Chapter 7

Our Crawler Implementation

We developed a Web crawler that implements the crawling model and architecture presented in Chapter **??**, and supports the scheduling algorithms presented in Chapter **??**. This chapter presents the implementation of the Web crawler in some detail. Source code and technical documentation, including a user manual are available at `http://www.cwr.cl/projects/WIRE/`.

The rest of this chapter is organized as follows: section 7.1 presents the programming environment used. Section 7.2 details the main programs, section 7.3 the main data structures and section 7.4 the configuration variables.

7.1 Programming environment and dependencies

The programming language used was C for most of the application. We also used C++ to take advantage of the C++ Standard Template Library to shorten development time; however, we did not use the STL for the critical parts of our application (e.g.: we developed a specialized implementation of a hash table for storing URLs). The crawler currently has approximatelly $25,000$ lines of code.

For building the crawler, we used the following software packages:

ADNS [Jac02] Asynchronous Domain Name System resolver, replaces the standard DNS resolver interface with non-blocking calls, so multiple host names can be searched simultaneously. We used ADNS in the "harvester" program.

LibXML2 [lib02] An XML parser developed in C for the Gnome project. It is very portable, and it is also an efficient and very complete specification of the XPath language. We used XPath for the configuration file of the crawler, and for parsing the "robots.rdf" file used for Web server cooperation during the crawl, as shown in Chapter **??**.

We made extensive use of the `gprof` utility to improve the speed of the application.

7.2 Programs

In this section, we will present the four main programs: manager, harvester, gatherer and seeder. The four programs are run in cycles during the crawler's execution, as shown in Figure **??**.

7.2.1 Manager: long-term scheduling

The "manager" program generates the list of K URLs to be downloaded in this cycle (we used $K = 100,000$ pages by default). The procedure for generating this list is outlined below.

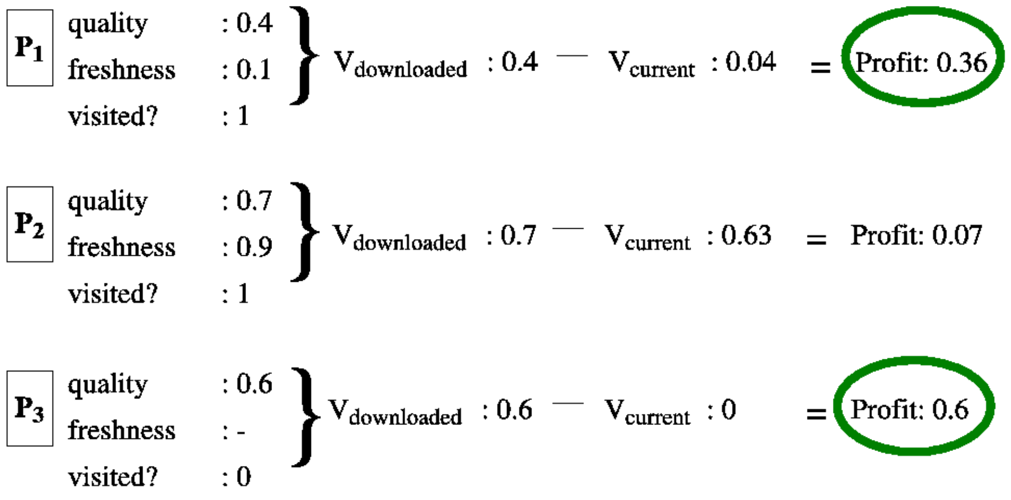

Figure 7.1: Operation of the manager program with $K = 2$. The two pages with the highest expected profit are assigned to this batch.

The current value of a page is $\text{IntrinsicQuality}(p) \times Pr(\text{Freshness}(p) = 1) \times \text{RepresentationalQuality}(p)$, where $\text{RepresentationalQuality}(p)$ equals 1 if the page has been visited, 0 if not. The value of the downloaded page is $\text{IntrinsicQuality}(p) \times 1 \times 1$. In Figure 7.1, the manager should select pages P_1 and P_3 for this cycle.

