12-4-2006

# A Domain Based Approach to Crawl the Hidden Web

Milan Pandya

A DOMAIN BASED APPROACH TO CRAWL THE HIDDEN WEB

by

MILAN C PANDYA

Under the Direction of Raj Sunderraman

## ABSTRACT

There is a lot of research work being performed on indexing the Web. More and more sophisticated Web crawlers are been designed to search and index the Web faster. But all these traditional crawlers crawl only the part of Web we call *"Surface Web"*. They are unable to crawl the hidden portion of the Web. These traditional crawlers retrieve contents only from surface Web pages which are just a set of Web pages linked by some hyperlinks and ignoring the hidden information. Hence, they ignore tremendous amount of information hidden behind these search forms in Web pages. Most of the published research has been done to detect such searchable forms and make a systematic search over these forms. Our approach here will be based on a Web crawler that analyzes search forms and fills tem with appropriate content to retrieve maximum relevant information from the database.

INDEX WORDS: Web crawler, Search spider, Web bot, Best first crawler, Focused Web crawler, Web page, Domain based.

# A DOMAIN BASED APPROACH TO CRAWL THE HIDDEN WEB

by

MILAN C PANDYA

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

In the College of Arts and Sciences

Georgia State University

2006

# A DOMAIN BASED APPROACH TO CRAWL THE HIDDEN WEB

by

MILAN C PANDYA

Major Professor: Raj Sunderraman

Committee: Saeid Belkasim

Ying Zhu

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Science

Georgia State University

November2006

## ACKNOWLEDGEMENTS

I am very much thankful to my advisor, Professor Raj Sunderraman for his constant support and encouragement without which I could not have accomplished this thesis. I am thankful for his interest in this new field and allowing me to work in it. He has constantly supervised guided and suggested ideas for improving my work and led it to a new dimension.

I am also thankful to my committee members, Professor Saeid Belkasim and Professor Ying Zhu for their valuable encouragement and support. I am thankful for their innovative ideas and their interest in new technologies which motivated me to go forward in this prototype.

I would also like to thank Serpil Tokdemir for help and support she provided throughout the academic program. She constantly made sure that I put my best efforts in this work. Without her co-operation it would be difficult for me to make this achievement.

# TABLE OF CONTENTS

# LIST OF FIGURES

# 1. INTRODUCTION

There are billions and billions of Web pages published over the internet via World Wide Web. All of us rely on internet as a source of information. This source of information is available in various forms; Websites, databases, images, sound, videos and many more. Web based information is usually organized and is available in an ad hoc manner. Due to its vastness most of the information is gathered by us through search engines.

A search engine classifies the Search results by keyword matches, link analysis, or other mechanisms perhaps not entirely clear to a front end user. These initial search results are later on fine-tuned by individual users through supplying additional keyword phrases or restricting selection to few Web pages for the results. While people usually rely on search engines to search information from the Web, the results obtained can be variable and often additional browsing must be done within the results. Even powerful commercial search engines such as Google, Yahoo, AltaVista etc., can have problems finding relevant results for the users. About 80% of the Web users uses search engine to retrieve information from the Web [1]. Powerful search engines such as Google which can crawl and index millions of Web pages everyday, provide a good start for information retrieval but may not be sufficient for complex information inquiry tasks that require not just an initial search, but then relevant classification of a large volume of results.

One such area where the search engines lack their expertise is in classifying and indexing the information available from the databases. To give a fair understanding of

what has just been stated, consider an example of a shopping Website. Shopping is the most popular thing one does on the internet. According to internet studies, shopping is one wide part of internet that covers of almost 35 % of it [2]. We see numerous Websites from where we can do e-shopping. E-shopping is one of the fastest growing sectors of internet. Now, thinking as a user, if one wants to shop on internet for a particular product, the best way to begin is to start browsing the Websites popular in shopping for e.g. amazon.com, ebay.com, bestbuy.com and many more. More likely a shopper will surf for like 3 different shopping Websites before he concludes as to where to buy. Now imagine why not to use the search engines to shop. Each time a buyer has to visit at least 3 different shopping Websites, making searches and gathering all the information he needs. Observing closely, the buyer is manually doing the work of the WebCrawler, which is visiting the shopping Websites to gather necessary information and then picking up the right one for him.

Search engines help us to gather information from their own indexed databases. The WebCrawler of these search engines are expert in crawling various Web pages to gather huge source of information. However, majority of the WebCrawler of these search engines crawls only what we call as surface Web. Whenever any WebCrawler visits a Web server, they gather information of all the Web pages been stored on that Web server. They are not able to penetrate deep into the Web server to access their databases and use the Web services served by that particular Web server. According to studies conducted in 2000, deep Web contains 7500 terabytes of information in comparison to 19 terabytes on surface Web. There are more than 200,000 deep Web sites [3].

So one can imagine such a vast source of information is just out of reach because of the limitation of current WebCrawler for unable to access these public databases. In the 21st century A D, where the advancement of the technology is in leaps and bounds, the design of the WebCrawler is still lame. There is not much of a significant improvement in the way the WebCrawler fetches the surface Web pages. However, from the rise of Web Crawlers since 1994 [4], there is been significant change in the way the WebCrawler, crawls the Web. Thus, we can get a clear view that more technology and investment is laid on the side where crawling of WebCrawler is an issue. This large investment in its crawl is understandable, since new Web pages are added very month [3, 5]. It is necessary that the WebCrawler crawls and indexes these pages at a very faster rate. A typical Google WebCrawler can index its whole database of Web pages in just 4 weeks [6]. In spite of these fast crawls by the WebCrawler, we still find the information available through search engines as stale.

As we see the left out, unseen or ignored part of the WebCrawler was its ability to penetrate deep inside the Web server to gather information from the databases. The following chapters of this report will propose a prototype of a WebCrawler as to how can the WebCrawler access these hidden databases and get the most information from it.

Chapter 2 will discuss about the background and related works and few models of the WebCrawler made so far. Chapter 3 will give an overview of “The Hidden Web”. In chapter 4, we will talk about the architecture of the proposed WebCrawler. Chapter 5 will be presenting with the various aspects of the WebCrawler’s functions and working of it. Chapter 6 will cover the results obtained from the proposed prototype and chapter 7 will conclude the paper.

# 2. BACKGROUD AND RELATED WORKS

The development of WebCrawler began in January 27, 1994 by Brain Pinkerton, a student at University of Washington. He built the WebCrawler in his spare time. In the beginning, the WebCrawler was just a desktop application; not like a Web service as of today. It was first launched on the Web in April 20, 1994. From 1994 to present, the WebCrawler has undergone through most significant and radical changes.

The initial WebCrawler's were not very good in speed and in their precision. With passing of time, better search algorithms were used that helped crawlers to crawl more speedily and with higher precision. We will now take a look at the design of the basic WebCrawler.

## 2.1. The Design of the WebCrawler

The World Wide Web is decentralized by nature. Anybody can add Web pages, servers, documents and hypertext links. In WebCrawler's, discovering new Web pages is an important feature due to the dynamic organization of the Web. For any WebCrawler, discovering Web pages means to identify Uniform Resource Locators (URLs). These URLs point to different network resources. Once the WebCrawler has that URL, it can decide weather to search the document and retrieve it or not. Once the document is been retrieved, the WebCrawler will place the URL into the retrieved URL database. Each and every single time, when WebCrawler comes across any URL, it performs three basic steps.

In the first step, it checks the URL with the retrieved URL database to see if the entry is there or not. If the entry is already in the retrieved URL database, then it means that the WebCrawler has already visited that page and hence no need to retrieve it again. After placing the entry in the retrieved URL database, in the second step the WebCrawler will then parse the Web page for any outbound and inbound links. These outbound links are check and stored in retrieved URL database later on, while the inbound links are stored in a separate small hash table designed for that particular Website. After the Web page is parsed for the outbound and inbound links, during the third step the Web page is parsed and the information collected is indexed into the main database.

