CREATING A POLITE, ADAPTIVE AND SELECTIVE INCREMENTAL CRAWLER

Christos Bouras
Research Academic Computer Technology Institute and
Computer Engineering and Informatics Dept., University of Patras,
N. Kazantzaki, University Campus, GR26500, Rio, Patras
bouras@cti.gr

Vassilis Poulopoulos
Research Academic Computer Technology Institute and
Computer Engineering and Informatics Dept., University of Patras,
N. Kazantzaki, University Campus, GR26500, Rio, Patras
poulop@ceid.upatras.gr

Athena Thanou
Computer Engineering and Informatics Dept., University of Patras,
Building B, University Campus,GR26500, Rio, Patras
thanou@ceid.upatras.gr

ABSTRACT

The expansion of the World Wide Web has led to a chaotic state where the users of the internet have to face and overcome the major problem of discovering information. For the solution of this problem, many mechanisms were created based on crawlers who are browsing the www and downloading pages. In this paper we will describe a crawling mechanism which is created in order to support data mining and processing systems and to obtain a history of the web's content. A crawler has to be efficient and polite, trying not to harm or overload the pages it is visiting. Therefore, it is extremely important to follow specific rules when crawling. In addition to these rules, the mechanism we created includes a selective incremental algorithm, which is used to make the crawler more efficient and more polite in parallel. The structure and design of the mechanism is simple, but the experimental results showed us that this simplicity makes our crawler a very strong and stable mechanism.

KEYWORDS

Crawler, Incremental and Adaptive Crawler, Data Mining, Crawling policies.

1. INTRODUCTION

The World Wide Web has grown from a few thousand pages in 1993 to more than two billion pages at present. The contributing factors to this explosive growth include the widespread use of microcomputers, advances in hardware technologies (microprocessors, memory and storage), increased ease of use in computer software packages, and most importantly tremendous opportunities that the Web offers to businesses. The consequence of the popularity of the Web as a global information system is that it is flooded with a large amount of data and information and hence finding useful information on the Web is often a tedious and frustrating experience. New tools and techniques are crucial for intelligently searching for useful information on the Web.

However, the mechanisms that were invented to make Web seem less chaotic need information and waste a great amount of time in order to collect it. Web crawlers are an essential component of all search engines and are increasingly becoming important in data mining and other indexing applications. Web crawlers (also known as spiders, robots, walkers and wanderers) are programs which browse the Web in a methodical, automated manner. They are mainly used to create a copy of all the visited pages for later processing by

mechanisms, that will index the downloaded pages to provide fast searches and further processing. However, simple crawlers can also be used by individuals to copy an entire web site to their hard drive for local viewing.

Web crawlers are dated to the period World Wide Web appeared. The first crawler was implemented by Matthew Gray Wanderer in the spring of 1993. Many papers on web crawling were presented at the first two conferences on World Wide Web [6, 7, 8]. Nevertheless, at that time, Web was not as extended as it is nowadays and thus that crawling systems did not have to confront scaling problems as they do now [12].

The crawlers are widely used today and as it was expected, the last few years, many crawlers were designed but they have not managed to collect the information required for a mechanism that implements data mining. Ubicrawler (Boldi et al., 2004) is a distributed crawler written in Java, and it has no central process. It is composed of a number of identical "agents"; and the assignment function is calculated using consistent hashing of the host names. The crawler is designed to achieve high scalability and to be tolerant to failures. WebRACE (Zeinalipour-Yazti and Dikaiakos, 2002) is a crawling and caching module implemented in Java. The system receives requests from users for downloading Web pages. The most outstanding feature of WebRACE is that it is continuously receiving new starting URLs to crawl from ("seed"). PolyBot [Shkapenyuk and Suel, 2002] is a distributed crawler written in C++ and Python. It is composed of a "crawl manager", one or more "downloaders" and one or more "DNS resolvers". Collected URLs are added to a queue on disk, and processed later to search for seen URLs in batch mode.