1. **Filter out pages that were downloaded too recently** In the configuration file, a criteria for the maximum frequency of re-visits to pages can be stated (e.g.: no more than once a day or once a week). This criteria is used to avoid accessing only a few elements of the collection, and is based on the observations by Cho and Garcia-Molina [CGM03].

2. **Estimate the intrinsic value of Web pages** The manager program calculates the value of all the Web pages in the collection according to a ranking function. The ranking function is specified in the con-

figuration file, and it is a combination of one or several of the following: Pagerank [PBMW98], static hubs and authority scores [Kle99], weighted link rank [Dav03, BYD04], page depth, and a flag indicating if a page is static or dynamic. It can also rank pages according to properties of the Web sites that contain the pages, such as "Siterank" (which is like Pagerank, but calculated over the graph of links between Web sites) or the number of pages that still have not been downloaded from that specific Web site, this is, the stragey presented in Chapter **??**.

3. **Estimate the freshness of Web pages** The manager programs estimates $Pr(Freshness_p = 1)$ for all pages that have been visited, using the information collected from past visits and the formulas presented in Section **??** (page **??**).

4. **Estimate the profit of retrieving a Web page** The program considers that the representational quality of a Web page is either 0 (page not downloaded) or 1 (page downloaded). Then it uses the formula given in Section **??** (page **??**) with $\alpha = \beta = \gamma = 1$ to obtain the profit, in terms of the value of the index, obtained by downloading the given page. This is high, e.g.: if the intrinsic value of the page is high, and the page copy is not expected to be fresh, so important pages are crawled more often.

5. **Extract top K pages according to expected profit** Or less than K pages if there are fewer URLs available. Pages are selected according to how much their value in the index will increase if they are downloaded now.

An hypothetical scenario for the manager program with $K = 2$ is depicted in Figure 7.1. The manager objective is to maximize the profit in each cycle.

For parallelization, the batch of pages generated by the manager is stored in a series of files that include all the URLs and metadata of the required Web pages and Web sites. It is a closed, independent unit of data that can be copied to a different machine for distributed crawling, as it includes all the information the harvester needs. Several batches of pages can be generated during the same cycle by taking more URLs from the top of the list.

7.2.2 Harvester: short-term scheduling

The "harvester" programs receives a list of K URLs and attempts to download them from the Web.

The politeness policy chosen is to never open more than one simultaneous connection to a Website, and to wait a configurable amount of seconds between accesses (default 15). For the larger Websites, over a certain quantity of pages (default 100), the waiting time is reduced (to a default of 5 seconds). This is because by the end of a large crawl only a few Web sites remain active, and the waiting time generates inefficiencies in the process that are studied in Chapter **??**.

As shown in Figure 7.2, the harvester maintains a queue for each Web site. At a given time, pages are being transferred from some Web sites, while other Web sites are idle to enforce the politeness policy. This is implemented using a priority queue in which Web sites are inserted according to a timestamp for their next visit.