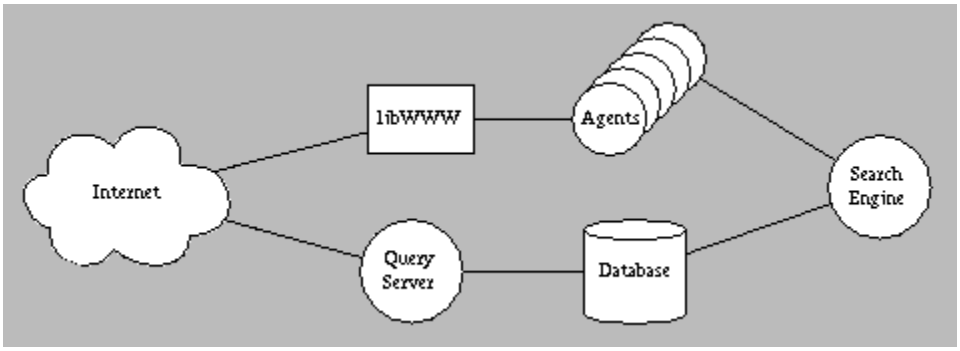



**Figure 1: Software components of the WebCrawler** [7]

For each of the three steps discussed above, the WebCrawler has a particular software component designed. The connections between the components are as shown in Figure 1

In the architecture, as shown above, the search engine controls the progress of WebCrawler. The search engine will decide which document should be explored first for the initiating the information retrieval. Normally, every search engine begins with a preset amount of Web links, formed in the Web directory [7]. The database handles the persistent storage of the Web page metadata, the links between them, and the full-text

index. The "agents" are responsible for retrieving the Web pages from the network at the direction of the search engine. Finally, the Query Server implements the query service provided to the Internet. Each of these components is described in detail below.

### 2.1.1. The Search Engine

The WebCrawler discovers new Web pages by starting with a known set of Web pages available from Web directory. Examining the outbound links from them, the WebCrawler will follow one of the links that leads to a new Web page, and then repeating the whole process. Another way to think of it is that the Web is a large directed graph and that the WebCrawler is simply exploring the graph using a graph traversal algorithm.

Figure 2 shows an example of the Web as a graph. Imagine that the WebCrawler has already visited Web page A on Server 1 and Web page E on Server 3 and is now deciding which new Web pages to visit. Web page A has links to Web page B, C and E, while Web page E has links to Web page's D and F. The WebCrawler will select one of Web pages B, C or D to visit next, based on the details of the search it is executing.

The search engine is responsible not only for determining which Web pages to visit, but which types of Web pages to visit. The file which WebCrawler is unable to index, such as pictures, sound, PostScript or binary data is not retrieved from the Web. If they are erroneously retrieved, they are ignored during the indexing step. This file type discrimination is applied during both kinds of searching. The only difference between running the WebCrawler in indexing mode and running it in real-time search mode is the discovery strategy employed by the search engine.

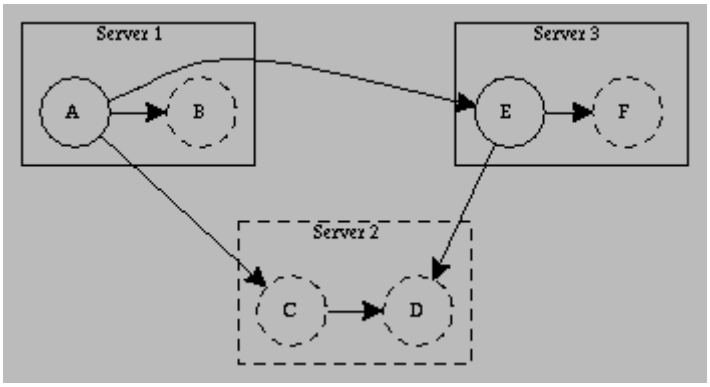



**Figure 2: The Web as a graph** [7]

### 2.1.2. Indexing mode

The ultimate goal of any search engine is to build databases of larger indexes. If the WebCrawler index has enough space for 50,000 Web pages, then those Web pages should be more relevant ones. For a Web index, one solution is that those Web pages should come from as many different servers as possible. The WebCrawler takes the following approach: it uses a modified breadth-first algorithm to ensure that every server has at least one Web page represented in the index. This strategy is very effective. The most frequent feedback about the WebCrawler is that it has great coverage and that nearly every server is represented.

In detail, a WebCrawler indexing run proceeds as follows: every time a Web page on a new server is found, that server is placed on a list of servers to be visited right away. Before any other Web pages are visited, a Web page on each of the new servers is retrieved and indexed. When all known servers have been visited, indexing proceeds sequentially through a list of all servers until a new one is found, at which point the process repeats. During indexing, the WebCrawler runs either for a certain amount of time, or until it has retrieved some number of Web pages. Normally, the WebCrawler can

build an index at the rate of about 1000 Web pages an hour on a 486-based PC running NEXTSTEP [7].

### 2.1.3. Real-time search mode

The real time search mode is when the WebCrawler has to find the relevant Web pages based on the users query. The key thing in such kind of search mode is to follow the URLs from the Web pages that are similar to what the user wants, and thus these URLs will lead to more relevant Web pages. Such a kind of approach roughly captures the way people navigate the Web: they find a Web page about a topic related to what they are looking for and follow links from there.

In order to find an initial list of similar Web pages, the WebCrawler runs the user's query against its index. A single Web page can also be used as a starting point or the whole category/sub category of the Web directory is used, but using the index is much more efficient. From the list, the most relevant Web pages are noted, and any unexplored links from those Web pages are followed. As new Web pages are retrieved, they are added to the index, and the query is re-run. The results of the query are sorted by relevance, and new Web pages near the top of the list become candidates for further exploration. The process is iterated either until the WebCrawler has found enough similar Web pages to satisfy the user or until a time limit is reached.

However, there is a problem with this approach. The WebCrawler blindly follows links from the Web pages, which may end up in following irrelevant path. For e.g. if the WebCrawler is searching for pages about sport news, it may come across Web pages that have sporting accessories. Whenever, people browse through the Web pages they follow

the links based on the anchor text, these are the words that describes about other Web pages. People click on these links and traverse in that particular direction.

Ideally, the WebCrawler should choose among several of these links, preferring the one that made the most sense. Although the WebCrawler's reasoning ability is somewhat less than that of a human, it does have a basis for evaluating each link: the similarity of the anchor text to the user's query. To evaluate this similarity, the WebCrawler makes a small full-text index from the anchor text in a document and applies the users query to select the most relevant link. Searching the anchor text in this way works well, but anchor texts are usually short and full-text indexing does not work as well as it could. More sophisticated full-text tools would help greatly, particularly the application of a thesaurus to expand the anchor text.

The basic idea of following different Web pages from one to another using the links came was first demonstrated to work in the Fish search [8]. The WebCrawler extends that concept to initiate the search using the index, and to follow links in an intelligent order.

### 2.1.4. Agents

In order to actually retrieve Web pages, the search engines will invoke “agents”. The task of the agent is simple, to retrieve the URL; in response to the task assigned, the agent will either returned the whole Web page or a reason why the Web page could not be retrieved. The agent uses the CERN WWW library [9], which gives it the ability to access several types of content with several different protocols, including HTTP, FTP and Gopher.

These agents are all ran as separate process. Since there might be few issues like server been down or the network having a bottleneck. A typical WebCrawler contains

like 15 agents in parallel. The search engine decides a new URL, finds a free agent and then assigns that URL to the free agent. When that agent responds back, it gets new URL. As a practical matter, running agents in separate processes helps isolate the main WebCrawler process from memory leaks and errors in the agent.

### 2.1.5. The Database

The WebCrawler's database contains two separate pieces: a full-text index and a representation of the Web as a graph. The database is stored on disk, and is updated as Web pages are added. To protect the database from system crashes, updates are made under the scope of transactions that are committed every few hundred documents.

The index is inverted in order to execute queries faster by looking up a word and producing a list of pointers to the Web pages that contain that word. More complex queries are handled by combining the Web page lists for several words with conventional set operations. The index uses a vector-space model for handling queries [10]. Indexing by titles is a problem. The titles are an optional part of an HTML document, and 20% of the documents that the WebCrawler visits do not have them. Even if that figure is an overestimate of the missing titles in the network as a whole, basing an index only on titles would omit a significant fraction of documents from the index. Furthermore, titles don't always reflect the content of a document.

The WebCrawler captures more of what people want to know by indexing titles and content. Most people who use the WebCrawler to find what they are looking for, takes them several tries.

In order to prepare a Web page for indexing, a lexical analyzer breaks it down into a stream of words that includes tokens from both the title and the body of the Web page.

The words are run through a "stop list" to prevent common words from being indexed, and they are weighted by their frequency in the Web page divided by their frequency in a reference domain. Words that appear frequently in the Web page, and infrequently in the reference domain are weighted most highly, while words that appear infrequently in either are given lower weights. This type of weighting is commonly called peculiarity weighting. Information retrieval systems are usually measured in two ways: *precision* and *recall* [10]. Precision is the measurement of how well the retrieved Web pages match the query, while recall indicates what fraction of the relevant Web pages are retrieved by the query. For example, if an index contained ten Web pages, five of which were about laptops, then a query for `computers' that retrieved four Web pages about laptops and two others would have a precision of 0.66 and a recall of 0.80.

