section 5.1 describes an algorithm to generate a keywords for a generic text box, and we evaluate it in Section 5.2. We discuss typed text boxes in Section 5.3.

5.1 Generic text boxes

We first consider the problem of identifying good candidate keywords for generic text boxes. Conceivably, we could have designed word lists in various domains to enter into text boxes. However, we quickly realized that there are far too many concepts and far too many domains. Furthermore, for generic text boxes, even if we identified inputs in two separate forms to be the same concept in the same domain, it is not necessarily the case that the same set of keywords will work on both sites. The best keywords often turn out to be very site specific. Since our goal was to scale to millions of forms and multiple languages, we required a simple, efficient and fully automatic technique.

We adopt an iterative probing approach to identify the candidate keywords for a text box. At a high level, we assign an initial seed set of words as values for the text box and construct a query template with the text box as the single binding input. We generate the corresponding form submissions and extract additional keywords from the resulting documents. The extracted keywords are then used to update the candidate values for the text box. We repeat the process until either we are unable to extract further keywords or have reached an alternate stopping condition. On termination, a subset of the candidate keywords is chosen as the set of values for the text box.

Iterative probing has been proposed in the past as a means to retrieving documents from a text database [1, 13]. How-

Table 4: Examples of HTML forms comparing the URLs generated and the number of records retrieved against the Cartesian product and the actual size of the database.

No.	URL with HTML form	Cartesian product	URLs Generated	Estimated Database	Records Retrieved
1	http://www2.imm.dtu.dk/pubdb/public/index_public.php	53550	1349	4518	4518
2	http://dermatology.jwatch.org/cgi/archive	32400	179	3122	1740
3	http://www.kies.com.au/listings.php	129654	81	62	61
4	http://aarpmagazine.org/food/recipeguide	150822	365	314	273
5	http://www.shadetrees.org/search.php	313170	181	1699	1756
6	http://www.dasregistry.org/listServices.jsp	2 million	627	295	267
7	http://www.nch.org.uk/ourservices/index.php?i=402	2.5 million	34	27	27
8	http://www.lodgerealestate.co.nz/?nav=rentsearch	4.7 million	26	87	74
9	http://www.pipa.gov.ps/search_db.asp	68 million	1274	409	409
10	http://www.biztransact.com/	257 million	13,209	100,000+	10,937
11	http://projects.bebif.be/enbi/albertinerift/common/search	948 billion	2406	2287	1257
12	http://oraweb.aucc.ca/showcupid.html	3.2 trillion	34	27	27

ever, these approaches had the goal of achieving maximum coverage of specific sites. As a consequence, they employ site-aware techniques. For example, in [1] the expression for the number of results returned for a query is used to select keywords, while in [13] individual result documents are retrieved from the result pages to maintain correlations between candidate keywords. We now discuss the techniques we employ to adapt iterative probing to our context.

Seeds and candidate keywords selection: We start by applying the informativeness test on the template for the text box assuming a seed set of input values. Our experiments indicate that if the template fails the informativeness test, then text box is unlikely to be a generic text box. Since we want to cover all possible languages, we cannot start with from a dictionary of terms. Hence, we select the seeds from the words on the form page. Likewise, since we do not extract individual result records or documents, we select additional keywords from the words of the web pages generated by the form submissions.

We select words from a page by identifying the words most relevant to its contents. For this we use the popular Information Retrieval measure TF-IDF [16]. Briefly, the term frequency (TF) measures the importance of the word on that particular web page. Suppose a word w occurs $n_{w,p}$ times a web page p and there are a total of N_p terms (including repeated words) on the web page, then $tf(w,p) = \frac{n_{w,p}}{N_p}$. The inverse document frequency (IDF) measures the importance of the word among all possible web pages. Suppose w occurs on d_w web pages and there are a total of D web pages in the search engine index, then $idf(w) = \log \frac{D}{d_w}$. TF-IDF balances the word's importance on the page with its overall importance and is given by $tfidf(w,p) = tf(w,p) \times idf(w)$.

For the seeds, we select the top $N_{initial}$ words on the form page sorted by their TF-IDF scores. For the candidate keywords for iteration $i+1$, suppose that W_i is the set of all web pages generated and analyzed until iteration i. Let C_i be the set of words that occur in the top N_{probe} words on any page in W_i. From C_i, we eliminate words

- if they have so far occurred in too many of the pages in W_i (say 80%), since they are likely to correspond to boiler plate HTML that occurs on all pages on the form site (e.g., menus, advertisements), or
- if they occur only on one page in W_i, since they can be nonsensical or idiosyncratic words that are not representative of the contents of the form site.