In this paper, we describe a different crawler that executes a very simple process as it only focuses on the content collection of web pages, ignoring completely images, stylesheet, javascript and document files. The information the crawler collects will be used by mechanisms that wish to implement data mining and they only need the content of web pages in text form. It can also be used by mechanisms that are interested in the history of a page as our crawler maintains the content of pages that have been crawled in the past. The fact that the crawler produces very simple and useful data in combination with its characteristics - it is polite, adaptive and selective – make him very powerful, useful and efficient.

The remainder of the paper is structured as follows. Section 2 presents the architecture of the crawler. Section 3 describes design issues and in particular how the crawler succeeds to be polite, adaptive and selective. Section 4 reports traps and how the crawler manages to bypass them. Section 5, presents a real life scenario and section 6 the conclusions.

2. ARCHITECTURE

The basic algorithm, executed by the web crawler, reads URLs from the Data Base as its input and executes the following loop. It reads a URL from the URL table downloads the corresponding document and extracts any links contained in it. For each of the extracted links, it ensures that it is an absolute URL and adds it to the table of URLs to download, provided it has not been encountered before.

A more detailed description of the architecture of our distributed crawler follows. We partition the crawling system into four major components. First, a number of crawlers download the URLs from the Web. Second a general manager is in charge of storing the information the crawlers collect from the URLs both in a Database and on the local disk, in a data structure of directories and subdirectories. As a result, our crawling system also consists of a Directory based system manager and a Database manager. Moreover, a Link Extractor Manager is responsible for the extraction of external links, their filtering and their addition to the main table with the URLs to be crawled. Finally, a Frequency Manager is charged with managing the visiting frequency to web pages.

Each one of the following systems, the downloading system, the system that manages external links and the system that is responsible of the visiting frequency, can execute their process in parallel. This is achieved by using either multiple terminals or threads. If terminals are used, each system should constantly be informed. When a terminal produces a result, it should immediately notify the manager of it and then, the manager is responsible for making known the new information to the rest of the terminals. On the contrary, threads' usage is preferable to different terminals because the system is always informed of its current condition and there is no need of informing either the manager or the rest of the terminals.

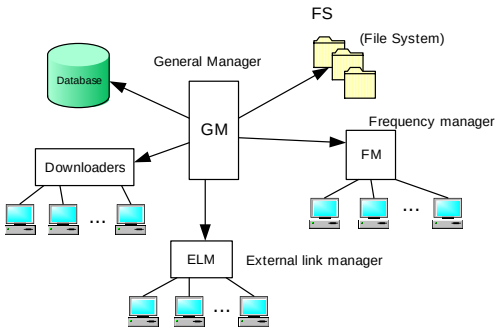

Figure 1 - The architecture of our crawling system

Nevertheless, in both cases there is a restriction in parallelism when the systems communicate with the Database and the Filesystem. If more than one thread or terminal accesses simultaneously the FileSystem or the Database in order to write or read data, then there will definitely arise a problem. Thus, writing and reading procedures should be done sequentially and when one thread accesses the repository, no other can access it at the same time.

As mentioned above, our crawler uses a Database save system and a Directory save system in order to maintain and easily explore the information acquired from crawling. The Database save system provides for a fast lookup. For example to check whether a page has been crawled or not, how often a page changes or when was the last time a page was crawled. Information about the size and the type of a page is also stored in the database. On the other hand, the Directory save system enables maintenance of the crawl history and further processing of the information retrieved. In every crawl, a directory is created on the local disk named by the date and time of creation. In this directory, the system creates other directories that take their names by the domains of the crawled URLs. The HTML code of a page is stored in the corresponding directory where there are also created more directories for the internal links of this page. For example, if the crawler should crawl the URL http://www.mysite.com/d1/d2, then a directory named www.mysite.com is constructed and another two subdirectories named d1 and d2 are created inside the directories www.mysite.com and d1 respectively.

The Link Extractor Manager is responsible for the links of web pages. During the parsing of a page, its links are extracted and they are added temporarily to a vector. Then they are compared to URLs that they have already been crawled and to URLs that are disallowed according to the Robot Exclusion Protocol.