Recall is satisfactory in the WebCrawler, and in the other indexing systems available on the Internet today i.e. finding of enough relevant Web pages is not the problem. Instead, precision suffers because these systems give many false positives. These false positives occur when people modify their Web pages with a list of keywords that helps that Web page to increase its rank in the search engine database. Web pages returned in response to a keyword search need only contain the requested keywords and may not be what the user is looking for. Weighting the Web pages returned by a query helps, but does not completely eliminate irrelevant Web pages.

Another factor which limits the precision of queries is that users do not submit well-focused queries. In general, a precise query will have more words in it. For e.g. if a person is looking for “laptop portable speakers” then it will be useless if he enters words like “computer speakers” or “laptop” or “laptop speakers”. Unfortunately, the average

number of words in a query submitted to the WebCrawler are about 1.5, barely enough to narrow in on a precise set of documents [7].

However, there are still issues over the scheme been applied in recall and precision calculation. Since the Web is not just source of one kind of information, it has lots of information all of different kind. Hence, using the technique of recall and precision to calculate the relevant of the pages might be correct for one type of information, but might fail for another type.

The second part of the database stores data about Web pages, links and servers. Entire URLs are not stored; instead, they are broken down into objects that describe the server and the Web page itself. A link in a Web page is simply a pointer to another Web page. Each object is stored in a separate btree on disk; Web pages in one, servers in another, and links in the last. Separating the data in this way allows the WebCrawler to scan the list of servers quickly to select unexplored servers or the least recently accessed server.

### 2.1.6. The Query Server

The Query Server implements the index for the WebCrawler, a search service available via a Web page on the Web [11]. The query model as shown here is a simple vector-space query model on the full-text database described above. Users enter keywords as their query, and the titles and URLs of Web pages containing some or all of those words are retrieved from the index and presented to the user as an ordered list sorted by relevance. In this model, relevance is the sum, over all words in the query, of the product of the word's weight in the Web page and its weight in the query, all divided by the number of words in the query. Although simple, this interface is powerful and can find related documents with ease.

The servers are normally built by breadth-first search which ensures that every single server having useful content has several pages represented in the index. Such a type of scheme is important to users, as they can usually navigate within a server more easily than navigating across servers. If a search tool identifies a server as having some relevant information, users will probably be able to find what they are looking for.

This strategy has another advantage as well. Indexing Web pages from one server at a time spreads the indexing load among servers. In a run over the entire Web, each server might see an access every few hours, at worst. This load is negligible, and for most servers is lost in the background of everyday use. When the WebCrawler is run in more restricted domains (for e.g. any university server), each server will see more frequent accesses, but the load is still less than that of other search strategies.

## 3. THE HIDDEN WEB AND VARIOUS APPROACHES TO DISCOVER IT

People find the information they need by visiting numerous Web pages and also by following the links obtained from those Web pages respectively. A user has an ability to browse through the Web pages he desires by following those links. Even though if the Web page is not made public. There are numerous resources over the internet. Most of these resources are hidden behind the database. Hence, the user is only able to access it once he makes a query into the database. With the help of search engines, the general problem of finding necessary and important data is been resolved.

According to recent studies, the data in the hidden Web can be around 7,500 and 91,850 terabytes [12] and it is still increasing. Which means that there needs to be a mechanism to extract the resources from this hidden Web and present it to the user. The entry point for the hidden Web is the forms. [13, 14] Whenever a WebCrawler encounters a form, it needs to know the domain in which it acquired the form and the type of contents that it will be requiring to fill up the form. [12, 15, 16, 17]. There are many factors which makes the task complicated. The Web is a constantly changing resource. New resources are added and the old resources are modified or removed. A scalable solution, suitable for a large-scale integration or deep-crawling task, is to automatically find the hidden-Web sources.

Another problem been encountered is that no Web form has same structure. For e.g. any user looking for books on the Web will encounter different types of forms on different Websites. Hence, we have different schemas in the form structure which makes reading and understanding of the forms more complicated. [12, 15, 16, 17]. Hence, the

search for forms should be in a broader sense. According to recent study, there are like 307,000 deep Websites [12] and an average of 4.2 query interfaces with the help of forms for those Web sites [12, 18]. Hence, searching for tens, hundreds or even thousands of forms, which are relevant to the integration task among billions of Web pages, is really like looking for a few needles in a haystack.

In order to be practical, the search process must be *efficient* and should avoid visiting large unproductive portions of the Web. One possible solution for this problem would be to perform a full crawl of the Web, but this would be highly inefficient. The reason is that the crawling of the Web will take weeks and looking for Websites that have forms in it is in small quantity. Hence, we would be doing unnecessary crawling for a long time. Another approach will be to use a focused crawler. They try to retrieve only a subset of the pages on the Web that are relevant to a particular topic. Such kind of approach will lead to better indexing and it will also improve the crawling efficiency than the first approach [12]. This approach might not work accurately in some cases, where we encounter more forms within parent forms. For e.g. in a shopping Web site a user fills up the keywords in the form on the front, when the search is made, there is another form that can be used to sort the results by price or by popularity and so on.

The approach been discussed in this paper will look for the first page on the shopping Website and will extract all the forms present over there and will make searches accordingly in order to eliminate the other forms.

# 4. APPROACH

Consider the following scenario. A user is doing shopping for a “computer” on the Web. He visits numerous shopping Websites looking for different types of computer. He looks up the price on all different Websites. Finally, after collecting information from various shopping Websites, he summarizes them and makes his choice. Now as we see in this, its kind of a tedious work the user is doing; which involves visiting numerous Websites and entering the keywords into the forms and then collecting the results.

## 4.1. The Basic Structure

The prototype of the WebCrawler been presented here will do the same way as described above.

The whole crawler can be divided into 4 different stages.

1) Analyzer

2) Parser

3) Composer

4) Result analyzer

### 4.1.1. Analyzer

The World Wide Web has millions of Web pages. Some of them will have forms. These forms will act as an entry point for the vast information hidden behind them. Our prototype WebCrawler will be scanning millions of Web pages to find such kinds of form. However, different domain Websites will have different types of forms in it. For e.g. a hospital Website might have a form used to find information about the patients and other health diseases. A Website related with weather information, will have a form used

to get the weather and other related information for that particular area. Our prototype will be analyzing most of the Websites and will extract the forms that are related with shopping. The part of going to the Web and looking for maximum URLs and then extracting specially shopping URLs which has forms will be taken care by our Analyzer part.

The analyzer will be analyzing each and every Web page the crawler comes across. It will scan the Web page to see if the Web page can be used as search page to retrieve information or not. It will basically see if the Web page has some form fields or not. Like for e.g. product name or type of product looking for. Also, the analyzer will help us to find the difference between a registration page and a query page also. This is one of the most important components while looking into the shopping Web pages since most of the shopping Web pages also have registration forms. Hence, in order to increase the efficiency of the analyzer in searching for best shopping Web pages and thereby providing us with the best results; in the crawler, we have to place a module which can identify between shopping forms and other types of forms as well.

### 4.1.2. Parser

A form can have various kinds of input fields. Once the Web page is detected from where the form can be filled and made a search; parser is called. The main job of the parser is to look for various types of forms found on that Web page. Once a suitable form is found from where the query can be made; that particular form is been separated out and passed it on to the composer. It is very essential that the form which is been extracted out by the parser can be used to perform users query. For e.g. a user wants to search for “laptop”, hence the form been extracted should be able to make a search with keyword

“laptop”. Sometimes the Web page have many other forms like ones for sign in. hence, since the fields for the sign in form does not match with the ones in which “laptop” keyword should fit, such sign in form can be ignored.

### 4.1.3. Composer

The user is been asked to store the keywords that he would like to be in the search. These keywords are been stored and are used by composer in filling up the forms. Sometimes Web pages are been designed to trick WebCrawler’s. Hence, it is more likely that parser would have selected the wrong form to fill.

The composer will try to fill up the form with the information provided by the user. While trying to do so, if the information does not fit in the form it means that such kind of form is useless. For e.g. WebCrawler is trying to make a search with keywords “laptop” in a shopping Website of jewelers.

Those forms that are been successfully filled are then executed and the results obtained are been stored for result analyzer. The designing of the composer needs to be taken special care because once these forms are filled and submitted; most of the shopping Websites will return cookies along with the search results. These cookies are needed to be handled or else the program will start throwing exceptions. Thus, composer is one of the key components in our WebCrawler.

