IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Web Crawlers and Search Engines

Ritika Hans[*1], Gaurav Garg[2]
[*1,2] AITM Palwal, India

Abstract

In large distributed hypertext system like the World-Wide Web; users find resources by following hypertext links. As the size of the system increases the users must traverse increasingly more links to find what they are looking for, until precise navigation becomes impractical. The WebCrawler is a tool that solves these problems by indexing and automatically navigating the Web. This paper describes the basic definition of web and search engine and we have also explored the web crawler with its types.

Introduction

WWW on the Web is a service that resides on computers that are connected to the Internet and allows end users to access data that is stored on the computers using standard interface software and we can say that it is a driving force behind the internet. The World Wide Web is the universe of network-accessible information, an embodiment of human knowledge.

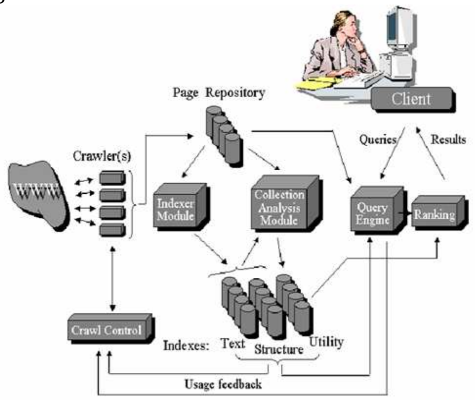
The web creates new challenges for information retrieval. The amount of information on the web is growing rapidly, as well as the number of new users inexperienced in the web research. People are likely to surf the web using its link graph, often starting with high quality human maintained indices such as Yahoo! or the search engines like Google. A web crawler is a program that downloads and stores Web pages, often for a Web search engine by placing an initial set of URLs in a queue ,where all URLs to be retrieved are kept and prioritized.

Search Engines

The Internet, particularly the World Wide Web, is a vast source of information that is growing t an explosive rate .More than 7 million publicly available pages are added to the World Wide Web everyday. This growth rate means the 2.1 billion unique pages available on the Web will double in the span of about one year. Search engines must be fast enough to crawl the exploding the exploding volume of new Web pages in order to provide the most up-to-date information. As the number of pages on the Web grows, so will the number of results search engines return. There are differences in the ways various search engines work, but they all perform three basic tasks:

- They search the Internet based on important words.
- They keep an index of the words they find, and where they find them.
- They allow users to look for words or combinations of words found in that index.

In the process of crawling, pages are extracted all the words from each page, and records the URL where each word occurred. The result is a generally very large “lookup table" that can provide all the URLs that point to pages where a given word occurs. The table is of course limited to the pages that were covered in the crawling process. As mentioned earlier, text indexing of the Web poses special difficulties, due to its size, and its rapid rate of change.



In addition to these quantitative challenges, the Web calls for some special, less common kinds of indexes. Such indexes would not be appropriate for traditional text collections that do not contain links. During a crawling and indexing run, search engines must store the pages they retrieve from the Web. The page repository represents the possibly temporary collection. Sometimes search engines maintain a cache of the pages they have visited beyond the time required to build the index. This cache allows them to serve out result pages very quickly, in addition to providing basic search facilities.

If the user returns to a page fairly soon, it is likely that the data will not be retrieved from the source web server, as above, again. By default, browsers cache all web resources on the local hard drive. An HTTP request will be sent by the browser that asks for the data only if it has been updated since the last download. If it has not, the cached version will be reused in the rendering step. This is particularly valuable in reducing the amount of web traffic on the internet.

The Some systems, such as the Internet Archive, have aimed to maintain a very large number of pages for permanent archival purposes. Storage at such a scale again requires special consideration.

The query engine module is responsible for receiving and fulfilling search requests from users. The engine relies heavily on the indexes, and sometimes on the page.

Types Of Search Engine

1. Crawler Based Search Engines

Crawler based search engines create their listings automatically. Computer programs 'spiders' build them not by human selection. They are not organized by subject categories; a computer algorithm ranks all pages. Such kinds of search engines are huge and often retrieve a lot of information -- for complex searches it allows to search within the results of a previous search and enables you to refine search results. These types of search engines contain full text of the Web pages they link to. So one can find pages by matching words in the pages one wants.

