Web-Crawling Approaches in Search Engines

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

Thapar University, Patiala

By:
Sandeep Sharma
(80632022)

Under the supervision of:

Mr. Ravinder Kumar
Lecturer, CSED

JUNE 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis report entitled, **"Web Crawling Approaches in Search Engines"**, submitted by me in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering at Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ravinder Kumar* and refers other researcher's works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(*Sandeep Sharma*)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ravinder Kumar)
Lecturer
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. SEEMA BAWA)
Professor & Head
Computer Sc. & Engg. Department,
Thapar University, Patiala.

(Dr. R.K.SHARMA)
Dean(Academic Affairs)
Thapar University,
Patiala

Acknowledgment

I express my sincere and deep gratitude to my guide Mr. Ravinder Kumar, Lecturer, Computer Science & Engineering Department, Thapar University Patiala, for the invaluable guidance, support and encouragement. He provided me all resource and guidance throughout thesis work.

I am heartfelt thankful to Dr. (Mrs.) Seema Bawa, Head of Computer Science & Engineering department Thapar University Patiala, for providing us adequate environment, facility for carrying out thesis work.

I would like to thank to all staff members who were always there at the need of hour and provided with all the help and facilities, which I required for the completion of my thesis.

I would also like to express my appreciation to my good friends Mohit, Arun, Neeraj and Rajeev for the timely motivation and providing interesting work environment. It was great pleasure in working with them during this thesis work.

At last but not the least I would like to thank God and mine parents for not letting me down at the time of crisis and showing me the silver lining in the dark clouds.

Sandeep Sharma

(80632022)

.

Abstract

The number of web pages is increasing into millions and trillions around the world. To make searching much easier for users, web search engines came into existence. Web Search engines are used to find specific information on the World Wide Web. Without search engines, it would be almost impossible for us to locate anything on the Web unless or until we know a specific URL address. Every search engine maintains a central repository or databases of HTML documents in indexed form. Whenever a user query comes, searching is performed within that database of indexed web pages. The size of repository of every search engine can't accommodate each and every page available on the WWW. So it is desired that only the most relevant pages are stored in the database so as to increase the efficiency of search engines. To store most relevant pages from the World Wide Web, a suitable and better approach has to be followed by the search engines. This database of HTML documents is maintained by special software .The software that traverses web for capturing pages is called "***Crawlers" or "Spiders".***

In this thesis, we discuss the basics of crawlers and the commonly used techniques of crawling the web. We discuss the pseudo code of basic crawling algorithms, their implementation in C language along with simplified flowcharts.

In this work, firstly we describe how search engine works along with implementation of various crawling algorithms into programs using C language and then the implementation results of various crawling algorithms have been discussed and a comparison study is given in a table in last.

Keywords: Web Search Engines, Crawlers, Seed URL**,** Frontier**,** Queue, Priority Queue Data structures.

Table of Contents

List of Figures

Chapter 1
Introduction

According to Internet World Stats survey [1], as on March 31, 2008, 1.407 billion people use the Internet. The vast expansion of the internet is getting more and more day by day.
The World Wide Web (commonly termed as the Web) is a system of interlinked hypertext documents accessed via the Internet. With a Web browser, a user views Web pages that may contain text, images, videos, and other multimedia and navigates between them using hyperlinks.

Difference between Web and Internet

One can easily get confused by thinking that both World Wide Web and the Internet is the same thing. But the fact is that both are quite different.
The Internet and the World Wide Web are not one and the same. The Internet is a collection of interconnected computer networks, linked by copper wires, fiber-optic cables, wireless connections, etc. In contrast, the Web is a collection of interconnected documents and other resources, linked by hyperlinks and URLs. The World Wide Web is one of the services accessible via the Internet, along with various others including e-mail, file sharing, online gaming and others described below. However, "the Internet" and "the Web" are commonly used interchangeably in non-technical settings.

1.1 Working of Web

To view a Web page on the World Wide Web , the procedure begins either by typing the URL of the page into a Web browser, or by following a hyperlink to that page or resource. The Web browser then initiates a series of communication messages, behind the scenes, in order to fetch and display it. First, the server-name portion of the URL is resolved into an IP address using the global, distributed Internet database known as the domain name system, or DNS. This IP address is necessary to contact and send data packets to the Web server. The browser then requests the resource by sending an HTTP

request to the Web server at that particular address. In the case of a typical Web page, the HTML text of the page is requested first and parsed immediately by the Web browser, which will then make additional requests for images and any other files that form a part of the page. All this searching within the Web is performed by the special engines, known as Web Search Engines.

1.2 Thesis Organization

This section discusses the organization of this thesis. This thesis is organized as follows:

Chapter 2 reviews working of Search Engines and their components.

Chapter 3 describes Web Crawlers and their architecture in detail.

Chapter 4 illustrates the work done in the field of various web crawling algorithms along with their description.
These 3 chapters constitute Literature Survey and the work done in the field of search engines and Crawling.

Chapter 5 describes problem statement; the issue related with any crawling algorithm and also briefly gives modified algorithm description.

Chapter 6 gives the Implementation framework, the assumptions taken into consideration during the implementation and the results obtained after implementation.

Chapter 7 concludes the thesis work; illustrates future work and improvements that can be done to enhance performances of the thesis work.

Chapter 2

Working Of Web Search Engine

In the previous chapter we briefly discussed about the vast expansion occurring in the World Wide Web. As the web of pages around the world is increasing day by day, the need of search engines has also emerged. In this chapter, we explain the basic components of any basic search engine along with its working. After this, the role of web crawlers, one of the essential components of any search engine, is explained.

2.1 Basic Web Search Engine

The plentiful content of the World-Wide Web is useful to millions. Some simply browse the Web through entry points such as Yahoo, MSN etc. But many information seekers use a search engine to begin their Web activity [2]. In this case, users submit a query, typically a list of keywords, and receive a list of Web pages that may be relevant, typically pages that contain the keywords. By Search Engine in relation to the Web, we are usually referring to the actual search form that searches through- databases of HTML documents

2.2 Types of Search Engine

There are basically **3 types** of search engines:

- Those that are powered by robots (called ***crawlers***, ***ants or spiders***)
- Those that are powered by human submissions
- And those that are a hybrid of the two.

2 main types of search engines are explained briefly below.

2.2.1 Crawler Based Search Engine

Such search engines use automated software agents (called crawlers) that visit a Web site, read the information on the actual site, read the site's meta tags and also follow the

links that the site connects to performing indexing on all linked Web sites as well. The crawler returns all that information back to a *central depository*, where the data is indexed. The crawler will periodically return to the sites to check for any information that has changed. The frequency with which this happens is determined by the administrators of the search engine.

2.2.2 Human-powered Search Engine

Such search engines rely on humans to submit information that is subsequently indexed and catalogued. Only information that is submitted is put into the index. This type of search engines are rarely used at large scale. But these are useful in the organizations where small scale of data is dealt with.