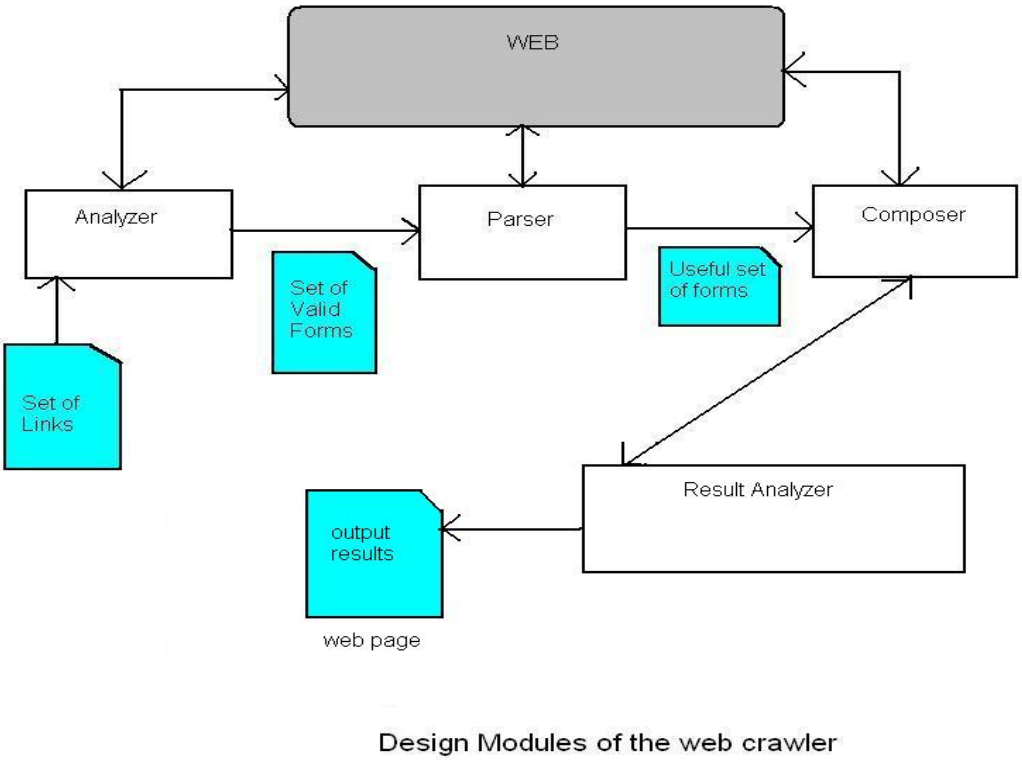
### 4.1.4. Result analyzer

Once the composer fills up the forms and executes them, the results of the forms are been obtained in the form of Web pages. Now it is the job of the result analyzer to

analyze each and every result on that particular Web page and then extract the necessary results from it.

The input to the result analyzer will be a set of Web pages containing results obtained after executing the forms. These results will be evaluated depending upon the type of search made. If a particular Web page has results that are more in number corresponding to the other ones then we can say that particular Web site is more likely to be the right one and hence, if similar keywords search will be made on that particular Web site then the search results will be more likely to be the right ones. Hence, result analyzer will play an important role to determine the accuracy of the WebCrawler. The result analyzer will be modifying the knowledge base till it reaches optimization level.

In the Figure 3, we can see various different modules of our prototype WebCrawler




Design Modules of the web crawler

**Figure 3: Prototype of WebCrawler**

Certain assumptions were been made before the designing part of the WebCrawler. These assumptions include

1. Domain is been specified already
2. Set of Websites found in that particular domain is been written down in a text file. These Websites were all from one single domain, however there were good Websites and bad Websites also.

#### 4.1.5. Characteristics of a bad Website

The Website was classified as bad Website when it encountered any one or more of the following issues.

1. The Website when called loaded cookies in extensive numbers so as to cause the analyzer to throw exceptions.
2. Websites has numerous html forms tag in it that makes the crawlers internal cache run out of resources and hereby causing memory leaks.
3. Websites which are designed using Ajax and JavaScript's are a limitation to our prototype. The limitation has been aroused because of the development environment been used does not support fully JavaScript's and Ajax.

### 4.2. Design Modules

The design modules for prototype of WebCrawler are as below.

#### 4.2.1. Analyzer

The main task of the analyzer is to look for Web pages that have forms in it. A set of URL's of Websites are been given to the analyzer from a text file. The analyzer will read

each and every single URL from that text file and will then start loading them into URL class. The URL class object is then passed on to the WebClient object which will then load the URL object and will return a Web page object. This Web page object has the whole Web page of that particular Website into it. It's just like the Document class of java.

Once the Web page object is been obtained, it is been passed on to a separate function to look for forms in it. In order to look for forms into a Web page object we are using a DOM Parser. This parser is been provided with a regular expression "//table" which means to look for html tag table at any depth in that particular Web page object. It is been observed by performing various experiments that using of regular expression like "//table" gives more inaccurate results then compared to "//table//table". The regular expression "//table//table" means the parser has to look for table at any depth and for that particular table there should be another table at any depth.

The function been called is of the form

```
ArrayList a =
      getHtmlElementSByXPath ("//table//table", HtmlParser object);
```

As we see above, we are passing the regular expression and a parser object. Below, is the code that is been executed when the function is been called. We see that DOM Parser is been provided with the regular expression path. Later on, the DOM Parser returns with all the selected Nodes that are been obtained corresponding to the regular expression. All these nodes will be containing html tag table and hence, they will be representing actual tables in the form of nodes. All these nodes, where each node corresponds to a particular table will be stored in ArrayList object.

```
ArrayList getHtmlElementSByXPath (final String exp, final
HtmlPage page) throws Exception {
     Final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
          return (ArrayList) xpath.selectNodes(page);
}
```

### 4.2.2. Parser and Composer

The main job of the parser is to look for html forms on a Web page and see if those forms are actually the search forms and not registration forms. Based upon the survey been made on large number of Websites, it is been observed that these forms which are designed for search function has specific type of format. Normally, the submit buttons of these forms are found to be named as “GO” or “Search”. Besides this, they will contain one common structure of like having a text field named as “Keywords” or “ks” (Keyword Search) or no name at all and a submit button. However, this find of format of having text field with that particular name and a submit button are not found in all the Websites, but they are the most common structures found and it also results into large number of successful search results. Hence, in our prototype design, we will be using these structures in our parser.

The function that does the job of the Parser is as shown below:-

```
static void formRipper(String UrlLink){
try{
  final WebClient webClient = new WebClient();
  final URL url = new URL(UrlLink);
  final HtmlPage page1 = (HtmlPage)webClient.getPage(url);
```

```
List l = page1.getForms();
int i=0;
HtmlInput ip = null;
HtmlTextInput textField = null;
  while(i < l.size())
  {
   final HtmlForm form = (HtmlForm)l.get(i);
   try{
     ip = (HtmlInput)form.getInputByName("Search");
   }catch(Exception e){}

   if(ip == null)      // Search did not work
   {
     try{
       ip = (HtmlInput) form.getInputByName("search");
     }catch(Exception e){}

   if(ip == null) // search did not work
   {
     try{
       ip = (HtmlInput) form.getInputByName("go");
     }catch(Exception e){}

   if(ip == null)
   {
     try{
       ip = (HtmlInput) form.getInputByName("GO");
     }catch(Exception e){}
   }   // if ends
   }   // if ends
   }   // if ends
```

```
     if(ip == null)
     {
/* get all inputs and see if there is any image input with
* name search or go
*/
ArrayList AllInputs = new
ArrayList(form.getInputsByName(""));

      try{
        ip = (HtmlInput)AllInputs.get(0);
      }
      catch(Exception e){}
      /* searching for image ends */

      if(ip == null)
      { i++;
        continue;
      }
     }

     // now looking for the text part in the same form
ArrayList ListOfText = new
ArrayList(form.getInputsByValue(""));

     if(ListOfText.size() == 0)
       ListOfText = new
       ArrayList(form.getInputsByName(“keywords”));

     for(int k=0;k < ListOfText.size(); k++)
       System.out.println(k + “ “ +
                          ListOfText.get(k).toString());
```

```
if(ListOfText.size() >= 1){
  textField = (HtmlTextInput)ListOfText.get(0);
try{
  submitThisForm(ip,textField);
}
catch(Exception e){}
break;
}
else
  i++;
}    // while ends
}catch(Exception e)
{System.out.println(e);}
}    // formRipper ends
```

We begin by picking up a URL from the input file and submit it to WebClient. The WebClient will return an HtmlPage object. Then we will be using a method called getForms( ) which will be internally using a parser having regular expression “//form”. This will return to us a list of all nodes that begin with html tag form. Now, we enter into the heart of the parser.