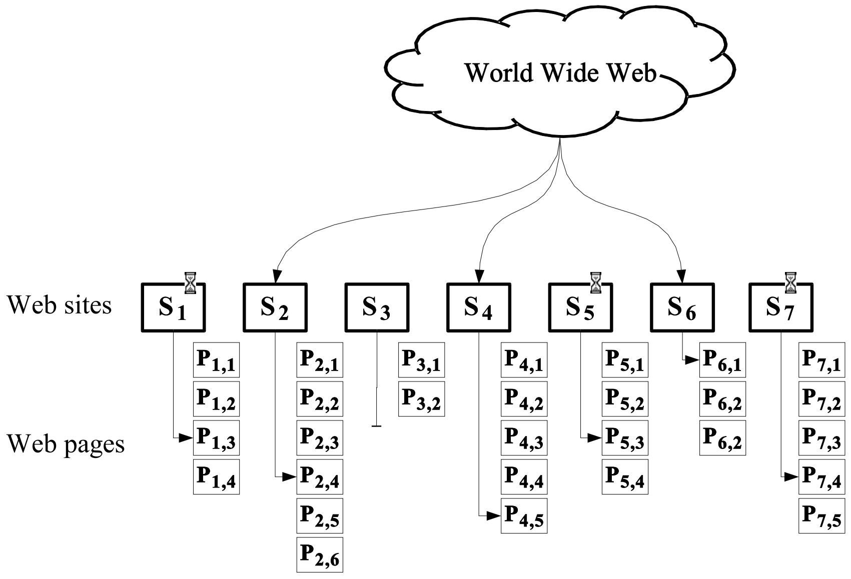
World Wide Web
Web sites
Web pages
S_1
S_2
S_3
S_4
S_5
S_6
S_7
$P_{1,1}$ $P_{1,2}$ $P_{1,3}$ $P_{1,4}$
$P_{2,1}$ $P_{2,2}$ $P_{2,3}$ $P_{2,4}$ $P_{2,5}$ $P_{2,6}$
$P_{3,1}$ $P_{3,2}$
$P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{4,4}$ $P_{4,5}$
$P_{5,1}$ $P_{5,2}$ $P_{5,3}$ $P_{5,4}$
$P_{6,1}$ $P_{6,2}$ $P_{6,2}$
$P_{7,1}$ $P_{7,2}$ $P_{7,3}$ $P_{7,4}$ $P_{7,5}$

Figure 7.2: Operation of the harvester program. This program creates a queue for each Web site and opens one connection to each active Web site (sites 2, 4, and 6). Some Web sites are "idle", because they have transfered pages too recently (sites 1, 5, and 7) or because they have exhausted all of their pages for this batch (3).

Our first implementation used Linux threads [Fal97] and did blocking I/O on each thread. It worked well, but was not able to go over 500 threads even in PCs with processors of 1GHz and 1GB of RAM. It seems that entire thread system was designed for only a few threads at the same time, not for higher degrees of parallelization.

Our current implementation uses a single thread with non-blocking I/O over an array of sockets. The `poll()` system call is used to check for activity in the sockets. This is much harder to implement than the multi-threaded version, as in practical terms it involves programming context switches explicitly, but the performance was much better, allowing us to download from over 1000 Web sites at the same time with a very lightweight process.

The output of the harvester is a series of files containing the downloaded pages and metadata found (e.g.: server response codes, document lengths, connection speeds, etc.). The response headers are parsed to obtain metadata, but the pages themselves are not parsed at this step.

7.2.3 Gatherer: parsing of pages

The “gatherer” program receives the raw Web pages downloaded by the harvester and parses them. In the current implementation, only `text/plain` and `text/html` pages are accepted by the harvester, so these are the only MIME types the gatherer has to deal with.

The parsing of HTML pages is done using an events-oriented parser. An events-oriented parser (such as SAX [Meg04] for XML) does not build an structured representation of the documents: it just generates function calls whenever certain conditions are met, as shown in Figure 7.3. We found that a substantial amount of pages were not well-formed (e.g.: tags were not balanced), so the parser must be very tolerant to malformed markup.

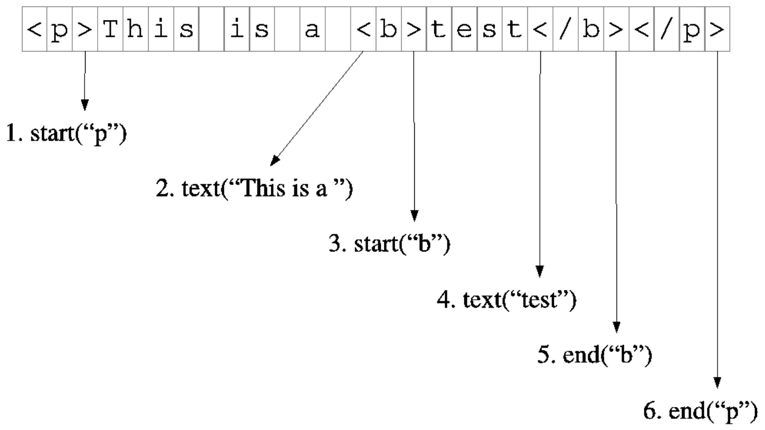

Figure 7.3: Events-oriented parsing of HTML data, showing the functions that are called while scanning the document.

