Implementing A Web Crawler

By

Jai Aditya Shaunik 071279
Jaspreet Singh 071285
Neha Nagpal 071345

Under the Supervision of

Prof. Ravi Rastogi

Submitted in partial fulfillment
of the requirements for the degree of

BACHELOR OF TECHNOLOGY
(CSE & / IT)

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY
WAKNAGHAT
SOLAN, HIMACHAL PRADESH
INDIA
2011

CONTENTS

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

WAKNAGHAT

SOLAN, HIMACHAL PRADESH

Date: 09 May 2011

CERTIFICATE

This is to certify that the work entitled Implementing A Web Crawler, submitted by Jai Aditya Shaunik, Jaspreet Singh and Neha Nagpal in partial fulfilment for award of degree of Bachelor of Technology (Computer Science & Engineering) of Jaypee University of Information Technology has been carried out in my supervision. This work has not been submitted partially or wholly to any other university or institution for award of this or any other degree programme.

Ravi Rastogi

Project Supervisor

Prof. Ravi Rastogi

Senior Lecturer, JUIT

ACKNOWLEDGEMENTS

This project has been an outcome of sustained and continual efforts on part of every group member. We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project.

We are deeply indebted to our project guide **Prof. Ravi Rastogi** for his help, stimulating suggestions and encouragement given throughout the project. We feel fortunate to be taught by him.

Also, we are grateful for **Prof. Suman Saha,** as he was our project guide last semester and always gave us his coherent guidance and unwavering support.

[illegible]	[illegible]	Neha Nagpal
Jai Aditya Shaunik	Jaspreet Singh	Neha Nagpal
071279	071285	071345

SUMMARY

As the importance of the internet grows by leaps and bounds every year, it has become an ever-increasing and unmanageable repository of data. To convert this data into information, we use the software, Web crawler.

After giving a basic understanding of the Web Crawler, we have introduced more complex policies that make it more effective and efficient with a higher rate of output accuracy.

The project is implemented on a small scale so as to act as a prototype.

The project is fully scalable and can be deployed on larger scale as well.

Ravi Rastogi

Project Supervisor

Prof. Ravi Rastogi

Senior Lecturer, JUIT

LIST OF FIGURES

Chapter 1

PROJECT DETAILS

1.1 PROBLEM STATEMENT

It is proposed to design software that can download pages from the World Wide Web and process them to obtain further Uniform Resource Locators (URL) which too will be processed for specific information. The program is meant to be automatic and have an indefinite runtime with no human intervention after automation begins.

The software will comprise of four policies, selection, refresh, politeness and parallelization, which will define its working.

The project will involve the following tasks:

- PART 1 (7th Semester)
  - Understanding fundamentals regarding crawling policies
  - Implementing a basic working Web crawler
  - Using basic Selection and Refresh Policies
  - Using Robots Exclusion Protocol
- PART 2 (8th Semester)
  - Establishing an efficient refresh policy
  - Creating a user-friendly GUI
  - Researching on parallelization policies that can implement distributed web crawlers
  - Display of results on the WWW

1.2 METHODOLOGY

The Software requires constant access to the net. It is an automated application i.e. once it is initialized; it requires no interference by the user.

Firstly, a basic back end processing is done that can fetch a web page and process its source code searching for hyperlinks. These hyperlinks are stored in a data structure to be downloaded later.

A selection policy is used to give an importance metric to the URLs. A refresh policy has also been used to keep the pages updated and fresh. Also, a politeness policy in the form of Robots Exclusion Protocol has been implemented to make sure that the web crawler is used in an ethical fashion.

1.3 LIMITATIONS

The efficiency of the crawler is low at present, due to low download speeds available to us. Further, the crawler is not able to run at a continuous indefinite fashion, since limited internet hours in a day are available. A computer capable of high powered processing will also be needed to execute an optimized crawler. Even if these requirements are met, a distributed web crawler using multiple nodes is necessary to get any effective results.

1.4 RESOURCES

Hardware:

- AMD Phenom™ II X4 945 Processor – 3.00GHz
- 4GB DDR3 RAM
- 6 hour per day internet access at 128kbps speed

Software

- Java SDK 1.6.022
- Java HotSpot(TM) Client VM 17.1-b03
- NetBeans IDE 6.9.1
- Windows 7 Ultimate 64-bit OS

Chapter 2

INTRODUCTION

2.1 DEFINITION

“An automated computer program that browses the World Wide Web to obtain data.”

A crawler is a program that retrieves and stores pages from the Web, commonly for a Web search engine. A crawler often has to download hundreds of millions of pages in a short period of time and has to constantly monitor and refresh the downloaded pages.

A Web crawler is a computer program that browses the World Wide Web in a methodical, automated manner or in an orderly fashion. Other terms for Web crawlers are ants, automatic indexers, bots, or Web spiders, Web robots.

2.2 EXAMPLES OF CRAWLERS

The following is a list of published Crawler Architectures:-

1. Yahoo! Slurp: is the name of the Yahoo Search crawler.

2. Msnbot: is the name of Microsoft's Bing web crawler.

3. FAST Crawler: is a distributed crawler, used by Fast Search & Transfer

Open-Source Crawlers:-

1. AspSeek: is a crawler, indexer and a search engine written in C++ and licensed under the GPL

2. Crawler4j: is a crawler written in Java and released under an Apache License. It can be configured in a few minutes and is suitable for educational purposes.

3. DataparkSearch: is a crawler and search engine released under the GNU General Public License.

4. GNU Wget: is a command-line-operated crawler written in C and released under the GPL. It is typically used to mirror Web and FTP sites.

2.3 CRAWLER POLICIES

There are important characteristics of the Web that make crawling very difficult:

- its large volume,
- its fast rate of change, and
- Dynamic page generation.

The large volume implies that the crawler can only download a fraction of the Web pages within a given time, so it needs to prioritize its downloads. The high rate of change implies that by the time the crawler is downloading the last pages from a site, it is very likely that new pages have been added to the site, or that pages have already been updated or even deleted.

The number of possible crawl-able URLs being generated by server-side software has also made it difficult for web crawlers to avoid retrieving duplicate content. Endless combinations of HTTP GET parameters exist, of which only a small selection will actually return unique content

A crawler must carefully choose at each step which pages to visit next.

The behavior of a Web crawler is the outcome of a combination of policies:

- Selection policy that states which pages to download
- Re-visit policy that states when to check for changes to the pages
- Politeness policy that states how to avoid overloading Web sites
- Parallelization policy that states how to coordinate distributed Web crawlers

Chapter 3

IMPLEMENTING AN EFFECTIVE WEB CRAWLER

3.1 INTRODUCTION

We already know that a Web crawler (also known as a Web spider or Web robot) is a program or automated script which browses the World Wide Web in a methodical and automated manner.

This process is called Web crawling or spidering. Many legitimate sites, in particular search engines, use spidering as a means of providing up-to-date data. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine, which will index the downloaded pages to provide fast searches. Crawlers can also be used for automating maintenance tasks on a Web site, such as checking links or validating HTML codes. Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting e-mail addresses (usually for spam).

A Web crawler is one type of bot, or software agent. In general, it starts with a list of URLs to visit, called the seeds. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, called the crawl frontier.

3.2 Why do we need a web crawler?

Following are some reasons to use a web crawler:

- To maintain mirror sites for popular Web sites.
- To test web pages and links for valid syntax and structure.
- To monitor sites to see when their structure or contents change.
- To search for copyright infringements.
- To build a special-purpose index.for example, one that has some understanding of the content stored in multimedia files on the Web.

3.3 How does a web crawler work?

A typical web crawler starts by parsing a specified web page: noting any hypertext links on that page that point to other web pages. The Crawler then parses those pages for new links, and so on, recursively. A crawler is a software or script or automated program which resides on a single machine. The crawler simply sends HTTP requests for documents to other machines on the Internet, just as a web browser does when the user clicks on links. All the crawler really does is to automate the process of following links.

This is the basic concept behind implementing web crawler, but implementing this concept is not merely a bunch of programming. The next section describes the difficulties involved in implementing an efficient web crawler.

3.4 Difficulties in implementing efficient web crawler

There are two important characteristics of the Web that generate a scenario in which Web crawling is very difficult:

1. Large volume of Web pages.
2. Rate of change on web pages.

A large volume of web page implies that web crawler can only download a fraction of the web pages and hence it is very essential that web crawler should be intelligent enough to prioritize download.

Another problem with today.s dynamic world is that web pages on the internet change very frequently, as a result, by the time the crawler is downloading the last page from a site, the page may change or a new page has been placed/updated to the site.

3.5 Solutions - Right strategies

The difficulties in implementing efficient web crawler clearly state that bandwidth for conducting crawls is neither infinite nor free. So, it is becoming essential to crawl the web in not only a scalable, but efficient way, if some reasonable amount of quality or freshness of web pages is to be maintained. This ensues that a crawler must carefully choose at each step which pages to visit next.

Thus the implementer of a web crawler must define its behavior.

Defining the behavior of a Web crawler is the outcome of a combination of below mentioned strategies:

- Selecting the better algorithm to decide which page to download.
- Strategizing how to re-visit pages to check for updates.
- Strategizing how to avoid overloading websites.

3.6 Selecting the right algorithm

Given the current size of the web, it is essential that the crawler program should crawl on a fraction of the web. Even large search engines in today.s dynamic world crawls fraction of web pages from web. But, a crawler should observe that the fraction of pages crawled must be most relevant pages, and not just random pages.

While selecting the search algorithm for the web crawler an implementer should keep in mind that algorithm must make sure that web pages are chosen depending upon their importance. The importance of a web page lies in its popularity in terms of links or visits, or even its URL.

3.7 Algorithm types

3.7.1 Path-ascending crawling

We intend the crawler to download as many resources as possible from a particular Web site. That way a crawler would ascend to every path in each URL that it intends to crawl. For example, when given a seed URL of http://foo.org/a/b/page.html, it will attempt to crawl /a/b/, /a/, and /.

The advantage with Path-ascending crawler is that they are very effective in finding isolated resources, or resources for which no inbound link would have been found in regular crawling.

3.7.2 Focused crawling

The importance of a page for a crawler can also be expressed as a function of the similarity of a page to a given query. In this strategy we can intend web crawler to download pages that are similar to each other, thus it will be called focused crawler or topical crawler.

The main problem in focused crawling is that in the context of a Web crawler, we would like to be able to predict the similarity of the text of a given page to the query before actually downloading the page. A possible predictor is the anchor text of links; to resolve this problem proposed solution would be to use the complete content of the pages already visited to infer the similarity between the driving query and the pages that have not been visited yet.

The performance of a focused crawling depends mostly on the richness of links in the specific topic being searched, and a focused crawling usually relies on a general Web search engine for providing starting points.