2.3 Structure & Working of Search Engine

The basic structure of any crawler based search engine is shown in figure 2.1. Thus the main steps in any search engine are-

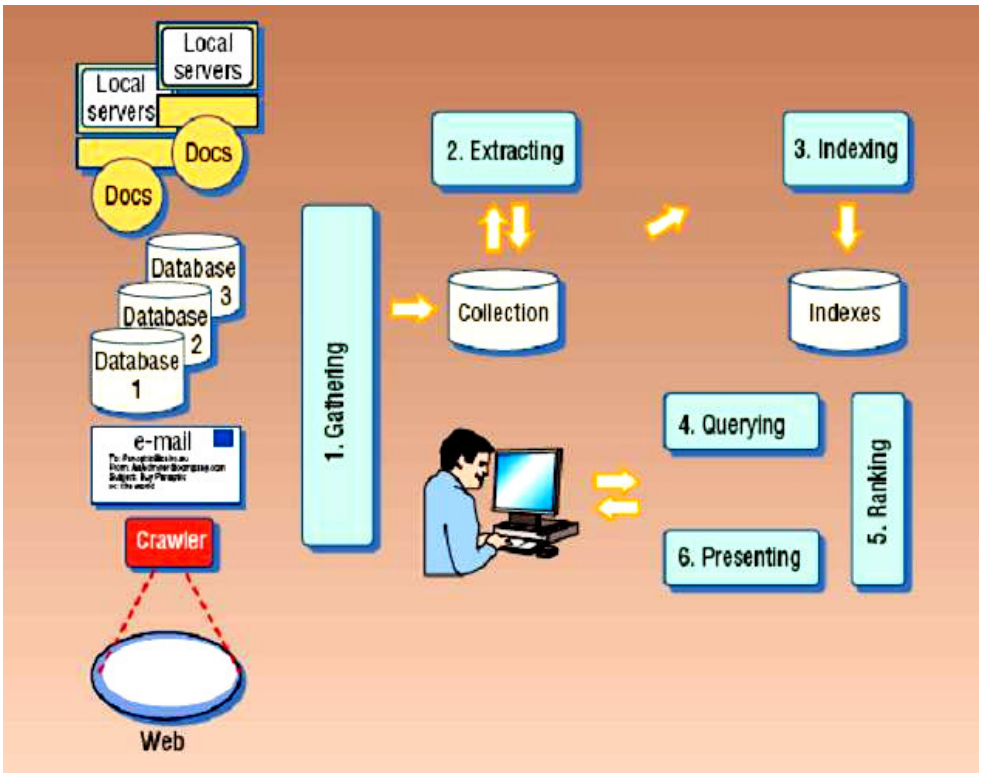


Figure 2.1: Generic Structure of a Search Engine [3]

2.3.1 Gathering also called “Crawling”

Every engine relies on a crawler module to provide the grist for its operation. This operation is performed by special software, called “Crawlers”

Crawlers are small programs that `browse' the Web on the search engine's behalf, similarly to how a human user would follow links to reach different pages. The programs are given a starting set of URLs, whose pages they retrieve from the Web. The crawlers extract URLs appearing in the retrieved pages, and give this information to the crawler control module. This module determines what links to visit next, and feeds the links to visit back to the crawlers.

2.3.2 Maintaining Database/Repository

All the data of the search engine is stored in a database as shown in the figure 2.1.All the searching is performed through that database and it needs to be updated frequently. During a crawling process, and after completing crawling process, search engines must store all the new useful pages that they have retrieved from the Web. The page repository (collection) in Figure 2.1 represents this possibly temporary collection. Sometimes search engines maintain a cache of the pages they have visited beyond the time required to build the index. This cache allows them to serve out result pages very quickly, in addition to providing basic search facilities.

2.3.3 Indexing

Once the pages are stored in the repository, the next job of search engine is to make a index of stored data. The indexer module extracts all the words from each page, and records the URL where each word occurred. The result is a generally very large “lookup table" that can provide all the URLs that point to pages where a given word occurs. The table is of course limited to the pages that were covered in the crawling process. As mentioned earlier, text indexing of the Web poses special difficulties, due to its size, and its rapid rate of change. In addition to these quantitative challenges, the Web calls for some special, less common kinds of indexes. For example, the indexing module may also create a structure index, which reflects the links between pages.

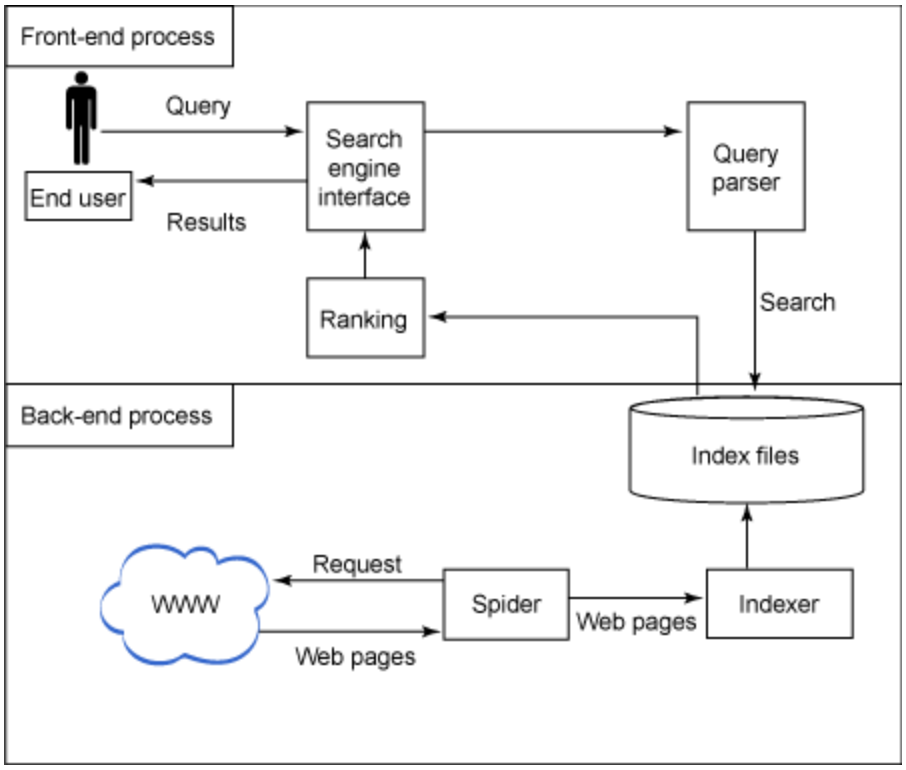


Figure 2.2: Working steps of search engine [4]

2.3.4 Querying

This sections deals with the user queries. The query engine module is responsible for receiving and filling search requests from users. The engine relies heavily on the indexes, and sometimes on the page repository. Because of the Web's size, and the fact that users typically only enter one or two keywords, result sets are usually very large.