The remaining keywords in C_i are the new candidate keywords for iteration $i+1$. The choice of $N_{initial}$ and N_{probe} determines the aggressiveness of the algorithm. By choosing lower values, we might not be able to extract sufficient keywords, but very high values can result in less representative candidate keywords. Our experiments indicate $N_{initial} = 50$ and $N_{probe} = 25$ to be good values.

Text box value selection: To limit the number of URLs generated from the form, we place restrictions on the maximum number of keywords for a text box. While we would like to choose the subset that provides us with the most Deep-Web coverage of the form site, we cannot maintain accurate models of estimated coverage (unlike [13, 18]). Instead we use a strategy that is simpler, and tries to ensure the diversity among selected keywords.

Let $P(w)$ be the top N_{probe} words on the web page corresponding to the candidate keyword w. We first cluster the candidate keywords based on their $P(w)$'s and randomly select one candidate keyword from each cluster. The rest of the keywords within each cluster are believed to have similar content and are hence omitted. We sort the chosen candidate keywords based on the page length of the corresponding form submission and proceed down the list of candidate values in decreasing order of page length. If S is the set of values already selected, then we select the candidate w_i into S only if $P(w_i) \cap (\cup_{w \in S} P(w)) \geq k$, where k is a small number. We start with $k = 5$ and re-iterate through the sorted list of candidate keywords with decreasing values of k until we have selected the desired number of keywords for the text box.

We note that placing a single maximum limit on the number of keywords per text box is unreasonable because the contents of form sites might vary widely from a few to tens to millions of results. We use a back-off scheme to address this problem. We start with a small maximum limit per form. Over time, we measure the amount of search engine traffic that is affected by the generated URLs. If the number of queries affected is high, then we increase the limit for that form and restart the probing process.

5.2 Experimental Results

HTML forms can have multiple text boxes. In our study, we only consider one text box per form. Based on a manual analysis, we believe that in a large fraction of forms, the most likely generic text box is the first text box. Hence, we apply the iterative probing approach to the first text box.

In the experiments below, we consider a different dataset of 500,000 HTML forms for which were able to generate key-

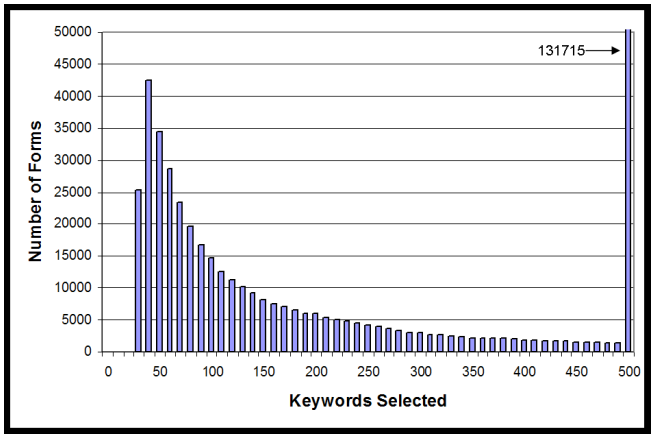

Figure 5: Distribution of HTML forms by the number of keywords selected.

words. For each form, we try to select 500 keywords. We stop extracting candidate keywords either after 15 iterations or once we have extracted 1500 candidates for keyword selection, whichever occurs first. We extract more than 500 keywords, so that we have a larger and potentially more diverse set of candidate keywords from which to select the final set of values for the text box.

We first consider the distribution of candidate keywords our algorithm generates. We then demonstrate that we are able to achieve good coverage of the underlying database. We then consider the interaction between the text boxes and the select menus that might be present in the same form. Finally, we show how the URLs we surface can serve as seeds for a web crawler that will then organically increase the database coverage over time.

Keyword Generation: In the experiments, the distribution of reasons for which iterative probing terminated is: for 70% of the forms no new keywords can be extracted after a few iterations, for 9% of the forms the maximum number of iterations is reached, and for the remaining 21% termination occurs once sufficient keywords have been extracted.

Figure 5 shows the distribution of HTML forms by the number of keywords selected for the form. The histogram is constructed by bucketing forms based on the number of keywords selected. We make two observations. First, there are no forms with 0-20 keywords because we consider text boxes with fewer that 20 extracted keywords to be uninformative. These text boxes are unlikely to be generic search boxes. Second, the number of forms in the last bucket is significantly more because it groups all forms with 500 or more candidate keywords into a single bucket. A more complete analysis of the distribution of keywords extracted and its implications will be considered in future work.