One of the important characteristics of this crawler is that it is stand-alone. The four systems of the crawler are separate but not completely independent. The operation of one system depends on the results that are produced by the operation of another. They should be executed sequentially and parallelism can be achieved only to some extent. The program that crawls pages should be executed first and then the processes that manage visiting frequency and external links can be executed in parallel. In the following figure, number one represents crawling process, number two represents the procedure of recalculating visiting frequency and number three represents the process that manages external links.

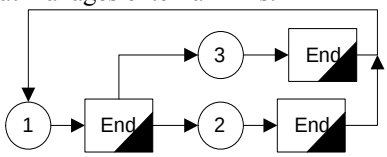

Figure 2 - The execution sequence of the three systems

Thus the four stand-alone systems of the crawler manage to increase the speed of crawling and reduce the load of the pages to be crawled.

3. DESIGN ISSUES

This section describes design issues and in particular three essential characteristics of our crawling system that make it efficient and different from other systems.

3.1 A polite crawler – Crawling Policies

The crawler that we describe is friendly to the pages it visits. Firstly, it implements the Robots Exclusion Protocol, which allows web masters to declare parts of their sites off limits to crawlers [3]. The Robots Exclusion Protocol requires a web crawler to fetch a special document containing these declarations from a web site before downloading any real content from it. Our crawling system before adding a new URL to the table with the URLs to be crawled, it examines whether this URL is excluded by the robot protocol or not. If a URL is excluded, it will not be added to the table and thus will not be crawled.

Secondly our crawler pays attention to its visits between pages of the same domain. It starts visiting all the URLs of the table whose level is not zero and extracts their URLs. Then, it repeats this procedure for all internal URLs it finds. In other words, it does not take the first URL of the table, finds its URLs and then visits these URLs to extract their internal URLs. The crawler visits the pages of the same domain with such a frequency that does not overload the pages and this is very important especially for the portals that are visited million times every day. Anecdotal evidence from access logs shows that access intervals from known crawlers vary between 20 seconds and 3-4 minutes. The lower bound of access intervals of the crawler that we describe is 10 seconds.

Another crucial point of our crawling system is that the time it visits the same pages changes daily. Therefore the pages are not overloaded the same time every day and they are able to meet efficiently requests of users during the day.

3.2 Content Based Adaptation of the Crawler

The second important characteristic of our crawling system is adaptation. Our crawler is capable of adapting its visiting frequency to the rate of change of the page content. This is implemented by the following procedure. After a page has been crawled several times, the system examines the rate of change of its content and proportionally increases or decreases the frequency it visits this page.

In more detail, the table of the Database, where the URLs that will be crawled are stored, has among others, the following fields: size, date, fr (frequency) and fr_max (maximum frequency). Size represents the size of the page, date represents the last date the page was crawled, fr_max represents the crawler's visiting frequency to the page and fr represents the number of crawlings that should be done before the page can be recrawled again. In each crawl, fr is decreased until it becomes zero. Then, the page should be crawled and after the crawling procedure, fr takes the value of fr_max. We present a flow diagram below that explains better this part of our crawling system.

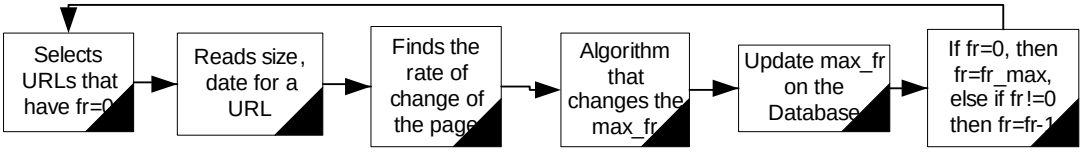

Figure 3 - Flow diagram that shows how the system manipulates visiting frequency

The algorithm that calculates new max_fr, based on the rate of change of a page is the following:

For

```
records → the entries of a page at the table
limit → the number of records that we examine for changes on the content of the page.
counter → a variable that increases when the content of two sequential records differs.


variable x takes the minimum value between records and limit
if counter is equal to zero
    then variable b takes the value of x
    and new_maxfr becomes the lower bound of (maxfr + b) / 2
else if counter is not equal to zero
    then variable b takes the value of (x / counter)
    and new_maxfr becomes the lower bound of (maxfr + b) / 2
```

example, we assume that at the table, there are 45 records of a page that its max_fr is 3 and we would like to examine the last 20 records for changes on its content. If it has changed 10 times during the period of the last 20 crawlings, then according to the above algorithm, we have:

x = 20;
b = 20/10 = 2;
new_maxfr = (3+2)/2 = 5/2 = 2,5 → 2.