2.3.5 Ranking

Since the user query results in a large number of results, it is the job of the search engine to display the most appropriate results to the user. To do this efficient searching, the ranking of the results are performed. The ranking module therefore has the task of sorting the results such that results near the top are the most likely ones to be what the user is looking for.

Once the ranking is done by the Ranking component, the final results are displayed to the user. This is how any search engine works.

2.4 Examples of Search Engines

There are a number of web search engines available in the market. The list of the most important and famous search engines can be listed as below:

- Google
- Yahoo
- MSN
- AltaVista
- E-Bay
- AOL Search
- Live Search
- You tube

And there are so many more search engines available in the market to assist the interne users.

2.5 Chapter Summary

The Web Search Engines are the key or the main-gate of World Wide Web. The evolution and their components are very important portion of study. Essential components of search engine are – Crawler, Parser, Scheduler and Database.
Some of the mostly used search engines are – Google, MSN etc.

Chapter 3
Web-Crawlers

As cleared from the previous chapter, one of the most essential jobs of any search engine is gathering of web pages, also called, ***"Crawling".*** This crawling procedure is performed by special software called, ***"Crawlers" or "Spiders".*** In this chapter, we will discuss the detailed structure of any web crawler.

3.1 Definition of Web-Crawler

A web-crawler is a program/software or automated script which browses the World Wide Web in a methodical, automated manner.

The structure of the World Wide Web is a graphical structure, *i.e*. the links given in a page can be used to open other web pages. Actually Internet is a directed graph, webpage as node and hyperlink as edge, so the search operation could be abstracted as a process of traversing directed graph. By following the linked structure of the Web, we can traverse a number of new web-pages starting from a Starting webpage. Web crawlers are the programs or software that uses the graphical structure of the Web to move from page to page [5]. Such programs are also called wanderers, robots, spiders, and worms. Web crawlers are designed to retrieve Web pages and add them or their representations to local repository/databases. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine that will index the downloaded pages that will help in fast searches. Web search engines work by storing information about many web pages, which they retrieve from the WWW itself. These pages are retrieved by a Web crawler (sometimes also known as a spider) — which is an automated Web browser that follows every link it sees. Web *crawlers* are programs that exploit the graph structure of the web to move from page to page. It may be observed that `crawlers' itself doesn't indicate speed of these programs, as they can be considerably fast working programs.

Web crawlers are software systems that use the text and links on web pages to create search indexes of the pages, using the HTML links to follow or crawl the connections between pages.

3.2 A Survey of Web Crawlers [6]

Web crawlers are almost as old as the web itself. The first crawler, Matthew Gray's Wanderer, was written in the spring of 1993, roughly coinciding with the first release of NCSA Mosaic. Several papers about web crawling were presented at the first two World Wide Web conferences. However, at the time, the web was three to four orders of magnitude smaller than it is today, so those systems did not address the scaling problems inherent in a crawl of today's web.

Obviously, all of the popular search engines use crawlers that must scale up to substantial portions of the web. However, due to the competitive nature of the search engine business, the designs of these crawlers have not been publicly described. There are two notable exceptions: the Google crawler and the Internet Archive crawler.

The original Google crawler [7] (developed at Stanford) consisted of five functional components running in different processes. A *URL server process* read URLs out of a file and forwarded them to multiple crawler processes. Each *crawler process* ran on a different machine, was single-threaded, and used asynchronous I/O to fetch data from up to 300 web servers in parallel. The crawlers transmitted downloaded pages to a single *StoreServer process*, which compressed the pages and stored them to disk. The pages were then read back from disk by an *indexer process*, which extracted links from HTML pages and saved them to a different disk file. A *URL resolver process* read the link file, derelativized the URLs contained therein, and saved the absolute URLs to the disk file that was read by the URL server. Typically, three to four crawler machines were used, so the entire system required between four and eight machines.

Research on web crawling continues at Stanford even after Google has been transformed into a commercial effort. The Stanford WebBase project has implemented a high-performance distributed crawler, capable of downloading 50 to 100 documents per second .Cho and others have also developed models of document update frequencies to inform the download schedule of incremental crawlers.

The Internet Archive also used multiple machines to crawl the web. Each crawler process was assigned up to 64 sites to crawl, and no site was assigned to more than one crawler. Each single-threaded crawler process read a list of seed URLs for its assigned sites from disk into per-site queues, and then used asynchronous I/O to fetch pages from these queues in parallel. Once a page was downloaded, the crawler extracted the links contained in it. If a link referred to the site of the page it was contained in, it was added to the appropriate site queue; otherwise it was logged to disk. Periodically, a batch process merged these logged "cross-site" URLs into the site-specific seed sets, filtering out duplicates in the process.

3.3 Basic Crawling Terminology

Before we discuss the working of crawlers, it is worth to explain some of the basic terminology that is related with crawlers. These terms will be used in the forth coming chapters as well.

3.3.1 Seed Page: By crawling, we mean to traverse the Web by recursively following links from a starting URL or a set of starting URLs. This starting URL set is the entry point though which any crawler starts searching procedure. This set of starting URL is known as "Seed Page". The selection of a good seed is the most important factor in any crawling process.

3.3.2 Frontier (Processing Queue): The crawling method starts with a given URL (seed), extracting links from it and adding them to an un-visited list of URLs. This list of un-visited links or URLs is known as, **"Frontier".** Each time, a URL is picked from the frontier by the Crawler Scheduler**.** This frontier is implemented by using Queue, Priority Queue Data structures. The maintenance of the Frontier is also a major functionality of any Crawler.

3.3.3 Parser: Once a page has been fetched, we need to parse its content to extract information that will feed and possibly guide the future path of the crawler. Parsing may imply simple hyperlink/URL extraction or it may involve the more complex process of tidying up the HTML content in order to analyze the HTML tag tree. The job of any

parser is to parse the fetched web page to extract list of new URLs from it and return the new un-visited URLs to the Frontier.

3.4 Working of Basic Web Crawler

From the beginning, a key motivation for designing Web crawlers has been to retrieve web pages and add them or their representations to a local repository. Such a repository may then serve particular application needs such as those of a Web search engine. In its simplest form a crawler starts from a ***seed* page** and then uses the external links within it to attend to other pages. The structure of a basic crawler is shown in figure 3.

The process repeats with the new pages offering more external links to follow, until a sufficient number of pages are identified or some higher level objective is reached. Behind this simple description lies a host of issues related to network connections, and parsing of fetched HTML pages to find new URL links.

