

Siphoning Hidden-Web Data through Keyword-Based Interfaces

Luciano Barbosa¹

Juliana Freire^{1,2}

¹Department of Computer Science & Engineering
OGI/OHSU

²School of Computing
University of Utah

{lab,juliana}@cse.ogi.edu

Abstract

In this paper, we study the problem of automating the retrieval of data hidden behind simple search interfaces that accept keyword-based queries. Our goal is to automatically retrieve all available results (or, as many as possible). We propose a new approach to siphon hidden data that automatically generates a small set of representative keywords and builds queries which lead to high coverage. We evaluate our algorithms over several real Web sites. Preliminary results indicate our approach is effective: coverage of over 90% is obtained for most of the sites considered.

1. Introduction

The volume of electronically available information is growing at a startling pace. Increasingly, databases are made available on the Web, from product catalogs and census data to gene databases. This information is often *hidden*, placed behind HTML forms [15] and Web services interfaces [19, 21], and it is only published, on demand, in response to users' requests.

Information in the hidden Web is often *hidden* for good reasons. Placing data behind form or Web services interfaces allows users to retrieve just the content they want, which can be more convenient than sifting through a large file or a large number of files. It also prevents unnecessary overload on the Web server for transferring large files. These interfaces, however, can be quite restrictive, disallowing *interesting* queries and hindering data exploration.

This is a serious limitation, especially since most of the Web-accessible information is hidden, and a significant portion of this information is of high-quality [18]. In a 1998 study, Lawrence and Giles [16] estimated that 80% of all the data in the Web could only be accessed via form interfaces. A more recent study by BrightPlanet [4] estimates an even bigger disparity: the hidden Web contains 7,500 terabytes of information and is 400 to 500 times larger than the visible Web.

Many applications need, and others could greatly benefit from, a more flexible and prompt access to hidden data. For example, a data integration application may need to pose queries that are not directly supported by the search interface provided; for data mining, performance may be significantly improved if the data is materialized locally; and by indexing high-quality hidden content, a topic-specific search engine may return higher-quality answers.

Not surprisingly, retrieving and querying hidden-Web data is a problem that has attracted a lot of attention (see e.g., [10, 18, 13, 14]). Scripts (or wrappers) can be handcrafted to retrieve

data from a particular site – these scripts can be created either manually (e.g., using a high-level languages such as Perl) or semi-automatically using wrapper generators (see e.g., [2, 10]). While scripts are effective and can be made efficient, they may require significant human input. In addition, since scripts have to be designed specifically for a given Web site and search interface, this approach has limited scalability.

An alternative, and more scalable approach is to use a hidden-Web crawler (HWC). The key problem that must be addressed by a HWC is how to automatically provide meaningful value assignments to the attributes of forms it encounters while crawling. Some form elements, such as pull-down lists, actually expose the set possible input values, which can be automatically submitted by the crawler. However, for open-ended attributes, such as text fields, knowledge of the domain is required and the set of possible values must be supplied to the crawler. Consequently, HWCs still require significant human input and their performance is highly dependent on the quality of the input data [18]. While progress has been made on automatically filling out forms, existing proposals [18, 14] focus exclusively on retrieving data from structured data sources through multi-attribute form interfaces. For example, HiWe [18] ignores forms with fewer than three attributes.

In this paper, we study the problem of automating the retrieval of data hidden behind simple search interfaces that accept keyword-based queries. These interfaces are commonplace on the Web. Although they have become popular for searching document collections, they are also being increasingly used for structured databases [5, 1] – in addition to structured (advanced) search, online databases often provide a simpler keyword-based search facility.

Keyword-based interfaces simplify querying because they do not require detailed knowledge of the schema or structure of the underlying data. If they make it easier for humans, what about for HWCs? We set out to investigate if and how it is possible to automatically retrieve all the available results in a collection through a keyword-based interface. We developed sampling-based algorithms that automatically discover keywords which result in high recall; and use these keywords to build queries that *siphon* all available results in the database (or, as many as possible). We evaluated our algorithms over several real Web sites. Preliminary results are promising: they indicate our approach is effective and it is able to obtain coverages of over 90% for most of the sites considered.

Outline. The rest of the paper is organized as follows. In Section 2 we give an overview of the main problems involved in retrieving data behind search interfaces and describe our approach to siphon these data. Experimental results are discussed in Section 3. Related work is reviewed in Section 4. We conclude in Section 5 with directions for future work.