Database Coverage: Table 5 lists examples of real form sites whose size was possible to determine either by inspection of by submitting a carefully designed query. The table shows the performance of our algorithm in extracting keywords for the sites. In each of these examples, a maximum of 500 keywords were selected for the first text box in the form. We consider the URLs generated using the selected keywords for the text box and default values for all other inputs. To estimate our coverage, we counted the number of database records retrieved by manually inspecting the site to determine patterns that identify unique records on the result web pages.

First, we consider only the contents on the first results page (column `first` in Table 5), since these correspond directly to our generated URLs. We observe that in examples 1 and 2, when the databases are small (less than 500 records), we are able to reach almost all database records. Further, our algorithm terminates with fewer keywords than the estimated database size. As the estimated database sizes increase (examples 3 to 8), we stop at 500 selected keywords and we are only able to get to a fraction of the total records. In general, while the absolute number of records retrieved increases, the fraction retrieved decreased. Not surprisingly, we get to more records when there are more results per page. As already highlighted, our algorithm is language independent – example 9 is a Polish site, while 10 is a French one. In all, the results in Figure 5 include forms in 54 languages.

Text boxes and select menus: The second column in Table 5 shows the number of results we obtained from considering only select menus in the forms. The table shows that the coverage of the select menus is much smaller, therefore it is important to consider both select menus and text boxes. It is important to note that the records extracted from text boxes did not necessarily subsume the ones extracted from select menus.

Once we have the values for a text input, we can treat them similar to select menus as described in Section 4. We now compare the relative contribution of URLs generated from text boxes and those generated from select menus to the resulting query traffic. As in Section 4.4, we count the number of times the URLs appear in the first 10 results for some query on the search engine. We consider the 1 million forms in the datasets in this section and in Section 4.4. The URLs we generate fall into three categories: those generated using templates having (1) only one text box as a binding input, (2) one or more select menus, and (3) one text box and one or more select menus. Overall, we find that URLs in these categories contribute to search results in the ratio $\langle 0.57, 0.37, 0.06 \rangle$. The ratio is $\langle 0.30, 0.16, 0.54 \rangle$ when we restrict our attention to forms that generated URLs from both text boxes and select menus. Clearly, we need to consider text boxes, select menus and their combinations.

Seeding the web crawler: Given our surfaced URLs the search engine's web crawler will automatically over time pursue some of the outgoing hyperlinks on the corresponding web pages, e.g., follow the *Next* links. Hence, in order to get an estimate of the database coverage assuming such a crawling pattern, we also included the web pages that are outlinks from the first result page (column `first++` in Table 5). As can be seen, our coverage is much greater. Observe that while in example 7, the coverage only doubles, but in example 9 it is significantly more. This is because in the former, there is only a single *Next* link, while in the latter, there are links to multiple subsequent result pages.

5.3 Typed text boxes

One of the main observations from our work is that there are relatively few types that, if recognized, can be used to index *many* domains, and therefore appear in many forms. For example, the type zip code is used as an input in many domains including cars, public records and real-estate, and the type date is used often as an index of many domains, such as events and articles archives. We describe an experiment that validates this observation.

The two ideas we build on are the following. First, a

Table 5: Examples of HTML forms with text boxes comparing the the actual size of the database (number of records) against the number of URLs generated and the number of records retrieved: (first) on the first results page when using only the text box, (select) on the first page using only select menus, and (first++) on the first page and the pages that have links from it when using only the text box,

No.	Form URL	Estimated Database	Keywords Selected	Results Per Page	Records Retrieved		
					first	select	first++
1	http://kbase.gofrugaltech.com/	382	169	20	348	n.a.	381
2	http://nsidc.org/data/index.html	506	355	20	475	102	478
3	http://www.hhmi.org/news/search.html	1700	500	25	1485	320	1585
4	http://ww.azom.com/	4700	500	10	2136	n.a.	3124
5	https://secure.financeweek.co.uk/cgi-bin/iadmin.cgi?page=68	5000	500	50	4118	1316	4118
6	http://www.angis.org.au/Databases/BIRX/	18200	500	100	5315	n.a.	n.a.
7	http://english.ibd.com.cn/databank.asp	21200	500	10	2661	306	4697
8	http://federalgovernmentjobs.us/	60300	500	10	3096	n.a.	12127
9	http://praca.gratka.pl/	15800	500	20	4012	290	11043
10	http://www.rhdsc.gc.ca/fr/ministeriel/recherche/index.shtml	21800	500	200	13324	724	19332

Table 6: Detecting Input Type: Each entry records the results of applying a particular type recognizer (rows, e.g., city-us, date) on inputs whose names match different patterns (columns, e.g., `*city*`, `*date*`). `*` includes inputs that do not match any of the patterns. Likewise, the row not recognized includes inputs that are not recognized by any of the identifiers.