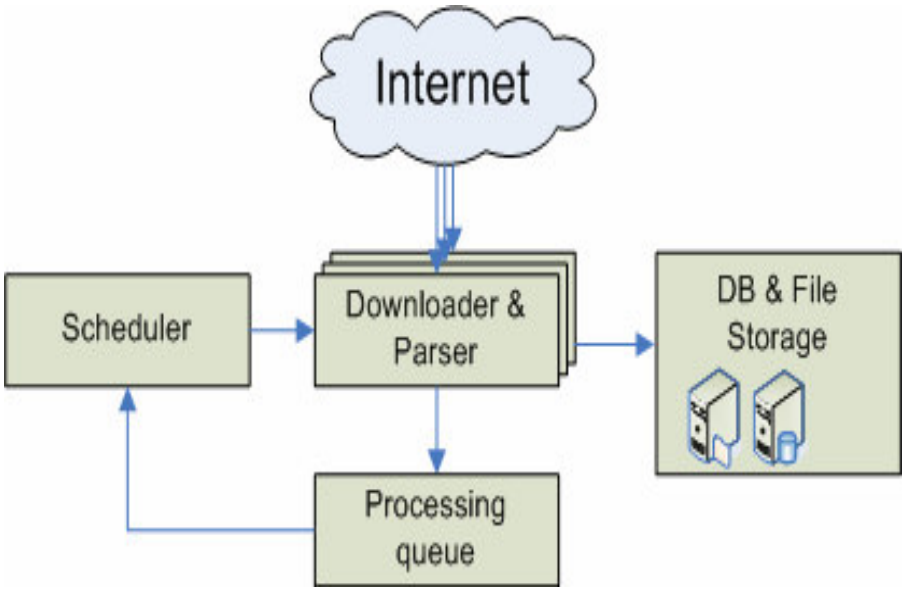


Figure 3.1: Components of a web-crawler

Common web crawler implements method composed from following steps:

- Acquire URL of processed web document from processing queue
- Download web document
- Parse document's content to extract set of URL links to other resources and update processing queue
- Store web document for further processing

The basic working of a web-crawler can be discussed as follows:

- Select a starting seed URL or URLs
- Add it to the frontier
- Now pick the URL from the frontier
- Fetch the web-page corresponding to that URL
- Parse that web-page to find new URL links
- Add all the newly found URLs into the frontier
- Go to **step 2** and repeat while the frontier is not Empty

Thus a crawler will recursively keep on adding newer URLs to the database repository of the search engine. So we can see that the main function of a crawler is to add new links into the frontier and to select a new URL from the frontier for further processing after each recursive step.

The working of the crawlers can also be shown in the form of a flow –chart **(Figure 3.2).** Note that it also depicts the 7 steps given earlier [8]. Such crawlers are called sequential crawlers because they follow a sequential approach.

In simple form, the flow chart of a web crawler can be stated as below:

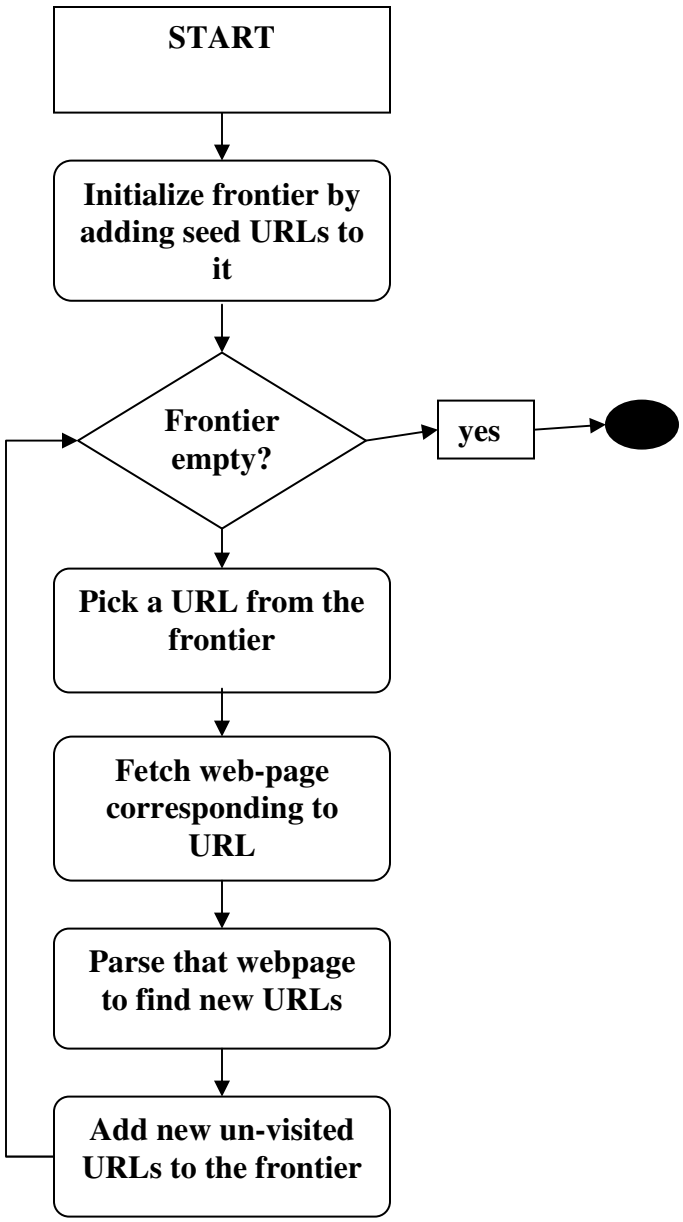


Figure 3.2: Sequential flow of a Crawler

3.5 Designing Issues of Web-Crawler

The crawler module retrieves pages from the Web for later analysis by the indexing module. As discussed above, a crawler module typically starts off with an initial set of URLs called, *Seed*. Roughly, it first places *Seed* in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue. This process is repeated until the crawler decides to stop. Given the enormous size and the change rate of the Web, many issues arise, including the following:

What pages should the crawler download? In most cases, the crawler cannot download all pages on the Web. Even the most comprehensive search engine currently indexes a small fraction of the entire Web. Given this fact, it is important for the crawler to carefully select the pages and to visit “important" pages first by prioritizing the URLs in the queue properly, so that the fraction of the Web that is visited (and kept up-to-date) is more meaningful.

- **How should the crawler refresh pages?** Once the crawler has downloaded a significant number of pages, it has to start *revisiting* the downloaded pages in order to detect changes and refresh the downloaded collection. Because Web pages are changing at very different rates, the crawler needs to carefully decide what page to revisit and what page to skip, because this decision may significantly impact the “freshness" of the downloaded collection. For example, if a certain page rarely changes, the crawler may want to revisit the page less often, in order to visit more frequently changing ones.

- **How should the load on the visited Web sites be minimized?** When the crawler collects pages from the Web, it consumes resources belonging to other organizations. For example, when the crawler downloads page *p* on site *S*, the site needs to retrieve page *p* from its file system, consuming disk and CPU resource. Also, after this retrieval the page needs to be transferred through the network,

which is another resource, shared by multiple organizations. The crawler should minimize its impact on these resources. Otherwise, the administrators of the Web site or a particular network may complain and sometimes completely block access by the crawler.