That means that the crawler should crawl this page more often and in particular every two days and not three.

Finally, we should mention that an efficient crawler is only interested in collecting the differences in the content of a certain page. Visiting frequently a page whose content does not change, provides no information to the system and burdens both the system and the page.

3.3 Describing a Selective Incremental Algorithm

Crawlers whose procedure is quite simple, crawl all pages without carrying out any inspection to find out whether the pages have changed or not. Thus they are crawling pages whose content is exactly the same to that of the last crawling, extracting no information and managing only to overload both pages and the system.

However, there are also crawlers that choose not to crawl a page that has not changed recently. This is clever enough but there is a drawback as the crawling system may not be informed of the changes of the pages the internal URLs point at. For instance, the following figure represents a page and its internal links. If page A will not be crawled by the system because it has not changed recently, then none of the pages B, C, D, E, F and G will be crawled. Nevertheless, their content may have changed but the system will not be able to be informed of it.

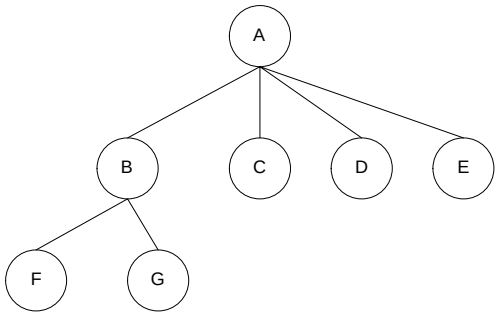

Figure 4 - A page and its internal links

A selective incremental crawler is capable to resolve the situation when a page has not changed but the pages its internal URLs point at have changed. For example, if page A has not changed, the crawler examines whether pages B, C, D and E have changed or not. If a high percentage of them have changed, page A will be crawled, otherwise it will not and the crawler will take the next URL from the table with the URLs to be crawled. From experiments that we made, we came to the conclusion that we crawl page A when 40% of its pages have changed. This percentage is not constant but we recalculate it every time is the turn of page A to

be crawled. In proportion to the number of pages that have changed or not, we increase or decrease the percentage respectively.

Another way of recrawling pages is by checking every page separately whether it has changed or not. For instance, the crawler will first examine whether page A has changed and if it has, it is crawled and then the next URL of the list is taken to be examined. Otherwise, if page A has not changed, the crawler examines its internal pages for changes. Thus, it checks page B. If its content differs from the one that was crawled the last time, it is crawled and afterwards page C is examined. If page B is the same, pages F and G are then checked. This way of recrawling may be the simplest, but it is a very time-consuming procedure and it presupposes that each page is not unknown to the system. The crawling system should know the tree diagram of each page, in other words its internal links.

On the contrary, the recrawling policy that is followed by the crawler that we describe, seems to be more clever than the one that we just reported. It is quicker, flexible and efficient but the basic and most important difference from the above policy is that each page is considered as a black box, in other words as something unknown that the system should approach and analyze. If the examination of the content of a page shows that the page has changed then it is still treated as something unknown to the system. The page ceases to seem unfamiliar to the system only when it is found that its content has not changed. In this case, the system knows the internal URLs of the page which examines to find out how many of them have changed and decide whether to crawl the page or not.

Our crawler is quite friendly to the pages it visits as it crawls them only when they have changed. Moreover, it is very efficient because it is interested in crawling information that is fresh, ignoring data that are not up-to-date. We should also remark on the cleverness of the crawler we describe. With just a look at the database, it avoids visiting pages whose content is stable and their crawling would put additional load both on the crawling system and on the pages. Finally the crawler learns more about the pages as it runs and its low speed which is a consequence of the fact that it is a selective incremental crawler, is not a disadvantage of the system but it makes it seem even more friendly to the pages it visits.