2. Human Powered Directories

These are built by human selection i.e. they depend on humans to create listings. They are organized into subject categories and subjects do classification of pages. Human powered directories never contain full text of the Web page they link to. They are smaller than most search engines.

3. Hybrid Search Engine

A hybrid search engine differs from traditional text oriented search engine such as Google or a directory-based search engine such as Yahoo in which each program operates by comparing a set of metadata, the primary corpus being the metadata derived from a Web crawler or taxonomic analysis of all internet text, and a user search query. In contrast, hybrid search engine may use these two bodies of metadata in addition to one or more sets of metadata that can, for example, include situational metadata derived from the client's network that would model the context awareness of the client.

Web Crawlers

A crawler is a program that downloads and stores Web pages, often for a Web search engine. Roughly, a crawler starts off by placing an initial set of URLs, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler removes URL (in some order), downloads the page, extracts any URLs in the downloaded page, and adds the new URLs in the queue. This process is repeated. Collected pages are indexed, cached and stored for further processing by Web search engines. Most crawlers abide by the Robots Exclusion Principle.

Robot exclusion principle

In 1993 and 1994 there have been occasions where robots have visited WWW servers where they weren't welcome for various reasons. Sometimes these reasons were robot specific, e.g. certain robots swamped servers with rapid-fire requests, or retrieved the same files repeatedly. In other situations robots traversed parts of WWW servers that weren't suitable, e.g. very deep virtual trees, duplicated information, temporary information, or cgi-scripts with side-effects (such as voting). These incidents indicated the need for established mechanisms for WWW servers to indicate to robots which parts of their server should not be accessed. This standard addresses this need with an operational solution.

The Method

The method used to exclude robots from a server is to create a file on the server, which specifies an access policy for robots. This file must be accessible via HTTP on the local URL "/robots.txt". This approach was chosen because it can be easily implemented on any existing WWW server, and a robot can find the access policy with only single document retrieval.

The Format

The format and semantics of the "/robots.txt" file is as follows: The file consists of one or more records separated by one or more blank lines. Each record contains lines of the form "<field> :< optional space><value><optional space>". The field name is case insensitive. Comments can be included in file using UNIX Bourne shell conventions: the '#' character is used to indicate that preceding space (if any) and the remainder of the line up to the line termination is discarded. Lines containing only a comment are discarded completely and therefore do not indicate a record boundary. The record starts with one or more User-agent lines, followed by one or more Disallow lines, as detailed below. UNRECOGNIZED HEADERS ARE IGNORED.

User-Agent

The value of this field is the name of the robot the record is describing access policy for. If more than one User-agent field is present the record describes an identical access policy for more than one robot. At least one field needs to be present per record. If the value is '*', the record describes the default access policy for any robot that has not matched any of the other records. It is not allowed to have multiple such records in the "/robots.txt" file. The value of this field specifies a partial URL that is not to be visited. This can be a full path, or a partial path; any URL that starts with this value will not be retrieved. For example, Disallow: /help disallows both /help.html and /help/index.html, whereas Disallow: /help/ would disallow /help/index.html but allow /help.html. Any empty value indicates that all URLs can be retrieved. At least one Disallow field needs to be present in a record. The presence of an empty "/robots.txt" file has no explicit associated semantics, it will be treated as if it was not present, i.e. all robots will consider themselves welcome.

Architecture of Web Crawler

In order to download a document, the crawler picks up its seed URL, and depending on the host protocol, downloads the document from the web server. For instance when a user accesses an HTML page using its URL, the documents is transferred to the requesting machine using Hyper Text Transfer Protocol (HTTP) [2, 3, 4]

The browser parses the document and makes it available to the user. Roughly, a crawler starts off by placing an initial set of URLs, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler extracts a URL, downloads the page, extracts URLs from the downloaded pages, and places the new URLs in the queue. This process is repeated and the collected pages are later used by other applications, such as a Web search engine.

Algorithm for the Crawler The algorithm of the typical crawler is given below:

Step 1. Read a URL from the set of seed URLs.

