Web Crawling: Important Factor for Web Search Engines in Information Retrieval

Bharati Neelkant Kharade
G. H. Raisoni College of Engineering and Management, Wagholi, Pune
Author University: University of Pune
bharati27@gmail.com

ABSTRACT
Web search is currently generating more than 13% of the traffic to Web sites. The main problem search engines have to deal with is the size of the Web, which currently is in the order of thousands of millions of pages. This large size induces a low coverage, with no search engine indexing more than one third of the publicly available Web. The large size and the dynamic nature of the Web highlight the need for continuous support and updating of Web based information retrieval systems. Crawlers facilitate the process by following the hyperlinks in Web pages to automatically download a partial snapshot of the Web. While some systems rely on crawlers that exhaustively crawl the Web, others incorporate “focus” within their crawlers to harvest application or topic specific collections. We discuss the basic issues related with developing a crawling infrastructure. This is followed by a review of several topical crawling algorithms, and evaluation metrics that may be used to judge their performance. While many innovative applications of Web crawling are still being invented, we take a brief look at some developed in the past.

KEYWORDS
Web Crawlers, Information Retrieval, Frontier, Fetching, Parsing, Stopping, and Stemming,

1. INTRODUCTION
Web crawlers are programs that exploit the graph structure of the Web to move from page to page. Web crawler can be one of the most sophisticated yet fragile parts of the application in which it is embedded. Web crawlers have been created to retrieve Web pages and add them or their representations to a local repository. Such a repository may then serve particular application needs such as those of a Web search engine. A crawler starts from a seed page and then uses the external links within it to attend to other pages. The process repeats with the new pages offering more external links to follow, until a sufficient number of pages are identified or some higher level objective is reached. Behind this simple description lies a host of issues related to network connections, spider traps, canonicalizing URLs, parsing HTML pages, and the ethics of dealing with remote Web servers. In fact a current generation Web crawler can be one of the most sophisticated yet fragile parts of the application in which it is embedded. Crawlers can be selective about the pages they fetch and are then referred to as preferential or heuristic-based crawlers. These may be used for building focused repositories, automating resource discovery, and facilitating software agents. Preferential crawlers built to retrieve pages within a certain topic are called topical or focused crawlers. Synergism between search engines and topical crawlers is certainly possible with the latter taking on the specialized responsibility of identifying subspaces relevant to particular communities of users. Techniques for preferential crawling that focus on improving the freshness of a search engine have also been suggested [1].

2. INFRASTRUCTURE REQUIRED FOR WEB CRAWLING
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. Before the URLs are added to the frontier they may be assigned a score that represents the estimated benefits of visiting the page corresponding to the URL. The crawling process may be terminated when a certain number of pages have been crawled. If the crawler is ready to crawl another page and the frontier is empty, the situation signals a dead-end for the crawler. The crawler has no new page to fetch and hence it stops [1].

2.1 FRONTIER
The frontier is the to-do list of a crawler that contains the URLs of unvisited pages. In graph search terminology the frontier is an open list of unexpanded (unvisited) nodes. Although it may be necessary to store the frontier on disk for large scale crawlers, we will represent the frontier as an in-memory data structure for simplicity. Based on the available memory, one can decide the maximum size of the frontier. Due to the large amount of memory available on PCs today, a frontier size of 100,000 URLs or more is not exceptional. Given a maximum frontier size we need a mechanism to decide which URLs to ignore when this limit is reached. Note that the frontier can fill rather quickly as pages are crawled. One can expect around 60,000 URLs in the frontier with a crawl of 10,000 pages, assuming an average of about 7 links per page. The frontier may be implemented as a FIFO queue in which case we have a breadth-first crawler that can be used to blindly crawl the Web. The URL to crawl next comes from the head of the queue and the new URLs are added to the tail of the queue. Due to the
limited size of the frontier, we need to make sure that we do not add duplicate URLs into the frontier. Once the frontier reaches its maximum size, the breadth-first crawler can add only one unvisited URL from each new page crawled. If the frontier is implemented as a priority queue we have a preferential crawler which is also known as a best-first crawler. The priority queue may be a dynamic array that is always kept sorted by the estimated score of unvisited URLs. At each step, the best URL is picked from the head of the queue. Once the corresponding page is fetched, the URLs are extracted from it and scored based on some heuristic. They are then added to the frontier in such a manner that the order of the priority queue is maintained. We can avoid duplicate URLs in the frontier by keeping a separate hash-table for lookup. Once the frontier's maximum size (MAX) is exceeded, only the best MAX URLs are kept in the frontier. If the crawler finds the frontier empty when it needs the next URL to crawl, the crawling process comes to a halt. With a large value of MAX and several seed URLs the frontier will rarely reach the empty state. At times, a crawler may encounter a spider trap that leads it to a large number of different URLs that refer to the same page. One way to improve this problem is by limiting the number of pages that the crawler accesses from a given domain. The code associated with the frontier can make sure that every consecutive sequence of k (say 100) URLs, picked by the crawler, contains only one URL from a fully qualified host name (e.g. www.cnn.com). As side-effects, the crawler is polite by not accessing the same Web site too often and the crawled pages tend to be more diverse [1][2].