During the parsing, URLs are detected and added to a list that is passed to the “seeder” program. At this point, exact duplicates are detected based on the page contents, and links from pages found to be duplicates are ignored to preserve bandwidth, as the prevalence of duplicates on the Web is very high [BBDH00].

The parser does not remove all HTML tags. It cleans superfluous tags and leaves only document structure, logical formatting, and physical formatting such as bold or italics. Information about colors, backgrounds, font families, cell widths and most of the visual formatting markup is discarded. The resulting file sizes are typically 30% of the original size and retain most of the information needed for indexing. The list of HTML tags are kept or removed is configurable by the user.

7.2.4 Seeder: URL resolver

The “seeder” program receives a list of URLs found by the gatherer, and adds some of them to the collection, according to a criteria given in the configuration file. This criteria includes patterns for accepting, rejecting, and transforming URLs.

Accept Patterns for accepting URLs include domain name and file name patterns. The domain name patterns are given as suffixes (e.g.: `.cl`, `.uchile.cl`, etc.) and the file name patterns are given as file extensions. In the later case, accepted URLs can be enqueued for download, or they can be just logged on a file, which is the current case for images and multimedia files.

Reject Patterns for rejecting URLs include substrings that appear on the parameters of known Web applications (e.g. `login`, `logout`, `register`, etc.) that lead to URLs which are not relevant for a search engine. In practice, this manually-generated list of patterns produces significant savings in terms of requests for pages with no useful information.

Transform To avoid duplicates from session ids, which are discussed in Section **??** (page **??**), we detect known session-id variables and remove them from the URLs. Log file analysis tools can detect requests coming from a Web crawler using the "user-agent" request header that is provided, so this should not harm Web server statistics.

The seeder also processes all the "robots.txt" and "robots.rdf" files that are found, to extract URLs and patterns:

robots.txt This file contains directories that should not be downloaded from the Web site [Kos96]. These directories are added to the patterns for rejecting URLs in a per-site basis.

robots.rdf This file contains paths to documents in the Web site, including their last-modification times. It is used for server cooperation, as explained on Chapter **??**.

The seeder also recognizes filename extensions for known programming languages used for the Web (e.g. `.php`, `.pl`, `.cfm`, etc.) and mark those URLs as "dynamic pages". Dynamic pages may be given higher or lower scores during long term scheduling.

To initialize the system, before the first batch of pages is generated by the manager, the seeder program is executed with a file providing the starting URLs for the crawl.

7.3 Data structures

7.3.1 Metadata

All the metadata about Web pages and Web sites is stored in files containing fixed-sized records. The records contain all the information about a Web page or Web site except for the URL and the contents of the Web page.

There are two files: one for metadata about Web sites, sorted by site-id, and one for metadata about Web pages, sorted by document-id. Metadata currently stored for a Web page includes information about:

Web page identification Document-id, which is an unique identifier for a Web page, and Site-id, which is an unique identifier for Web sites.

HTTP response headers HTTP response code and returned MIME-type.

Network status Connection speed and latency of the page download.

Freshness Number of visits, time of first and last visit, total number of visits in which a change was detected and total time with no changes. These are the parameters needed to estimate the freshness of a page.

Metadata about page contents Content-length of the original page and of the parsed page, hash function of the contents and original doc-id if the page is found to be a duplicate.

Page scores Pagerank, authority score, hub score, etc. depending on the scheduling policy from the configuration file.

Metadata currently stored for a Web site includes:

Web site identification Site-id.

DNS information IP-address and last-time it was resolved.

Web site statistics Number of documents enqueued/transfered, dynamic/static, erroneous/OK, etc.

Site scores Siterank, sum of Pagerank of its pages, etc. depending on the configuration file.

In both the file with metadata about documents, and the file with metadata about Web sites, the first record is special, as it contains the number of stored records. There is no document with doc-id$=0$ nor Web site with site-id$=0$, so identifier 0 is reserved for error conditions and record for document i is stored at offset $\text{sizeof}(\text{docid}) \times i$.