Step 2. Determine the IP address for the host name.

Step 3. Download the Robot.txt file which carries downloading permissions and also specifies the files to be excluded by the crawler.

Step 4. Determine the protocol of underlying host like http, ftp, gopher etc.

Step 5. Based on the protocol of the host, download the document.

Step 6. Identify the document format like doc, html, or pdf etc.

Step 7. Check whether the document has already been downloaded or not.

Step 8. If the document is fresh one Then Read it and extract the links or references to the other cites from that documents.

Else Continue;

Step 9. Convert the URL links into their absolute IP equivalents.

Step 10. Add the URLs to set of seed URLs.

Web crawling is a very time-consuming task - some search engines show off that their crawlers completely recheck their searched pages at least once a month. This isn't very useful if you're expecting to find current information via that crawler. The problem is one of sheer volume - search engines have to go through billions of pages of information and this takes a huge amount of time.

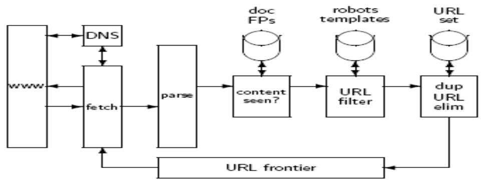
Types of Web Crawler

Focused Crawler

The rapid growth of the worldwide web poses unprecedented scaling challenges for general-purpose crawlers and search engines leading to a new hypertext resource discovery system called a *Focused Crawler* [5]. A focused crawler may be described as a crawler, which returns relevant web pages on a given topic in traversing the web. It takes as input one or several related web pages and attempts to find similar pages on the web, typically by recursively following links in a best first manner. Ideally, the focused crawler should retrieve all similar pages while retrieving the fewest possible number of irrelevant documents. The goal of a focused crawler is to selectively seek out pages that are relevant to a pre-defined set of *topics*. The topics are specified not using keywords, but using exemplary documents. Rather than collecting and indexing all accessible web documents to be able to answer all possible ad-hoc queries, a focused crawler analyzes its crawl boundary to find the links that are likely to be most relevant for the crawl, and avoids irrelevant regions of the web. This leads to significant savings in hardware and network resources, and helps keep the crawl more up-to-date.

A focused crawler has the following main components: (a) A way to determine if a particular web page is relevant to the given topic, and (b) a way to determine how to proceed from a known set of pages. An early search engine which deployed the focused crawling strategy was proposed in based on the intuition that relevant pages often contain relevant links. It searches deeper when relevant pages are found, and stops searching at pages not as relevant to the topic.

Unfortunately, the above crawlers show an important drawback when the pages about a topic are not directly connected in which case the crawling might stop pre-maturely. This problem is tackled in where reinforcement learning permits credit assignment during the search process, and hence, allowing off-topic pages to be included in the search path.



Following is the process block diagram for a simple crawler

However, this approach requires a large number of training examples, and the method can only be trained offline. In a set of classifiers are trained on examples to estimate the distance of the current page from the closest on-topic page. But the training procedure is quite complex. There are a number of issues related to existing focused crawlers, in particular the ability to ``tunnel" through lowly ranked pages in the search path to highly ranked pages related to a topic which might re-occur further down the search path. Two parameters, viz., degree of relatedness, and depth describe a simple focused crawler. Both provide an opportunity for the crawler to ``tunnel" through lowly ranked pages. A Focused Crawler seeks, acquires, indexes, and maintains pages on a specific set of topics that represent a relatively narrow segment of the web.

Architecture of Focused Crawler

The system architecture model adopted is distributed and object-oriented, which is a multi-tier implementation approach. The focused crawler system is composed of three tiers following the MVC (Model / View / Controller) paradigm.

The system is based on 3-tier architecture: the first tier is the presentation layer (User Interface), the second (the middle) tier is composed of the request manager and the crawlers' structure, the third tier is the data-storage layer (Web pages collection system, cache module).