2. Using Sampling to Retrieve Hidden Data

In order to automatically retrieve data (results) through a keyword-based interface, it is necessary to determine which queries to issue and which keywords to use. A naïve solution would be to issue a query for each word in the dictionary. This solution, however, leads to unacceptable performance due to the large number of unnecessary queries with possibly overlapping results. The ideal would be to determine a *small* number of queries that retrieve *all* the results.

Instead of blindly issuing queries, we propose a sampling-based approach to discover words that result in high coverage. The intuition is that words in the actual database or document

Algorithm 1 SampleKeywords(URL,form)

```
1: page = get(URL);
2: // retrieve the first set of results
3: initialKeywordList = generateList(page);
4: word = nextHigherOccurrency(initialKeywordList);
5: resultPage = submitQuery(form,word);
6: while resultPage == errorPage do
7:   word = nextHigherOccurrency(initialKeywordList);
8:   resultPage = submitQuery(form,word);
9: end while
10: // initialize keyword list
11: acceptStopword = checkStopword(form);
12: if acceptStopword then
13:   keywordList = generateListWithStopWords(resultPage);
14: else
15:   keywordList = generateList(resultPage);
16: end if
17: // iterate and build list of candidate keyword/occurences
18: numSubs = 0;
19: while numSubs < maxSubs do
20:   // randomly selects a word
21:   word = selectWord(keywordList);
22:   resultPage = submitQuery(form,word);
23:   // adds new keywords, and updates the frequency of existing keywords
24:   if acceptStopword then
25:     keywordList += genListWithStopWords(resultPage);
26:   else
27:     keywordList += genList(resultPage);
28:   end if
29:   numSubs++;
30: end while
31: return keywordList;
```

Algorithm 2 ConstructQuery(keywordList,form)

```
1: numSubs = numWords = totalBefore = totalCurrent = 0;
2: query = queryTemp = null;
3: while (numSubs < maxSubs) && (numWords < maxTerms) do
4:   // selects word with highest frequency
5:   word = nextElement(listOfOccurrency);
6:   queryTemp = addWordToQuery(query,word);
7:   page = submit(form, queryTemp);
8:   totalCurrent = getNumberOfResults(page,form);
9:   if (totalBefore * minimumIncrease) <= totalCurrent then
10:    query = queryTemp;
11:    totalBefore = totalCurrent;
12:    numberWords++;
13:   end if
14:   numSubs++;
15: end while
```

collection¹ are more likely to result in higher coverage than randomly selected words. Our approach consists of two phases: sample the collection to select a set of high-frequency keywords (Algorithm 1, *SampleKeywords*); and use these high-frequency keywords to construct a query that has high coverage (Algorithm 2, *ConstructQuery*).

As described in Algorithm 1, probe queries are issued to learn new keywords from the contents of the query results and their relative frequency with respect to all results retrieved in this phase.² The first step is to retrieve the page where the search form is located, select and submit a keyword (lines 1–16). Once a results page is successfully retrieved, the algorithm builds a list of candidate keywords by iteratively: submitting a query using a selected word (line 22); and using the results to insert new high-frequency words into the candidate set, as well as to update the frequencies of existing keywords (lines 23–28).

The candidate (high-frequency) keywords are input to Algorithm 2, which uses a greedy strategy to construct the query with the highest coverage. It iteratively selects the keyword with highest frequency from the candidate set, and adds it to the query if it leads to an increase in coverage. Note that although the query construction phase can be costly, once a high-coverage query is determined, it can be re-used for later searches, e.g., a hidden-Web search engine can use the query to refresh its index periodically.

These algorithms involve multiple choices: how to set the number of result pages retrieved; how to select keywords; when to stop. In what follows, we discuss our choices, the trade-offs, and the issues involved in a practical implementation.

Selecting keywords. The initial keyword can be selected from many sources. For example, from the set of keywords constructed by a focused crawler [9] on a topic related to the search interface. It has been shown, however, that the choice of initial term has little effect on the final results and on the speed of learning, as long as the query returns some answers [8]. In our implementation, we select a word from the initial page, where the form is located. As described above, the *SampleKeywords* algorithm proceeds to find additional keywords by iteratively sub-

¹We use database and document collection interchangeably in the remainder of this paper.

