SmartCrawl: A New Strategy for the Exploration of the Hidden Web

Augusto de Carvalho Fontes
Universidade Tiradentes
Aracaju SE, Brazil
augusto_carvalho@unit.br

Fábio Soares Silva
Universidade Tiradentes
Aracaju SE, Brazil
fabio_soares@unit.br

ABSTRACT

The way current search engines work leaves a large amount of information available in the World Wide Web outside their catalogues. This is due to the fact that crawlers work by following hyperlinks and a few other references and ignore HTML forms. In this paper, we propose a search engine prototype that can retrieve information behind HTML forms by automatically generating queries for them. We describe the architecture, some implementation details and an experiment that proves that the information is not in fact indexed by current search engines.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models, Search process*; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Navigation*

General Terms

Algorithms, Experimentation

Keywords

Hidden Web, Search Engine, Label Extraction

1. INTRODUCTION

The gigantic growth in content present in the World Wide Web has turned search engines into fundamental tools when the objective is searching for information. A study in 2000 [11] discovered that they are the most used source for finding answers to questions, positioning themselves above books, for example.

However, a great deal of relevant information is still hidden from general-purpose search engines like *AlltheWeb.com* or *Google*. This part of the Web, known as the Hidden Web [7], the Invisible Web [6, 9] or the Deep Web [1] is growing constantly, even more than the visible Web, to which we are accustomed [6].

This happens because the crawler (the program that is responsible for autonomous navigating the web, fetching pages) used by current search engines cannot reach this information.

There are many reasons for this to occur. The Internet's own dynamics, for example, ends up making the index of search engines obsolete because even the quickest crawlers only manage to access only a small fraction each day of the total information available on the Web.

The cost of interpretation of some types of files, as for example *Macromedia Flash* animations, compressed files, and programs (executable files) could be high, not compensating for the indexing of the little, or frequently absent, textual content. For this reason, that content is also not indexed for the majority of search engines.

Dynamic pages also cause some problems for indexing. There are no technical problems, since this type of page generates ordinary HTML as responses for its requests. However, they can cause some challenges for the crawlers, called *spider traps* [8], which can cause, for example, the crawler to visit the same page an infinite number of times. Therefore, some search engines opt not to index this type of content.

Finally, there are some sites that store their content in databases and utilize HTML forms as an access interface. This is certainly the major barrier in the exploration of the hidden Web and the problem that has fewer implemented solutions. Nowadays none of the commercial search engines that we use explore this content, which is called the *Truly Invisible Web* [9].

Two fundamental reasons make crawling the hidden Web a non-trivial task [7]. First is the issue of scale. Another study shows that the hidden content is actually much greater than what is currently publicly indexed [1]. As well as this, the interface for access to this information serves through the HTML forms are projected to be manipulated and filled by humans, creating a huge problem for the crawlers.

In this paper, we propose a search engine prototype called *SmartCrawl*, which is capable of automatically attaining pages that are not actually recoverable by current search engines, and that are "secreted" behind HTML forms.

The rest of this article is organised in the following manner. Section 2 shows related work and in the sequence, we explain the construction of HTML forms and how they can be represented. In sections 4 and 5, we describe the prototype. In Section 6 the experimental results are highlighted and finally, in Section 7, we conclude the paper.

WIDM'04, November 12–13, 2004, Washington, DC, USA.

2. RELATED WORK

There are some proposals for the automatic exploration of this hidden content. Lin and Chen's solution [6] aims to build up a catalogue of small search engines located in sites and, given the user searching terms, choose which ones are more likely to answer them. Once the search engines are chosen by a module called Search Engine Selector, the user query is redirected by filling the text field of the form. The system submits the keywords and waits for the results that are combined subsequently and sent to the users' interface.

The HiWE [7] is a different strategy which aims to test combinations of values for the HTML forms at the moment of the crawling (autonomous navigation), making the indexing of the hidden pages possible. Once a form in a HTML page is found, the crawler makes several filing attempts, analyses and indexes the results of the obtained pages.

Moreover, the HiWE has a strategy to extract the labels of HTML forms by rendering the page. This is very useful to obtain information and classify forms and helps to fill in its fields.