7.3.2 Page contents

The contents of Web pages are stored in variable-sized records indexed by document-id. Inserts and deletions are handled using a free-space list with first-fit allocation.

This data structure also implements duplicate detection: whenever a new document is stored, a hash function of its contents is calculated. If there is another document with the same hash function and length, the contents of the documents are compared. If they are equal, the document-id of the original document is returned, and the new document is marked as a duplicate.

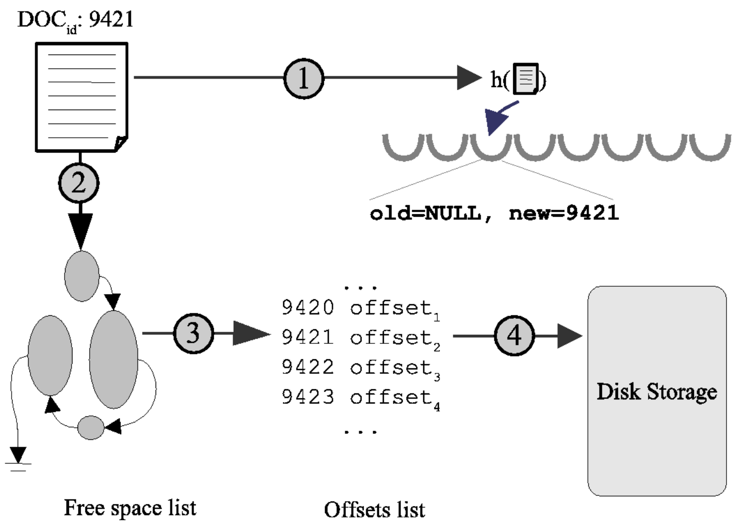

Figure 7.4: Storing the contents of a document requires to check first if the document is a duplicate, then searching for a place in the free-space list, and then writing the document to disk.

The process for storing a document, given its contents and document-id, is depicted in Figure 7.4:

1. The contents of the documents are checked against the content-seen hash table. If they have been already seen, the document is marked as a duplicate and the original doc-id is returned.
2. A free space is searched in the free-space list. This returns a document offset in the disk pointing to an available position with enough free space.
3. This offset is written to the index, and will be the offset for the current document.
4. The document contents are written to the disk at the given offset.

This module requires support to create large files, as for large collections the disk storage grows over 2GB, and the offset cannot be provided in a variable of type "`long`". In Linux, the LFS standard [Jae04] provides offsets of type "`long long`" that are used for disk I/O operations. The usage of continuous, large files for millions of pages, instead of small files, can save a lot of disk seeks, as noted also by Patterson [Pat04].

7.3.3 URLs

The structure that holds the URLs is highly optimized for the most common operations during the crawling process:

- Given the name of a Web site, obtain its site-id.
- Given the site-id of a Web site and a local link, obtain the doc-id for the link.
- Given a full URL, obtain both its site-id and doc-id.

The implementation uses two hash tables: the first for converting Web site names into site-ids, and the second for converting "site-id + path name" to a doc-id. The process for converting a full URL is shown in Figure 7.5.

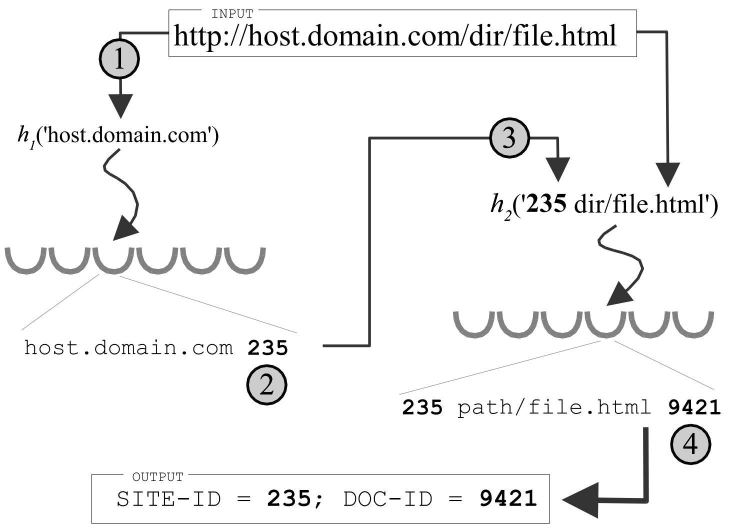

Figure 7.5: For checking a URL: (1) the host name is searched in the hash table of Web site names. The resulting site-id (2) is concatenated with the path and filename (3) to obtain a doc-id (4).