3.8 How to Re-visit web pages

The optimum method to re-visit the web and maintain average freshness high of web page is to ignore the pages that change too often.

The approaches could be:

- Re-visiting all pages in the collection with the same frequency, regardless of their rates of change.
- Re-visiting more often the pages that change more frequently.

(In both cases, the repeated crawling order of pages can be done either at random or with a fixed order.)

The re-visiting methods considered here regard all pages as homogeneous in terms of quality ("all pages on the Web are worth the same"), something that is not a realistic scenario.

3.9 How to avoid overloading websites

Crawlers can retrieve data much quicker and in greater depth than human searchers, so they can have a crippling impact on the performance of a site. Needless to say if a single crawler is performing multiple requests per second and/or downloading large files, a server would have a hard time keeping up with requests from multiple crawlers.

The use of Web crawler is useful for a number of tasks, but comes with a price for the general community. The costs of using Web crawlers include:

- Network resources, as crawlers require considerable bandwidth and operate with a high degree of parallelism during a long period of time.
- Server overload, especially if the frequency of accesses to a given server is too high.
- Poorly written crawlers, which can crash servers or routers, or which download pages they cannot handle.
- Personal crawlers that, if deployed by too many users, can disrupt networks and Web servers.

To resolve this problem we can use robots exclusion protocol, also known as the robots.txt protocol.

The robots exclusion standard or robots.txt protocol is a convention to prevent cooperating web spiders and other web robots from accessing all or part of a website. We can specify the top level directory of web site in a file called robots.txt and this will prevent the access of that directory to crawler.

This protocol uses simple substring comparisons to match the patterns defined in robots.txt file. So, while using this robots.txt file we need to make sure that we use final ./. character appended to directory path. Else, files with names starting with that substring will be matched rather than directory.

Example of robots.txt files that tells all crawlers not to enter into four directories of a website:

```
User-agent: *
Disallow: /cgi-bin/
Disallow: /images/
Disallow: /tmp/
Disallow: /private/
```

3.10 Web crawler architectures

A crawler must have a good crawling strategy, as noted in the previous sections, but it also needs a highly optimized architecture.

3.11 Pseudo code for a web crawler

Here's a pseudo code summary of the algorithm that can be used to implement a web crawler:

```
Ask user to specify the starting URL on web and file type that crawler should crawl.

Add the URL to the empty list of URLs to search.

While not empty ( the list of URLs to search )
{

        Take the first URL in from the list of URLs
        Mark this URL as already searched URL.

        If the URL protocol is not HTTP then
                break;
                go back to while
        If robots.txt file exist on site then
                If file includes .Disallow. statement then
                        break;
                        go back to while

        Open the URL

        If the opened URL is not HTML file then
                Break;
                Go back to while

        Iterate the HTML file

        While the html text contains another link {

                If robots.txt file exist on URL/site then
                        If file includes .Disallow. statement then
                                break;
                                go back to while

                If the opened URL is HTML file then
                        If the URL isn't marked as searched then
                                Mark this URL as already searched URL.

                Else if type of file is user requested
                        Add to list of files found.

        }
}
```

Chapter 4

SELECTION POLICY

4.1 DEFINITION

The crawler must deal with huge volumes of data. Unless it has unlimited computing resources and unlimited time, it must carefully decide what URLs to download and in what order. In this chapter we address this important challenge: How should a crawler select URLs to download from its list of known URLs? If a crawler intends to perform a single scan of the entire Web, and the load placed on target sites is not an issue, then any URL order will suffice. That is, eventually every single known URL will be visited, so the order is not critical. However, most crawlers will not be able to visit every possible page for two main reasons:

1. The crawler or its client may have limited storage capacity, and may be unable to index or analyze all pages. Currently the Web is believed to have several terabytes of textual data and is growing rapidly, so it is reasonable to expect that most clients will not want or will not be able to cope with all that data.

2. Crawling takes time, so at some point the crawler may need to start revisiting previously retrieved pages, to check for changes. This means that it may never get to some pages. It is currently estimated that there exist more than one billion pages available on the Web and many of these pages change at rapid rates.

It is important for the crawler to visit "important" pages first, so that the fraction of the Web that is visited is more meaningful. In the following sections, we present several different useful definitions of importance, and develop crawling priorities so that important pages have a higher probability of being visited first.

4.2 IMPORTANCE METRICS

Not all pages are necessarily of equal interest to a crawler's client. For instance, if the client is building a specialized database on a particular topic, then pages that refer to that topic are more important, and should be visited as early as possible. Similarly, a search engine may use the number of Web URLs that point to a page, the back-link count, to rank user query results. If the crawler cannot visit all pages, then it is better to visit those with a high back-link count, since this will give the end-user higher ranking results. Given a Web page, we can define the importance of the page, in one of the following ways.

1. **Similarity to a Driving Query:** A query drives the crawling process, and importance is defined to be the textual similarity between web page and query. One common way to compute the significance is to multiply the number of times the ith word appears in the document by the inverse document frequency of the ith word. The idf factor is one divided by the number of times the word appears in the entire "collection," which in this case would be the entire Web. The idf factor corresponds to the content discriminating power of a word. A term that appears rarely in document has a high idf , while a term that occurs in many documents has a low idf.

2. **Back-link Count:** The value of importance if the page is the number of links to web page that appear over the entire Web. We use query to refer to this importance metric. Intuitively, a page that is linked to by many pages is more important than one that is seldom referenced.

3. **Page Rank:** The Back-Link metric treats all links equally. Thus, a link from the goggle home page counts the same as a link from some individual's home page. However, since the Google home page is more important, it would make sense to value that link more highly. The Page-Rank back-link metric, recursively defines the importance of a page to be the weighted sum of the importance of the pages that have back-links to web page. Such a metric has been found to be very useful in ranking results of user queries.

4. **Forward Link Count:** A page with many outgoing links is very valuable, since it may be a Web directory. This metric can be computed directly from web page. This kind of

metric has been used in conjunction with other factors to reasonably identify index pages. We could also define a weighted forward link metric, analogous to Information Retrieval of web page.

5. **Location Metric:** The importance of page web page is a function of its location, not of its contents. If URL leads to web page, then important location of web page is a function of URL. For example, URLs ending with ".com" may be deemed more useful than URLs with other endings, or URLs containing the string "home" may be more of interest than other URLs.

4.3 FOCUSED CRAWLING

4.3.1 Introduction

The importance of a page for a crawler can also be expressed as a function of the similarity of a page to a given query. Web crawlers that attempt to download pages that are similar to each other are called focused crawler or topical crawlers.

The main problem in focused crawling is that in the context of a Web crawler, we would like to be able to predict the similarity of the text of a given page to the query before actually downloading the page. A possible predictor is the anchor text of links; this was the approach in a crawler developed in the early days of the Web. In focused crawling, the purpose is to use the complete content of the pages already visited to infer the similarity between the driving query and the pages that have not been visited yet. The performance of a focused crawling depends mostly on the richness of links in the specific topic being searched, and a focused crawling usually relies on a general Web search engine for providing starting points.

4.3.2 Summary

The rapid growth of the World-Wide Web poses unprecedented scaling challenges for general-purpose crawlers and search engines. We have now described a hypertext resource discovery system called the Focused Crawler. 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.

To achieve such goal-directed crawling, we make use of two hypertext mining programs that guide the crawler: a classifier that evaluates the relevance of a hypertext document with respect to the focus topics, and a distiller that identifies hypertext nodes that are great access points to many relevant pages within a few links. Focused crawling acquires relevant pages steadily while standard crawling quickly loses its way, even though they are started from the same root set. Focused crawling is robust against large perturbations in the starting set of URLs. It discovers largely overlapping sets of resources in spite of these perturbations. It is also capable of exploring out and discovering valuable resources that are dozens of links away from the start set, while carefully pruning the millions of pages that may lie within this same radius. Hence, focused crawling is very effective for building high-quality collections of web documents on specific topics, using modest desktop hardware.

4.4 OCCURING PROBLEMS

Our goal is to design a crawler that if possible visits high important query pages before lower ranked ones, for some definition of importance of query. Of course, the crawler will only have available important values, so based on these it will have to guess what the high importances of pages to fetch next are.

Our general goal can be stated more precisely in three ways, depending on how we expect the crawler to operate:

1. Crawl & Stop
2. Crawl & Stop with Threshold
3. Limited Buffer Crawl

1. **Crawl & Stop:**

The crawler starts at its initial page and stops after visiting some pages. At this point an "ideal" crawler would have visited pages r1. . . rK, where r1 is the page with the highest importance value, r2 is the next highest, and so on. We call pages r1 through rK the "hot" pages. The performance of the ideal crawler is of course 1. A crawler that somehow manages to visit pages entirely at random, and may revisit pages, would have a performance of K/T, where T is the total number of pages in the Web.

2. **Crawl & Stop with Threshold:**

We again assume that the crawler visits K pages.However, we are now given an importance target let's saying G, and any page with important query ≥ G is considered hot. Let us assume that the total number of hot pages is H. The performance of the crawler is the fraction of the H hot pages that have been visited when the crawler stops. If K < H, then an ideal crawler will have performance K/H. If K ≥ H, then the ideal crawler has the perfect performance 1. A purely random crawler that revisits pages is expected to visit (H/T) K hot pages when it stops. Thus, its performance is K/T. Only when the random crawler visits all T pages is its performance expected to be 1.

3. **Limited Buffer Crawl:**

In this model we consider the impact of limited storage on the crawling process. We assume that the crawler can only keep *B* pages in its buffer.

Thus, after the buffer fills up, the crawler must decide what pages to flush to make room for new pages. An ideal crawler could simply drop the pages with lowest *I(p)* value, but a real crawler must guess which of the pages in its buffer will eventually have low *I(p)* values. We allow the crawler to visit a total of *T* pages, equal to the total number of Web pages. At the end of this process, the fraction of the *B* buffer pages that are hot gives us the performance *PBC(C)*. We can define hot pages to be those with $I(p) \geq G$, where *G* is a target importance, or those with $I(p) \geq I(rB)$, where *rB* is the page with the *B*th highest importance value. The performance of an ideal and a random crawler are analogous to those in the previous cases.

Note that to evaluate a crawler under any of these metrics, we need to compute the actual *I(p)* values of pages, and this involves crawling the "entire" Web.

4.5 ORDERING METRICS

A crawler keeps a queue of URLs it has seen during a crawl, and must select from this queue the next URL to visit. The ordering metric *O* is used by the crawler for this selection, i.e., it selects the URL *u* such that *O(u)* has the highest value among all URLs in the queue.

The *O* metric can only use information seen (and remembered if space is limited) by the crawler. The *O* metric should be designed with an importance metric in mind. For instance, if we are searching for high *IB(p)* pages, it makes sense to use *O(u)* = *IB_(p)*, where *p* is the page *u* points to. However, it might also make sense to use *O(u)* = *IR_(p)*.