There are other approaches that focus on the data extraction. Lage et al. [4] claims to automatically generate agents to collect hidden Web pages by filling HTML forms.

In addition to this, Liddle et al. [5] perform a more comprehensive study about form submissions and results processing. This study focus on how valuable information can be obtained behind Web forms, but do not include a crawler to fetches them.

3. EXTRACTING DATA FROM BEHIND THE FORM

HTML forms are frequently found on the web and are generally used for filtering a large amount of information.

As shown in Figure 1, from a page with one form the user can provide several pieces of data which will be passed on to a process in the server, which generates the answer page.

The current crawlers do not fill in form fields with values, making them the major barrier for exploration of the hidden Web. In order to achieve this, it is vital to extract several pieces of information from the form.

An HTML form can be built on different manners, including various types of fields such as *comboboxes*, *radio buttons*, *checkboxes*, *text fields*, *hidden fields* and so on. However, the data sent to the server through the *Common Gateway Interface* (CGI) is represented by proper codified pairs $(name, value)$. This way, we can characterise a form with which has n fields as a tuple:

$$F = \{U, (N_1, V_1), (N_2, V_2), ..., (N_n, V_n)\} \quad (1)$$

where U is the URL for the data that has been submitted, and (N_n, V_n) are the pairs $(name, value)$ [5].

However, this is a simplification, since there are much more information associated with HTML forms. An example is the method by which the form data will be sent to the server, that is, by HTTP GET or POST. Moreover, some fields possess domain limitations (e.g. text fields with a maximum size, comboboxes).

To do an analysis of the form and extract relevant information is not an easy task, but the most difficult step surely is to extract the field's labels. This is because generally there is not a formal relationship between them in the HTML code. For example, the label for a text field can be placed above it, separated by a `BR` tag, it can be beside it, or it can be inserted inside table cells.

All these pieces of data are absolutely necessary to be extracted for surpassing HTML forms and fetching the results page.

4. THE SMARTCRAWL

The aim of *SmartCrawl* is to bring a strategy that allows a more complete exploration of the hidden content. To achieve this, it managed to generate values for a largest number of forms.

Furthermore, it has an architecture very similar to current commercial search engines, which means it permits an easier implantation of strategies vastly used to gain performance and scalability as presented by [10] or [2].

4.1 Execution of the Prototype

SmartCrawl is above all a search engine and, therefore, contains all its essential components. The difference is in the fact that each component has adaptations and some extra features which enable them to explore the hidden content of the Web. The main goal is to index only the pages that potentially are in the non-explorable part of the Web.

To extract the content from behind these forms, *SmartCrawl* generates values for its fields and submits them. These values are chosen in two different moments: in the indexing and when an user performs a search.

In the indexing phase, once it finds a form, the *SmartCrawl* extracts a set of pieces of information from it that allow queries (combinations of possible values for the form) to be created.

New queries are also generated when the user performs a search. For this, the forms that are more likely to answer to the search receive the supplied keywords. However, contrary to the implementation of Lin and Chen [6] the obtained results are also scheduled for indexing and not only returned to the user interface.

The process of execution of *SmartCrawl* is constituted in the following steps: (1) finding the forms, (2) generating queries for them, (3) going to the results and (4) searching created indexes.

4.1.1 Finding forms

The first step in the execution process of the *SmartCrawl* is the creation of a number of crawlers that work in parallel searching for pages that include HTML forms. Every page found is then compressed and stored for further analysis. At this moment, it acts like a common crawler, following only links and references to frames.

The pages stored by the crawler are decompressed afterwards by a indexing software component which extracts pieces of information from each of the forms found and catalogues them. Beyond this, every page is indexed and associated with the forms. If the same form is found in distinct pages, all of them are indexed. Nevertheless, there will be only one representation of the form.

4.1.2 Generating queries for the forms

Another component of the indexing software is in charge of generating values for the encountered forms. The generation of queries is based on the collected information about the form and its fields.