4. AVOIDING TRAPS – ENHANCING SECURITY

A crawler trap is a URL or set of URLs that cause a crawler to crawl indefinitely. Some crawler traps are unintentional. For example, a symbolic link within a file system can create a cycle. Other crawler traps are introduced intentionally. For example, people have written traps using CGI programs that dynamically generate an infinite web of documents. The crawler we describe does not read CGI programs and thus it is not threatened.

There is no automatic technique for avoiding crawler traps. However, sites containing crawler traps are easily noticed due to the large number of documents discovered there. A human operator can verify the existence of a trap and manually exclude the site from the crawler's purview using the customizable URL filter.

The URL filtering mechanism provides a customizable way to control the set of URLs that are downloaded. Before adding a URL to the table with the URLs to be crawled, the crawling system consults the user-supplied URL filter. The URL filter class has a single crawl method that takes a URL and returns a boolean value indicating whether or not to crawl that URL.

Nevertheless, it is possible a crawler to include a collection of different URL filter subclasses that provide facilities for restricting URLs by domain, prefix or protocol type and for computing conjunction, disjunction or negation of other filters. Users may also supply their own custom URL filters, which are dynamically loaded at start-up.

The crawler, we describe in this paper also uses a time limit in order to avoid traps. As we mentioned above, obvious traps are gigabyte-sized or even infinite web documents. Similarly, a web site may have an infinite series of links (eg. "domain.com/home?time=101" could have a self-link to "domain.com/home?time=102" which contains link to "…103", etc…). Deciding to ignore dynamic pages results in a lot of skipped pages and therefore the problem is not completely fixed. Our crawler uses a time limit which refers to the period a page is being crawled. If a page while crawling passes this limit, it stops being crawled and the next page is fetched to be crawled, encountering efficiently this kind of traps.

Finally, our crawling system manages enhanced security because it is only interested in extracting content and therefore it avoids traps that are caused by images and other complicated data.

5. EXPERIMENTING WITH THIS CRAWLER - REAL LIFE SCENARIO

In this section, we describe an experimental procedure of the usage of this crawler. Through the process that we describe, we explain how the crawler manages to be so efficient and at the same time also be polite and adaptive. The following table initiates our crawler.

Id	url	Date	Fr	Fr_max	Level
1	http://www.mysite.com	10 – 06 – 2005	0	2	2
2	http://www.mysite2.com	10 – 06 – 2005	0	2	2

Table 1. URL seed

After some days of crawling, we receive the following results for the urls of the above table and the crawler adapts its behavior to each URL separately, in proportion to these results.

url_id	Internal links (the average / crawl)	Changes of size during last 10 crawls	Changes of size of internal page during last 10 crawls (%)
1	586	9	10%
2	718	1	89%

Table 2. Url analysis after 10 crawl procedures

As the speed is concerned, we noticed that the crawler needs about 2,5 hours to download all the pages of a specific URL, up to level 2. This means that the pages of the server are accessed every 15 seconds and therefore our crawler adds little load to the network of the server. As a consequence, ostensible delay which definitely does not influence crawler's operation, results to a more polite behavior of the crawler towards server.

This is one of the ways that are used by the system to prevent any kind of problems that may be caused on the network and at the sites we visit. Next time the crawler will run and start crawling the new URLs, it will begin its function at a different time from the one it started the last time. Although the specific procedure may seem unnecessary to the crawler's execution, we will show, through a real example, how we ameliorate the response of the systems we have access to.

We tried to crawl a specific international news portal at the same time every day for about a week and simultaneously we were visiting the same pages with the usage of a simple browser. Browsing was repeated after four hours and finally we came to the conclusion that the time of the second browsing was reduced by 5%. In spite of the fact that the decrease in time may not be exclusively a cause of the crawler's process, the increase in speed is obvious.