	`*city*`	`*date*`	`*price*`	`*zip*`	`*`
city-us	**60**	6	4	14	113
date	3	**46**	12	8	7
price-small	3	6	**40**	4	18
price-large	2	8	**34**	0	12
zip-us	4	2	13	**136**	3
generic	8	0	2	3	**392**
not recognized	92	295	369	111	300
total	172	363	475	276	845

typed text box will produce reasonable result pages only with type-appropriate values. We use this to set up informativeness tests using known values for popular types. We consider finite and continuous types. For finite types, e.g., zip codes and state abbreviations in the US, we can test for informativeness using a sampling of the known values. For continuous types, we can test using sets of uniformly distributed values corresponding to different orders of magnitude. Second, popular types in forms can be associated with distinctive input names. We can use such a list of input names, either manually provided or learned over time (e.g., as in [6]), to select candidate inputs to apply our informativeness tests on. We conducted the following experiment with four types: US zip codes, US city names, prices and dates. For price, we consider two sub-types, price-small (0 - 5,000) and price-large (50,000 - 500,000), with the former targeting product sites and the latter real-estate sites. We consider a dataset of about 1400 forms chosen randomly such that there are at least 200 forms each with a text box whose name matches the pattern `*city*`, `*date*`, `*price*`, and `*zip*`. For each of these forms we consider the first text box and other text boxes whose name match any of the mentioned patterns. On each of the selected inputs, we perform the informativeness tests for *all* fives types as well as the informativeness test for a generic text box (using the seed keywords picked from the form page).

Table 6 shows our results. The type recognized for an input is the one that has the highest distinctness fraction. However, none of the types are deemed recognized if the best distinctness fraction is lesser than 0.3. The row *not recognized* indicates that the text boxes that were not recognized with any of the types.

The table shows that when we exclude *not recognized*, we find that the vast majority of type recognitions are correct. We make the following observations. First, some `*price*` inputs get recognized by *zip-us*. This is not surprising, since zip code values being integers are valid entries for price. Second, some `*zip*` inputs get recognized by *city-us* and some `*city*` inputs get recognized by *zip-us*. On closer inspection, these inputs turn out to be ones that accept either city names or zip codes for location information. Third, a number of `*` inputs get recognized by *city-us*. This turns out to be the case because these inputs are generic search boxes and city names turn out are after all English words that seem work well for those sites. Fourth, `*date*` inputs are particularly hard to recognize, since there are multiple possible date formats, and since we used only one date format (mm/dd/yy) we do not recognize as many inputs. Experimenting with multiple candidate formats is likely to improve performance. Lastly, we found that the informativeness test can be used to identify input names associated with specific types, e.g., the 3 `*` inputs recognized by *zip-us* have the name *postalcode*.

Our results indicate that text box types can be recognized with high precision. We also found that, of all the English forms in our web index that are believed to be hosted in the US, as many as 6.7% have inputs that match the patterns we mention. This leads us to believe that a degree of *type*-specific modeling can play an important role in expanding Deep-Web coverage. Recent work on understanding forms, e.g., [8, 19, 17], can potentially be leveraged to identify candidate inputs for domain specific types that can then be verified using the informativeness test.

6. RELATED WORK

As noted earlier, in specific domains we can use data integration techniques to create vertical search engines. We implement such an engine by constructing semantic mappings from a mediated form to collections of forms with a specific domain. Structured queries are routed from the mediated form to the relevant sources forms. Recent work has focused on easing the process of building and maintaining vertical search engines by automatically identifying similar form inputs and selecting the best mediated form, e.g., [8, 19, 17]. The difficulty in pre-defining all possible domains

for forms on the Internet makes it hard to create mappings as well as routing queries, thereby making it infeasible for web search. However, in cases where the domains involve complex modeling of multiple inter-related attributes, e.g., airline reservations, or when forms use the `post` methods (see Section 2), virtual integration remains an attractive and also only possible solution.

Research on text databases has focused on two problems: (1) computing keyword distributions that summarize the database contents to facilitate the selection the correct database for any user query, and (2) extracting the text documents that are in the database using the restricted keyword interface. In order to create keyword summaries, a probing approach was proposed in [4] where, starting from an initial set of keywords, more are extracted by repeatedly querying the text database with a known set of keywords. In [7], the number of documents retrieved from a text databases for well chosen probes were used to classify and organize text databases into a topic hierarchy, while in [10] keyword summaries for text databases are from the documents retrieved from well chosen probes.