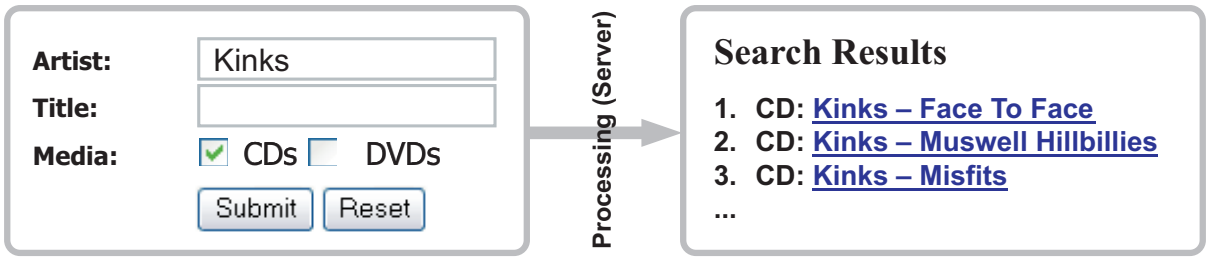

Figure 1: A form processing

The first generated query is always the default, that is, the one which uses all the defined values in the HTML code of the form. Next, a pre-defined number k of other possible combinations is generated. To generate values for the text fields (which possess an infinite domain) a table that stores a list of values for a data category is consulted based on the field label.

For every generated query, a further visit is scheduled for the crawler. The parameters (set of field names and values) are stored and a new item is added to the queue of URLs that must be visited by crawlers.

4.1.3 Visiting the results

The crawler is in charge of executing its second big goal which is to submit the scheduled queries. To accomplish this it needs an extra feature: the capacity to send parameters in HTTP requests using both the GET and POST methods. If it perceives that an item in the queue of URLs is a query, it submits the parameters and analyses the HTML code obtained as a result in the same way that it does to others. The page is then compressed, stored and associated with the information of the original query.

The indexing software decompress pages that contain results of form submissions and indexes them. From this index, the classification and search software finds results that contain all the search terms formulated by the user.

4.1.4 Searching for stored pages

As soon as the user performs a search, two steps are executed by the classification and search software. Firstly, the indexes created by the indexing software are consulted to find the keywords formed by the user and the results are returned in an organised form (the most relevant come first) in an HTML page.

Subsequently, based on these indexed pages which are associated with the forms, *SmartCrawl* selects forms that are more likely to answer to the user's search and generate new queries which will be visited afterwards by the crawlers and indexed in the same way by the indexing software.

5. ARCHITECTURE AND IMPLEMENTATION

The architecture at the high level of this application is divided into: crawler, indexing software, ranking and search software and storage components.

As we have seen, the crawler is responsible for obtaining Web pages, submitting queries and storing the results. The indexing software, on the other hand, indexes the obtained pages and generates form queries. The ranking and search software uses the indexes to answer searches made by the user or redirects other forms and storage components take care of the storage of all the information used by other components.

Figure 2 shows how the main components are available and how they interact with the storage components, that are represented by the rounded boxes. They are the *Form Parser*, *Form Inquirer* and *Form Result Indexer* of the indexing software, the *Document Seeker* of the ranking and search software and the *Crawler Downloader* of the crawler.

Two storage components support the crawling: *URL Queue* and *URL List*. The first is responsible for storing the line of URLs that the *Crawlers Downloaders* need to visit, in this way, the URLs which will serve as seeds to the autonomous are also added in it. As soon as a *Crawler Downloader* extracts links from an HTML page, it is in this component that the new URLs will also be inserted for a further visit. The URL list, on the other hand, stores the URLs that have already been visited, allowing the *Crawler Downloader* keep track of them.

When the page includes a form, or a result to a query, it needs to be stored for subsequent indexing. The crawler stores the compressed content in a storage component named *Warehouse*, where it is given a number called `storeId`.

Two components of the indexing software are responsible for decompressing these pages that have been stored in the *Warehouse*. The first, called the *Form Parser*, extracts information from all the forms contained within the page, and sends them to the storage component *Form List*, where a number, called `formId`, is associated with every form. The *Form Parser* is also responsible for indexing the page which contains forms and associating it to each `formId` of the forms contained within.