### 2.2 HISTORY AND PAGE REPOSITORY

The crawl history is a time-stamped list of URLs that were fetched by the crawler. In effect, it shows the path of the crawler through the Web starting from the seed pages. A URL entry is made into the history only after fetching the corresponding page. This history may be used for post crawl analysis and evaluations. For example, we can associate a value with each page on the crawl path and identify significant events (such as the discovery of an excellent resource). While history may be stored occasionally to the disk, it is also maintained as an in-memory data structure. This provides for a fast lookup to check whether a page has been crawled or not. This check is important to avoid revisiting pages and also to avoid adding the URLs of crawled pages to the limited size frontier. For the same reasons it is important to canonicalize URLs (section 2.4) before adding them to the history. Once a page is fetched, it may be stored/indexed for the master application (such as a search engine). In its simplest form a page repository may store the crawled pages as separate files. In that case, each page must map to a unique file name. The page repository can also be used to check if a URL has been crawled before by converting it to its 32 character file name and checking for the existence of that file in the repository [1].

In order to fetch a Web page, we need an HTTP client which sends an HTTP request for a page and reads the response. The client needs to have timeouts to make sure that an unnecessary amount of time is not spent on slow servers or in reading large pages. In fact we may typically restrict the client to download only the first 10-20KB of the page. The client needs to parse the response headers for status codes and redirections. We may also like to parse and store the last-modified header to determine the age of the document. Error checking and exception handling is important during the page fetching process since we need to deal with millions of remote servers using the same code. In addition it may be beneficial to collect statistics on timeouts and status codes for identifying problems or automatically changing timeout values. Modern programming languages such as Java and Perl provide very simple and often multiple programmatic interfaces for fetching pages from the Web. However, one must be careful in using high level interfaces where it may be harder to find lower level problems. For example, with Java one may want to use the java.net.Socket class to send HTTP requests instead of the more ready-made java.net.HttpURLConnection class. No discussion about crawling pages from the Web can be complete without talking about the Robot Exclusion Protocol. This protocol provides a mechanism for Web server administrators to communicate their file access policies; more specifically to identify files that may not be accessed by a crawler. This is done by keeping a file named robots.txt under the root directory of the Web server (such as http://www.biz.uiowa.edu/robots.txt). This file provides access policy for different User-agents (robots or crawlers). A User-agent value of `*` denotes a default policy for any crawler that does not match other User-agent values in the file. A number of Disallow entries may be provided for a User-agent. Any URL that starts with the value of a http://java.sun.com. Disallow field must not be retrieved by a crawler matching the User-agent. When a crawler wants to retrieve a page from a Web server, it must first fetch the appropriate robots.txt file and make sure that the URL to be fetched is not disallowed. It is efficient to cache the access policies of a number of servers recently visited by the crawler [1][2].

