Oriental Journal of Computer Science & Technology Vol. **1**(1), 49-54 (2008)

An algorithm for effective web crawling mechanism of a search engine

B. VIJAYA BABU[1], M. SURENDRA PRASAD BABU[2] and CHETAN PRASAD Y.[3]

[1-3]Department of IST, K.L. College of Engineering, Vaddeswaram - 522 502 (India)
[2]Department of CS&SE, College of Engineering, Andhra University, Visakhapatnam (India).

(Received: February 12, 2008; Accepted: April 04, 2008)

ABSTRACT

Broad web search engines as well as many more specialized search tools rely on web crawlers to acquire large collections of pages for indexing and analysis. Such a web crawler may interact with millions of hosts over a period of weeks or months, and thus issues of robustness, flexibility, and manageability are of major importance. In addition, I/O performance, network resources, and OS limits must be taken into account in order to achieve high performance at a reasonable cost.

Current-day crawlers retrieve content only from the publicly indexable Web, i.e., the set of web pages reachable purely by following hypertext links, ignoring search forms and pages that require authorization or prior registration. In particular, they ignore the tremendous amount of high quality content "hidden" behind search forms, in large searchable electronic databases. Also even if there is good data collection that has been indexed we would be able to look at those sites having these info only if we are connected to the internet and may be the days where hourly based nets used to be major providers of internet are gone, these days the broadband facilities, high speed net connections are available to the common man. But, the growth in the usage of laptops is even growing at same pace, in that case one may not be able to access the net where ever he moves and if any important pages on the net in a particular web site would be of no use even he has good configuration, as still it takes time for the wi-fi networks to come in to full swing, until then saving every page of a particular website may be a hectic task. In this paper, would provide a framework for addressing the problem of browsing the web even when offline.

Key words: Web crawler, search engine, indexer, frontier, crawl manager.

INTRODUCTION

Search Engines

A search engine is a program designed to help one access files stored on a computer, for example a public server on the World Wide Web[2]. Unlike an index document that organizes files in a predetermined way, a search engine looks for files only after the user has entered search criteria[1].

A search engine reference model

Most practical and commercially operated Internet search engines are based on a centralized architecture that relies on a set of key components, namely Crawler, Indexer and Searcher. This architecture can be seen in systems including WWW [McB94], Google, and our own FAST Search Engine.

Crawler

A crawler is a module aggregating data from the World Wide Web in order to make them searchable. Several heuristics and algorithms exists for crawling, most of them are based upon following links.

Indexer

A module that takes a collection of documents or data and builds a searchable index from them. Common practices are inverted files,

vector spaces, suffix structures and hybrids of these[1].

Searcher

The searcher is working on the output files from the indexer. The searcher accepts user queries, runs them over the index, and returns computed search results to issuer.

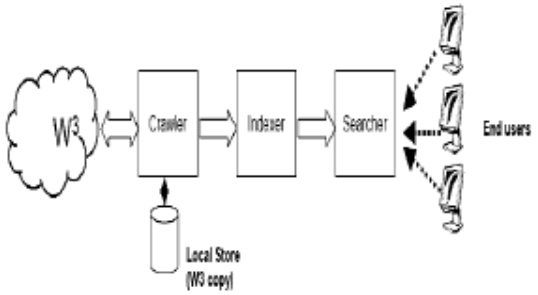

Fig. 1: A search engine model

Local store

A local store copy is a snapshot of the web at the given crawling time for each document.

Systems usually run the crawler, indexer, and searcher sequentially in cycles[1]. First the crawler retrieves the content, then the indexer generates the searchable index, and finally, the searcher provides functionality for searching the indexed data. To refresh the search engine, this *indexing cycle* is run again[2,4].

Crawling the web

Introduction to web crawlers

Web *crawlers* are programs that exploit the graph structure of the Web to move from page to page. In their infancy such programs were also called wanderers, robots, spiders, fish, and worms, words that are quite evocative of web imagery. It may be observed that the noun 'crawler' is not indicative of the speed of these programs, as they can be considerably fast[2].