To index these documents, *SmartCrawl* uses a technique called inverted index or inverted file. The *Document List*, *Wordmatch* and *Lexicon* are the three components that carry out storing an indexed page. The *Document List* stores the title, a brief description of each indexed page and its `docId`. The *Lexicon* and *Wordmatch* store the inverted index itself. The first contains a list of pairs (`wordId`, *word*) for each one of the words used in indexed documents. The second contains a list of occurrences of the words in the indexed documents and their position (offset) in the text. *Wordmatch* is formed therefore by the values of `docId`, `wordId` and `offset`

The *Form Inquirer* is the indexing software component whose objective is to generate queries for the stored forms in the *Form List*. To generate values for the text fields, *Form Inquirer* consults the list of categories and values through the component *Categories*. Each query generated is sent to the *Query List* where a `queryId` is associated to it and a new URL is added to the *URL Queue*.

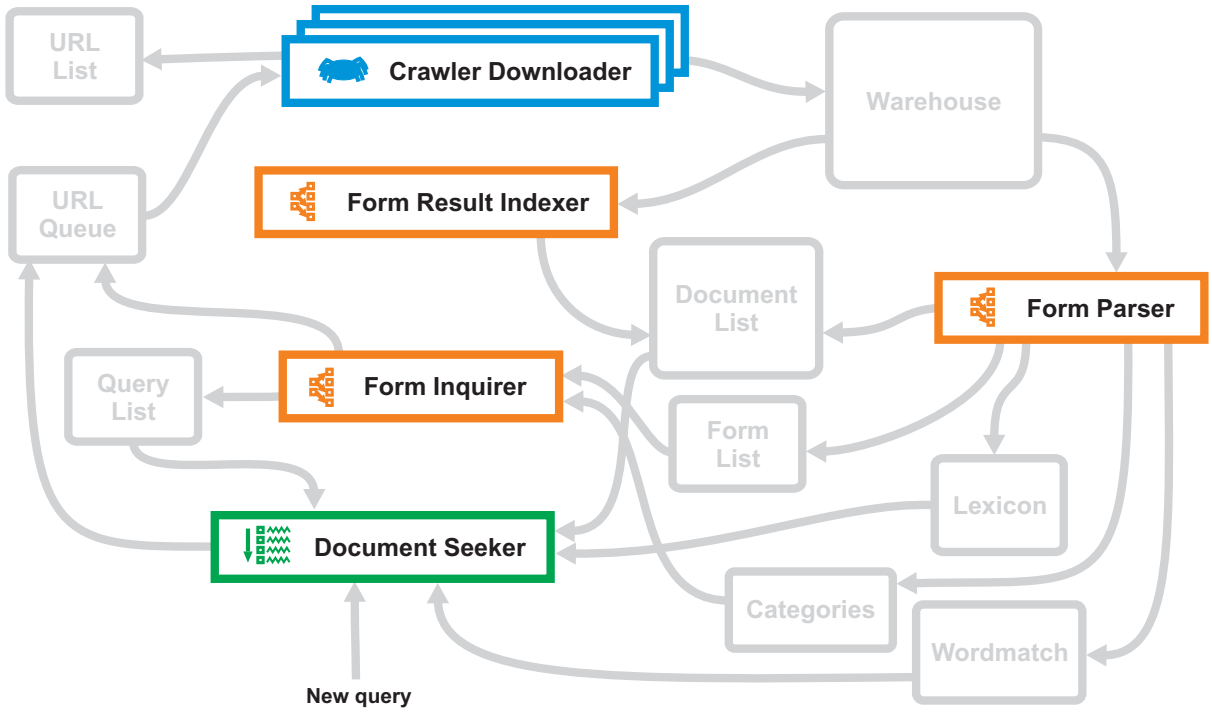

Figure 2: Architecture at a high level

The second component that extracts compressed pages from the *Warehouse* is the *Form Result Indexer*. Its job is too much easier than the others, as its aim is only to index the pages that contain results of submitted queries and to associate the proper `queryId`.