For a location importance metric *IL(p)*, we can use that metric directly for ordering since the URL of *p* directly gives the *IL(p)* value. However, for forward link *IF(p)* and similarity *IS*(*p*) metrics, it is much harder to devise an ordering metric since we have not seen *p* yet.

As we will see, for similarity, we may be able to use the text that anchors the URL *u* as a predictor of the text that *p* might contain.
Thus, one possible ordering metric *O*(*u*) is *IS*(*A*,*Q*), where *A* is the anchor text of the URL *u*, and *Q* is the driving query.

4.6 EXPERIMENTS

These experiments were done by Junghoo Cho for his PHD dissertation at Stanford University. We have briefly included his presentations as they have a high relevance to the development of our Web Crawler.

To avoid network congestion and heavy loads on the servers, the experimental evaluation was done in two steps. In the first step, all Web pages were physically crawled and a local repository of the pages was built. After we built the repository, we ran our virtual crawlers on it to evaluate different crawling schemes. Even though we had the complete image of the in the repository, our virtual crawler based its crawling decisions only on the pages it saw for itself. In this section we briefly discuss how the particular database was obtained for our experiments.

Description of dataset:

To download an image of the Web pages, we started Web Base with an initial list of URLs. These 89,119 URLs were obtained from an earlier crawl. Also, we limited the actual data that we collected for two reasons.

The first is that many heuristics are needed to avoid automatically generated, and potentially infinite, sets of pages. For example, any URLs containing "/cgi-bin/" are not crawled, because they are likely to contain programs which generate infinite sets of pages, or produce other undesirable side effects such as an unintended vote in an online election. We used similar heuristics to avoid downloading pages generated by programs.

Another way the data set is reduced is through the robots exclusion protocol, which allows Webmasters to define pages they do not want crawled by automatic systems.

At the end of the process, we had downloaded 375,746 pages and had 784,592 known URLs to the pages. The crawl was stopped before it was complete, but most of the uncrawled URLs were on only a few servers, so we believe the dataset we used to be a reasonable representation of Web. In particular, it should be noted that 352,944 of the known URLs were on one server, which has a program that could generate an unlimited number of Web pages.
Since the dynamically-generated pages on the server had links to other dynamically generated pages, we would have downloaded an infinite number of pages if we naively followed the links.

Our dataset consisted of about 225,000 crawled "valid" HTML pages, 1 which consumed roughly 2.5GB of disk space. However, out of these 225,000 pages, 46,000 pages were unreachable from the starting point of the crawl, so the total number of pages for our experiments was 179,000.

We should stress that the virtual crawlers that will be discussed next do not use WebBase directly. As stated earlier, they use the dataset collected by the Web Base crawler, and do their own crawling on it. The virtual crawlers are simpler than the WebBase crawler.

For instance, they can detect if a URL is invalid simply by seeing if it is in the dataset. Similarly, they do not need to distribute the load to visited sites. These simplifications are fine, since the virtual crawlers are only used to evaluate ordering schemes, and not to do real crawling.

Backlink-based crawlers

In this section we study the effectiveness of various ordering metrics, for the scenario where importance is measured through backlinks (i.e., either the IB(p) or IR(p) metrics). We start by describing the structure of the virtual crawler, and then consider the different ordering metrics.

For the PageRank metric we use a damping factor d of 0.9 (for both IR(p) and IR_(p)) for all of our experiments. First algorithm shows our basic virtual crawler. The crawler manages three main data structures. Queue url queue contains the URLs that have been seen and need to be visited. Once a page is visited, it is stored (with its URL) in crawled pages. links holds pairs of the form (u1, u2), where URL u2 was seen in the visited page with URL u1. The crawler's ordering metric is implemented by the function reorder queue(). We used three ordering metrics:

(1) Breadth-first
(2) Backlink count IB_(p)d
(3) PageRank IR_(p).

The breadth-first metric places URLs in the queue in the order in which they are discovered, and this policy makes the crawler visit pages in breadth-first order. We considered a page valid when its Web server responded with the HTTP header "200 OK."

Algorithm: Crawling algorithm (backlink based)

Function description:

Enqueue(queue, element): append element at the end of queue Dequeue(queue) : remove the element at the beginning

of queue and return it

Reorder queue(queue) : reorder queue using information in links

```
Input: starting url: seed URL

Procedure:
[1] enqueue(url queue, starting url)
[2] while (not empty(url queue))
[3] url = dequeue(url queue)
[4] page = crawl page(url)
[5] enqueue(crawled pages, (url, page))
[6] url list = extract urls(page)
[7] for each u in url list

[8] enqueue(links, (url, u))
[9] if (u/∈url queue and (u,-)/∈crawled pages)
[10] enqueue(url queue, u)
[11] reorder queue(url queue)

(1) breadth first
        do nothing (null operation)
(2) backlink count, IB_(p)
        for each u in URL queue,
        backlink count[u] = number of terms (-,u) in links
        sort url queue by backlink count[u]
(3) PageRank IR_(p)
        solve the following set of equations:
        IR[u] = (1 − 0.9) + 0.9_i
        IR[vi]
        ci, where
        (vi, u) ∈ links and ci is the number of links in the page vi
        sort url queue by IR(u)
```

Backlink ordering metric

In this scenario, the importance metric is the number of backlinks to a page (I(p)=IB(p)) and we consider a Crawl & Stop with Threshold model with G 3, 10, or 100. Recall that a page with G or more backlinks is considered important, i.e., hot. Under these hot page definitions, about H = 85,000 (47%), 17,500 (10%) and 1,400 (0.8%) pages out of 179,000 total Web pages were considered hot, respectively.

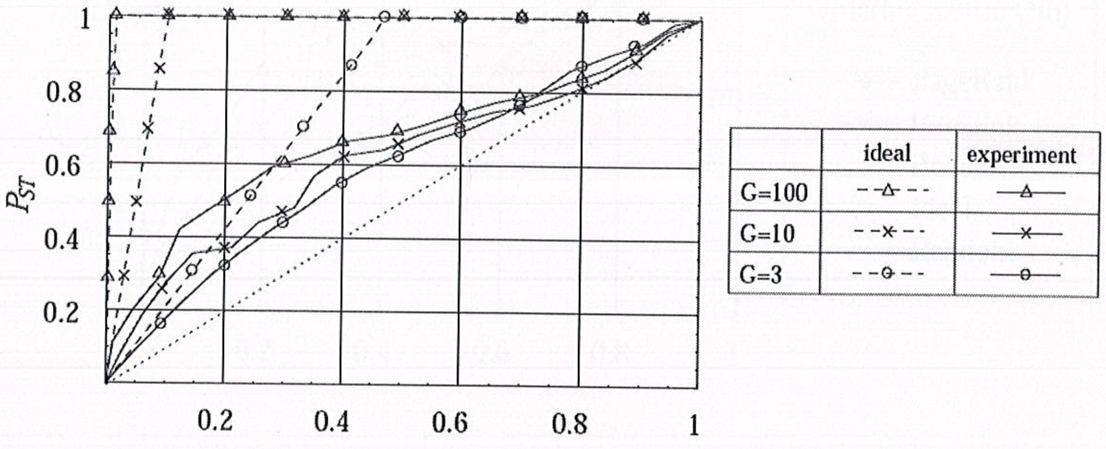


Figure 1: the horizontal axis is the fraction of the Web pages that has been crawled over time. At the right end of the horizontal axis, all 179,000 pages have been visited. The vertical axis represents PST , the fraction of the total hot pages that has been crawled at a given point. The solid lines in the figure show the results from our experiments.

For example, when the crawler in our experiment visited 0.2 (20%) of the pages, it crawled 0.5 (50%) of the total hot pages for G = 100. The dashed lines in the graph show the expected performance of ideal crawlers. An ideal crawler reaches performance 1 when H pages have been crawled. The dotted line represents the performance of a random crawler, and it increases linearly over time.

The graph shows that as our definition of a hot page becomes more stringent (larger G), the faster the crawler can locate the hot pages. This result is to be expected, since pages with many backlinks will be seen quickly after the crawl starts. Even if G is large, finding the "last" group of hot pages is always difficult. That is, to the right of the 0.8 point on the horizontal axis, the crawler finds hot pages at roughly the same point.

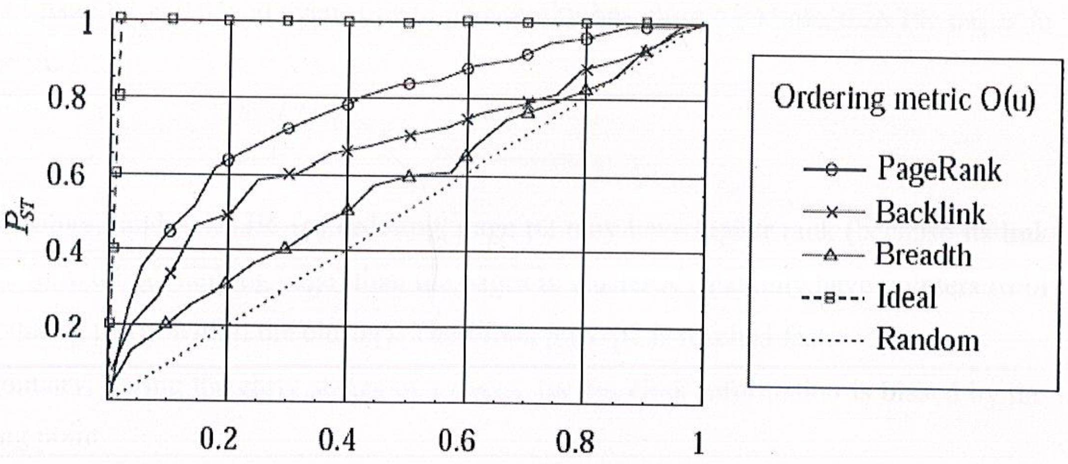


Figure 2: Fraction of Web crawled vs. PST . I(p) = IB(p); G = 100. Rate as a random crawler.

In our next experiment we compare three different ordering metrics:

1) Breadth-first
2) Backlink-count
3) PageRank

We continue to use the Crawl & Stop with Threshold model, with G = 100, and a IB(p) importance metric. Figure B shows the results of this experiment. The results are rather counterintuitive. Intuitively one would expect that a crawler using the backlink ordering metric IB_(p) that matches the importance metric IB(p) would perform the best.

However, this is not the case, and the PageRank metric IR_(p) outperforms the IB_(p) one. To understand why, we manually traced the crawler's operation. We noticed that often the IB_(p) crawler behaved like a depth-first one, frequently visiting pages in one "cluster" before moving on to the next.