In the First tier the user interface is a HTML page, through which the user can launch the focused search based on keywords, a classifier, a bookmark file (file containing the URL to be used for the training of the classifier or as initial seed). In the second tier it gets as input the query string data receives a XML request message that contains a list of URLs identifying documents that need to be acquired and processed at higher layers.

The message specifies also the ID of the agent crawler who is sending the request. In output it returns the requested document in XML format to the agent who sent the request. In the third tier a set of links provided by the second tier are stored in the cache module.

Incremental Crawler

An incremental crawler [6] is one, which updates an existing set of downloaded pages instead of restarting the crawl from scratch each time. This involves some way of determining whether a page has changed since the last time it was crawled.
A crawler, which will continually crawl the entire web, based on some set of crawling cycles. An adaptive model is used, which uses data from previous cycles to decide which pages should be checked for updates, thus high freshness and results in low peak load is achieved.

Architecture of Incremental Crawler

Here we first identify two goals for the incremental crawler and explain how the incremental crawler conceptually operates. From this operational model, we will identify two key decisions that an incremental crawler constantly makes. Based on these observations, we propose architecture for the incremental crawler.

Two goals for an incremental crawler

The incremental crawler continuously crawls the web, revisiting pages periodically. During its continuous crawl, it may also purge some pages in the local collection, in order to make room for newly crawled pages. During this process, the crawler should have two goals:

- ***Keep the local collection fresh:*** freshness of a collection can vary widely depending on the strategy used. Thus, the crawler should use the best policies to keep pages fresh. This includes adjusting the revisit frequency for a page based on its estimated change frequency.
- ***Improve quality of the local collection:*** The crawler should increase the quality of the local collection by replacing less important pages with more important ones. This refinement process is necessary for two reasons; pages are constantly created and removed. Some of the new pages can be more important than existing pages in the collection, so the crawler should replace the old and less important pages with the new

and more important pages. Second, the importance of existing pages changes over time. When some of the existing pages become less important than previously ignored pages, the crawler should replace the existing pages with the previously ignored pages.

Operational model of an incremental crawler

In this case the conceptual operation of the crawler is shown; it is not an efficient or complete implementation. All URL records the set of all URLs, and CollUrls records the set of URLs in the collection. We assume that the local Collection maintains affixed number of pages and that the collection is at its maximum capacity from the beginning.

If the page already exists in the collection, the crawler updates its image in the collection. If not, the crawler discards an existing page from the collection saves the new page and updates CollUrls Finally, the crawler extracts links (or URLs) in the crawled page to add them to the list of all URLs.
When the crawler decides to crawl a new page, it has to discard a page from the collection to make room for the new page. Therefore, when the crawler decides to crawl a new page, the crawler should decide what page to discard. We refer to this selection/discard decision as the refinement decision. This refinement decision should be based on the importance of pages.

To measure importance, the crawler can use a number of metrics, including Page Rank and Authority. Clearly, the importance of the discarded page should be lower than the importance of the new page. In fact, the discarded page should have the lowest importance in the collection, to maintain the collection of the highest quality. Together with the refinement decision, the crawler decides on what page to update.

That is, instead of visiting a new page, the crawler may decide to visit an existing page to refresh its image. To maintain the collection fresh, the crawler has to select the page that will increase the freshness most significantly, and we refer to this decision as update decision.

To achieve the two goals for incremental crawlers, and to effectively implement the corresponding decision process, we propose the architecture for an incremental crawler. The architecture consists of three major modules (Ranking Module, Update Module and Crawl Module) and three data structures (Allures, CollUrls and Collection). The lines and arrows show data flow between modules, and the labels on the lines show the corresponding commands.

Two data structures, AllUrls and CollUrls, maintain information similar to that. AllUrls records all URLs that the crawler has discovered, and CollUrls records the URLs that are/will be in the Collection. CollUrls is implemented as a priority-queue, where the URLs to be crawled early are placed in the front. The URLs in CollUrls are chosen by the Ranking Module. The Ranking Module constantly scans through AllUrls and the Collection to make the refinement decision. For instance, if the crawler uses Page Rank as its importance metric, the Ranking Module constantly reevaluates the Page Ranks of all URLs, based on the link structure. When a page not in CollUrls turns out to be more important than a page within CollUrls, the Ranking Module schedules for replacement of the less-important page in CollUrls with that more-important page.