From the indexes created and stored in *Lexicon* and *Wordmatch*, the *Document Seeker* answers to user searches. Every document stored in the in the *Document List* possesses a `formId` associated with it, and optionally, a `queryId` when the document is the response to a query. With these two numbers, the *Document Seeker* consults the *Form List* and the *Query List* to obtain the necessary information about the way it locates an indexed page on the World Wide Web. For example, a query to a form that points to the URL `http://search.cnn.com/cnn/search` using the HTTP GET method can be represented by `http://search.cnn.com/cnn/search?q=brazil`, if it possesses only one parameter with name `q` and value `brazil`.

The *Document Seeker* should return the result set in an ordered form, so that the most relevant documents will be taking first place. A simple solution for this is to only take into consideration the position of the words in the text, and the number of occurrences. Considering o_i, as the offset of the *i-th* encountered word in the document which is amongst the terms of the user, the relevance is given by this:

$$r = \sum_{i=0}^{n} \frac{1000}{(o_i + 1)} \tag{2}$$

In equation 2, we compute the rank of a page by summing the offsets of all user search terms that appears in the document and then divide the result by an arbitrary number (in this case 1000) so that the most relevant entry receives the smallest number.

Another important role of the *Document Seeker* is to redirect the search terms supplied by the user to some of the several forms catalogued in the *Form List*. To accomplish this, it looks in the *Document List* for pages that contain search engines (forms with one, and only one, text field) and that possess in its text words related to the terms sought by the user. New queries are then added to the *Query List* and new URLs are added to the *URL Queue* to be visited by the *Crawler Downloader*.

5.1 Labels extraction algorithm

A very important task is performed by a secondary component called *Form Extractor*. It is in charge of extracting diverse pieces of form information present in an HTML page.

To facilitate the content analysis of a page, the HTML code is converted into a DOM[1] tree provided by *Cyberneko Html Parser* [3]. From the DOM tree, the *Form Extractor* looks for nodes which represent forms and separate them from the rest of the tree. Each of these sub-trees, which encompass all the tags which are positioned between the `<form>` and the `</form>`, is submitted for processing.

Amongst the data which should be obtained, undoubtedly the fields and their labels are the most challenging ones. In spite of having a tag in the HTML specification called `label` for the declaration of a label, it is almost not used and, therefore, we do not possess a formal declaration of labels in the HTML code.

The solution encountered was to establish a standard that the labels must have. For the *Form Extractor*, labels are continuous segments of texts which use the same format and have the maximum of n words and k characters. These values can be defined in the configuration file.

[1] *Document Object Model*

(a) HTML form example

	1	2
1	{**Label** "Artist:"}	{**Textfield** "artist"}
2	{**Label** "Title:"}	{**Textfield** "title"}
3	{**Label** "Media:"}	{**Checkbox** "cds", **Label** "CDs", **Checkbox** "dvds", **Label** "DVDs"}
4	{**Button** "Submit", **Button** "Reset"}	

(b) Table representing the position of the elements of the form

Figure 3: Representing the positions of the components of an HTML form

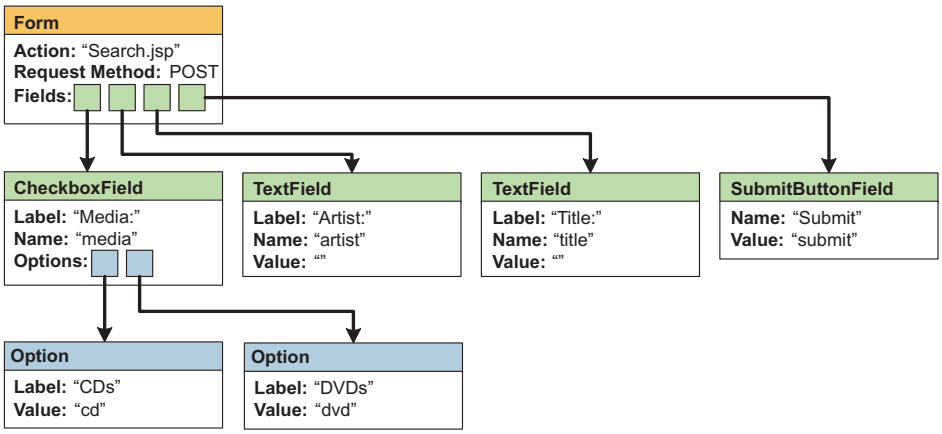

Figure 4: An example of a HTML form representation

From the sub-tree which contains a certain form information, the *Form Extractor* generates a table which represents the positioning of the elements contained in it. Figure 3 shows an example of the table generated by a simple form.