²This algorithm is a variation of the algorithm proposed in [8].

mitting queries using keywords obtained in previous iterations. Since our goal is to retrieve as many results as possible, a simple strategy is to also issue queries using stopwords (e.g., a, the), since their frequency in the collection is likely to be high. As we discuss in Section 3, higher coverages are indeed obtained by the algorithm if stopwords are indexed.

The response page for a query may contain information that is not relevant to the actual query (e.g., ads, navigation bars that are part of the site template), and this information may negatively impact the keyword selection process. Thus, it may be beneficial to remove these extraneous pieces of information before selecting the candidate words and computing their frequencies. A simple strategy is to discard the HTML markup. More sophisticated solutions are possible. A *smart* and generic wrapper, such as the response analysis module of HiWe [18], can be used to extract only the results list in the page. In Section 3, we discuss how these choices influence the effectiveness of our approach.

Stopping condition. While building the keyword sample, an important question is how big the sample needs to be in order to achieve high coverage – how many iterations should the Algorithm 1 go through. As we discuss in Section 3, the ideal number of iterations depends on the collection.

For constructing the coverage query, there is also a choice of when to stop. Algorithm 2 iterates until it gathers `maxTerms` keywords, or `maxSubmissions` probe queries are submitted (see Algorithm 2, line 3). The best choices for these parameters are collection-dependent.

Since the cost of running both algorithms is proportional to the number of requests issued, it is desirable to keep the number of requests to a minimum. Thus, it is crucial that *good* stopping conditions be determined.

Determining the number of results. Often, search interfaces return the total number of answers in the results. Heuristics can be developed to locate and extract this information (e.g., extract the number close to the string `results`). However, in some sites, the number of results returned is only an approximation. Google, for example, returns a rough estimate of the total number of results for a given query. If this approximation deviates too much from the actual number, the quality of the selected keywords is likely degrade.

Detecting error pages. While issuing the probe queries, it is possible that they lead to errors and no results are returned. To prevent error pages from negatively impacting the selection of keywords, it is important that they be automatically identified. In our implementation, we follow the simple heuristic proposed in [11]: issue queries using *dummy* words (e.g., `ewrwdwewdwddasd`) and record the results in an error template. As queries are issued, their results can be checked against the error template (see Algorithm 1, line 6).

Detecting interface characteristics. In order to get the highest coverage for a given collection or database, the ideal would be understand the characteristics of the search interface (e.g., indexing choices, allowed queries, collection size) and tailor the various choices of the siphoning algorithm to the collection. However, this is often an unreasonable assumption, especially in non-cooperative environment, where Web sites do not provide this information; and for large-scale tasks such as crawling the hidden Web. Instead, techniques are needed to automatically infer these properties. For example, in order to detect whether stopwords are indexed, a set of probe queries with stopwords can be issued and if they yield an error page (Algorithm 1, line 11), one can safely assume stopwords are not indexed.

Table 1: Description of sites used in the experiments

Site	Size (number of results)	Description
nwfusion.com – Network World Fusion	60,000	News information about information technology
apsa.org – American Psychoanalytic Association	34,000	Bibliographies of psychoanalytic literature
cdc.gov – Centers for Disease Control and Prevention	461,194	Health-related documents
epa.gov – Environment Protection Agency	550,134	Environment-related documents
georgetown.edu – Georgetown University	61,265	Search interface to the site
chid.nih.gov – Combined Health Information Database	127,211	Health-related documents
www.gsfc.nasa.gov – NASA Goddard Space Flight Center	24,668	Astronomy-related documents
www.ncbi.nlm.nih.gov/pubmed – NCBI PubMed	14,000,000	Citations for biomedical articles

Table 2: Coverage obtained for different numbers of iterations

Site	5 iterations	10 iterations	15 iterations	Use stopwords
nwfusion.com	94.8	94.4	94.4	true
apsa.org	86.6	88.5	91.6	true
cdc.gov	90.4	90.4	90.4	true
epa.gov	94.2	94.2	94.2	true
georgetown.edu	98.3	97.9	97.9	true
chid	35.9	22.8	22.8	true
gsfc.nasa.gov	99.9	99.9	99.9	false
pubmed	33.8	34.6	48.9	false

3. Experiments and Experiences

In order to assess the effectiveness of our approach, we ran some preliminary experiments using real Web sites. We selected sites from different domains and of different sizes. Since we wanted to measure coverage, we restricted our experiments to sites for which we were able to determine the total collection size. Some of the sites actually publish this information. For the others, we obtained the collection sizes from the site administrator. The description of the sites is given in Table 1.³ In what follows we analyze several features of our siphoning strategy, and discuss how different choices for tuning the algorithm influence the results.