Once, we get the list of forms, we start iterating through it one after another. While iterating through each and every form, we will do the following actions.

1. Look for a keyword “search” or “go”. If we encounter any of these keywords, it will mean that we found the submit button of the form. If the form does not have any of these keywords, it will mean that the form does not possess any submit button. Hence we can go to step 2.

2. Look for images that have keywords like “search” or “go”. Shopping Websites are now more interactive and attractive. Hence, in place of using traditional looking submit button, it might be possible that they have used an image button with name search or go. And looking at the number of URL’s found on the Web for shopping, around 60 % of the Websites will use image or any other method rather than a plain submit type button. Once, we find such kind of input type button, we can proceed to step 3 or we can pick up the next form and proceed to step 1.
3. After finding the submit button, now its time to look for text field. The text field will be helpful in filling up the keywords that user has specified. Once we find the text field we are all set to fill it up with the user keywords (done using text field.setAttributes (“user keywords”) and clicking on submit button. It is done through submitButton.click ( ) the function submitButton.click ( ) will return a new Web page object that will have the result page from that particular Website. After the result page is been obtained, we pass it on to Result Analyzer to analyze the results from that particular Web page.

Please note, in all of the above steps, getInputsByName ( ) is the function been used to obtain the nodes that correspond to specific name. For e.g. getInputsByName (“search”) will return with the node that has name = “search”. Since the function getInputsByName ( ) is case-sensitive, the results obtained by getInputsByName (“search”) and getInputsByName (“Search”) will be different.

### 4.2.3. Result Analyzer

The result analyzer will analyze each and every Web page it receives for search results obtained by making the query done by composer. The piece of code that does this is presented below:-

```
void findTables(HtmlPage hp,PrintStream p) throws Exception
{
ArrayList a = getHtmlElementSByXPath("//table//table",hp);

findExactOne(a,p);
}    // fn ends

/* The function given below will take one single table as
* input and will extract all the rows from it. It will look
* for the one that has “$” sign.
*/

void findExactOne(ArrayList a, PrintStream p)
{
  try{
     for(int i=0;i<a.size();i++)
     {
     HtmlTable ht = (HtmlTable)a.get(i);
     List rows = ht.getRows();
   for(final Iterator rowIterator = rows.iterator();
   rowIterator.hasNext();)
   {
     final HtmlTableRow row = (HtmlTableRow)
                              rowIterator.next()
     final List cells = row.getCells();
```

```
    for(final Iterator cellIterator = cells.iterator();
    cellIterator.hasNext();)
    {
    Final HtmlTableCell cell = (HtmlTableCell)
                                cellIterator.next();

    if(cell.asText().indexOf("$",0) > 0){
          rowExtractor(ht,p);
    }
    }
}
}
}catch(Exception e)
{
System.out.println("found in findExactOne :"+e);
}
}    // fn ends

/* this function will extract rows and will stuff it into a
* file it will extract only those rows that has $ sign in
* it
*/

void rowExtractor(HtmlTable ht,PrintStream p)
{
int rowSize = ht.getRows().size();
List rows = ht.getRows();
p.println("<p>");

  for(int i=0; i<rowSize; i++)
```

```
  {
     HtmlTableRow htr = (HtmlTableRow)rows.get(i);
     p.println("<i>Entry # "+i+"</i>");

     for(int j=0; j<htr.getCells().size();j++)
     {
     p.println("<div align='left'>");
     p.println(ht.getCellAt(i,j).asText());
     p.println("</div>");
     }
     }
       p.println("</p>");

     }   // fn ends

/* this function will extract one single row from a given
table */
void rowCellExtractor(HtmlTableCell htc,PrintStream p)
{
          p.println(htc.asXml());
}   // fn ends
```

The result analyzer is called by passing a Web page as an object. Once the Web page object is obtained, the following steps take place.

The Web page is looked for html tag table. Once all these tables are obtained in the form of ArrayList they will be passed on to step 2.

In this step, the function findExactTable ( ) is called which will take one table at a time and will look for “$”. Since each and every Web page result obtained after making a

search on shopping Website will have $ tag, we will be looking for all the tables that have this tag in their rows. If the table has that particular tag, it means that we found the right table.

Once the right table is found, rowExtractor ( ) is called. This function will extract each and every single row of the table preceding it. The results on selecting the table preceding it leads to better accuracy rather than selecting the table that has $ tag. The rows of the preceding table is been printed into the output file.

## 5. OBSERVATIONS AND RESULTS

Several tests were run by making searches with following keywords:-

a. “the alchemist”

b. “computer”

c. “speakers”

d. diamond

The observations obtained by making the above searches is represented as below

| Keyword Search | Number of successful URL’s | Total number of URL’s | Total number of entries T | Relevant entries (approx.) R | %age of success (R/T) * 100 |
|---|---|---|---|---|---|
| The alchemist | 24 | 49 | 862 | 215 | 24.94 |
| Computer | 29 | 51 | 1069 | 267 | 24.97 |
| Speakers | 28 | 53 | 976 | 244 | 25.00 |
| diamond | 29 | 53 | 1133 | 258 | 22.77 |

**Figure 4: with //table//table regular expression for 284 URLs**

### 5.1 //table OR //table//table

A change was made in the analyzer by changing the regular expression from //table//table to //table. Due to this, the analyzer will look only for the top level table tags. These table tags may or may not contain sub tables in them. This should degrade the performance of the analyzer and as well as the WebCrawler itself. The reason on selecting //table//table to be more suitable is that this regular expression will look for table tags which are child tables of some parent table. It is because the table tag at the top level is mainly used for formatting the whole Web page. The subsequent table tags are then used for actual contents in them. Hence, selecting //table tag will give us the master table and also the subsequent child tables. Thus, there will be more duplication of the results been produced by the WebCrawler if master table is introduced. The performance chart is shown below in Figure 5.

| Keyword Search | Number of successful URL's | Total number of URL's | Total number of entries T | Relevant entries (approx.) R | %age of success (R/T) * 100 |
|---|---|---|---|---|---|
| The alchemist | 24 | 49 | 563 | 141 | 25.04 |
| Computer | 29 | 51 | 817 | 204 | 24.96 |
| Speakers | 28 | 53 | 652 | 163 | 25.00 |
| diamond | 29 | 53 | 914 | 228 | 24.94 |

**Figure 5: with //table regular expression for 284 URL's**

**5.2 Good and Bad URL's**

Figure 4 and Figure 5 are based on the URL's that are related to shopping Websites. Out of 100 URL's been searched over the internet, on average 17 were recognized with its form credibility. In other words, the WebCrawler was presented with 100 shopping URL's. During Parsing of these 100 URL's, the WebCrawler was only able to find 17 good URL's. These 17 URL's were the ones that matched the form criteria (the form will have a submit/image button saying search or go and it will contain one textfield for keywords entry). Though 17 out of 100 is not a good number to say, but there were many other issues as well that prevented the WebCrawler from identifying the Web page.

Today looking at the World Wide Web, the Web pages are just not written in plain HTML. There are numbers other scripting languages like PHP, JavaScript, AJAX been used in one single Web page. The key component that is been used in designing this WebCrawler is HtmlUnit [tm.] The limitation of HtmlUnit is that it's still not able to function properly on advanced scripting languages like JavaScript and AJAX where the Web page is generated on the fly. This limitation of HtmlUnit was one main drawback that resulted into accuracy loss in identifying potential shopping URL's.

Some of the pioneering Web sites like amazon.com, ebay.com had special piece of code within that would prevent the WebCrawler from reading the Web page successfully. Hence, when the parser came across such Web sites it would normally throw exceptions and would continue searching for another URL.