On the other hand, the IR_(p) crawler combined breadth and depth in a better way. To illustrate, let us consider the Web fragment of Figure 2.5. With IB_(p) ordering, the crawler visits a page like the one labeled p1 and quickly finds a cluster A of pages that point to each other. The A pages temporarily have more backlinks than page p2, so the visit of page p2 is delayed even if page p2 actually has more backlinks than the pages in cluster A.

On the other hand, with IR_(p) ordering, page p2 may have higher rank (because its link comes from a high ranking page) than the pages in cluster A (that only have pointers from low ranking pages within the cluster). Therefore, page p2 is reached faster.

In summary, during the early stages of a crawl, the backlink information is biased by the starting point.

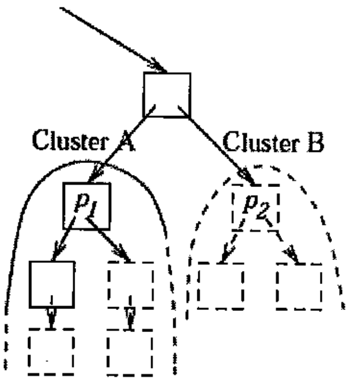


Figure 2.5: Crawling order

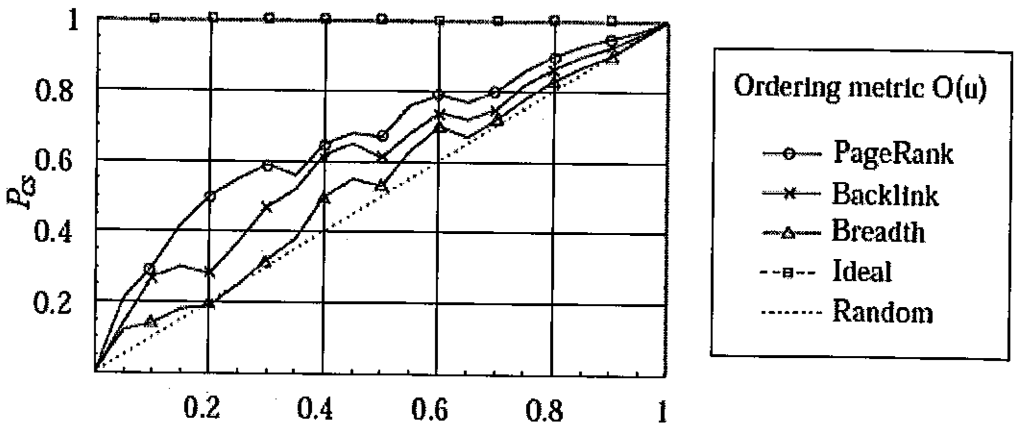


Figure 3: Fraction of Web crawled vs. PCS. I(p) = IB(p)

If the crawler bases its decisions on this skewed information, it tries getting locally hot pages instead of globally hot pages, and this bias gets worse as the crawl proceeds. On the other hand, the IR_(p) PageRank crawler is not as biased towards locally hot pages, so it gives better results regardless of the starting point.

Remember that under the Crawl & Stop model, the definition of hot pages changes over time. That is, the crawler does not have a predefined notion of hot pages, and instead, when the crawler has visited, say, 30% of the entire Web, it considers the top 30% pages as hot pages. Therefore, an ideal crawler would have performance 1 at all times because it would download pages in the order of their importance.

1) Breadth-first
2) Backlink
3) PageRank

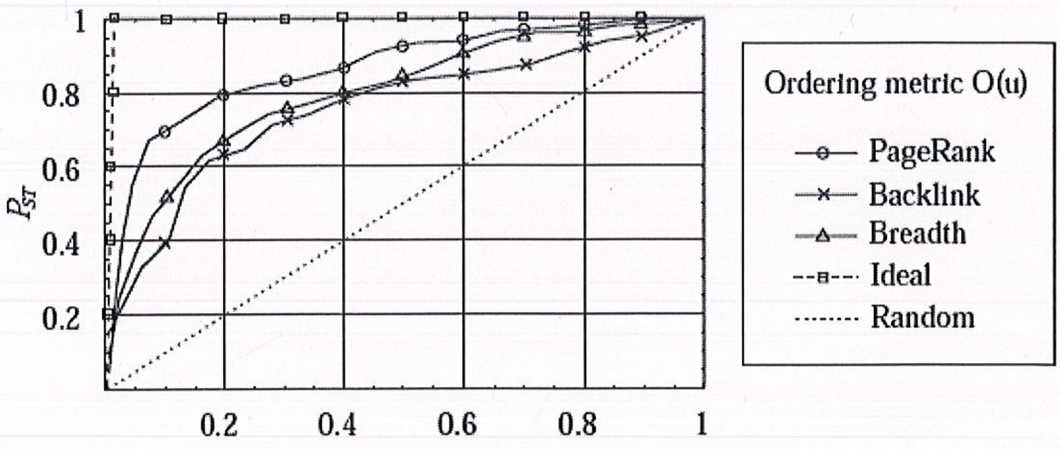


Figure 4: Fraction of Web crawled vs. PST . I(p) = IR(p); G = 13 metric under this model.

The vertical axis represents PCS, the crawled fraction of hot pages at each point under the varying definition of hot pages. The figure shows that the results of the Crawl & Stop model are analogous to those of the Crawl & Stop with Threshold model: The PageRank ordering metric shows the best performance. Returning to the Crawl & Stop with Threshold model, Figure D shows the results when we use the IR(p) PageRank importance metric with G = 13.2 Again, the PageRank ordering metric shows the best performance. The backlink and the breadth-first metrics show similar performance. Based on these results, we recommend using the PageRank ordering metric for both the IB(p) and the IR(p) importance metrics.

Chapter 5

POLITENESS POLICY

5.1 DESCRIPTION

Crawlers can retrieve data much quicker and in greater depth than human searchers, so they can have a crippling impact on the performance of a site. Needless to say, if a single crawler is performing multiple requests per second and/or downloading large files, a server would have a hard time keeping up with requests from multiple crawlers.

The use of Web crawlers is useful for a number of tasks, but comes with a price for the general community. The costs of using Web crawlers include:

- Network resources, as crawlers require considerable bandwidth and operate with a high degree of parallelism during a long period of time
- Server overload, especially if the frequency of accesses to a given server is too high
- Poorly-written crawlers, which can crash servers or routers, or which download pages they cannot handle
- Personal crawlers that, if deployed by too many users, can disrupt networks and Web servers

A partial solution to these problems is the robots exclusion protocol, also known as the robots.txt protocol that is a standard for administrators to indicate which parts of their Web servers should not be accessed by crawlers. This standard does not include a suggestion for the interval of visits to the same server, even though this interval is the most effective way of avoiding server overload. Recently commercial search engines like Ask Jeeves, MSN and Yahoo are able to use an extra "Crawl-delay:" parameter in the robots.txt file to indicate the number of seconds to delay between requests.

5.2 ROBOTS EXCLUSION PROTOCOL EXAMPLE

This Robots Exclusion Protocol has been taken from Google. Its URL is *http://www.google.com/robots.txt*

```
User-agent: *
Disallow: /search
Disallow: /groups
Disallow: /images
Disallow: /catalogs
Disallow: /catalogues
Disallow: /news
Allow: /news/directory
Disallow: /nwshp
Disallow: /setnewsprefs?
Disallow: /index.html?
Disallow: /?
Disallow: /addurl/image?
Disallow: /pagead/
Disallow: /relpage/
Disallow: /relcontent
Disallow: /imgres
Disallow: /imglanding
Disallow: /keyword/
Disallow: /u/
Disallow: /univ/
Disallow: /cobrand
Disallow: /custom
```

```
Disallow: /advanced_group_search
Disallow: /googlesite
Disallow: /preferences
Disallow: /setprefs
Disallow: /swr
Disallow: /url
Disallow: /default
Disallow: /m?
Disallow: /m/?
Disallow: /m/images?
Disallow: /m/local?
Disallow: /m/movies?
Disallow: /m/news?
Disallow: /m/news/i?
Disallow: /m/place?
Disallow: /m/products?
Disallow: /m/products/
Disallow: /m/setnewsprefs?
Disallow: /m/search?
Disallow: /m/swmloptin?
Allow: /alerts/manage
Sitemap: http://www.gstatic.com/s2/sitemaps/profiles-sitemap.xml
Sitemap: http://www.google.com/hostednews/sitemap_index.xml
Sitemap: http://www.google.com/ventures/sitemap_ventures.xml
Sitemap: http://www.google.com/sitemaps_webmasters.xml
Sitemap: http://www.gstatic.com/trends/websites/sitemaps/sitemapindex.xml
Sitemap: http://www.gstatic.com/dictionary/static/sitemaps/sitemap_index.xml
```

5.3 ROBOTS EXCLUSION PROTOCOL INTERPRETATION

The keyword, 'User-agent' states the web crawler for which the following instructions apply. A Robots Exclusion Protocol document can have different instructions for different web crawlers.

If an asterisk, *, is stated after User-agent, it implies that all web crawlers should follow the instructions given below.

The keyword, 'Disallow' states that the specified URL should not be crawled by the aforementioned crawler.

Similarly, 'Allow' states that the specified URL can be crawled by the aforementioned crawler specified in User-agent.

By reading Google's Robots file, one can surmise that it's search, images, and news sections among others should not be crawled by any web crawler.

Only a few areas such as '/news/directory' and '/alerts/manage' have been permitted to be crawled.

Conversely, sites such as http://www.apple.com allow all crawlers to access each and every area of its website.

Chapter 6

PARALLEL CRAWLERS

6.1 INTRODUCTION

Our main aim here is how we should parallelize the crawling process so that we can maximize the download rate while minimizing the overhead from parallelization. As the size of the Web grows, it becomes more difficult – or impossible – to crawl the entire Web by a single process. Many search engines run multiple processes in parallel. We refer to this type of crawler as a *parallel crawler*.

While many existing search engines already use a parallel crawler internally, there has been little scientific research conducted on the parallelization of a crawler. In particular, we believe the following issues make the study of a parallel crawler difficult, yet interesting:

• ***Overlap:*** When multiple processes run in parallel to download pages, different processes
may download the same page multiple times. One process may not be aware of the fact that another process has already downloaded a page. Clearly, multiple downloads should be minimized to save network bandwidth and increase the crawler's effectiveness. How can we coordinate the processes to prevent overlap?

• ***Quality:*** A crawler wants to download "important" pages first. However, in a parallel crawler, each process may not be aware of the whole image of the Web that they have collectively downloaded so far. For this reason, each process may make a crawling decision solely based on *its own* image of the Web (thatit has downloaded) and thus make a poor decision. How can we make sure that the *quality* of downloaded pages is as good for a parallel crawler as for a single-process crawler?

• ***Communication bandwidth:*** In order to prevent overlap, or to improve the quality of the downloaded pages, crawling processes need to periodically communicate and coordinate with each other. However, this communication may grow significantly as the number of crawling processes increases. Exactly what do the processes need to communicate and how significant would the communication overhead be? Can we minimize this communication while maintaining the effectiveness of the crawler?