- **How should the crawling process be parallelized?** Due to the enormous size of the Web, crawlers often run on multiple machines and download pages in parallel. This parallelization is often necessary in order to download a large number of pages in a reasonable amount of time. Clearly these parallel crawlers should be coordinated properly, so that different crawlers do not visit the same Web site multiple times, and the adopted crawling policy should be strictly enforced. The coordination can incur significant communication overhead, limiting the number of simultaneous crawlers.

In the next section, the concept of parallel crawlers is given in detail along with their design issues.

3.6 Parallel Crawlers

As the size of the Web grows, it becomes more difficult to retrieve the whole or a significant portion of the Web using a single sequential crawler. Therefore, many search engines often run multiple processes in parallel to perform the above task, so that download rate is maximized. We refer to this type of crawler as a ***parallel crawler***.

Parallel crawlers work simultaneously to grab pages from the World Wide Web and add them to the central repository of the search engine.

The parallel crawling architecture is shown in the figure. Each crawler is having its own database of collected pages and own queue of un-visited URLs.

Once the crawling procedure finishes, the collected pages of every crawler are added to the central repository of the search engine.

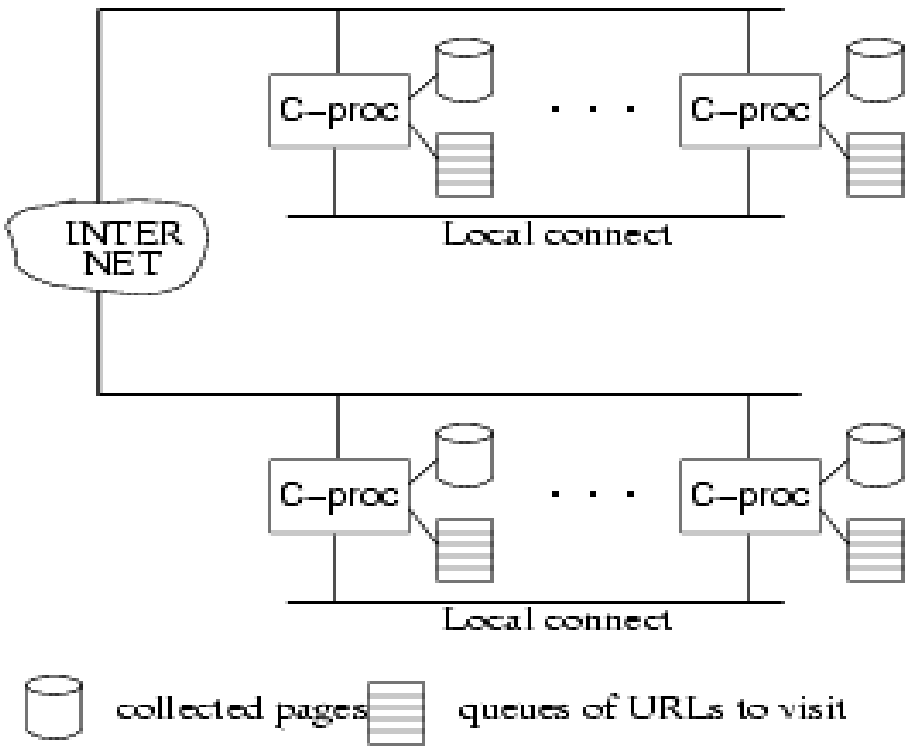


Figure 3.3: Structure of Parallel crawlers

Parallel crawling architecture no doubt increases the efficiency of any search engine but there are certain problems or issues related with the usage of parallel crawlers [9]. These are discussed in following section.

3.6.1 Issues related with Parallel Crawlers

- **Risk of Multiple copy download***:* When multiple crawlers run in parallel to download pages, it is possible that different processes download the same page multiple times. One process may not be aware that another process has already downloaded the page. Clearly, such multiple downloads should be minimized to save network bandwidth and increase the crawler's effectiveness.

- **Quality:** Every crawler wants to download "important" pages first, in order to maximize the "quality" of the downloaded collection. However, in a parallel

crawler environment, each crawler 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 (that itself has downloaded) and thus make a poor crawling decision. Thus ensuring the *quality* of the downloaded pages in a parallel crawler environment is a major issue.

- **Communication bandwidth:** In order to prevent overlap, or to improve the quality of the downloaded pages, crawling processes need to periodically communicate to coordinate with each other. However, this communication may grow significantly as the number of crawling processes increases.

Thus we can see that it's not very easy to implement parallel crawling environment in any search engine. So a lot of care has to taken into account before designing any parallel crawling structure.

3.6.2 Advantages of Parallel Crawler Architecture

Although parallel crawling architecture is challenging to implement, still we believe that a parallel crawler has many important advantages, compared to a single-process crawler or a sequential crawler.

- **Scalability:** The size of Web is huge. Due to this size of the Web, it is often useful to run a parallel crawler. A single-process crawler simply cannot achieve the required download rate in certain time space.
- **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 Sweden may download all European pages, while another one in India crawls all Asian pages. In this way, one can easily 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.

- **Reduction in Network-load:** In addition to the dispersing load, a parallel crawler may actually reduce the network load also. For example, assume that a crawler in India 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-Asia inter-continental network and finally the network in India. Instead, if a crawling process in Europe collects all European pages, and if another process in Asia crawls all pages from Asia, the overall network load will be reduced, because pages go through only "local" networks

Thus we can conclude that Parallel Crawler Architecture is a better option as compared to single crawler architecture. Also the number of web pages around the globe is huge. So to crawl the whole web, only parallel crawlers can do this job in short span of time by keeping in consideration the issues listed above.

3.7 Examples of Web-Crawler

The following is a list of crawler architectures [10] for general-purpose crawlers (excluding focused web crawlers), with a brief description that includes the names given to the different components and their features:

- **RBSE** (Eichmann, 1994) is the first published web crawler. It is based on two programs: the first program, "`spider`" maintains a queue in a relational database, and the second program "`mite`", is a modified `www` ASCII browser that downloads the pages from the Web.
- **WebCrawler** (Pinkerton, 1994) is used to build the first publicly-available full-text index of a subset of the Web. It is based on lib-WWW to download

pages, and another program to parse and order URLs for breadth-first exploration of the Web graph.