The table is generated considering the nodes in the DOM tree which represent a common HTML table. If there are more than one defined table in the HTML code, similar representations are created. Each cell has a collection of the form's elements. The third step of the process is to extract the labels of each one of the fields in the form (except for hidden fields and buttons) and generate an object-orientated representation for the forms.

To extract the labels, the *Form Extractor* passes twice by the generated table to the form. The first time, for each field in the form which require a label, it is verified exactly what exists on the left side of the field (even in one adjacent cell). If a label is found in this position, this label is immediately associated to the field. In the second passage, the fields which still do not have association with labels are observed again, however this time the search for the label is done in the above cell.

For the fields of the *checkbox* and *combobox* kind, the treatment is special, because apart from the conventional label, the items which represent their domain also have labels.

As in the case of "DVDs" and "CDs" labels in Figure 3(a). The domain labels are extracted from the right, and the label of the set of items is obtained from the left. In the example, the *checkboxes* are grouped by their names, which in this case is "media." The labels of each item are extracted from the right ("CDs" and "DVDs") and the label of the set of *checkboxes* is extracted from the left of the first field ("Media:").

Figure 4 shows how the object-oriented structure for the example above would be.

5.2 The list of categories and values

The *Categories* component is in charged of controlling a list of categories and values which helps the *Form Inquirer* to generate values for the text fields. In order to guarantee better results, the name of the category is normalized before comparing to the field label. The normalization aims to: (1) remove punctuation (leaving just the words), (2) convert graphic signing and other special characters into simple ones and (3) remove *stop words*.

Stop words is a concept given to the words which can be taken from sentences without changing its meaning, being largely used in normalization and some search engines even extinguish these words from their indexes. Great part of the *Stop Words* are prepositions, articles and auxiliary verbs, such as "the", "of" or "is."

The list of categories is automatically built by the *Form Parser*. Once it finds a field with finite domain (e.g. *comboboxes*), the values are extracted and added to the categories list associated to the field's label.

When the same value is added more than once to a category, it gets more priority in relation to the others. This is obtained by using a number which means the relevance of this value in the category. It is based on this number that the set of values is put in order before it is repassed to the *Query Inquirer*. Therefore, the most relevant values are tested first in text fields.

5.3 Redirecting queries

As mentioned before, associated to each form, there is a set of indexed pages where it has been found. These pages allow the *Document Seeker* to choose the forms to which the user's search terms will be redirected to.

In order to choose amongst several forms, two steps are taken: (1) finding which words or sequences of words are related to the terms of the user's search and (2) looking for these words on the pages which have small search engines.

To solve the first problem, we could use the stored index itself. However, the volume of indexed information is not so large as to provide a good set of words. It was used, therefore, the catalogue of a general purpose search engine called Gigablast[2], since it implements data mining techniques and provides, for the searching terms, a list of words or sentences which frequently appear in the returning documents.

From this set of words, the *Document Seeker* performs search on pages which have search engines looking for these terms, also putting them in order according to equation 2.

For the first n selected search engines, the *Document Seeker* creates new queries and add them to the queue so that they can be submitted afterwards in the same way done by the *Form Inquirer*. The queries are created by filling in form's only text field with the searching terms of the user and the other fields with default values.