While parallel crawlers are challenging to operate, we believe that they have many important advantages, compared to single-process crawlers:

• ***Scalability:*** Due to the enormous size of the Web, it may be imperative to run a parallel crawler to achieve the required download rate in certain cases.

• ***Network-load dispersion:*** Multiple crawling processes of a parallel crawler may run at geographically distant locations, each downloading "geographically-adjacent" pages. For example, a process in Germany may download all European pages, while another one in Japan crawls all Asian pages. In this way, we can *disperse* the network load to multiple regions. In particular, this dispersion might be necessary when a single network cannot handle the heavy load from a large-scale crawl.

• ***Network-load reduction:*** In addition to dispersing load, a parallel crawler may actually *reduce* the network load. For example, assume that a crawler in North America retrieves a page from Europe. To be downloaded by the crawler, the page first has to go through the network in Europe, then the Europe-to-North America inter-continental network and finally the network in North America. Instead, if a crawling process in Europe collects all European pages, and if another process in North America crawls all North American pages, the overall network load will be reduced, because pages only go through "local" networks.

Note that the downloaded pages may need to be transferred later to a central location, so that a central index can be built.

However, even in that case, we believe that the transfer can be made significantly smaller than the original page download traffic, by using some of the following methods:

– ***Compression:*** Once the pages are collected and stored, it is easy to *compress* the data before sending them to a central location.

– ***Difference:*** Instead of sending the entire image of all downloaded pages, we may first take the *difference* between the previous image and the current one and send only this difference. Since many pages are static and do not change very often, this scheme may significantly reduce network traffic.

– ***Summarization:*** In certain cases, we may need only a central index, not the original pages themselves. In this case, we may extract the necessary information for an index construction (e.g., postings lists) and transfer this information only.

In summary, we believe a parallel crawler has many advantages and poses interesting challenges. Here we makes the following contributions:
• We identify major issues and problems related to a parallel crawler and discuss how we can solve these problems.

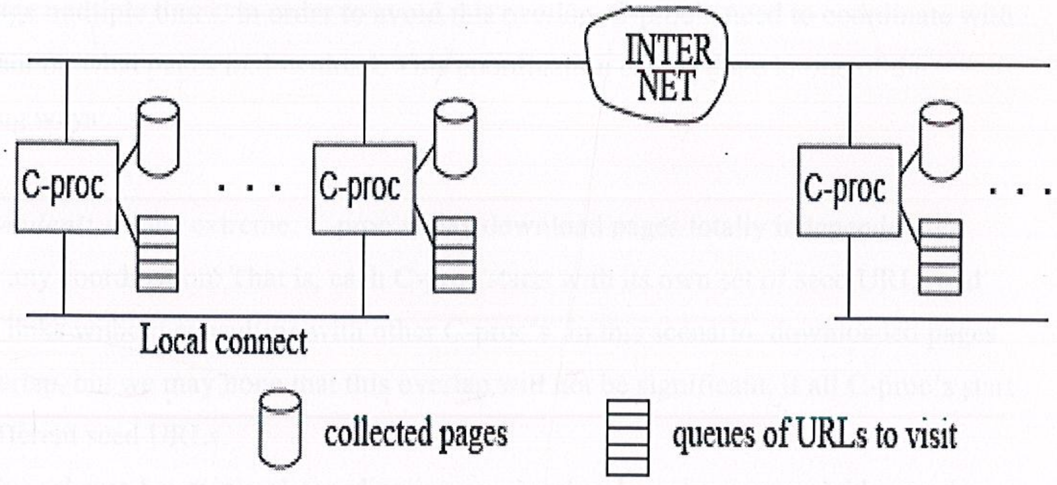
• We present multiple architectures and techniques for a parallel crawler and discuss their advantages and disadvantages. As far as we know, most of these techniques have not been described in open literature.

• Using a large dataset (40M web pages) collected from the Web, we experimentally compare the design choices and study their tradeoffs quantitatively.

• We propose various optimization techniques that can minimize the coordination effort among crawling processes so that they can operate more independently while maximizing their effectiveness.

6.2 ARCHITECTURE OF A PARALLEL CRAWLER

In the following figure we illustrate the general architecture of a 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 them 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 *intrasite parallel crawler*. This scenario corresponds to the case where all C-proc's run only on the local network on the left. In this case, all C-proc's use 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 a crawler's 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. As we discussed in the introduction, a distributed crawler can *disperse* and even *reduce* the load on the overall network. When C-proc's run at distant locations and communicate 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.

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. This coordination can be done in one of the following ways:

• ***Independent:*** At one extreme, C-proc's may download pages totally independently, without any coordination. That is, each C-proc starts with its own set of seed URLs and follows links without consulting with other C-proc's. In this scenario, downloaded pages may overlap, but we may hope that this overlap will not be significant, if all C-proc's start from different seed URLs.
While this scheme has minimal coordination overhead and can be very scalable, we do not directly study this option due to its overlap problem.

• ***Dynamic assignment:*** When there exists a central coordinator that logically divides the Web into small partitions (using a certain partitioning function) and dynamically assigns each partition to a C-proc for download, we call it *dynamic assignment*.

• ***Static assignment:*** When the Web is partitioned and assigned to each C-proc *before* the start of a crawl, we call it *static assignment*. In this case, every C-proc knows which C-proc is responsible for which page during a crawl, and the crawler does not need a central coordinator.

In this thesis, we mainly focus on static assignment and defer the study of dynamic assignment to future work. Note that in dynamic assignment, the central coordinator may become a major bottleneck because it has to maintain a large number of URLs reported from all C-proc's and has to constantly coordinate all C-proc's.

So far, we have mainly assumed that the Web pages are partitioned by Web sites. Clearly, there exist a multitude of ways to partition the Web, including the following:

1. **URL-hash based:** Based on the hash value of the URL of a page, we assign the page to a C-proc. In this scheme, pages in the same site can be assigned to different C-proc's. Therefore, the locality of link structure2 is not reflected in the partition, and there will be many inter-partition links.

2. **Site-hash based:** Instead of computing the hash value on the entire URL, we compute the hash value only on the *site name* of a URL (e.g., cnn.com in http: //cnn.com/index.html) and assign the page to a C-proc.

In this scheme, note that pages in the same site will be allocated to the *same* partition. Therefore, only some of *inter-site* links will be *inter-partition* links, and thus we can reduce the number of inter-partition links quite significantly compared to the URLhash based scheme.

3. **Hierarchical:** Instead of using a hash-value, we may partition the Web hierarchically based on the URLs of pages. For example, we may divide theWeb into three partitions (the pages in the .com domain, .net domain and all other pages) and allocate them to three C-proc's.

Note that this scheme may have even fewer inter-partition links than the site-hash based scheme, because pages may tend to be linked to the pages in the same domain.

In our later experiments, we will mainly use the site-hash based scheme as our partitioning function. We chose this option because it is simple to implement and because it captures the core issues that we want to study. For example, under the hierarchical scheme, it is not easy to divide the Web into equal size partitions, while it is relatively straightforward under the site-hash based scheme. Also, the URL-hash based scheme generates many inter-partition links, resulting in more URL exchanges in the exchange mode and less coverage in the firewall mode.

6.3 Evaluation models

In this section, we define the metrics that will let us quantify the advantages or disadvantages of different parallel crawling schemes. These metrics will be used later in our experiments.

1.Overlap:

When multiple C-proc's are downloading Web pages simultaneously, it is possible that different C-proc's will download the same page multiple times. Multiple downloads of the same page are clearly undesirable.More precisely, we define the *overlap of downloaded pages as* $N-I/I$ *. Here, N represents* the *total number of pages downloaded by the overall crawler, and I represents the* number of *unique pages downloaded, again, by the overall crawler. Thus, the goal of* a parallel crawler is to minimize the overlap.

2. Coverage:

When multiple C-proc's run independently, it is possible that they may not download all pages that they have to. In particular, a crawler based on the firewall mode may have this problem because its C-proc's do not follow inter-partition links or exchange the links with others.

To formalize this notion, we define the *coverage of downloaded pages as* I/U*, where U* represents the total number of pages that the overall crawler has to download and *I* is the number of unique pages downloaded by the overall crawler.

3.Quality:

Crawlers often cannot download the whole Web, and thus they try to download an "important" or "relevant" section of the Web.However, note that the quality of a crawler may vary depending on how often it exchanges backlink messages. For instance, if C-proc's exchange backlink messages after every page download, they will have essentially the same backlink information as a single-process crawler does).

Therefore, the quality of the downloaded pages would be virtually the same as that of a single-process crawler. In contrast, if C-proc's rarely exchange backlink messages, they do not have "accurate" backlink counts from the downloaded pages, so they may make poor crawling decisions, resulting in poor quality.

4. **Communication overhead:**

The C-proc's in a parallel crawler need to exchange messages to coordinate their work.In articular, C-proc's based on the exchange mode swap their inter-partition URLs periodically. To quantify how much communication is required for this exchange, we define *communication overhead as the average number of inter-partition URLs exchanged per downloaded* page.

For example, if a parallel crawler has downloaded 1,000 pages in total and if its C-proc's have exchanged 3,000 inter-partition URLs, its communication overhead is **3, *000/1, 000*** = *3.*

***Note that crawlers based on the firewall* do not have any communication overhead because they do not exchange any inter-partition URLs.**

Chapter 7

CRAWLER ACRCHITECTURE

7.1 DESCRIPTION

Each layer is oblivious to the implementation details of other layers; this makes it easy to replace one implementation of a layer with another, as long as their interface is kept the same. In a crawling loop a URL is picked from the frontier (a list of unvisited URLs), the page corresponding to the URL is fetched from the Web, the fetched page is processed through all the infrastructure layers, and the unvisited URLs from the page are added to the frontier. The networking layer is primarily responsible for fetching and storing a page and making the process transparent to the layers above it. The parsing and extraction layer parses the page and extracts the needed data such as hyperlink URLs and their contexts. The extracted data is then represented in a formal notation by the representation layer before being passed onto the intelligence layer that associates priorities or scores with unvisited URLs in the page. We implement the high level layered design as multithreaded objects in Java. Each thread implements a sequential crawling loop that includes the layers shown in above figure. All of the threads share a single synchronized frontier.

In terms of the high-level design the change in classifiers amounts to changing only the intelligence layer while all of the other layers remain constant. A topical crawler needs to process and represent Web content, apply intelligence and maintain a frontier of URLs based on their priorities. All of these tasks require some amount of memory-resident data structures that are not necessary for a generic crawler. This additional memory consumption by a topical crawler may affect its scalability when compared with a generic crawler.