Building a crawling infrastructure

The crawler maintains a list of unvisited URLs called the *frontier*. The list is initialized with seed URLs which may be provided by a user or another program. Each *crawling loop* involves picking the next URL to crawl from the frontier, fetching the page corresponding to the URL through HTTP, parsing the retrieved page to extract the URLs and application specific information, and finally adding the unvisited URLs to the frontier. The crawling process may be terminated when a certain number of pages have been crawled. Crawling can be viewed as a graph search problem[3,4]. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes[3].

Design of a web crawler

Requirements for a crawler

We now discuss the requirements for a good crawler, and approaches for achieving them:[3]

Flexibility

Low Cost and High Performance

Robustness

Etiquette and Speed Control

Manageability and Re-configurability

System architecture

Two major components in the architecture of a crawler are – crawling system and crawling application[4].

Crawling system

The crawling system itself consists of several specialized components, in particular a *crawl manager*, one or more *downloaders*, and one or more *DNS resolvers*. All of these components, plus the crawling application, can run on different machines (and operating systems) and can be replicated to increase the system performance[4,5]. The crawl manager is responsible for receiving the URL input stream from the applications and forwarding it to the available downloaders and DNS resolvers while enforcing rules about robot exclusion and crawl speed. A downloader is a high performance asynchronous HTTP client capable of downloading hundreds of web pages in parallel, while a DNS resolver is an optimized stub DNS resolver that forwards queries to local DNS servers[5,6].

Crawl manager

The crawl manager is the central component of the system, and the first component that is started up. Afterwards, other components are started and register with the manager to offer or request services. The manager is the only component visible to the other components, with the exception of the case of a parallelized application, described further below, where the different parts of the application have to interact[4]. The manager receives requests for URLs from the application, where each request has a pointer to a file containing several hundred or thousand URLs and located on some disk accessible via internet.

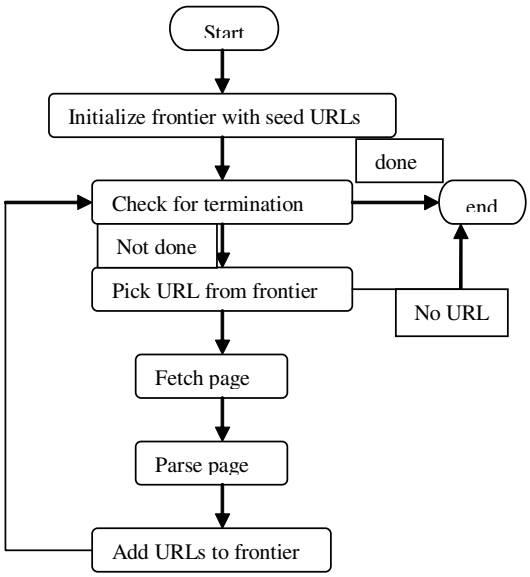

Fig. 2: Flow chart representing a web crawler

The manager will enqueue the request, and will eventually load the corresponding file in order to prepare for the download. In general, the goal of the manager is to download pages in approximately the order specified by the application, while reordering requests as needed to maintain high performance without putting too much load on any particular web server. After parsing the robots files and removing excluded URLs, the requested URLs are sent in batches to the downloaders, making sure that a certain interval between requests to the same server is observed. The manager later notifies the application of the pages that have been downloaded and are available for processing[6,7,8].

Design/Prototype

In addition to this, we would have to write a small code segment which does the following [10]:

- Take in a URL
- Open a connection
- Read in a page
- Parse the page

Step 1. Build gethttp

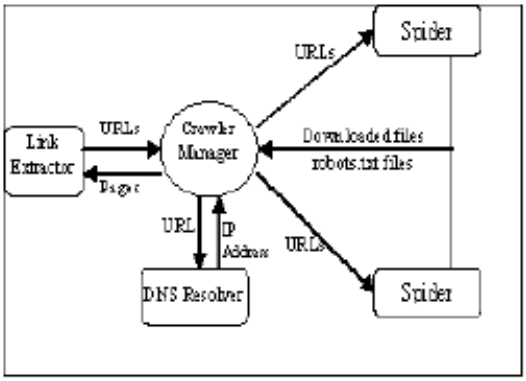

Fig. 3: Web crawler architecture

document. The tags are strings of the form "<...>". If the document is not "text/html" print "Error: document is not text/html" and exit. Also skip everything between an ampersand ("&") and a semi-colon (";"). This will eliminate escape characters such as . Additionally, eliminate the new lines and allow only one space between words. Also ignore any text between <script ...> ... </script> tags.