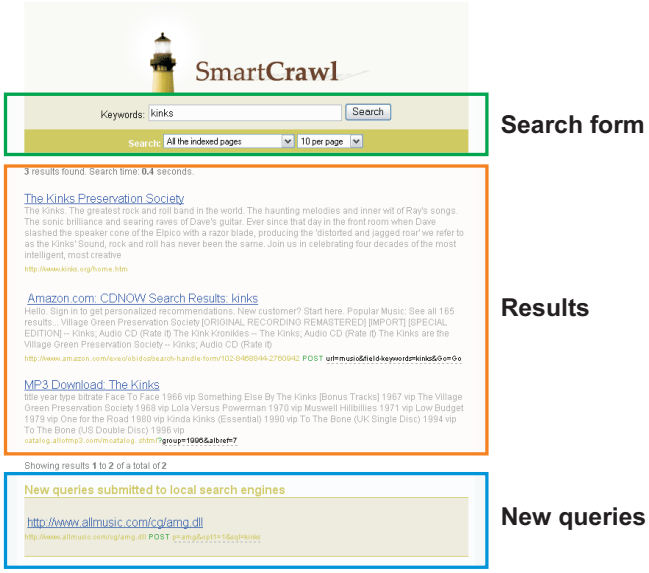

Figure 5: Interface for the search of indexed documents

5.4 The searching interface

A searching interface was build for the purpose of carrying out the tests. As shown in Figure 5, it is divided into three parts.

The **searching form** allows the user to provide the terms of the search. Furthermore, it is possible to choose what the target of the search is: all the pages, only pages which have forms or pages containing the results of forms. The **results** are shown on a list of documents that contain all the searching terms offered by the user. On pages that contain a query associated to them, it is possible to visualise also the parameters. In the area called **new queries** the generated queries which have been scheduled for the crawler's visit are displayed.

[2]http://www.gigablast.com/

6. EXPERIMENTAL RESULTS

The tests which have been carried out aim to test some strategies used in the implementation of the prototype, hence some important aspects of the system were tested separately. Besides that, an analysis of the indexed content regarding its absence or not in the current searching engines was carried out.

In order to support the tests, we started up the crawlers and kept them up until we have 15 thousands indexed pages (including only pages with HTML forms and form results). It worth repeat that our strategy does not index pages that do not offer any challenge to regular crawlers.

6.1 Label extraction algorithm evaluation

This phase aims to evaluate the algorithm used for the extraction of the labels from the form fields that was described in section 5.1. To do so, 100 forms were manually observed and compared to the information extracted by the *Form Extractor*. For each one of these fields it was verified whether the choice made by the algorithm was correct or not. From the 100 forms evaluated, 5 of them (5%) were not extracted.

The reason for this is that the HTML was malformed and the API used for the extraction of the DOM tree (NekoHtml [3]) did not manage to recover the error. This way, 189 fields from the 95 remaining forms were verified. For 167 of them (88%), the algorithm extracted the label correctly, making mistakes only in 22 labels (see Figure 6).

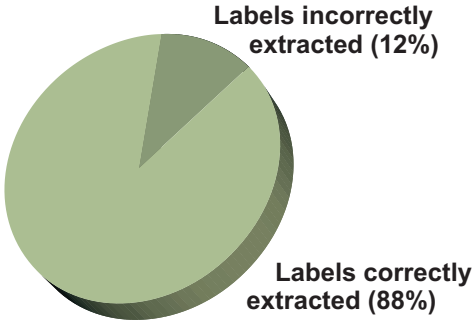

Figure 6: Fraction of labels extracted correctly

Some labels were not extracted correctly because they did not fit within the restrictions defined in section 5.1. Another problem faced was when the labels were not defined inside the tag `FORM`. In this case they were not present in the subtree analysed by the *Form Extractor*, making its extraction impossible.

Although our solution did not reach the HiWE [7] accuracy to extract labels, we prove that it is possible to get very close results without rendering the page (that consumes much computing resources). Moreover, many of the problems faced in this experiment can be fixed without much effort.

6.2 Relevances of the queries generated by the Document Seeker

In order to analyse the results obtained by the new generated queries from the searches of users, 80 search queries were submitted to the prototype by using arbitrary terms that are commonly used in general purpose search engines, such as "World Cup" or "Music Lyrics".

For each list of queries generated by the *Document Seeker*, the first five ones were submitted and analysed manually, totalizing 155 pages with results. Each page was verified whether the query was successful or not. A successful query is one which has one or more results in it, in contrast to pages with no result or pages that was not considered a search engine results page (e.g. mailing list registration form).