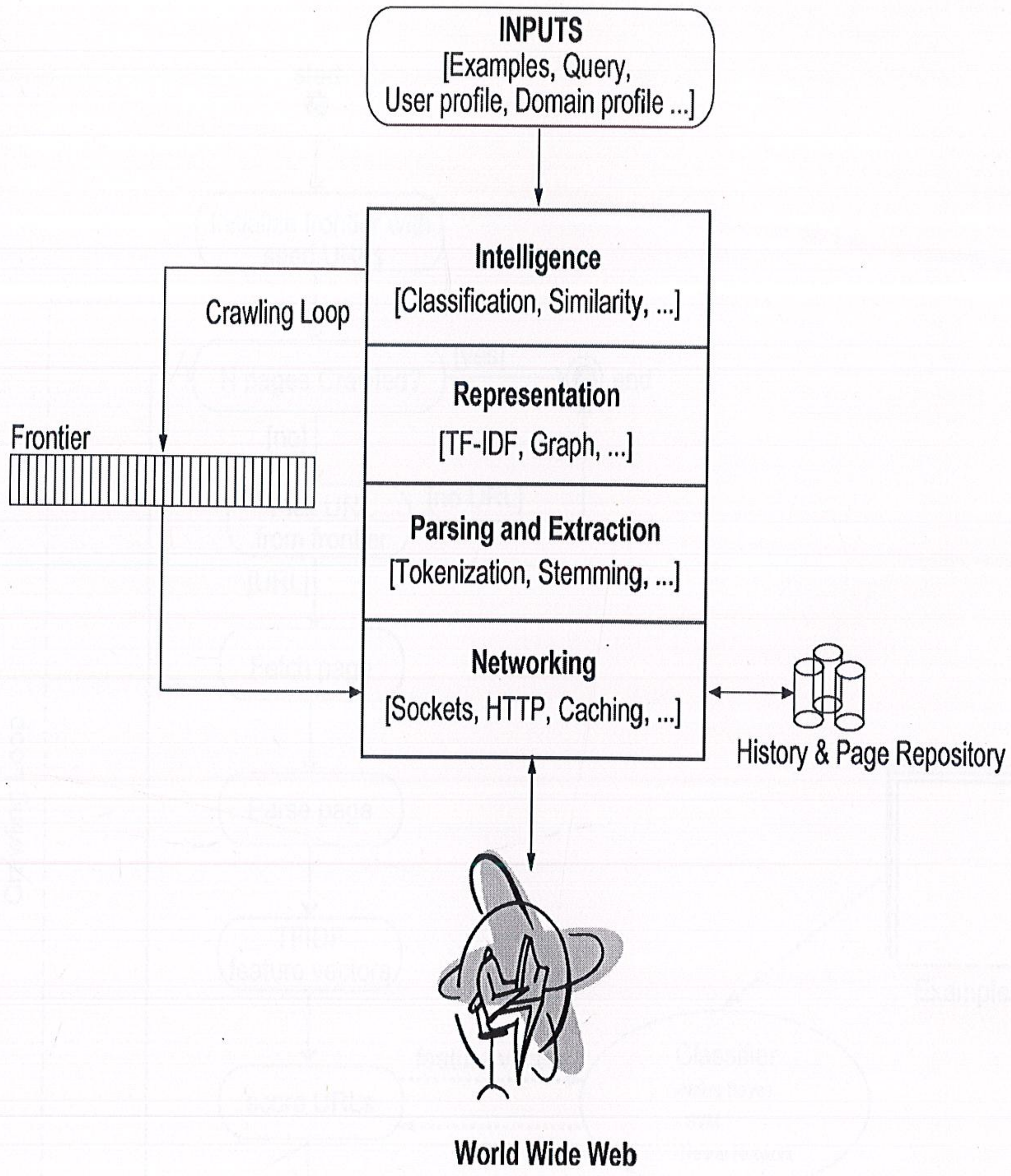

INPUTS
[Examples, Query,
User profile, Domain profile ...]
Intelligence
[Classification, Similarity, ...]
Crawling Loop
Representation
[TF-IDF, Graph, ...]
Frontier
Parsing and Extraction
[Tokenization, Stemming, ...]
Networking
[Sockets, HTTP, Caching, ...]
History & Page Repository
World Wide Web

Figure 5: high-level layered design for Web crawling infrastructure.

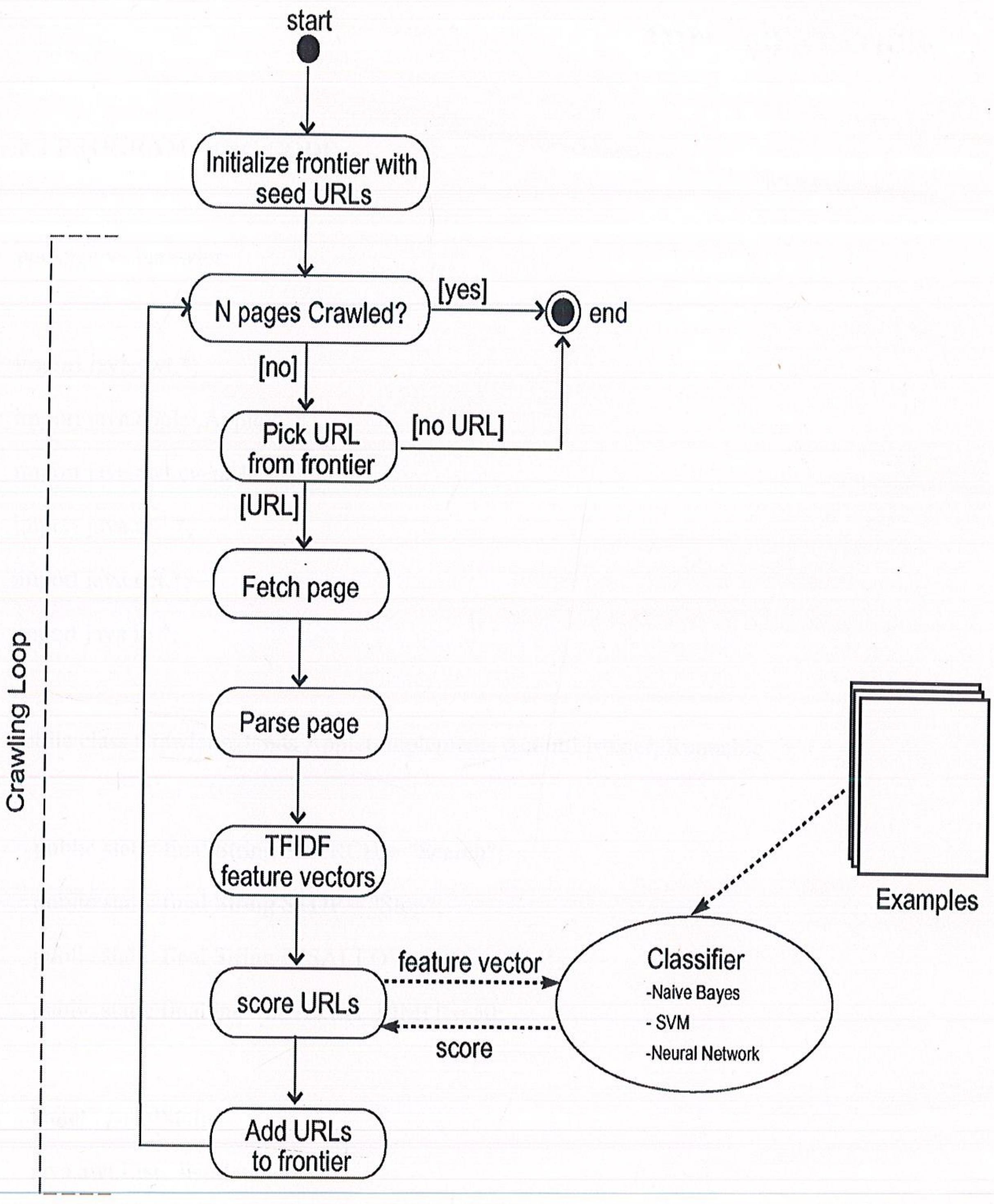


Figure 6: When a classifier is used to guide a topical crawler, each thread in implementation follows the steps shown in the figure

Chapter 8

IMPLEMENTATION

8.1 PROGRAMMING CODE

```
package webcrawler;


import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;


public class Crawler extends Applet implements ActionListener, Runnable
{
  public static final String SEARCH = "Search";
  public static final String STOP = "Stop";
  public static final String DISALLOW = "Disallow:";
  public static final int    SEARCH_LIMIT = 50;


  Panel   panelMain;
  java.awt.List   listMatches;
  Label   labelStatus;


  Vector vectorToSearch;
  Vector vectorSearched;
Vector vectorMatches;

Thread searchThread;


TextField textURL;

Choice   choiceType;
```