This process is optimized to exploit the locality on Web links, as most of the links found in a page point to other pages co-located in the same Web site.

7.3.4 Link structure

The link structure is stored on disk as an adjacency list of document-ids. This adjacency list is implemented on top of the same data structure used for storing the page contents, except for the duplicate checking. As only the forward adjacency list is stored, the algorithm for calculating Pagerank cannot access efficiently the list of back-links of a page, so it must be programmed to use only forward links. This is not difficult to do, and Algorithm 1 illustrates how to calculate Pagerank without back-links; the same idea is also used for hubs and authorities.

Our link structure does not use compression. The Web graph can be compressed by exploiting the locality of the links, the distribution of the degree of pages, and the fact that several pages share a substantial

Algorithm 1 Calculating Pagerank without back-links

```
Require: $G$ Web Graph.
Require: $q$ dampening factor, usually $q \approx 0.15$
 1: $N \leftarrow |G|$
 2: for each $p \in G$ do
 3:     $\text{Pagerank}_p = \frac{1}{N}$
 4:     $\text{Aux}_p = 0$
 5: end for
 6: while Pagerank not converging do
 7:     for each $p \in G$ do
 8:         $\Gamma^+(p) \leftarrow$ pages pointed by $p$
 9:         for each $p' \in \Gamma^+(p)$ do
10:             $\text{Aux}_{p'} = \text{Aux}_{p'} + \frac{\text{Pagerank}_p}{|\Gamma^+(p)|}$
11:         end for
12:     end for
13:     for each $p \in G$ do
14:         $\text{Pagerank}_p = \frac{q}{N} + (1-q)\text{Aux}_p$
15:         $\text{Aux}_p = 0$
16:     end for
17:     Normalize Pagerank: $\sum \text{Pagerank}_p = 1$
18: end while
```

portion of their links [SY01]. Using compression, a Web graph can be represented with as few as 3-4 bits per link [**?**].

7.4 Configuration

Configuration of the crawling parameters is done with a XML file. Internally, there is a mapping between XPath expressions (which represent parts of the XML file) and internal variables with native data types such as integer, float or string. When the crawler is executed, these internal variables are filled with the data given in the configuration file.

Table 7.1 shows the main configuration variables with their default values. For a detail of all the configuration variables, see the WIRE documentation at `http://www.cwr.cl/projects/WIRE/doc/`.

7.5 Conclusions

This chapter described the implementation of the WIRE crawler, which is based on the crawling model developed for this thesis. The Web as an information repository is very challenging, especially because of its dynamic and open nature; thus, a good Web crawler needs to deal with some aspects of the Web that become visible only while running an extensive crawl, and there are several special cases, as shown in Appendix **??**.

There are a few public domain crawling programs listed under Section **??** (page **??**). We expect to benchmark our crawler against some of them in the future, but there is still work to do to get the most out of this architecture. The most important task is to design a component for coordinating several instances of the Web crawler running in different machines, or to be able to carry two parts of the process at the same time, such as running the harvester while the gatherer is working on a previous batch. This is necessary because otherwise the bandwidth is not used while parsing the Web pages.

Our first implementation of the Web crawler used a relational database and threads for downloading the Web pages, and the performance was very low. Our current implementation, with the data structures presented in this chapter, is powerful enough for downloading collections in the order of tens of millions of pages in a few days, which is reasonable for the purposes of creating datasets for simulation and analysis, and for testing different strategies. There is plenty of room for enhancements, especially in the routines for manipulating the Web graph –which is currently not compressed, but should be compressed for larger datasets– and for calculating link-based scores.