The URL for this new page is placed on the top of CollUrls, so that the Update Module can crawl the page immediately. Also, the Ranking Module discards the less-important page from the Collection to make space for the new page. While the Ranking Module refines the Collection, the Update Module maintains the Collection fresh (update decision). It constantly extracts the top entry from CollUrls, requests the Crawl Module to crawl the page, and puts the crawled URL back into CollUrls.

The position of the crawled URL within CollUrls is determined by the page's estimated change frequency and its importance. (The closer a URL is to the head of the queue, the more frequently it will be revisited.) To estimate how often a particular page changes, the Update Module records the checksum of the page from the last crawl and compares that checksum with the one from the current crawl. From this comparison, the Update Module can tell whether the page has changed or not. we explain how the Update Module can frequency of a page. Note that it is also possible to keep update statistics on larger units than a page, such as a web site or a directory.

If web pages on a site change at similar frequencies, the crawler may trace how many times the pages on that site changed for last 6 months, and get a confidence interval based on the site-level statistics. In this case, the crawler may get a tighter confidence interval, because the frequency is estimated on larger number of pages (i.e., larger sample).

However, if pages on a site change at highly different frequencies, this average change frequency may not be sufficient to determine how often to revisit pages in that site, leading to a less-than optimal revisit frequency. Also note that the Update Module may need to consult the\importance of a page in deciding on revisit frequency. If a certain page is highly important and the page needs to be always up-to-date, the Update Module may revisit the page much more often than other pages with estimate the change frequency of a page based on this change history. In short, we propose two estimators, EP and EB, for the change The position of the crawled URL within CollUrls is determined by the page's estimated change frequency and its importance. (The closer a URL is to the head of the queue, the more frequently it will be revisited.) To estimate how often a particular page changes, the Update Module records the checksum of the page from the last crawl and compares that checksum with the one from the current crawl. From this comparison, the Update Module can tell whether the page has changed or not. we explain how the Update Module can estimate the change frequency of a page based on this change history. In short, we propose two estimators, EP and EB, for the change frequency of a page. Note that it is also possible to keep update statistics on larger units than a page, such as a web site or a directory.

If web pages on a site change at similar frequencies, the crawler may trace how many times the pages on that site changed for last 6 months, and get a confidence interval based on the site-level statistics. In this case, the crawler may get a tighter confidence interval, because the frequency is estimated on larger number of pages (i.e., larger sample).

However, if pages on a site change at highly different frequencies, this average change frequency may not be sufficient to determine how often to revisit pages in that site, leading to a less-than optimal revisit frequency. Also note that the Update Module may need to consult the\importance of a page in deciding on revisit frequency. If a certain page is highly important and the page needs to be always up-to-date, the Update Module may revisit the page much more often than other pages with similar change frequency.

To implement this policy, the Update Module also needs to record the importance of each page returning to our architecture, the Crawl Module crawls a page and saves/updates the page in the Collection, based on the request from the Update Module. Also, the Crawl Module extracts all links/URLs in the crawled page and forwards the URLs to AllUrls.

The forwarded URLs are included in AllUrls, if they are new. While we show only one instance of the Crawl Module in the Figure note that multiple Crawl Module's may run in parallel, depending on how fast we need to crawl pages. Separating the update decision (Update Module) from the refinement decision (Ranking Module) is crucial for performance reasons.

However, it may take a while to select/deselect pages for Collection, because computing the importance of pages is often expensive. For instance, when the crawler computes Page Rank, it needs to scan through the Collection multiple times, even if the link structure has changed little. Clearly, the crawler cannot recompute the importance of pages for every page crawled, when it needs to run at 40 pages/second. By separating the refinement decision from the update decision, the Update Module can focus on updating pages at high speed, while the Ranking Module carefully refines the Collection.

Hidden Web Crawler

Current-day crawlers retrieve content from the publicly index able 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. Crawling the hidden Web is a very challenging problem for two fundamental reasons: (1) scale (a recent study [3] estimates the size of the hidden Web to be about 500 times the size of the publicly index able Web) and (10) the need for crawlers to handle search interfaces designed primarily for humans.