// Graphics User Interface of the program is implemented in here. Also, the data structures used to store URLs are initialized here

```
public void init()

{

    panelMain = new Panel();

    panelMain.setLayout(new BorderLayout(5, 5));


    Panel panelEntry = new Panel();

    panelEntry.setLayout(new BorderLayout(5, 5));


    Panel panelURL = new Panel();

    panelURL.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));

    Label labelURL = new Label("Starting URL: ", Label.RIGHT);

    panelURL.add(labelURL);

    textURL = new TextField("", 40);

    panelURL.add(textURL);

    panelEntry.add("North", panelURL);


    Panel panelType = new Panel();
```

```
panelType.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));

Label labelType = new Label("Content type: ", Label.RIGHT);
panelType.add(labelType);
choiceType = new Choice();
choiceType.addItem("text/html");
choiceType.addItem("audio/basic");
choiceType.addItem("audio/au");
choiceType.addItem("audio/aiff");
choiceType.addItem("audio/wav");
choiceType.addItem("video/mpeg");
choiceType.addItem("video/x-avi");
panelType.add(choiceType);
panelEntry.add("South", panelType);

panelMain.add("North", panelEntry);


Panel panelListButtons = new Panel();
panelListButtons.setLayout(new BorderLayout(5, 5));

Panel panelList = new Panel();
panelList.setLayout(new BorderLayout(5, 5));
Label labelResults = new Label("Search results");
panelList.add("North", labelResults);
Panel panelListCurrent = new Panel();
panelListCurrent.setLayout(new BorderLayout(5, 5));
listMatches = new java.awt.List(10);
```

```
        panelListCurrent.add("North", listMatches);
        labelStatus = new Label("");

        panelListCurrent.add("South", labelStatus);
        panelList.add("South", panelListCurrent);

    panelListButtons.add("North", panelList);

        Panel panelButtons = new Panel();
        Button buttonSearch = new Button(SEARCH);
        buttonSearch.addActionListener(this);
        panelButtons.add(buttonSearch);
        Button buttonStop = new Button(STOP);
        buttonStop.addActionListener(this);
        panelButtons.add(buttonStop);
        panelListButtons.add("South", panelButtons);

        panelMain.add("South", panelListButtons);

        add(panelMain);
        setVisible(true);

        repaint();

        vectorToSearch = new Vector();
        vectorSearched = new Vector();
        vectorMatches = new Vector();
        URLConnection.setDefaultAllowUserInteraction(false);
    }

    public void start()
    {

    }

    public void stop()
    {
        if (searchThread != null)
        {
                setStatus("stopping...");
                searchThread = null;
        }
    }

    public void destroy()
    {

    }

// Robots Exclusion Protocol is implemented in this module

    boolean robotSafe(URL url)
    {
```

```
String strHost = url.getHost();

String strRobot = "http://" + strHost + "/robots.txt";
URL urlRobot;

try
{
urlRobot = new URL(strRobot);
} catch (MalformedURLException e)
{
                return false;
        }

String strCommands;
try
{
InputStream urlRobotStream = urlRobot.openStream();
byte b[] = new byte[1000];
int numRead = urlRobotStream.read(b);
  strCommands = new String(b, 0, numRead);
  while (numRead != -1)
  {
        if (Thread.currentThread() != searchThread)
                break;
        numRead = urlRobotStream.read(b);
        if (numRead != -1)
        {
```

```
                    String newCommands = new String(b, 0, numRead);
                    strCommands += newCommands;
                }
            }
            urlRobotStream.close();

        } catch (IOException e)
          {
            return true;
          }


         String strURL = url.getFile();
         int index = 0;
         while ((index = strCommands.indexOf(DISALLOW, index)) != -1)
         {
         index += DISALLOW.length();
         String strPath = strCommands.substring(index);
         StringTokenizer st = new StringTokenizer(strPath);

         if (!st.hasMoreTokens())
                 break;

         String strBadPath = st.nextToken();


         if (strURL.indexOf(strBadPath) == 0)
                 return false;
     }
```

```
        return true;
    }

    public void paint(Graphics g)
    {
        g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);

        panelMain.paint(g);
        panelMain.paintComponents(g);
    }

// All back end processing is done in this module

    public void run()
    {
        String strURL = textURL.getText();
        String strTargetType = choiceType.getSelectedItem();
        int numberSearched = 0;
        int numberFound = 0;
        if (strURL.length() == 0)
        {
            setStatus("ERROR: must enter a starting URL");
            return;
        }

        vectorToSearch.removeAllElements();
        vectorSearched.removeAllElements();
        vectorMatches.removeAllElements();
        listMatches.removeAll();
        vectorToSearch.addElement(strURL);

        while    ((vectorToSearch.size()    >    0)    &&    (Thread.currentThread()    ==
searchThread))
        {
                strURL = (String) vectorToSearch.elementAt(0);

                setStatus("searching " + strURL);

                URL url;
                try
                {
                        url = new URL(strURL);
                } catch (MalformedURLException e)
                        {
                                setStatus("ERROR: invalid URL " + strURL);
                                break;
                        }

                vectorToSearch.removeElementAt(0);
                vectorSearched.addElement(strURL);

                  if (url.getProtocol().compareTo("http") != 0)
                        break;
```

```
  if (!robotSafe(url))
    break;

try
{
  URLConnection urlConnection = url.openConnection();

  urlConnection.setAllowUserInteraction(false);



  InputStream urlStream = url.openStream();
  String type = urlConnection.guessContentTypeFromStream(urlStream);
  if (type == null)
    break;
  if (type.compareTo("text/html") != 0)
    break;

  byte b[] = new byte[1000];
  int numRead = urlStream.read(b);
  String content = new String(b, 0, numRead);
  while (numRead != -1)
  {
    if (Thread.currentThread() != searchThread)
      break;
    numRead = urlStream.read(b);
    if (numRead != -1)
    {
            String newContent = new String(b, 0, numRead);
            content += newContent;
        }
    }
    urlStream.close();

    if (Thread.currentThread() != searchThread)
        break;

    String lowerCaseContent = content.toLowerCase();

    int index = 0;
    while ((index = lowerCaseContent.indexOf("<a", index)) != -1)
    {
        if ((index = lowerCaseContent.indexOf("href", index)) == -1)
            break;
        if ((index = lowerCaseContent.indexOf("=", index)) == -1)
            break;

        if (Thread.currentThread() != searchThread)
            break;

        index++;
        String remaining = content.substring(index);

        StringTokenizer st = new StringTokenizer(remaining, "\t\n\r\">#");
        String strLink = st.nextToken();
```

```
URL urlLink;
try
{
  urlLink = new URL(url, strLink);
  strLink = urlLink.toString();
} catch (MalformedURLException e)
  {
    setStatus("ERROR: bad URL " + strLink);
    continue;
  }

if (urlLink.getProtocol().compareTo("http") != 0)
  break;

if (Thread.currentThread() != searchThread)
  break;

try
{
  URLConnection urlLinkConnection = urlLink.openConnection();
  urlLinkConnection.setAllowUserInteraction(false);
  InputStream linkStream = urlLink.openStream();

  String                    strType                    =
urlLinkConnection.guessContentTypeFromStream(linkStream);
  linkStream.close();
```

```
            if (strType == null)
                break;
            if (strType.compareTo("text/html") == 0)
            {
                if              ((!vectorSearched.contains(strLink))              &&
(!vectorToSearch.contains(strLink)))
                {
                    if (robotSafe(urlLink))
                        vectorToSearch.addElement(strLink);
                }
            }


            if (strType.compareTo(strTargetType) == 0)
            {
                if (vectorMatches.contains(strLink) == false)
                {
                    listMatches.add(strLink);
                    vectorMatches.addElement(strLink);
                    numberFound++;
                    if (numberFound >= SEARCH_LIMIT)
                        break;
                }
            }
        } catch (IOException e)
            {
                setStatus("ERROR: couldn't open URL " + strLink);
                continue;
              }
          }
      } catch (IOException e)
          {
            setStatus("ERROR: couldn't open URL " + strURL);
            break;
          }

      numberSearched++;
      if (numberSearched >= SEARCH_LIMIT)
        break;
  }

    if (numberSearched >= SEARCH_LIMIT || numberFound >= SEARCH_LIMIT)
    setStatus("reached search limit of " + SEARCH_LIMIT);
    else
      setStatus("done");
    searchThread = null;
}

void setStatus(String status)
{
    labelStatus.setText(status);
}
```

```
    public void actionPerformed(ActionEvent event)
    {
        String command = event.getActionCommand();

        if (command.compareTo(SEARCH) == 0)
      {
          setStatus("searching...");

          if (searchThread == null)
        {
              searchThread = new Thread(this);
          }
          searchThread.start();
        }

        else if (command.compareTo(STOP) == 0)
      {
          stop();
        }
    }

// Main module of application

    public static void main (String argv[])
    {
      Frame f = new Frame("SSN WebSpider 1.0b");
```

```
        Crawler applet = new Crawler();
          f.add("Center", applet);

        Properties props= new Properties(System.getProperties());
        props.put("http.proxySet", "true");
        props.put("http.proxyHost", "172.16.73.12");
        props.put("http.proxyPort", "3128");

        Properties newprops = new Properties(props);
        System.setProperties(newprops);
        applet.init();
        applet.start();
        f.pack();
        f.show();
    }
}
```

8.2 INDIVIDUAL MODULES

1. **Main**

- This is the primary module of the application. A frame with the name of our web crawler, "SSN WebSpider 1.0b" is created.
- Within this frame, the Crawler applet is integrated.
- Since the Crawler is accessing the World Wide Web through the juitnameserver, proxy settings have to be given. This is done by changing the System Propertiesof the application.
- Finally, the Web crawler applet is initiated.

2. **Init**

- In this module, the Graphics User Interface of the applet is created.
- The GUI is divided into multiple panels. Each panel has a specific function.
- E.g. In the panel, PanelURL, a label is added stating "Starting URL:" and a text field is provided to enter a Uniform Resource Locater.
- Buttons are also created, equipped with Action Listeners to link commands with the buttons.
- Finally three data structures of Vector data type are initialised which will be used to store the URLs that have been searched, that are scheduled to being searched and those that match with the URLs already used.

3. **Run**

- The initial URL that is fed by the user is stored and is the first URL to be downloaded.
- Once an URL has been searched it is removed from the 'vectorToSearch' data structure and added to data structure 'vectorSearched'. The latter data

structure is used to verify that a present URL being downloaded has not already been crawled before.

- Since this application is used only for HTTP protocols, any other protocol is rejected.
- From the run module, the RobotSafe module is executed to verify whether the particular URL has been allowed to be crawlable. If the result is false, the URL is rejected.
- Once an URL has not been rejected, it's source code is read and the <a href> tag is searched. If such a tag is found, the hyperlink is verified and added to the data structure, "vecotToSearch".
- A status message is displayed at all times. Normally, the string "Searching" with the URL being searched is output. If any error is encountered such as the URL being invalid or the page being inaccessible, an Error message is displayed. Even if an error has occurred, the Web Crawler does not stop processing further URLs.
- Finally, if no URLs remain to be searched, the Web Crawler exits from this module and the back-end processing of the program comes to an end. In a realistic scenario, such an event does not take place.

4. RobotSafe

- This module first verifies if the host URL has a '/robots.txt' file. If that is not the case, the module returns a true value, allowing the Run method to crawl it.
- If the URL has been mentioned in the DISSALLOW section of the Robots.txt file, a false value is returned preventing the Run Method from crawling it.

In this implementation, the selection policy has been utilized in the Run module.

The Politeness policy has been used in the RobotSafe module by verifying the Robots Exclusion Protocol of the URL.