Also, for scaling to billions of Web pages, some data structures should be analized on disk instead of in memory. This development is outside the scope of this thesis, but seems a natural continuation of this work.

The next two chapters present a study of a Web collection and the practical problems found while performing this large crawl.

XPath expression	Default value	Description
collection/base	/tmp/	Base directory for the crawler
collection/maxdoc	10 Mill.	Maximum number of Web pages.
collection/maxsite	100,000	Maximum number of Web sites.
seeder/max-urls-per-site	25,000	Max. pages to download from each Web site.
seeder/accept/domain-suffixes	.cl	Domain suffixes to accept.
seeder/ext/download/static	*	Extensions to consider as static.
seeder/ext/download/dynamic	*	Extensions to consider as dynamic.
seeder/ext/log/group	*	Extensions of non-html files.
seeder/sessionids	*	Suffixes of known session-id parameters.
manager/maxdepth/dynamic	5	Maximum level to download dynamic pages.
manager/maxdepth/static	15	Maximum level to download static pages.
manager/batch/size	100,000	URLs per batch.
manager/batch/samesite	500	Max. number of URLs from the same site.
manager/score	*	Weights for the different score functions.
manager/minperiod	*	Minimum re-visiting period.
harvester/resolvconf	127.0.0.1	Address of the name server(s).
harvester/blocked-ip	127.0.0.1	IPs that should not be visited.
harvester/nthreads/start	300	Number of simultaneous threads or sockets.
harvester/nthreads/min	10	Minimum number of active sockets.
harvester/timeout/connection	30	Timeout in seconds.
harvester/wait/normal	15	Number of seconds to wait (politeness).
harvester/maxfilesize	400,000	Maximum number of bytes to download.
gatherer/maxstoredsize	300,000	Maximum number of bytes to store.
gatherer/discard	*	HTML tags to discard.
gatherer/keep	*	HTML tags to keep.
gatherer/link	*	HTML tags that contain links.

Table 7.1: Main configuration variables of the Web crawler. Default values marked “*” can be seen at `http://www.cwr.cl/projects/WIRE/doc/`

Bibliography

[BBDH00] Krishna Bharat, Andrei Z. Broder, Jeffrey Dean, and Monika Rauch Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *Journal of the American Society of Information Science*, 51(12):1114–1122, 2000.

[BYD04] Ricardo Baeza-Yates and Emilio Davis. Web page ranking using link attributes. In *Alternate track papers & posters of the 13th international conference on World Wide Web*, pages 328–329. ACM Press, 2004.

[CGM03] Junghoo Cho and Hector Garcia-Molina. Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4), December 2003.

[Dav03] Emilio Davis. Módulo de búsqueda en texto completo para la web con un nuevo ranking estático, October 2003. Honors Thesis.

[Fal97] Sean Falton. Linux threads frequently asked questions. http://www.tldp.org/FAQ/Threads-FAQ/, January 1997.

[Jac02] Ian Jackson. ADNS. http://www.chiark.greenend.org.uk/∼ian/adns/, 2002.

[Jae04] Andreas Jaeger. Large file support in linux. http://www.suse.de/∼aj/linux_lfs.html, June 2004.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[Kos96] Martijn Koster. A standard for robot exclusion. http://www.robotstxt.org/wc/exclusion.html, 1996.

[lib02] Libxml - the xml c library for gnome. http://www.xmlsoft.org/, 2002.

[Meg04] David Megginson. Simple API for XML (SAX 2.0). http://sax.sourceforge.net/, 2004.

[Pat04] Anna Patterson. Why writing your own search engine is hard. *ACM Queue*, pages 49 – 53, April 2004.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation algorithm: bringing order to the web. In *Proceedings of the seventh conference on World Wide Web*, Brisbane, Australia, April 1998.

[SY01] Torsten Suel and Jun Yuan. Compressing the graph structure of the Web. In *Proceedings of the Data Compression Conference DCC*, pages 213 – 222. IEEE Computer Society, 2001.