Our iterative probing and keyword selection approach is similar in spirit to those proposed in [1] and [13]. As already mentioned in Section 5.1, these works have the goal of achieving coverage on specifically chosen sites and hence employ site-specific techniques to varying degrees. We have shown that the iterative probing approach can in fact be used on a large scale with absolutely no site-specific processing to select keywords for generic text inputs.

Multi-input forms are considered in [18] and [14]. However, in [18], ultimately only single-attribute value queries are chosen, albeit the attributes chosen in different queries can be different. Here again, the authors assume that they can extract records and their different attributes from the result pages and thus maintain reasonable models of coverage. In [14], to evaluate the each input, the authors use a simple measure of the fraction of non-error pages. Further, they assume the multiple-inputs to be independent and try to select specific URLs from the Cartesian product of inputs. In essence, they assume the Cartesian product to be informative and focus on selecting a sample from it.

In [3], the authors propose a specific approach for forms with zip code inputs and use a coverage metric that relies on prior knowledge of zip code distributions. We have shown that our informativeness measure can be re-used to identify the types of zip codes as well as other inputs.

In [17], the authors use occurrences of input keywords on result pages to map text boxes to concepts within a domain. However, their focus is disambiguating inputs when the domain of the form is already known. Our informativeness test is very different and we target cross-domain types.

7. CONCLUSION

We described the technical innovations underlying the first large-scale Deep-Web surfacing system. The results or our surfacing are currently enjoyed by millions of users per day world-wide, and cover content in over 700 domains, over 50 languages, and from several million forms. The impact on our search traffic is a significant validation of the value of Deep-Web content.

Our work illustrates three principles that can be leveraged in further investigations. First, the test of informativeness for a form input can be used as a basic building block for exploring techniques for indexing the Deep Web. Second, we believe efforts should be made to crawl well chosen subsets of Deep-Web sites in order to maximize traffic to these sites, reduce the burden on the crawler, and alleviate possible concerns of sites about being completely crawled. Third, while devising domain-specific methods for crawling is unlikely to scale on the Web, developing heuristics for recognizing certain common *data types* of inputs is a fruitful endeavor. We believe that building on these three principles it is possible to offer even more Deep-Web content to users.

Two more specific directions for future work are to handle forms powered by Javascript and to consider more carefully dependencies between values in different inputs of a form.

8. REFERENCES

[1] L. Barbosa and J. Freire. Siphoning hidden-web data through keyword-based interfaces. In *SBBD*, 2004.

[2] M. K. Bergman. The Deep Web: Surfacing Hidden Value. *Journal of Electronic Publishing*, 2001.

[3] S. Byers, J. Freire, and C. T. Silva. Efficient acquisition of web data through restricted query interfaces. In *WWW Posters*, 2001.

[4] J. P. Callan and M. E. Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems*, 19(2):97–130, 2001.

[5] Cars.com FAQ. http://siy.cars.com/siy/qsg/faqGeneralInfo.jsp#howmanyads.

[6] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *SIGMOD*, 2001.

[7] L. Gravano, P. G. Ipeirotis, and M. Sahami. QProber: A system for automatic classification of hidden-web databases. *ACM Transactions on Information Systems*, 21(1):1–41, 2003.

[8] B. He and K. Chang. Automatic Complex Schema Matching across Web Query Interfaces: A Correlation Mining Approach. *TODS*, 31(1), 2006.

[9] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the Deep Web: A survey. *Communications of the ACM*, 50(5):95–101, 2007.

[10] P. G. Ipeirotis and L. Gravano. Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection. In *VLDB*, pages 394–405, 2002.

[11] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based Schema Matching. In *ICDE*, 2005.

[12] J. Madhavan, S. Jeffery, S. Cohen, X. Dong, D. Ko, C. Yu, and A. Halevy. Web-scale Data Integration: You can only afford to Pay As You Go. In *CIDR*, 2007.

[13] A. Ntoulas, P. Zerfos, and J. Cho. Downloading Textual Hidden Web Content through Keyword Queries. In *JCDL*, pages 100–109, 2005.

[14] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *VLDB*, pages 129–138, 2001.

[15] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering Queries Using Templates with Binding Patterns. In *PODS*, 1995.

[16] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. 1983.

[17] J. Wang, J.-R. Wen, F. Lochovsky, and W.-Y. Ma. Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. In *VLDB*, 2004.

[18] P. Wu, J.-R. Wen, H. Liu, and W.-Y. Ma. Query Selection Techniques for Efficient Crawling of Structured Web Sources. In *ICDE*, 2006.

[19] W. Wu, C. Yu, A. Doan, and W. Meng. An Interactive Clustering-based Approach to Integrating Source Query Interfaces on the Deep Web. In *SIGMOD*, 2004.