- **World Wide Web Worm** (McBryan, 1994) is a crawler used to build a simple index of document titles and URLs. The index could be searched by using the `grep` Unix command.
- **Google Crawler** (Brin and Page, 1998) in C++ and Python. The crawler is integrated with the indexing process, because text parsing was done for full-text indexing and also for URL extraction.
- **Mercator** (Heydon and Najork, 1999; Najork and Heydon, 2001) is a distributed, modular web crawler written in Java. Its modularity arises from the usage of interchangeable "protocol modules" and "processing modules". Protocols modules are related to how to acquire the web pages (e.g.: by HTTP), and processing modules are related to how to process web pages.
- **WebFountain** (Edwards et al., 2001) is a distributed, modular crawler similar to Mercator but written in C++. It features a "controller" machine that coordinates a series of "ant" machines. After repeatedly downloading pages, a change rate is inferred for each page and a non-linear programming method must be used to solve the equation system for maximizing freshness.
- **WebRACE** (Zeinalipour-Yazti and Dikaiakos, 2002) is a crawling and caching module implemented in Java, and used as a part of a more generic system called eRACE.
- **Ubicrawler** (Boldi et al., 2004) is a distributed crawler written in Java, and it has no central process. It is composed of a number of identical "agents"; and the assignment function is calculated using consistent hashing of the host names.

3.7.1 Open-Source Web-crawlers

- **DataparkSearch** is a crawler and search engine released under the GNU General Public License.
- **GNU Wget** is a command-line operated crawler written in C. It is typically used to mirror web and FTP sites.
- **Heritrix** is the Internet Archive's archival-quality crawler, designed for archiving periodic snapshots of a large portion of the Web. It is written in Java.
- **WIRE - Web Information Retrieval Environment** - is a web crawler written in C++ and released under the GPL.
- **Web Crawler** Open source web crawler class for .NET (written in C#).

3.8 Chapter Summary

Web Crawler is one of the important components of any search engine. A number of web crawlers are available in the market. The job of a web crawler is to navigate the web and extract new pages for the storage in the database of the search engines. It also involves the traversing, parsing and other considerable issues. Also to achieve better and fast results, parallel crawlers are used.

Chapter 4
Web-Crawling Algorithms

In the previous chapter, we mentioned concepts of web crawlers, their working and the component functionality. The most important component is the **Scheduler**, whose job is to pick the next URL from the list of un-visited URLs for further processing.
In this chapter, we will mention the techniques or algorithms currently used by search engines to implement the Scheduling Process while crawling the web.

Crawling Algorithms

By looking at the basic working of crawlers, it is clear to us that at each stage crawler selects a new link from the frontier for processing. So the selection of the next link from the frontier is also a major aspect. The selection of the next link from the frontier entirely depends upon the crawling algorithm we are using.
So the job of selecting the next link from the frontier is something like selection of job by the CPU from Ready Queue (CPU Scheduling) in operating systems.

The basic algorithm executed by any web crawler takes a list of ***seed*** **URLs** as its input and repeatedly executes the following steps:

- **Remove a URL from the URL list**
- **Determine the IP address of its host name**
- **Download the corresponding document**
- **Extract any links contained in it**

For each of the extracted links, ensure that it is an absolute URL and add it to the list of URLs to download; provided it has not been encountered before. If desired, process the downloaded document in other ways (e.g., index its content).

Any basic Web crawling algorithm requires a number of functional components:

- A component (called the URL frontier) for storing the list of URLs to download;
- A component for resolving host names into IP addresses;
- A component for downloading documents using the HTTP protocol;
- A component for extracting links from HTML documents; and
- A component for determining whether a URL has been encountered before or not.

The simplest crawling algorithm uses a queue of URLs yet to be visited and a fast mechanism for determining if it has already seen a URL. This requires huge data structures. A simple list of 20 billion URLs contains more than a terabyte of data. [How things] The crawler initializes the queue with one or more "seed" URLs. A good seed URL will link to many high-quality Web sites—for example, www.dmoz.org or wikipedia.org. Crawling proceeds by making an HTTP request to fetch the page at the first URL in the queue. When the crawler fetches the page, it scans the contents for links to other URLs and adds each previously unseen URL to the queue. Finally, the crawler saves the page content for indexing. Crawling continues until the queue is empty.

4.1 Desired Features of a Good Crawling Algorithm

The features of a simple crawling algorithm must be extended to address the following issues:

- **Speed-** If each HTTP request takes one second to complete—some will take much longer or fail to respond at all—the simple crawler can fetch no more than 86,400 pages per day. At this rate, it would take 634 years to crawl 20 billion pages. In practice, crawling is carried out using hundreds of distributed crawling machines. A hashing function determines which crawling machine is responsible for a particular URL. If a crawling machine encounters a URL for which it is not

responsible, it passes it on to the machine that is responsible for it. Even hundredfold parallelism is not sufficient to achieve the necessary crawling rate. Each crawling machine therefore exploits a high degree of internal parallelism, with hundreds or thousands of threads issuing requests and waiting for responses.

- **Politeness-** Unless care is taken, crawler parallelism introduces the risk that a single Web server will be bombarded with requests to such an extent that it becomes overloaded. Crawler algorithms are designed to ensure that only one request to a server is made at a time and that a politeness delay is inserted between requests. It is also necessary to take into account bottlenecks in the Internet; for example, search engine crawlers have sufficient bandwidth to completely saturate network links to entire countries.
- **Excluded content-** Before fetching a page from a site, a crawler must fetch that site's robots.txt file to determine whether the webmaster has specified that some or the entire site should not be crawled.
- **Duplicate content-** Identical content is frequently published at multiple URLs. Simple checksum comparisons can detect exact duplicates, but when the page includes its own URL, a visitor counter, or a date, more sophisticated fingerprinting methods are needed .Crawlers can save considerable resources by recognizing and eliminating duplication as early as possible because unrecognized duplicates can contain relative links to whole families of other duplicate content. Search engines avoid some systematic causes of duplication by transforming URLs to remove superfluous parameters such as session IDs and by case folding URLs from case-insensitive servers.
- **Continuous crawling**- Carrying out full crawls at fixed intervals would imply slow response to important changes in the Web. It would also mean that the crawler would continuously re-fetch low-value and static pages, thereby incurring substantial costs without significant benefit. For example, a corporate site's 2002 media releases section rarely, if ever, requires re-crawling.
- **Spam rejection-** Primitive spamming techniques, such as inserting misleading keywords into pages that are invisible to the viewer—for example, white text on a white background, zero-point fonts, or meta tags—are easily detected. Modern

spammers create artificial Web landscapes of domains, servers, links, and pages to inflate the link scores of the targets they have been paid to promote. Search engine companies use manual and automated analysis of link patterns and content to identify spam sites that are then included in a blacklist. A crawler can reject links to URLs on the current blacklist and can reject or lower the priority of pages that are linked to or from blacklisted sites.

2 major approaches used for crawling are:

- **Blind Traversing approach**
- **Best – First Heuristic approach**

4.2 Blind Traversing Approach

In this approach, we simply start with a seed URL and apply the crawling process as stated earlier. It is called blind because for selecting next URL from frontier, no criteria are applied. Crawling links are selected in the order in which they are encountered in the frontier (in serial order)

One algorithm widely common to implement Blind traversing approach is – Breadth First Algorithm. It uses FIFO QUEUE data structure to implement the frontier; it is very simple and basic crawling algorithm.