With regard to the question of adaptability in each page, we present below a real example of two pages that began with the same elements (fr_max = 2) and they led in two completely different ones, because of adaptability. After 10 times of crawling, we received the values $fr\_max_1 = 1$ and $fr\_max_2 = 6$ for the two pages respectively (results from table 2). If a crawler could run every day then we would receive the double results than what we receive with this crawler. Quantitatively, we can say that even after two times of crawling, there is a profit of 28% (we download 28% less pages), percentage very big comparatively to a crawler that simply browses all pages without using any historical elements from them.

In the above example, we also add elements that concern the internal URLs of the pages in order to test the output of our crawler in extreme conditions and mainly to show how the element 'selective incremental' functions. The results, we have from the above table (table 2) show that while the first page changes continuously, its internal URLs do not change with the same frequency. On the contrary, the second page appears to change less often, compared to its internal URLs. Thus, in the first case, the crawler should crawl continuously the first page without though analyzing it, as the experiments show that its URLs change very little. In other words, we daily begin from a specific link but we analyze it rarely. In the second case, we

notice that the internal pages are continuously changed. In order to maintain a balance and the elements in the database to remain as much as possible renewed, we minimize the bound of fr_max_2 from 6 to 3, according to the algorithm [fr_max + (Changes of size [%] * fr_max)]/2.

We come to the conclusion, therefore, that the crawler we describe can 'learn' the pages it visits and as a result, after some period, it adapts to each URL, increasing its output as well as the quality of content that it stores.

5.1 Supporting Data Processing Systems

The mechanism that we created can represent neither a one-time crawler nor a simple content downloader because under these circumstances its performance is quite low. It is a system whose output can be evaluated after some time. At the same time, it is given great importance to the content and through this mechanism, it is possible to be maintained a content history and a database which contains information, useful to systems that process and analyze texts.

6. FUTURE WORK – CONCLUSION

Due to the dynamism of the Web, crawling forms the back-bone of applications that facilitate Web information retrieval. In this paper, we described the architecture and implementation details of our crawling system and presented some preliminary experiments. We explained the importance of extracting only content from web pages and how this can be implemented by a mechanism content analysis, corresponding crawling policies and clever systems that extract content of high quality.

There are obviously many improvements to the system that can be made. A major open issue for future work is a detailed study of how the system could become even more distributed, retaining though quality of the content of the crawled pages. When a system is distributed, it is possible to use only one of its components or easily add a new one to it.

REFERENCES

[1] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. Compaq Systems Research Center, Palo Alto, CA 94301.

[2] Vladislav Shkapenyuk and Torsten Suel. Design and Implementation of a High- Performance Distributed Web Crawler. CIS Department Polytechnic University Brooklyn, NY 11201.

[3] The Robots Exclusion Protocol
http://info.webcrawler.com/mak/projects/robots/exclusion.html

[4] J. Cho and H. Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. Department of Computer Science, Stanford, December 2,1999.

[5] The Web Robots Pages.
http://info.webcrawler.com/mak/projects/robots/robots.html

[6] David Eichmann. The RBSE Spider -- Balancing Effective Search Against Web Load. In *Proceedings of the First International World Wide Web Conference*, pages 113--120, 1994.

[7] Oliver A. McBryan. GENVL and WWWW: Tools for Taming the Web. In *Proceedings of the First International World Wide Web Conference*, pages 79--90, 1994.

[8] Brian Pinkerton. Finding What People Want: Experiences with the WebCrawler. In *Proceedings of the Second International World Wide Web Conference*, 1994.

[9] Gautam Pant, Padmini Srinivasan, and Filippo Menczer. Crawling the Web

[10] Web crawler
http://en.wikipedia.org/wiki/Web_crawler

[11] Dustin Boswell. Distributed High-performance Web Crawlers: A Survey of the State of the Art.

[12] F. Menczer, G. Pant and P. Srinivasan, Topic-Driven Crawlers: Machine Learning Issues. The University of Iowa, *ACM TOIT*, 2002

[13] X. Gao and L. Sterlning. Using Limited Common Sense Knowledge to Guide Knowledge Acquisition for Information Agents. *In Proceedings of the AI'97 workshop on knowledge acquisition*, Perth, Australia, pages 9.1--9.11, 1997.