For example:
gethttp -t http://www.cs.purdue.edu
will print the document without any of the tags or escape characters.

Finding hyperlinks

Add the argument **-a** that prints the hyperlinks (URLs) in a document. The hyperlinks in a document are found in tags of the form <a href=”url”>. If the document is not “text/html” print “Error: document is not text/html”and exit.
For example:
gethttp -t http://www.cs.purdue.edu
will print the hypelinks found in the document.

Building a web crawler program

The webcrawler program will have the syntax:
webcrawl [-u <maxurls>] url-list
Where maxurls is the maximum number of URLs that will be traversed. By default it is 1000. url-list is the list of starting URL's that will be traversed.

Web crawling procedure

You will visit the URLs in a breadth first search manner, that is, we will visit the URLs in the order they were inserted in the URL array. To do this you will keep a current index in the URL Array that you will increment by one until we have reached nURL.

During the web-crawiling we follow these steps

1. First insert the initial URLs from the URL list in the URL Array.
2. For currentURL = 0 to urlArray.nURL (For all the URLs in the URL array)

a) Get the document that corresponds to the URL.
b) If the type of the document is not text/html then go to the next URL
c) Get the first 200 characters of the document without tags. Add this description to the URLRecord for this URL.
d) Find all the hyperlinks of this document and add them to the URLArray and URLDictionary if they are not already in the URLDictionary. Only insert up to maxURL entries.
e) For each word in the document without tags, add this URLRecord to the word in the WordDictionary.

Web becomes offline…!!!

Our Idea

Given any web site address or URL, by a user, to this algorithm, it can present the user with the web graph or the site map of that particular web site. The algorithm now waits for the user to select the pages he want to save on to his hard disk, at the same time it preserves the links of the pages stored, i.e., the pages are stored in such a way that the links when clicked are accessible, only if the page pointed by that link is also stored by the user.

The algorithm also creates a new HTML file showing the web graph of the web site and also highlighting the links of all the pages that have been stored from that particular site, so that if any stored page has not been directly linked to any of the web pages it also can be viewed with out actually having to open the folder in which the files have been stored.

The procedure

First take the input from the user and convert it into the standard form i.e., in to all lower case letters and its domain form. Now apply time stamp before actually entering in to the process, so as to reduce huge time loss. Now parse the page for the hypertext links and then check if the links are in the same domain and if present, then we parse them by adding them in to the queue and adding the link to the queue which stores the already parsed links. Now display the web graph or the site map of the web site and then store the pages selected by the user to the hard disk and also change the links so that the saved pages links remain active if their corresponding page is also stored[11].

Algorithm

1. Take the input as IURL from the user
2. Process(URL)

{
Get the main URL in lower case
(ex: DSENGG.COM/NCACT08 should be processed as dsengg.com/ncact08)
if the web page exists
apply timestamp
else display error message and
exit
}
3. Initialize the queues ‘todo’=URL and done’
4. initialize n=1
5. Extract links(todo[k])

```
{
While(todo ! empty) do
If the page has frames then
{
For(i=1 to no.of frames) do
Extract links(URLi)
}
If the page has forms then
Omit them
Else
{
While(!EOF URL) do
{
Parse the document for anchor tags <a> Get the child URL – CURL
If the domains of the URL and CURL are equal then
Enter CURL in to 'todo'
queue
Else
Extract links(todo[++k])
}
} Enter the url in to 'done' queue and increment n
}
5. Initialize the array select[n]
6. Display the web graph
7. selection()
{
For (all web graph nodes) do
{
If (web graph node is selected)
Select[i]=1
Else
Select[i]=0
}
}
8. download(select[n])
{
For(i=1 to n)
If select[i]=1 then
Save done[i] in temp folder
}
9. Update (URLs)
{

Let the user choose the destination path on his disk.
Create a folder from the title of the URL.
Update the anchor tags such way that the files Having links to each other are linked properly.
Move files from the temp folder to the destination Folder.
Create a new HTML document which displays the web Graph of the web site and also highlighting the Saved pages.
}
10. Exit
```