Parallel Crawler

A crawler is a program that downloads and stores Web pages, often for a Web search engine. Roughly, a crawler starts off by placing an initial set of URLs, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process is repeated until the crawler decides to stop. Collected pages are later used for other applications, such as a Web search engine or a Web cache.

Architecture of General Parallel Crawler

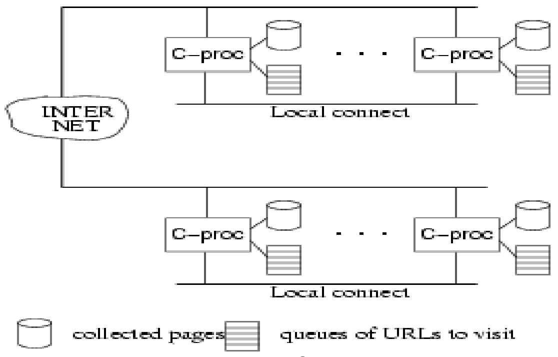
A parallel crawler consists of multiple crawling processes, which we refer to as C-proc's. Each C-proc performs the basic tasks that a single-process crawler conducts. It downloads pages from the Web, stores the pages locally, extracts URLs from the downloaded pages and follows links. Depending on how the C-proc's split the download task, some of the extracted links may be sent to other C-proc's. The C-proc's performing these tasks may be distributed either on the same local network or at geographically distant locations.

- *Intra-site parallel crawler:* When all C-proc's run on the same local network and communicate through a high speed interconnect (such as LAN), we call it an *intra-site parallel crawler*. In Fig 3.1, this scenario corresponds to the case where all C-proc's run only on the local network on the top.

In this case, all C-proc's uses the same local network when they download pages from remote Web sites. Therefore, the network load from C-proc's is centralized at a single location where they operate.

- *Distributed crawler:* When C-proc's run at geographically distant locations connected by the Internet (or a wide area network), we call it a *distributed crawler*. For example, one C-proc may run in the US, crawling all US pages, and another C-proc may run in France, crawling all European pages. A distributed crawler can *disperse* and even *reduce* the load on the overall network.

When C-proc's run at distant locations and communicates through the Internet, it becomes important how often and how much C-proc's need to communicate. The bandwidth between C-proc's may be limited and sometimes unavailable, as is often the case with the Internet.



General architecture of a parallel crawler

When multiple C-proc's download pages in parallel, different C-proc's may download the same page multiple times. In order to avoid this overlap, C-proc's need to coordinate with each other on what pages to download.

Conclusion

The hypertext documents are the most important component of WWW but their current structure poses a bottleneck for their retrieval by a crawler. By studying the different types of crawlers, a better retrieval technique can be suggested so as to improve the quality of the pages and the freshness of the pags.As the project matures, we are also interested to investigate the mechanisms to carefully select what page to download and store space in order to make the best use of its stored collection pages.

References

[1] A.K Sharma, J.P Gupta, D.P.Aggarwal,"PARACHYD: A Parallel Crawler based On Augmented Hyper text Documents", communicated to IASTED International Journal of computer applications, May.2005.

[2] Y.Yang, S.Slattery, and R.Ghani," A study of approaches to hypertext categorization ", Journal of Intelligent Information Systems.Kluwer Academic Press, 2001.

[3] S.Chakrabarti, M. van den berg, and B.Dom,"Distributed hypertext resources discovery through examples", Proceedings of the 25th International Conferences on Very large Database (VLDB) pages 375-386, 1999.

[4] A.K.Sharma, Charu Bishnoi, Dimple Juneja," *A Multi-Agent Framework for agent based focused crawlers*", Proc .Of International Conference on Emerging Technologies in IT Industry, pp-48, ICET -04, Punjab, India, November 2004.

[5] Junghoo Cho and Hector Garcia-Molina. *Estimating frequency of change, 2000.*Submitted to VLDB 2000, Research track

[6] The Deep Web: Surfacing Hidden Value. http:/www.completeplanet.com/Tutorials/DeepWeb/