Since this approach traverses the graphical structure of WWW breadth – wise, **Queue** data structure is used to implement the Frontier.

Algorithm that comes under Blind Crawling approach is- **Breadth First Algorithm**.

4.2.1 Breadth First Algorithm

A Breadth-First crawler is the simplest strategy for crawling. This algorithm was explored in 1994 in the WebCrawler as well as in more recent research [11]. It uses the frontier as a FIFO queue, crawling links in the order in which they are encountered. The problem with this algorithm is that when the frontier is full, the crawler can add only one link from a crawled page. The Breadth-First crawler is illustrated in Figure 3. Breadth-First Algorithm is usually used as a baseline crawler; since it does not use any knowledge

about the topic, it acts blindly. That is why, also called, Blind Search Algorithm. Its performance is used to provide a lower bound for any of the more sophisticated algorithms.

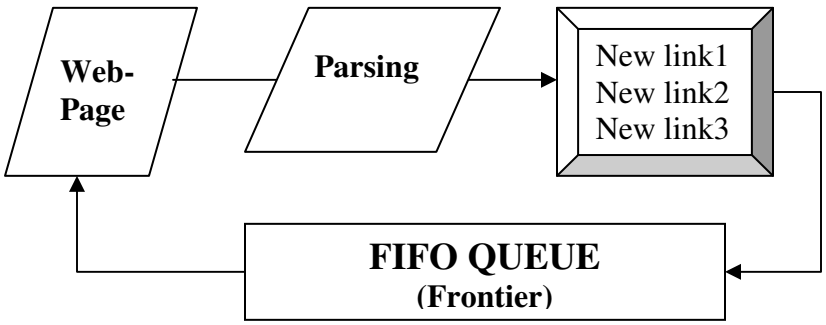


Figure 4.1 Breadth-First crawling Approach

4.2.2 Drawbacks of Breadth First Approach

In real WWW structure, there are millions of pages linked to each other. The size of the repository of any search engine can not accommodate all pages. So it is desired that we always store the most suitable and relevant pages in our repository. Problem with Blind Breadth First algorithm is that it traverses URLs in sequential order as these were inserted into the Frontier. It may be good when the total number of pages is small. But in real life, a lot of useless pages can produce links to other useless pages. Thus storing and processing such links in frontier is wastage of time and memory. So we should select a useful page from the frontier every time for processing irrespective of its position in the frontier. But Breadth first approach always fetched 1st link from the frontier, irrespective of its usefulness. So the Breadth First approach is not desirable.

4.3 Best –First Heuristic Approach

To overcome the problems of blind traverse approach, a heuristic approach called Best-First crawling approach have been studied by Cho et al. [1998] and Hersovici et al. [1998]. In this approach, from a given Frontier of links, next link for crawling is selected on the basis of some estimation or score or priority. Thus every time the best available link is opened and traversed. The estimation value for each link can be calculated by different pre-defined mathematical formulas. (Based purely on the needs of specific engine)

Following Web Crawling Algorithms use Heuristic Approach:

4.3.1 Naive Best - First Algorithm

One Best First Approach uses a relevancy Function rel ( ) to compute the lexical similarity between the desired key-words and each page & associate that value with corresponding links in the frontier. After each iteration, the link with the highest rel ( ) function value is picked from the frontier. That is, the best available link is traversed every time which is not possible in Breadth First Approach.

Since any link with highest relevancy value can be picked from the Frontier, most of the Best first algorithms use **– *Priority Queue*** as data structure.

The working of Heuristic Crawling Algorithms is illustrated in figure 4.

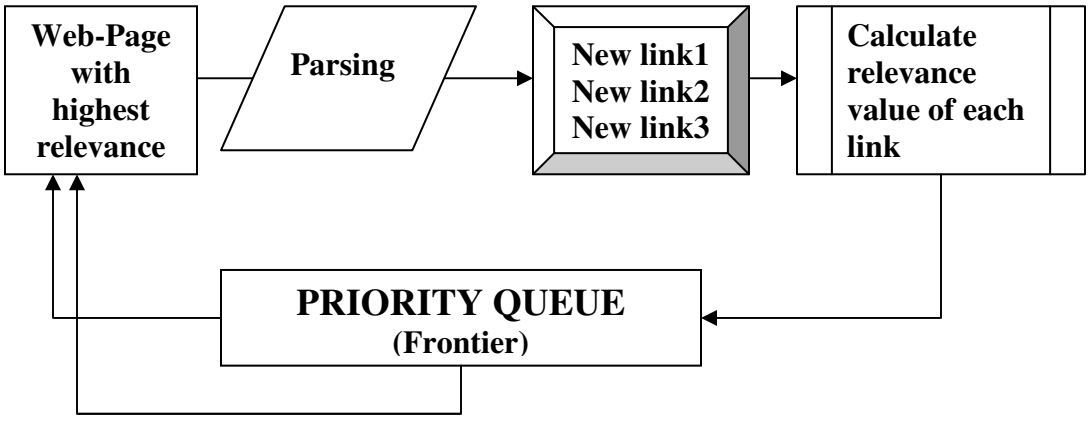


Figure 4.2: Best-First Crawling Approach

As clear from figure 4, web-page with highest relevance is picked from any position from Frontier for processing.

4.3.2 Page Rank Algorithm

PageRank was proposed by Brin and Page [12] as a possible model of user surfing behavior. The PageRank of a page represents the probability that a random surfer (one who follows links randomly from page to page) will be on that page at any given time. A page's score depends recursively upon the scores of the pages that point to it. Source pages distribute their PageRank across all of their outlinks.

Formally:

$$\mathbf{PR(A) = (1-d) + d\ (PR(T_1)/C(T_1) + ... + PR(T_n)/C(T_n))}$$

where PR(A) is the PageRank of a page A

$PR(T_1)$ is the PageRank of a page T_1

$C(T_1)$ is the number of outgoing links from the page T_1

d is a damping factor in the range $0 < d < 1$, usually set to 0.85

The PageRank of a web page is therefore calculated as a sum of the PageRanks of all pages linking to it (its incoming links), divided by the number of links on each of those pages (its outgoing links).

As originally proposed PageRank was intended to be used in combination with content-based criteria to rank retrieved sets of documents [Brin and Page1998].This is in fact how PageRank is used in the Google search engine. More recently PageRank has been used to guide crawlers [Cho et al. 1998] and to assess page quality.