Coverage. The coverage of a given keyword sample k_1, k_2, \dots, k_n is computed as follows: if the search interface accepts disjunctive queries, the coverage of the sample is simply the number of results returned for the query k_1 OR k_2 OR \dots OR k_n over the size of the database; otherwise,

³There was a discrepancy between the collection size published in the Web site (11 million) and the size given to us by the site administrator (14 million). In order to be conservative, we use the latter in our experiments.

Table 3: Keywords selected for coverage query

Site	With stopword	Without stopword
nwfusion	the,03,and	definition, data, latest, news, featuring
apsa	of,the,and,in,a,s,j	psychoanal, amer, review, psychoanalytic, int, new, study, family
cdc	cdc,search,health,of,the,to	health, department, texas, training, public, file, us, services
epa	epa,search,region,for,to,8	epa, environmental, site, data
georgetown	georgetown,the,and,of,to	university, georgetown, description, information
chid	chid,nih,hiv,for,aids,the,prevention,of,to,health	aids, health, disease, author, number, education, english
nasa	n/a	nasa
pubmed	n/a	nlm, nih, cells, cell, effects, expression, virus, after, proteins, human

if only conjunctive queries are allowed, the number of results is computed using the inclusion-exclusion formula [17]. Coverage obtained for the sites considered using different numbers of iterations for *SampleKeywords* (Algorithm 1) are shown in Table 2. In this experiment, and in the other experiments below, the settings for *ConstructQuery* (Algorithm 2) are as follows: `maxSubmissions` is 15 – this avoids overloading the sites with requests; and `maxTerms` is 10 – this prevents the generation of long queries that cannot be effectively (or efficiently) be processed by the site. Table 2 also indicates which sites index stopwords.

Two points are worthy of note in Table 2. For 6 out of the 8 sites, coverage of over 90% was obtained with as few as 5 iterations. These results indicate that our approach is promising, and although simple, it can be effective in practice.

For 2 sites, `chid` and `nwfusion`, the coverage decreases with the increased number of iterations. As we discuss below, in *Collection idiosyncrasies*, not always a keyword with high frequency leads to high coverage.

Effectiveness of stopwords in probe queries. Table 3 shows the list of keywords used in the query that obtains the coverage results of Table 2. This table shows two lists of keywords for each site: one with and one without stopwords. The lists of keywords reveal some of the properties of the *SampleQuery* algorithm and give some insight about its effectiveness. The lists without stopwords indeed contain words that are very relevant to the corresponding sites. A good example is `gsfc.nasa.org`, where the keyword *nasa*, found in the first few iterations of the algorithm, is enough to retrieve 99.9% of the documents in the site. These lists also reveal some of the characteristics of the indexers for the sites, e.g., `apsa.org` indexes single-letter words, and `epa.gov` indexes numbers.

Number of requests. Since we had no intuition for how many requests (iterations) were needed to build a *good* keyword sample, we tried different values. For most sites, the sample converges fast, after only 5 iterations. Using 15 iterations, the candidate set generated led to coverages of over 90% for all sites except for `chid` and `pubmed` (see Table 2). For `pubmed`, a higher number of iterations led to a significantly improved coverage, from 48.9% after 15 iterations to 79.8% after 50 iterations. Factors that contribute to this behavior include: the collection size – `pubmed` is orders of magnitude larger than the other collections; the heterogeneity of the documents –

1. Preventing Cryptosporidiosis.

Subfile: AIDS Education

Format (FM): 08 - Brochure.

Language(s) (LG): English.

Year Published (YR): 2003.

Audience code (AC): 084 - HIV Positive Persons. 157 - Persons with HIV/AIDS.

Corporate Author (CN): Project Inform, National HIV/AIDS Treatment Hotline.

Physical description (PD): 4 p.: b&w.