### 2.4 PARSING

Once a page has been fetched, we need to parse its content to extract information that will feed and possibly guide the future path of the crawler. Parsing may imply simple hyperlink/URL extraction or it may involve the more complex process of tidying up the HTML content in order to analyze the HTML tag tree (see section 2.5). Parsing might also involve steps to convert the extracted URL to a canonical form, remove stopwords from the page's content and stem the remaining words. These components of parsing are described next.

#### 2.4.1 URL Extraction and Canonicalization
HTML Parsers are freely available for many different languages. They provide the functionality to easily identify HTML tags and associated attribute-value pairs in a given HTML document. In order to extract hyperlink URLs from a Web page, we can use these parsers to find anchor tags and grab the values of associated href attributes. However, we do need to convert any relative URLs to absolute URLs using the base URL of the page from where they were retrieved. Different URLs that correspond to the same Web page can be mapped onto a single canonical form. This is important in order to avoid fetching the same page many times. File names such as index.html or index.htm may be removed from the URL with the assumption that they are the default files. If that is true, they would be retrieved by simply using the base URL. It is important to be consistent while applying canonicalization rules. It is possible that two seemingly opposite rules work equally well (such as that for port numbers) as long as you apply them consistently across URLs. Other canonicalization rules may be applied based on the application and prior knowledge about some sites (e.g., known mirrors). As noted earlier spider traps pose a serious problem for a crawler. The "dummy" URLs created by spider traps often become increasingly larger in size. A way to tackle such traps is by limiting the URL sizes to, say, 128 or 256 characters. [1]

2.4.2 Stoplisting and Stemming
When parsing a Web page to extract content information or in order to score new URLs suggested by the page, it is often helpful to remove commonly used words or stopwords such as "it" and "can". This process of removing stopwords from text is called stoplisting. In addition to stoplisting, one may also stem the words found in the page. The stemming process normalizes words by concatenating a number of morphologically similar words to a single root form or stem. For example, "connect," "connected" and "connection" are all reduced to "connect." Implementations of the commonly used Porter stemming algorithm are easily available in many programming languages [1].

2.5 HTML TAG TREE
Crawlers may assess the value of a URL or a content word by examining the HTML tag context in which it resides. For this, a crawler may need to utilize the tag tree or DOM structure of the HTML page. The <html> tag forms the root of the tree and various tags and texts form nodes of the tree. Unfortunately, many Web pages contain badly written HTML. For example, a start tag may not have an end tag (it may not be required by the HTML specification), or the tags may not be properly nested. In many cases, the <html> tag or the <body> tag is all-together missing from the HTML page. Thus structure-based criteria often require the prior step of converting a "dirty" HTML document into a well-formed one, a process that is called tidying an HTML page. This includes both insertion of missing tags and the reordering of tags in the page. Tidying an HTML page is necessary for mapping the content of a page onto a tree structure with integrity, where each node has a single parent. Hence, it is an essential precursor to analyze an HTML page as a tag tree. Analyzing the DOM structure is only necessary if the topical crawler intends to use the HTML document structure in a non-trivial manner. For example, if the crawler only needs the links within a page, and the text or portions of the text in the page, one can use simpler HTML parsers. Such parsers are also readily available in many languages [1][2].

CONCLUSION
Due to the dynamism of the Web, crawling forms the backbone of applications that facilitate Web information retrieval. While the typical use of crawlers has been for creating and maintaining indexes for general purpose search engine, diverse usage of topical crawlers is emerging both for client and server based applications. Topical crawlers are becoming important tools to support applications such as specialized Web portals, online searching, and competitive intelligence. 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. For a current example, Chakrabarti [2] have suggested the use of the pages' HTML tag tree or DOM structure for focusing a crawler. While they have shown some benefit of using the DOM structure, a thorough study on the merits of using the structure (in different ways) for crawling is warranted. Topical crawlers depend on various cues from crawled pages to prioritize the fetching of unvisited URLs. A good understanding of the relative importance of cues such as the link-context, linkage (graph) structure, ancestor pages etc. is also needed. Another potential area of research is stronger collaboration between search engines and crawlers and among the crawlers themselves.

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