Resources needed

Variables used

URL – To store the initial URL
todo – A queue to store the URLs to be parsed
done – A queue to store the URLs that have been parsed
n – Counter to store the no.of URLS parsed
select(n) – An array which specifies the pages to be stored
CURL – To store the child URL
i, k – General variables.

Functions used

Process(URL) – To standardize the input URL, and check for the availability of the web site.
Extract links(todo[k]) – To parse a web page, by dividing frames and checking for form
links and checking for domain equality.
Selection() – To recognize the pages selected by the user and to store the numbers of the
pages to be stored in the array select[n].
Download(select[n]) – To save the selected files, by using the select[n] array and saving the corresponding pages from done queue in temp folder
Update(URL) – To update the links in the saved pages and to create a new HTML document which shows the web site graph and highlights the web pages that have been stored by linking them.

CONCLUSION

We have described the architecture and implementation details of our crawling system, and presented some preliminary experiments. There are obviously many improvements to the system that can be made.

Our main interest is in using the crawler in our research group to look at other challenges in web search technology. A very good tool can be produced using open source software with minimal expense and time. A good tool should be developed

and updated using a short development cycle. Modular design can enable flexibility and speed for updates to evolving web standards.

A number of topical crawling algorithms have been proposed in the literature. Often the evaluation of these crawlers is done by comparing a few crawlers on a limited number of queries/tasks without considerations of statistical significance. As the web crawling field matures, the disparate crawling strategies will have to be evaluated and compared on common tasks through well-defined performance measures. In the future, we see more sophisticated usage of hypertext structure and link analysis by the crawlers.

REFERENCES

1. Sergey Brin and Larry Page. Google search engine.http://google.stanford.edu
2. Lawrence Page and Sergey Brin. The anatomy of a largescale hypertextual web search engine. In To Appear: Proceedings of the Seventh International Web Conference (WWW 98), (1998)
3. On Search, the Series (*http://www.tbray.org/ongoing/When/200x/2003/07/30/OnSearchTOC*); Tim Bray; A series of essays on search engine techniques (2003).
4. A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, **1**(1): (2001).
5. J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase : A repository of web pages. In *Proc. of the 9th Int. World Wide Web Conference*, May (2000).
6. M. Najork and J.Wiener. Breadth-first search crawling yields high-quality pages. In *10th Int. World Wide Web Conference*, 2001. The Deep Web: Surfacing Hidden Value. http://www.completeplanet.com/Tutorials/DeepWeb/.
7. Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: A new approach to topicspecific web resource discovery. In *Proceedings of the Eighth International World-Wide Web Conference*, (1999).
8. Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, Available at http://wwwdiglib. stanford.edu/cgi-bin/get/SIDL-WP-1999-0129 (2000).
9. Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the International Conference on Management of Data*, Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDLWP- 1999-0116 (2000).
10. Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *Proceedings of the Seventh International World-Wide Web Conference*, Available at http://www-diglib.stanford.edu/cgibin/ WP/get/SIDL-WP-1999-0103 (1998).
11. Chetan Prasad Y, Web Crawler.In Proceedings of International Conferrence on Systemics, Cybernetics and Informatics, ICSCI-2007, Jan 03-07, Hyderabad, India (2007).
12. B.Vijaya Babu, Prof Surendra Prasad Babu.M,A.Siddhartha and T.Viswanath An algorithm for effective web crawling mechanism of a search engine. In the Proceedings of international Conference on Advanced Computing Technologies, NCACT08, March 20, Perambalur, Tamilnadu, India (2008).