Availability (AV): Project Inform, National HIV/AIDS Treatment Hotline, 205 13th St Ste 2001, San Francisco, CA, 94103, (415) 558-8669, <http://www.projectinform.org>.

Abstract (AB): This information sheet discusses cryptosporidiosis (Crypto), a diarrheal disease caused by a parasite that can live in the intestines of humans and animals. This disease can be very serious, even fatal, in people with weakened immune systems. The information sheet describes, the symptoms, transmission, diagnosis, treatment, and prevention of Crypto, and gives examples of people who might be immuno-compromised or have a weakened immune system, such as people with AIDS or cancer, and transplant patients on immunosuppressive drugs. The information sheet also explains how crypto affects such people.

Major Descriptors (MJ): Disease Prevention. Disease Transmission. Guidelines. Hygiene. Opportunistic Infections. Sanitation.

Verification/Update Date (VE): 200304.

Notes (NT): This material is available in the following languages: AD0031721 Spanish.

Accession Number (AN): AD0031720.

Figure 1: Sample document from `chid.nih.gov`

there are documents about several different subjects within the biomedical area; and the lack of descriptions of the articles in the query results – only the titles of articles are available (see below for details). In addition, `pubmed` does not index stopwords.

Collection idiosyncrasies. The lowest coverage value was obtained for the `chid` collection. As Figure 1 illustrates, documents in this collection have some structure (a set of fields), and some of these fields are optional (e.g., *Notes*). In addition, different fields are indexed differently – some are indexed filtering out stopwords, while others are not. For example, the *Notes* field is indexed with stopwords, whereas *Abstract* is not. This poses a problem for our algorithm, since when it verifies if the database accepts stopwords, it assumes that all items in the collection have the same property. Consequently, the sample with stopwords will only be effective for a subset of the items which contain the *Notes* field. As shown in Figure 3, keyword samples without stopwords lead to a much increased coverage for `chid`: more than twice the value of the coverage of the lists with stopwords.

This also explains the reduction in coverage shown in Table 2. As the *SampleKeywords* algorithm iterates, it builds the candidate set and continuously updates the frequencies of the