Below is the piece of code that resulted parser to create an exception in amazon.com

```
<script type="text/javascript">//<![CDATA[
  n2RunThisWhen(
```

```
    n2sRTWTBS,
    function() {
      goLolPop = new N2SimplePopover();
      goN2Events.registerFeature('lolPop', 'goLolPop', 'n2MouseOverHotspot',
'n2MouseOutHotspot');
      goN2Events.setFeatureDelays('lolPop', 200, 400, 200);
      goLolPop.initialize('lolPopDiv', 'goLolPop', null, null, 'below', 'c');
      if (document.getElementById('lolPop_1') != undefined)
      {  goN2U.insertAdjacentHTML(document.getElementById('lolPop_1'), 'beforeEnd',
' \;<img src="http://ec1.images-amazon.com/images/G/01/x-
locale/common/icons/drop-down-icon-small-arrow.gif" width="11" style="margin:0px
2px -1px 4px;" height="11" border="0" />');
      }
    },
    'Your Lists popover' );
//]]></script>
<script type='text/javascript'>
  n2RunThisWhen(
      'holiday06Loaded',
      function()
      {
          goUSHolidayNavSwf.loadHolidaySwf(
              "swfContainer",
```

```
            "navbarTabsTable", "subnavTable", 11,
            "subnavAndSearchTable",
            "_po_holidayNavSnow",
            "http://ec1.images-amazon.com/images/G/01/nav2/images/skins/holiday-
2006/gw_snow08_nobtn._V37040087_.swf",
            "7.0r24"
         );
      },
      "initialize holiday swf logic"
    );
</script>
```

The piece of code that lead parser an exception on ebay.com is as below.

```
<script src="http://include.ebaystatic.com/js/e485/us/homepage_e4852us.js"
type="text/javascript"></script><link rel="stylesheet" type="text/css"
href="http://include.ebaystatic.com/aw/pics/us/css/homepage.css"><style
type="text/css">
  A.whitelinks:link{color:#ffffff;}
  .subtext{font-size: 11px;}
  .buttonsm {font-size: 11px; cursor: hand;}
  .btmbrdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/bottomDropShadow_20x20.gif)
repeat-x bottom;}
```

.rtbrdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/sideDropShadow_20x20.gif) repeat-y
left;}
.rt1brdr {background: #FFFFE5
url(http://pics.ebaystatic.com/aw/pics/userSitePrefs/dropshadow2_20x10.gif) repeat-y
left;}
.lftbrdr {background-color: #FEEEA3;border-left: 2px solid #F9B709;}
.lft1brdr {background-color: #FFFFE5;border-left: 2px solid #F9B709;}
.topbrdr {background-color: #FEEEA3;border-top: 2px solid #F9B709;}
.favSelect {width: 100%;}
.favNavHeader { margin: 0;padding: 0 5px 0 0;background-color: #CECEFF;border:
1px solid #A9A9F7;}
.favNavHeaderBuy {margin: 0;padding: 0 5px 0 0;background-color: #E2E3FF;border-
top: 1px solid #A9A9F7;border-right: 1px solid #A9A9F7;
border-left: 1px solid #A9A9F7;}
.favNavContent {margin: 0; padding: 5px;background-color: #FFF;border-width: 0 1px
1px 1px;border-style: solid;border-color: #A9A9F7;}
.favCenter {padding: 0 16px 0 16px;}
@media all {
IE\:HOMEPAGE {behavior:url(#default#homepage)}
}
</style><style type="text/css"><!--
.favsearchbar {

```
  margin-top:0px;
  width:785px;
  height:24px;
  padding-top:2px;
  background-color:#F3F3F3;
  padding-bottom:2px;
  padding-left:10px;
  margin-bottom:0px;}
--></style></head><body bgcolor="#FFFFFF" link="#0000FF"
onLoad="init();toolboxOnLoad();" onUnload="cleanUp();"><script
language="javascript" type="text/javascript"><!--
  if(document.all)
    document.write("<IE:HOMEPAGE ID = 'oHomePage' />");
//--></script>
```

The above were some pieces of code that resulted parser to throw the exception. Also, there were some Websites which mislead the analyzer to get the correct outputs. For e.g. in the Website of Barnes and Nobles (www.bn.com), they had an html table separately created for the price of the products. Hence, you will see a kind of generic structure like this,

```
<table>
  <tr>
```

```
        <td> Item # 1 </td>
 </tr>
<tr>
        <td> Item # 2 </td>
 </tr>
……
……
……
<tr>
        <td> Item # N </td>
 </tr>
</table>


<table>
  <tr>
        <td> Price of Item # 1 </td>
 </tr>
<tr>
        <td> Price of Item # 2 </td>
 </tr>
……
……
……
```

```
<tr>
        <td> Price of Item # N </td>
 </tr>
</table>


<table>
  Advertisements, books for sale at $ 5.00 only.
</table>
```

**Figure 6: Two separated tables**

As you see in Figure 6, there were two different tables, each one of them having different components of the same entity. Thus when the result analyzer of WebCrawler is feed with these pages, it will be looking of tables and will end up finding information not matched. Here, the analyzer will have difficult time matching the records from one table to another. These kinds of Web pages also resulted into lack of performance from the result analyzer point of view.

## 6. FUTURE WORKS

The prototype of the WebCrawler been presented here is a best-first crawler. It is been designed to search within shopping Websites. This prototype will execute queries on various shopping Websites and will gather the necessary results from them. Using this prototype, the WebCrawler can further be extended to enhance its performance by using a knowledge base. The Figure 7 shows new extended prototype of the WebCrawler.

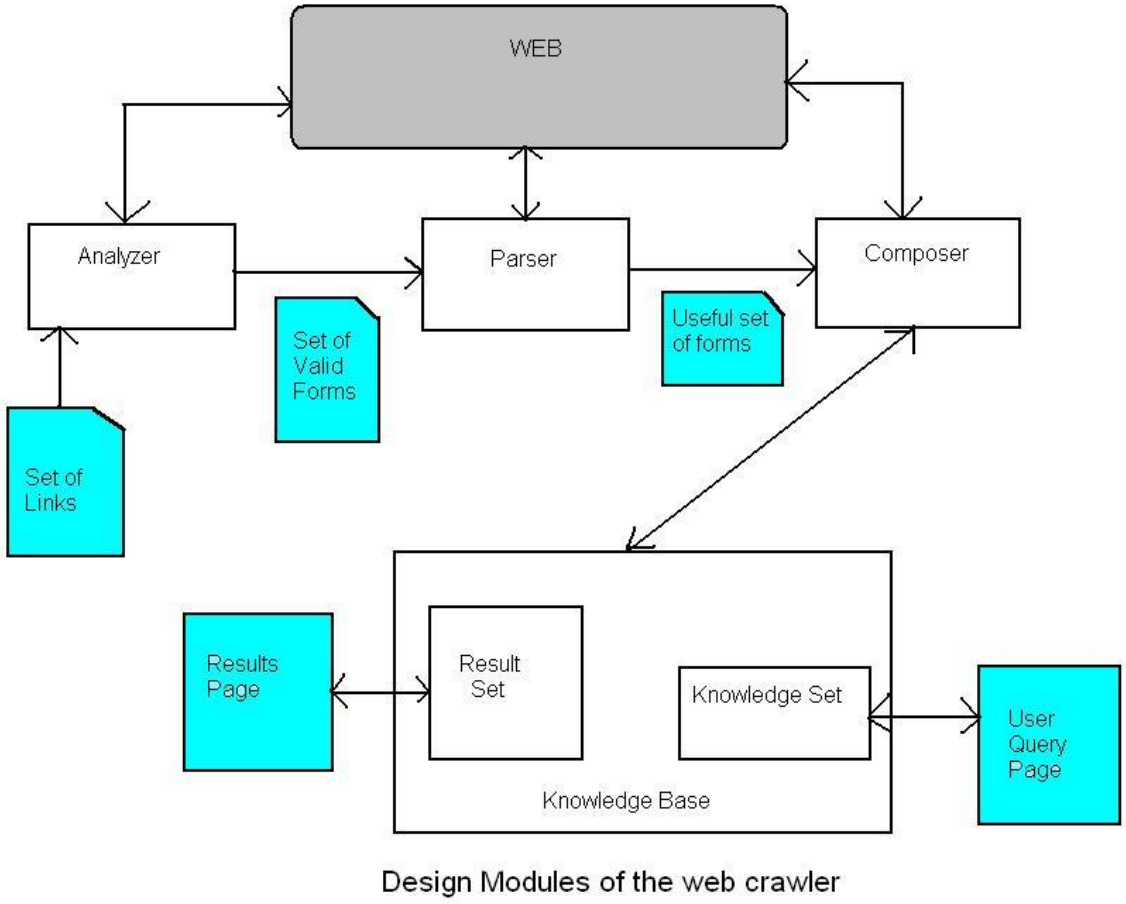



**Figure 7: Future prototype of WebCrawler**

The whole crawler can be divided into 5 different stages.