8.3 FLOW CHART OF CRAWLING PROCESS

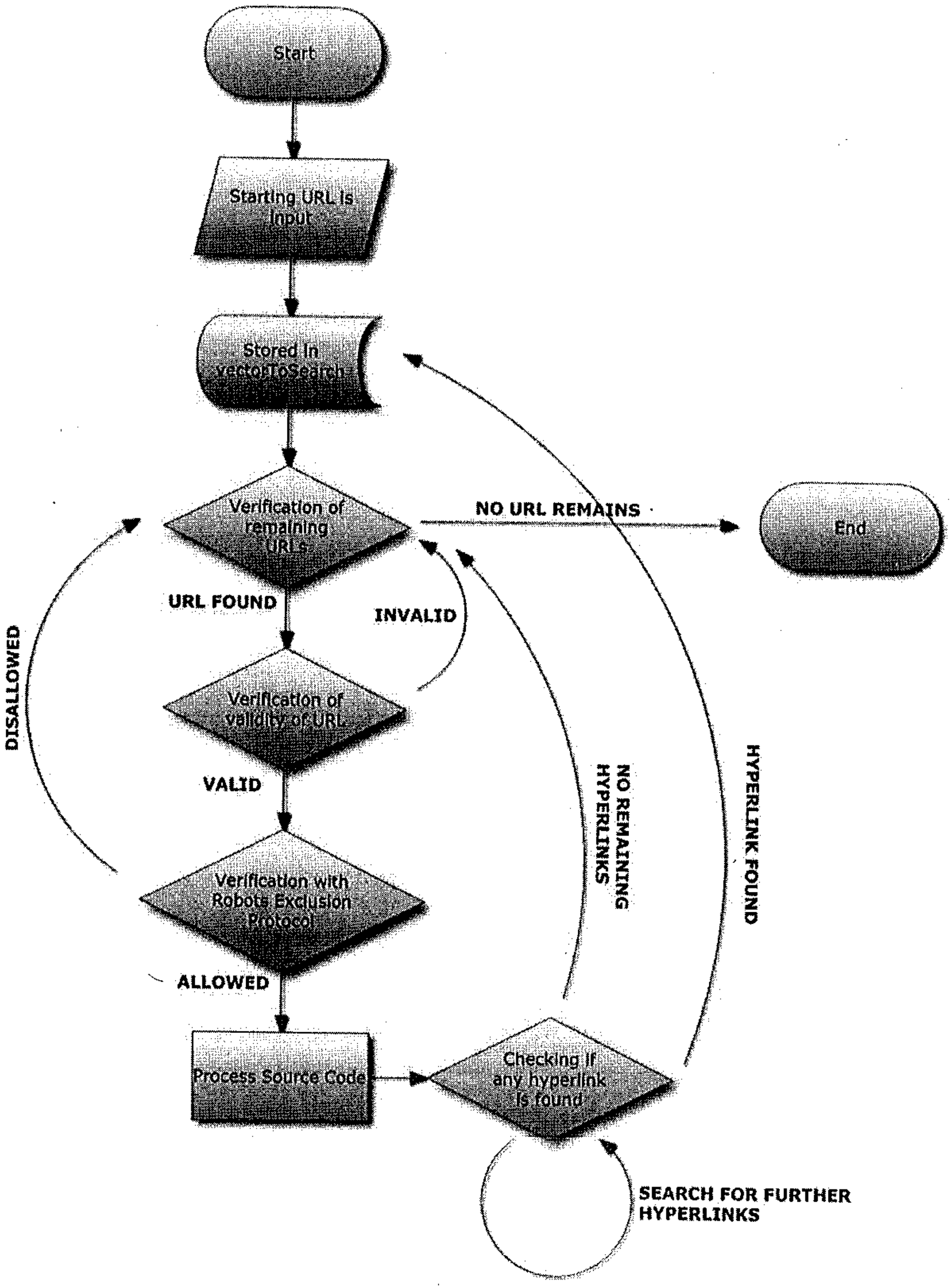
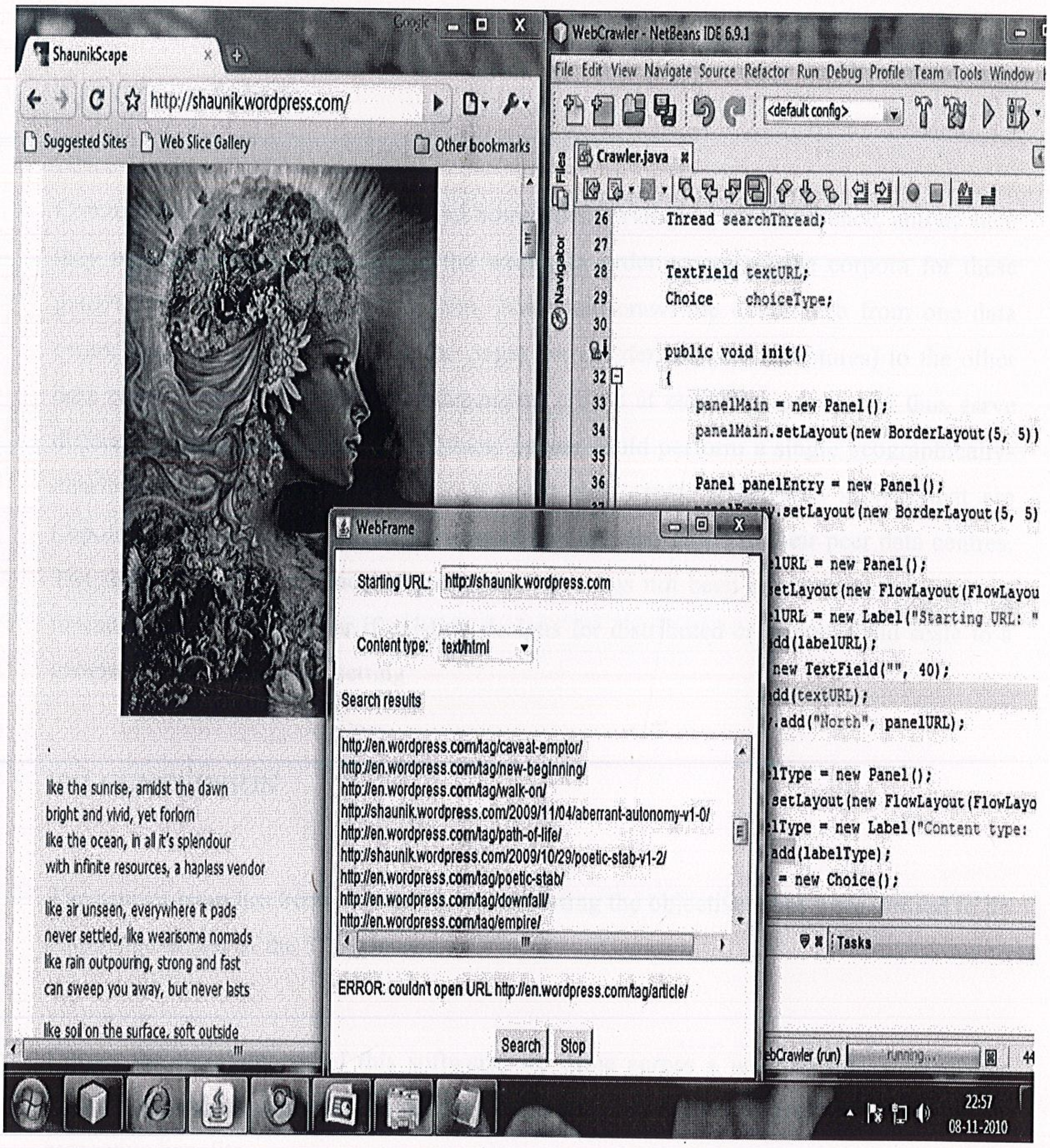

Start
Starting URL is input
Stored in vectorToSearch
Verification of remaining URLs
NO URL REMAINS
End
URL FOUND
INVALID
Verification of validity of URL
DISALLOWED
VALID
Verification with Robots Exclusion Protocol
NO REMAINING HYPERLINKS
HYPERLINK FOUND
ALLOWED
Process Source Code
Checking if any hyperlink is found
SEARCH FOR FURTHER HYPERLINKS

Figure 7: Flowchart depicting flow of events

8.4 CRAWLER AT WORK



like the sunrise, amidst the dawn
bright and vivid, yet forlorn
like the ocean, in all it's splendour
with infinite resources, a hapless vendor

like air unseen, everywhere it pads
never settled, like wearisome nomads
like rain outpouring, strong and fast
can sweep you away, but never lasts

like soil on the surface, soft outside

```
Thread searchThread;

TextField textURL;
Choice    choiceType;

public void init()
{
    panelMain = new Panel();
    panelMain.setLayout(new BorderLayout(5, 5))

    Panel panelEntry = new Panel();
    panelEntry.setLayout(new BorderLayout(5, 5)
    lURL = new Panel();
    etLayout(new FlowLayout(FlowLayou
    lURL = new Label("Starting URL: "
    dd(labelURL);
    new TextField("", 40);
    dd(textURL);
    .add("North", panelURL);

    lType = new Panel();
    setLayout(new FlowLayout(FlowLayo
    lType = new Label("Content type:
    add(labelType);
    e = new Choice();
```

WebFrame

Starting URL: http://shaunik.wordpress.com

Content type: text/html

Search results

```
http://en.wordpress.com/tag/caveat-emptor/
http://en.wordpress.com/tag/new-beginning/
http://en.wordpress.com/tag/walk-on/
http://shaunik.wordpress.com/2009/11/04/aberrant-autonomy-v1-0/
http://en.wordpress.com/tag/path-of-life/
http://shaunik.wordpress.com/2009/10/29/poetic-stab-v1-2/
http://en.wordpress.com/tag/poetic-stab/
http://en.wordpress.com/tag/downfall/
http://en.wordpress.com/tag/empire/
```

ERROR: couldn't open URL http://en.wordpress.com/tag/article/

Search | Stop

*Figure 8: An initial URL has been **input** in the Web Crawler. The content type to be processed has been selected as text/**html**. The Search Results display all URLs that have been further crawled. If there is any **error**, the user is notified*

Chapter 9
CONCLUSION

9.1 FUTURE WORK

Commercial search engines are global companies serving a global audience, and as such they maintain data centres around the world. In order to collect the corpora for these geographically distributed data centres, one could crawl the entire web from one data centre and then replicate the crawled pages (or the derived data structures) to the other data centres; one could perform independent crawls at each data centre and thus serve different indices to different geographies; or one could perform a single geographically-distributed crawl, where crawlers in a given data centre crawl web servers that are (topologically) close-by, and then propagate the crawled pages to their peer data centres. The third solution is the most elegant one, but it has not been explored in the research literature, and it is not clear if existing designs for distributed crawlers would scale to a geographically distributed setting

9.2 CONCLUSION

The project team has been successful in completing the objectives that were desired to be completed at end of the 8th Semester.

During the development of this software, we came across a wide variety of available techniques and implementations in the internet and also learnt a great deal about their respective benefits.

We also came across several hurdles while coding the software but realized that sustained efforts always pay off in the end.

APPENDIX

Appendix A

BIBLIOGRAPHY

1. Research Paper 1
   - Crawling the Web: Discovery and Maintenance of Large Scale Web data
   - Author: Junghoo Cho
   - Dissertation for PHD, Stanford University, November 2001

2. Research Paper 2
   - Learning to Crawl: Comparing Classification Schemes
   - Authors: Gautam Pant, Padmini Srinivasan
   - The University of Utah & the University of Iowa

3. Research Paper 3
   - Focused Crawling: A New Approach to Topic-specific Web Resource Discovery
   - Authors: Soumen Chakrabarti, Martin van den Berg, Byron Do
   - IIT Bombay, FX Palo Alto Laboratory, IBM Almaden Research Centre respectively,1999

4. Wikipedia References
   - http://en.wikipedia.org/wiki/Web_crawler
   - http://en.wikipedia.org/wiki/Focused_crawler

Appendix B

INDEX