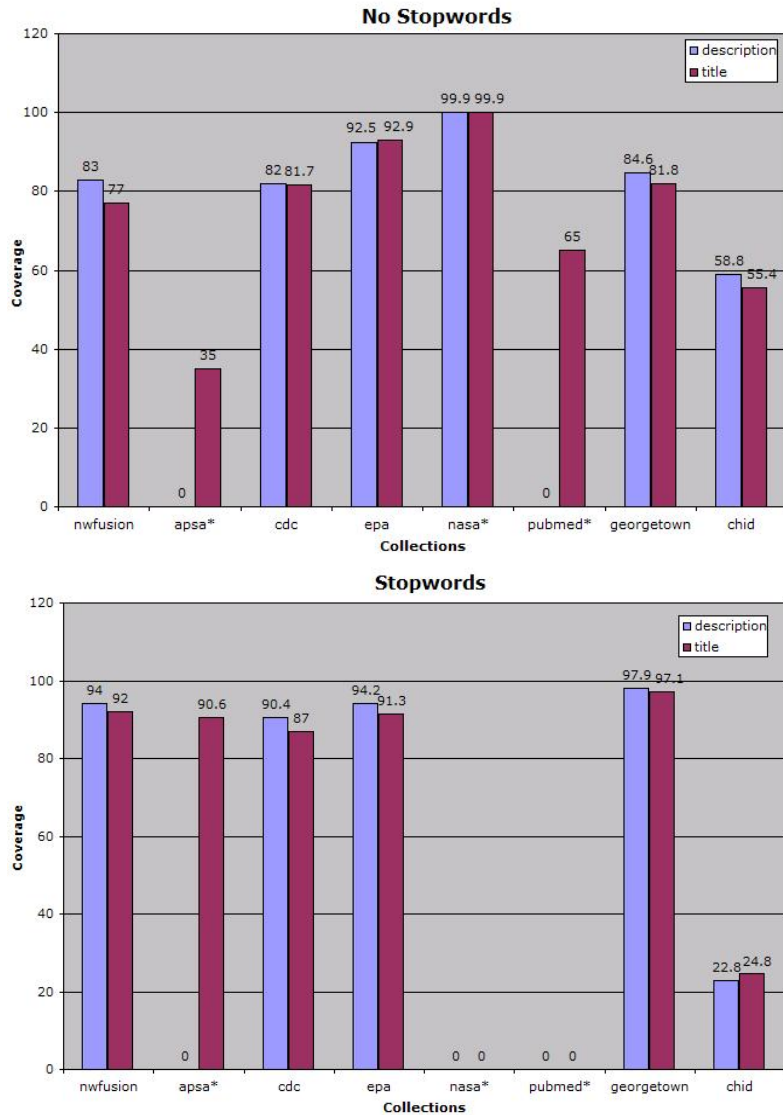


Figure 2: Selecting keywords: document descriptions vs. title

keywords in the set. When stopwords are considered, their frequencies are likely to grow at a faster pace than the frequencies of the other keywords. Since *ConstructQuery* selects the keywords with highest frequencies from the candidate set, at later iterations it is more likely to pick stopwords. In the case of *chid*, only the *Notes* field, which does not appear in all documents, indexes stopwords. As a result the stopwords-rich queries constructed based on the candidate sets for later iterations have lower coverage.

Selecting keywords. As discussed in Section 2, different strategies can be used to extract keywords from the query results. We experimented with four configurations, varying whether stopwords are indexed or not, and how the keywords are extracted. For the latter, we wrote specialized filters that select keywords either from the main entry of the result (e.g., the title of a document) or from its description (e.g., the document summary). The coverage results (for 15 iterations) are shown in Figures 2 and 3.⁴

⁴In these figures, the bars with value zero correspond to invalid configurations.

Figure 2 shows the difference in coverage between selecting keywords from the title of the documents and from document descriptions. Using the descriptions leads to slightly larger coverage, regardless of the presence or absence of indexing of stopwords. This is expectable, since a description is likely to contain more relevant information than a title. When the collection indexes stopwords, as shown in Figure 3, using stopwords leads to higher coverage for all collections, except `chid`, as explained above.