1) Analyzer

2) Parser

3) Composer

4) Knowledge base

5) Result analyzer

Analyzer, Parser, Composer will remain the same. There will be no change in designing and its implementation of them. Compared to our old prototype, this design contains Knowledge Base. Our old prototype of WebCrawler shows no intelligence. It just crawls for the Web pages and makes the query search depending upon what the user has provided. But by using this knowledge base concept, the new prototype will have intelligence which will improve the performance of the WebCrawler.

### 6.1. Knowledge base

Knowledge base will be like a database providing intelligence to the WebCrawler. User will provide a set of keywords that will be related with each other. For e.g. if the user is searching for computer speakers, he might enter words like computer desktop speakers, speakers for computer, surround speakers etc. This knowledge base will grow itself as the user will keep providing more and more words for making up the query.

The composer will be using knowledge set and will be making various queries to make search. The result set will be analyzing the results it obtained from the composer based on various searches been made using all the queries. The performance statistics (performance = total number of good results / total number of results) will be used to place the weights on the query words. For e.g. the results returned by making a search

“computer surround speakers” resulted into 65 % of overall results, while search named “computer speaker system” resulted into 85 % of overall results. Thus, using a mathematical formula, the weights of keywords computer and speakers will be more compared to “system” and “surround”. Thus, when user enters keywords like “computer speakers”, the performance of the results will be around 90 ~ 95 %.

### 6.2. Open Databases

Another suggested prototype of the WebCrawler can be for open databases. Currently, our prototype of the WebCrawler will look for shopping URL’s and extracting data from it. With the help of using open databases, the crawler can extract and analyze data flawlessly.

There can be numerous open databases. One of them is the names of students in a university. Every university in USA offers a directory service in which user can type in the name of the person and get its contact information. Our crawler can be programmed to visit such Web pages and perform a people search. In some cases, the results can be obtained from more than one particular university as well. For e.g. if making a search with a persons name can get us results from GSU and from other universities like NYU as well. This will mean that, that particular person with that name can be found in GSU and in NYU as well. Imagine the possibility of having access to numerous university databases to get information of people.

### 6.3. Image Search Crawler

The WebCrawler can also be used to search for images with a few modifications. The WebCrawler can search for images on the World Wide Web. The image search can be

based on captions [19]. Some of the WebCrawler's use a more complex algorithm to identify the images [20]. The modifications in our prototype will be done in analyzer and in parser.

In the image WebCrawler, the analyzer at the front will not be looking for shopping URL's anymore. It will crawl into the World Wide Web, potentially visiting every single Website. There will be no composer in this model. The next phase will be the parser, who will look for the image captions that matches the user interest.

An alternative model can involve the use of database in storing all the Web pages into the database from where the images can be searched by parser more speedily than looking dynamically on the Web.

### 6.4. Distributed WebCrawler

All the prototypes presented above or implemented in this report can be modified to perform crawling in distributed scale. A distributed WebCrawler will yield high performance of approximately 70 to 100 Web pages per second [21]. Recent studies focusing reduction of work load in distributed environment also involves use of p2p network [22]. Thus, we can see that implementing our prototype to work in a distributed environment can significantly improve the performance of the WebCrawler.

# 7. CONCLUSION

There are billions and billions of Web pages on World Wide Web. Generalized search on these Web pages is losing its accuracy. Hence, more and more focused crawlers are now been used. In this report, we talk about such a focused based best-first WebCrawler. The domain that it is been focused on is shopping Websites. Large number of internet users actively participates in buying and selling things using shopping Websites. Each shopping Websites have their own database to store information of all shopping items.

The prototype of the WebCrawler presented in this paper will visit all such shopping Websites and will perform users query in order to retrieve results from their databases. With the use of this WebCrawler the users does not have to go to each and every specific shopping Website and make its query search.

The prototype presented here is just an experiment combining focused WebCrawler and best first WebCrawler. Combining these two techniques we can see a reasonable performance been achieved. Using future techniques we can improve the prototype to work on large scale with high performance and hence achieving more accurate results.

## REFERENCE


[1] Kobayashi, M. AND Takeda , K. Information Retrieval of the Web, ACM Computing Surveys, 2000

[2] Kellerman, A. The Internet on Earth: Geography of Information, Publisher: John Wiley and Sons Ltd, 2002

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

[4] Resource Link: http://www.thinkpink.com/bp/WebCrawler/History.html

[5] Resource Link: http://news.netcraft.com

[6] Resource Link: http://www.thinkpink.com/bp/WebCrawler/History.html

[7] Pinkerton, B. Finding What People Want: Experiences with the WebCrawler, Second International WWW Conference,1994

[8] DeBra, P. and Post, R., Information Retrieval in the World-Wide Web: Making Client-based searching feasible, Proceedings of the second international WWW conference 1994 “Mosaic and the Web”,1994

[9] Resource Link: http://www.w3.org/Library/Status.html

[10] Salton, G. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer, Addison-Wesley Longman Publishing Co., Inc, 1989

[11] Pinkerton, C. B. The WebCrawler Index (http://www.Webcrawler.com)

[12] Barbosa, L AND Freire, J. Searching for Hidden-Web Databases, Eight International Workshop on Web and Databases, 2005

[13] Barbosa, L AND Freire, J. Siphoning Hidden-Web Data through Keyword-Based Interfaces. In Proc. of SBBD, 2004

[14] Raghavan, S AND Garcia-Molina, H. Crawling the Hidden Web. In Proc. of VLDB, 2001

[15] He, B AND Chang, K. Statistical Schema Matching across Web Query Interfaces. In Proc. of SIGMOD, 2003

[16] He, H, Meng, W, Yu, C AND Wu, Z. Automatic integration of Web search interfaces with WISE-Integrator. VLDB Journal, 2004

[17] Wu, W, Yu, C, Doan, A, AND Meng, W. An Interactive Clustering-based Approach to Integrating Source Query interfaces on the Deep Web. In Proc. of SIGMOD, 2004

[18] Chang, K., He, B., Li, C., Patel, M., AND Zhang, Z. Structured Databases on the Web: Observations and Implications. SIGMOD Record, 2004

[19] Rowe, N. Marie-4: A High-Recall, Self-Improving WebCrawler That Finds Images Using Captions, IEEE Intelligent Systems, 2002

[20] Kompatsiaris, I., Triantafyllou, E., AND Strintzis, M. A World Wide Web Region-Based image search engine, IEEE, 2001

[21] Shkapenyuk, V. AND Suel, T. Design and implementation of high performance distributed WebCrawler, Proceedings of the 18th International Conference on Data Engineering, IEEE, 2002

[22] Fei, L., Fan-Yuan, M., Yun-Ming, Y., Ming-Lu, L. AND Jia-Di, Y. Distributed High-Performance Web Crawler Based on Peer-to-Peer Network, Parallel and Distributed Computing, Applications and Technologies, 2004