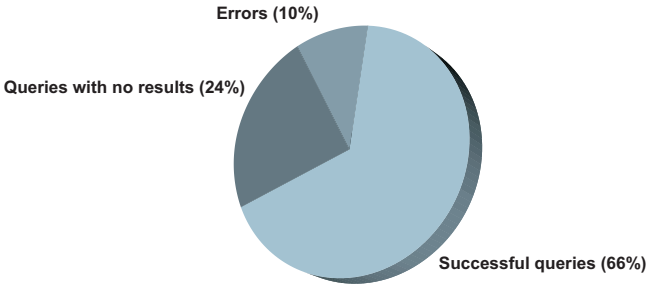

Figure 7: Utilizations of the new queries generated by *Document Seeker*

The result obtained was that 66% of the submitted queries brought some results back, 24% were not successful and 10% of the pages, for some reason, could not be recovered. Figure 7 illustrates better the obtained utilization.

Once the most relevant queries are returned taking first place, it is probable that, with a larger number of indexed pages, we will get better results.

6.3 Visibility of the indexed content

The implementation of *SmartCrawl* has as aim the pages generated from the filling of the HTML forms and which are potentially part of the hidden web. Despite the fact that the current commercial search engines do not have an automatic mechanism which fills in the forms fields and obtain these data, as the *SmartCrawl* does, through common links, part of this information can be explored.

For instance, a page with the results of a form that utilizes the HTTP method GET can be accessed through an usual link because all its parameters can be passed in the URL string itself. In addition to this, once the content is stored in databases, it is not so difficult to find pages that offer the same information through different interfaces.

This phase aimed to verify how much of the indexed content can also be accessed by *Google*. In order to do this, 300 pages with results of GET and POST forms were observed if, for one of the reasons stated before, they were not indexed by this general purpose search engine.

We found out that **62%** of these pages are not indexed by *Google*. When only queries which use the method HTTP POST (which are 59% from the total) are observed, this number becomes even greater, leaving just 14% reachable by **Google**.

7. CONCLUSIONS AND FUTURE WORK

This work proposed a search engine prototype which is capable of handling with HTML forms as well as filling them in automatically in order to obtain information which is unreachable by the current search engines. When compared to other solutions, as the HiWE [7] and the solution that redirects queries by Lin and Chen [6], the *SmartCrawl* brings a big differential which is the ability of surpassing great number of forms. The mentioned solutions have severe restrictions which directly affect the number of forms that receives queries.

There is a great deal of work still to be attached to the solution for a better exploration of the recovered content. An example of this is that *SmartCrawl* does not make any analysis of the pages obtained as results of the queries, therefore indexing pages which contain errors and no results. The implementation of an algorithm which recognizes these pages would increase the quality of the indexed data.

Besides, a high performance structure was not used for the storage of the indexes. This resulted in slow searching and indexing. A future work will be the implementation of a new indexing module.

8. REFERENCES

[1] M. K. Bergman. *The Deep Web: Surfacing Hidden Value*. 2001.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[3] A. Clark. Cyberneko html parser, 2004. http://www.apache.org/ andyc/.

[4] J. Lage, A. Silva, P. Golgher, and A. Laender. Collecting hidden web pages for data extraction. In *Proceedings of the 4th ACM International Workshop on Web Information and Data Management*, 2002.

[5] S. Liddle, D. Embley, D. Scott, and S. H. Yau. Extracting data behind web forms. In *Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business*, pages 38–49, 2002.

[6] K.-I. Lin and H. Chen. Automatic information discovery from the invisible web. In *Proceedings of the The International Conference on Information Technology: Coding and Computing (ITCC'02)*, pages 332–337, 2002.

[7] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 129–138, 2001.

[8] A. Rappoport. Checklist for search robot crawling and indexing, 2004. http://www.searchtools.com/robots/robot-checklist.html.

[9] C. Sherman and G. Price. *The Invisible Web: Uncovering Information Sources Search Engines Can't See*. CyberAge Books, 2001.

[10] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of the 18th International Conference on Data Engineering*, pages 357–368.

[11] D. Sullivan. Internet Top Information Resource, Study Finds, 2001.