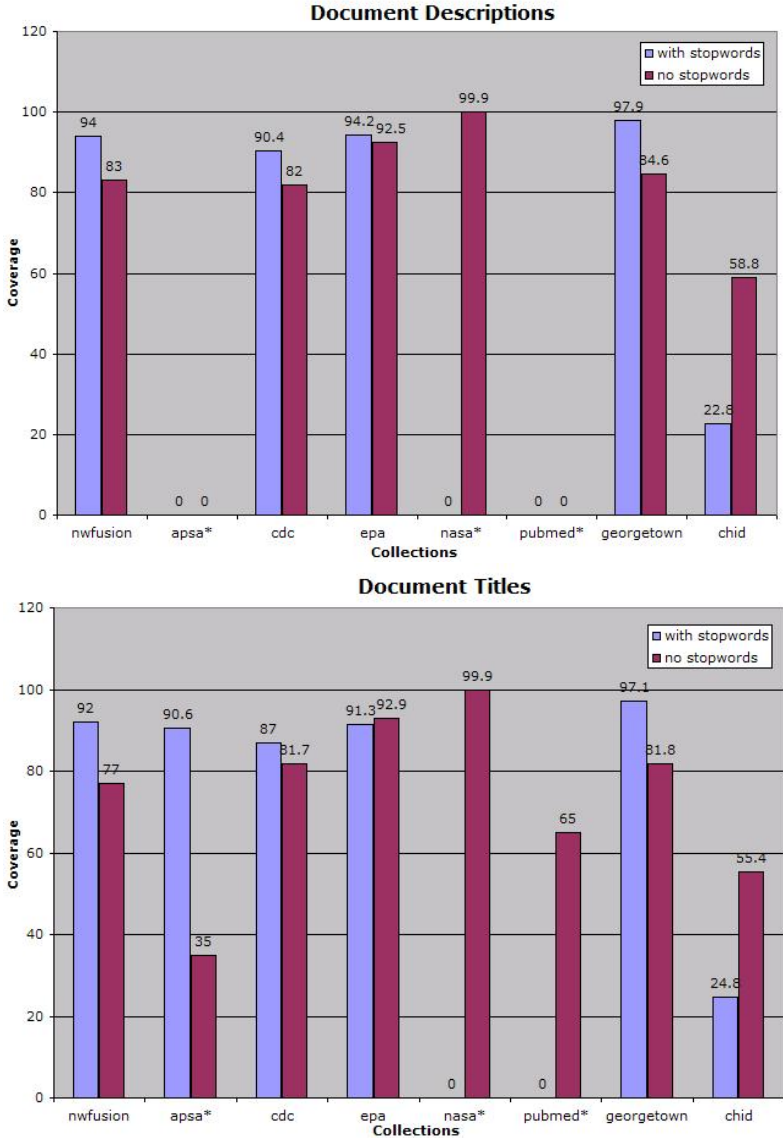


Figure 3: Selecting keywords: with vs. without stopwords

We also assessed the usefulness of using a wrapper to filter out extraneous information from the results page and extract only the actual results. As shown in Figure 4, the use of a wrapper leads to slightly, but not significantly, better coverage. A possible explanation for these non-intuitive results is that these collections have content-rich pages. The effectiveness of wrappers is more accentuated for `pubmed`, whose result pages follow a template that contains a large percentage of extraneous information.

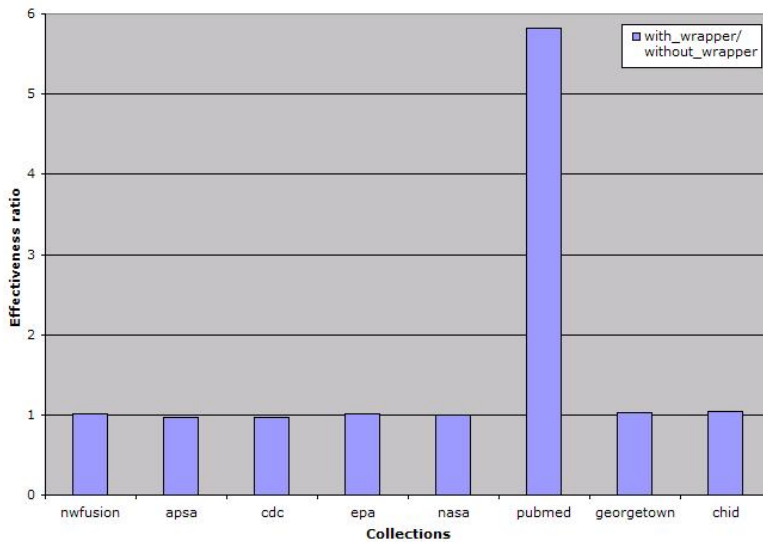


Figure 4: Wrapper effectiveness: ratio of results retrieved with and without a wrapper

4. Related Work

This work is, to the best of our knowledge, the first to consider the problem of automatically siphoning data hidden behind keyword-based search interfaces.

In [18], Raghavan and Garcia-Molina described HiWe, a task-specific hidden-Web crawler. Although they propose techniques to automatically fill out forms, their approach requires human input, and they only consider multi-attribute forms for structured data sources – single-element forms, such as keyword-based search interfaces, are ignored in the crawls. In addition, although they acknowledge the importance of coverage as a measure of crawler performance, no coverage analysis is provided for HiWe. Instead, they focus on *submission efficiency*. Note that in contrast with keyword-based interfaces, attributes in structured forms have well-defined (and sometimes strict) domains. Thus, an important measure of crawling effectiveness is submission efficiency, i.e., the percentage of submissions that use *correct* values. Techniques such as the statistical schema matching proposed by He and Chang [14] can greatly improve submission efficiency for multi-attribute forms. In [3], Benedikt et al propose techniques to automatically crawl a dynamic Web site, including filling out forms. Their focus, however, is on testing (verifying) the correctness of Web sites.

The problem of reconstructing Web databases through limited query interfaces was considered in [6]. Byers et al [6] study a subset of this problem, namely finding efficient covers for spatial queries over databases that are accessible through nearest-neighbor interfaces, and that return a fixed number of answers. The focus of [6] is on efficiency, i.e., how to minimize the number of queries required to reconstruct the database, and the approach relies on the availability of specialized wrappers to access the underlying databases.