## APPENDIX A: Code

```
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.*;
import java.io.*;

import com.gargoylesoftware.htmlunit.WebClient;
import com.gargoylesoftware.htmlunit.html.HtmlForm;
import com.gargoylesoftware.htmlunit.html.HtmlInput;
import com.gargoylesoftware.htmlunit.html.HtmlPage;
import com.gargoylesoftware.htmlunit.html.HtmlSubmitInput;
import com.gargoylesoftware.htmlunit.html.HtmlTextInput;
import com.gargoylesoftware.htmlunit.html.HtmlTable;
import com.gargoylesoftware.htmlunit.html.HtmlElement;
import com.gargoylesoftware.htmlunit.html.xpath.HtmlUnitXPath;
import com.gargoylesoftware.htmlunit.html.*;

/*
 * Created on Sep 24, 2006
 * this program will read the inputs from file c:/siteLinks.txt and will submit
 * url one by one to formFiller
 */

/**
 * @author milan
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class Dawn {

	static int found = 0;
	static int countFile = 0;

	/* file writing */
	static FileOutputStream out; // declare a file output object
   static PrintStream p; // declare a print stream object

   static String SEARCHFIELD;	// = new String("the alchemist");
```

```
   static int goodUrls = 0;

        public static void main(String[] args) throws Exception{

                out = new FileOutputStream("c:/results.txt");
                p = new PrintStream( out );

                // getting user keyword
                System.out.println("\n Please Enter your Search: ");
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                try {
               SEARCHFIELD = br.readLine();
             } catch (IOException ioe) {System.out.println("unable to recognize it, please
try again");}

                // reading a file
                //File f = new File("c:/goodOnes.txt");
                File f = new File("c:/siteLinks.txt");
                FileInputStream fis = new FileInputStream(f);
                BufferedInputStream bis = new BufferedInputStream(fis);
                DataInputStream dis = new DataInputStream(bis);
                String urlEntry = null;

                int totalUrls = 0;

                  while ( (urlEntry=dis.readLine()) != null ) {
                        totalUrls++;
                   formRipper(urlEntry);
                   }
                  p.close();

                  printGoodBad(totalUrls);

        }       // main ends

        /* this function will print goodurls, badurls into the result file */
        static void printGoodBad(int totalUrls)
        {
                PrintStream p = null; // declare a print stream object
                try{
                        FileOutputStream out; // declare a file output object

                    out = new FileOutputStream("c:\\finalResult.html",true);
                        p = new PrintStream( out );
```

```
                        p.println("<br></br> Total Urls:"+totalUrls+" Good
Ones:"+goodUrls);

                }catch(Exception e)
                {System.out.println("found exception in printGoodBad fn");
                }
                p.close();
        }

        /* this function will search for forms and will extract the submit button and text
field
         * for keywords entry
         */

        static void formRipper(String UrlLink)
        {

                try{
                        final WebClient WebClient = new WebClient();
                   final URL url = new URL(UrlLink); // overstock, buy,about,buyitonline
.....

                   // Get the first page
                   final HtmlPage page1 = (HtmlPage)WebClient.getPage(url);
                   List l = page1.getForms();

                   int i=0;
                   HtmlInput ip = null;
                   HtmlTextInput textField = null;

                   while(i < l.size())
                   {
                        //p.println("---------------------------------------------"+"\r\n");

                        final HtmlForm form = (HtmlForm)l.get(i);

                        try{

                        ip = (HtmlInput)form.getInputByName("Search");
                        }catch(Exception e)
                                {}              // writing try - catch to keep control over
here

                        if(ip == null)          // Search did not work
                        {
                                try{
```

```
                        ip = (HtmlInput) form.getInputByName("search");
                        }catch(Exception e)
                                {}
                        if(ip == null)   // search did not work
                        {
                                try{

                                ip = (HtmlInput) form.getInputByName("go");
                                }catch(Exception e)
                                        {}
                                if(ip == null)
                                {
                                        try{

                                        ip = (HtmlInput)
form.getInputByName("GO");
                                        }catch(Exception e)
                                                {}
                                }
                        }
                }
                if(ip == null)   // means we did not got anything from above ... then
just continue
                {
                        /* get all inputs and see if there is any image input with
name search or go */

                        ArrayList AllInputs = new
ArrayList(form.getInputsByName(""));

                        try{
                        ip = (HtmlInput)AllInputs.get(0);               // assuming we
will only get one input
                        }
                        catch(Exception e){}

                        /* searching for image ends */

                        /* no hope so lets contine ... */
                        if(ip == null)
                        {i++;
                        continue;
                        }

                }
```

```
                        // now looking for the text part in the same form

                        ArrayList ListOfText = new
ArrayList(form.getInputsByValue(""));

                        if(ListOfText.size() == 0)
                                ListOfText = new
ArrayList(form.getInputsByName("keywords"));

                        for(int k=0;k < ListOfText.size(); k++)
                                System.out.println(k + "  "+ListOfText.get(k).toString());

                        if(ListOfText.size() >= 1)
                                {

                                textField = (HtmlTextInput)ListOfText.get(0);
                                try{
                                submitThisForm(ip,textField);
                                }
                                catch(Exception e){}

                                break;
                                }

                        else
                        i++;

                  }     // while ends



                }catch(Exception e)
                        {System.out.println(e);}


        }       // formRipper ends

        /* submitThisForm will take ip (button),textField and form
         * it will fillup the text field of the form and will click on the button
         * the page it will get will be stored in arraylist
         * @author milan
         */

        private static void submitThisForm(HtmlInput ip,HtmlTextInput textField) throws
Exception
```

```
        {
                try{
                FileOutputStream out; // declare a file output object
           PrintStream p; // declare a print stream object

           out = new FileOutputStream("c:\\finalResult.html",true);
                p = new PrintStream( out );

                p.println("<html><body>");

                goodUrls++;

//                Change the value of the text field
           textField.setValueAttribute(SEARCHFIELD);

           final HtmlPage page2 = (HtmlPage)ip.click();
           System.out.println("\n Web page is ->"+page2.getTitleText());
           p.println("<b> Web page is:- "+page2.getTitleText()+"</b>");
           p.println("<a href='"+page2.getWebResponse().getUrl()+"'>Click Here To See
Original Page</a>");
           p.println();
           findTables(page2,p);

           p.println("</body></html>");
           }
                catch(Exception e){}


           p.close();

          countFile++;
        }

        private static void findTables(HtmlPage hp,PrintStream p) throws Exception
        {

                ArrayList a = getHtmlElementSByXPath("//table//table",hp);
                //ArrayList a = getHtmlElementSByXPath("//table",hp);      // not a good
change, use only for experiment

                findExactOne(a,p);

                if(a.size() > 0)
                        System.out.println("\n numbers : "+a.size());

        }
```

```
        private static HtmlElement getHtmlElementByXPath(final String exp, final
HtmlPage page)
   throws Exception
        {
                        final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
                        return (HtmlElement) xpath.selectSingleNode(page);
}

        private static ArrayList getHtmlElementSByXPath(final String exp,final
HtmlPage page) throws Exception
        {
                final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
                return (ArrayList) xpath.selectNodes(page);
        }

        private static ArrayList getHtmlElementSByXPathForm(final String exp,final
HtmlForm form) throws Exception
        {
                final HtmlUnitXPath xpath = new HtmlUnitXPath(exp);
                return (ArrayList) xpath.selectNodes(form);
        }

        private static void findExactOne(ArrayList a, PrintStream p)
        {

                try{
                        for(int i=0;i<a.size();i++)
                        {

                                HtmlTable ht = (HtmlTable)a.get(i);
                                List rows = ht.getRows();

                                for (final Iterator rowIterator = rows.iterator();
rowIterator.hasNext(); )
                                {
                                   final HtmlTableRow row = (HtmlTableRow)
rowIterator.next();
                                   //System.out.println("Found row");
                                   final List cells = row.getCells();
                                   for (final Iterator cellIterator = cells.iterator();
cellIterator.hasNext();)
                                   {
                                      final HtmlTableCell cell = (HtmlTableCell)
cellIterator.next();
                                      System.out.println("   Found cell : "+ cell.asText());
```

```
                                        //finding if the cell has price tag, then its the one
                                        // we are looking for
                                        if(cell.asText().indexOf("$",0) > 0)                    //
returns > 0 if $ is found
                                        {       System.out.println("index is:
"+cell.asText().indexOf("$",0));

                                                rowExtractor(ht,p);
                                                //rowCellExtractor(cell,p);

                                        }
                                    }
                                }
                        }

                }catch(Exception e)
                {
                        System.out.println("found in findExactOne :"+e);
                }
        }       // fn ends

        /* this function will extract rows and will stuff it into a file
         * it will extract only those rows that has $ sign in it
         */
        private static void rowExtractor(HtmlTable ht,PrintStream p)
        {

                System.out.println("rowExtractor is been called");
                //p.println("\n\t ------------------------------------------------");
                int rowSize = ht.getRows().size();
                List rows = ht.getRows();
                p.println("<p>");

                for(int i=0; i<rowSize; i++)
                {
                        HtmlTableRow htr = (HtmlTableRow)rows.get(i);
                        p.println("<i>Entry # "+i+"</i>");

                        for(int j=0; j<htr.getCells().size();j++)
                        {

                                p.println("<div align='left'>");
                                p.println(ht.getCellAt(i,j).asText());
                                p.println("</div>");
```

```
                    }
                }
                p.println("</p>");

        }   // fn ends

        private static void rowCellExtractor(HtmlTableCell htc,PrintStream p)
        {

                p.println(htc.asXml());

        }   // fn ends

}       // class ends
```

## APPENDIX B: HtmlUnit

HtmlUnit is a java unit testing framework for testing Web based applications. It is similar in concept to httpunit (http://sourceforge.net/projects/httpunit) but is very different in implementation. Which one is better for you depends on how you like to write your tests. HttpUnit models the http protocol so you deal with request and response objects. HtmlUnit on the other hand, models the returned document so that you deal with pages and forms and tables.

HtmlUnit was originally written by Mike Bowler of Gargoyle Software and released under an apache style license. Since then, it has received many contributions from other developers and would not be where it is today without their assistance.

HtmlUnit is not a generic unit testing framework. It is specifically a way to simulate a browser for testing purposes and is intended to be used within another testing framework such as JUnit

For more information, please visit http://htmlunit.sourceforge.net/.