In [8], Callan and Connell’s proposed a technique to automatically create descriptions (i.e., a set of representative words) of document databases that are sufficiently accurate and detailed for use by automatic database selection algorithms (e.g., [7, 12]). They show that accurate descriptions can be learned by sampling a text database with simple keyword-based queries. The problem of retrieving all the results in a hidden collection or database also requires that a

representative set of words be learned, but with the goal achieve the highest possible coverage of the collection. Note that the quality of selected terms is measured differently in the two problems. Whereas terms for descriptions must be *meaningful* (e.g., terms such as numbers, or short words with fewer than 3 characters are discarded), for siphoning hidden data, the best term is simply a term that is present in the largest number of items, this term can even be a stopword. Our *SampleKeywords* algorithm adapts and extends the approach proposed in [8].

5. Conclusion

In this paper we examined the problem of siphoning data hidden behind keyword-based search interfaces. We have proposed a simple and completely automated strategy that can be quite effective in practice, leading to very high coverages.

The fact that such a simple strategy is effective raises some security issues. As people publish data on the Web, they maybe unaware of how much access is actually provided to their data. Our preliminary study suggests some simple guidelines that can be followed to make it harder for information to be hijacked from search interfaces, e.g., avoid indexing stopwords. However, an interesting direction for future work is to better characterize search interfaces, and devise techniques that guarantee different notions and levels of security.

Although our results are promising, we have just scratched the surface of the problem. There are several open problems that we intend to investigate. One such problem is how to achieve coverage for collections whose search interfaces fix the number of returned results.

As discussed in Section 3, the effectiveness of the algorithm depends both on the choice for the parameters used in the algorithm (e.g., the number of iterations) as well as on features of the collection (e.g., the indexing, size, nature of contents – homogeneous vs. heterogeneous). In order to provide a comprehensive solution to the problem, techniques are needed to automatically obtain this information and dynamically tune the algorithm.

Our initial experiments focused on more document-oriented collections. We are currently experimenting with more structured collections, specifically, with the sites catalogued in [20]. A preliminary exploration of these sites indicates that many do provide keyword-based interfaces; and some actually index stopwords (e.g., job listing sites).

Acknowledgments. The National Science Foundation partially supports Juliana Freire under grant EIA-0323604.

References

- [1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of ICDE*, pages 5–16, 2002.
- [2] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web navigation with the WebVCR. In *Proceedings of WWW*, pages 503–517, 2000.
- [3] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: A platform for automating web site testing. In *Proceedings of WWW*, 2002.
- [4] M. K. Bergman. The deep web: Surfacing hidden value (white paper). *Journal of Electronic Publishing*, 7(1), August 2001.
- [5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proceedings of ICDE*, pages 431–440, 2002.

- [6] S. Byers, J. Freire, and C. T. Silva. Efficient acquisition of web data through restricted query interfaces. In *Poster Proceedings of WWW*, pages 184–185, 2001.
- [7] J. Callan, Z. Lu, and W. Croft. Searching distributed collections with inference networks. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, 1995.
- [8] J. P. Callan and M. E. Connell. Query-based sampling of text databases. *Information Systems*, 19(2):97–130, 2001.
- [9] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.
- [10] H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan. A layered architecture for querying dynamic web content. In *Proceedings of SIGMOD*, pages 491–502, 1999.
- [11] R. B. Doorenbos, O. Etzioni, and D. S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 39–48, 1997.
- [12] L. Gravano, H. Garcia-Molina, and A. Tomasic. The effectiveness of GLOSS for the text-database discovery problem. In *Proceedings of SIGMOD*, pages 126–137, 1994.
- [13] L. Gravano, P. G. Ipeirotis, and M. Sahami. Qprober: A system for automatic classification of hidden-web databases. *ACM TOIS*, 21(1):1–41, 2003.
- [14] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *Proceedings of SIGMOD*, pages 217–228, 2003.
- [15] Hypertext markup language (HTML). <http://www.w3.org/MarkUp>.
- [16] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.
- [17] N. Linial and N. Nisan. Approximate inclusion-exclusion. In *ACM Symposium on Theory of Computing*, pages 260–270, 1990.
- [18] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *Proceedings of VLDB*, pages 129–138, 2001.
- [19] SOAP version 1.2 part 0: Primer. W3C Recommendation, June 2003. <http://www.w3.org/TR/2003/REC-soap12-part0-20030624>.
- [20] The UIUC Web integration repository. <http://metaquerier.cs.uiuc.edu/repository>.
- [21] Web service definition language (WSDL). <http://www.w3.org/TR/wsdl.html>.