The algorithm of page rank [13] is given in **figure 4.3**

```
PageRank (topic, starting_urls, frequency) {
   foreach link (starting_urls) {
      enqueue(frontier, link);
   }
   while (visited < MAX_PAGES) {
      if (multiplies(visited, frequency)) {
         recompute_scores_PR;
      }
      link := dequeue_top_link(frontier);
      doc := fetch(link);
      score_sim :=  sim(topic, doc);
      enqueue(buffered_pages, doc, score_sim);
      if (#buffered_pages >= MAX_BUFFER) {
         dequeue_bottom_links(buffered_pages);
      }
      merge(frontier, extract_links(doc), score_PR);
      if (#frontier > MAX_BUFFER) {
         dequeue_bottom_links(frontier);
      }
   }
}
```

Figure 4.3: Page Rank Algorithm

4.3.3 Fish – Search Algorithm

Web search services typically use a previously built index that is actually stored on the search service server(s). This approach is pretty efficient for searching large parts of the Web, but it is basically static. The actual search is performed on the server on all the data stored in the index. Results are not guaranteed to be valid at the time the query is issued (note that many search sites may take up to one month for refreshing their index on the full Web). In contrast, dynamic search actually fetches the data at the time the query is issued. While it does not scale up, dynamic search guarantees valid results, and is preferable to static search for discovering information in small and dynamic sub-Webs.

One of the first dynamic search heuristics was the "fish search" [14], that capitalizes on the intuition that relevant documents often have relevant neighbors. Thus, it searches

deeper under documents that have been found to be relevant to the search query, and stops searching in "dry" areas.

Fish-search algorithm treats Internet as a directed graph, webpage as node and hyperlink as edge, so the search operation could be abstracted as a process of traversing directed graph. For every node we judge whether it is relative, I for relevant, 0 for irrelevant. Fish-search algorithm maintains a list, which keeps URL of page to be searched. The URLs have different priority, the URL with more superior priority will be located at the front of the list, and will be searched sooner than others. If relative page is found, it stands for that the food has been found by the fish, and more healthy reproduction, the same as more relative links covered by the page. The key point of Fish-search algorithm lies in the maintenance of URL's order. Based on the value of potential-score, Fish-search algorithm changes the order in the list.

The fish search algorithm can be stated as follows:

- **Get** as Input parameters, the initial node, the width *(width)*, depth *(D)* and size *(S)* of the desired graph to be explored, the time limit, and a search query
- **Set** the depth of the initial node as *depth = D*, and **Insert** it into an empty list
- **While** the list is not empty, and the number of processed nodes is less than S, and the time limit is not reached
    1. **Pop** the first node from the list and make it the current node
    2. **Compute** the relevance of the current node
    3. **If** *depth > 0*:
        1. **If** *current_node* is irrelevant

           Then
            - **For** each child, *child_node*, of the first *width* children of *current_node*
                - **Set** *potential_score(child_node) = 0.5*
            - **For** each child, *child_node*, of the rest of the children of *current_node*

- **Set** *potential_score(child_node) = 0*

Else

- **For** each child, *child_node*, of the first *(α * width)* children of *current_node*
  (where α is a pre-defined constant typically set to *1.5*)
  - **Set** *potential_score(child_node) = 1*
- **For** each child, *child_node*, of the rest of the children of *current_node*
  - **Set** *potential_score(child_node) = 0*

2. **For** each child, *child_node*, of *current_node*,
   - **If** *child_node* already exists in the priority list,
     Then
     1. **Compute** the maximum between the existing score in the list to the newly computed potential score
     2. **Replace** the existing score in the list by that maximum
     3. **Move** *child_node* to its correct location in the sorted list if necessary

     Else
     Insert *child_node* at its right location in the sorted list according to its *potential_score* value

3. **For** each child, *child_node*, of *current_node*,
   - **Compute** its depth, *depth(child_node)*, as follows:
     0. **If** *current_node* is relevant,
        Then Set *depth(child_node) = D*
        Else *depth(child_node) = depth(current_node) – 1*

1. **If** *child_node* already exists in the priority list
   Then
   1. **Compute** the maximum between the existing depth in the list to the newly computed depth
   2. **Replace** the existing depth in the list by that maximum.

End While

Figure 4.4: Fish Search Algorithm

The fish-search algorithm, while being attractive because of the simplicity of its paradigm, and its dynamic nature, presents some limitations.
First, it assigns a relevance score in a discrete manner (*1* for relevant, 0 or 0.5 for irrelevant) using primitive string- or regular-expression match. More generally, the key problem of the fish-search algorithm is the very low differentiation of the priority of pages in the list. When many documents have the same priority, and the crawler is restricted to a fairly short time, arbitrary pruning occurs — the crawler devotes its time to the documents at the head of the list. Documents which are further along the list whose scores may be identical to some further along may be more relevant to the query. In addition, cutting down the number of addressed children by using the width parameter is arbitrary, and may result in loosing valuable information.

Clearly, the main issue that needs to be addressed is a finer grained scoring capability. This is problematic because it is difficult to assign a more precise potential score to documents which have not yet been fetched.

4.3.4 Shark Search Algorithm

Considering the limitations of the Fish Search algorithm as stated above, a powerful improved version of Fish Search algorithm is developed known as- **Shark Search** [15]

In this algorithm, One immediate improvement is that instead of the binary (relevant/irrelevant) evaluation of document relevance, it returns a "fuzzy" score, i.e., a score between **0 and 1** (0 for no similarity whatsoever, 1 for perfect "conceptual" match) rather than a binary value.

These heuristics are so effective in increasing the differentiation of the scores of the documents on the priority list, that they make the use of the width parameter redundant, there is no need to arbitrarily prune the tree. Therefore, no mention is made of the width parameter in the shark-search algorithm that is more formally described in Fig. 4.5

In the fish-search algorithm, replace step for computing the child's potential score, with the following:

1. **Compute** the inherited score of *child_node*, *inherited_score(child_node)*, as follows:
   - **If** *relevance(current_node) > 0* (the current node is relevant)
     Then *inherited_score(child_node)* = δ * *sim(q,current_node)*
     where δ is a predefined decay factor.
     Else *inherited_score(child_node)* = δ * *inherited_score(current_node)*
2. **Let** *anchor_text* be the textual contents of the anchor pointing to *child_node*, and *anchor_text_context*, the textual context of the anchor (up to given predefined boundaries)
3. **Compute** the relevance score of the anchor text as *anchor_score* = *sim(q,anchor_text)*
4. **Compute** the relevance score of the anchor textual context as follows:
   - **If** *anchor_score > 0*,
     Then *anchor_context_score = 1*

Else *anchor_context_score = sim(q,anchor_text_context)*

5. **Compute** the score of the anchor, that we denote *neighbourhood_score* as follows:

   neighbourhood_score $= \beta$ * *anchor_score* + *(1* $- \beta$*)* * *anchor_context_score*

   where β is a predefined constant.

6. **Compute** the potential score of the child as - potential_score (child_node) = γ *inherited_score(child_node) + (1 $- \gamma$) * neighbourhood_score(child_node)

   where γ is a predefined constant

Figure 4.5: Shark Search Algorithm

Rest of the algorithm is same as fish Algorithm.

The Shark Search Algorithm improved working of Fish Search algorithm in a lot of manner. The child inherits the discounted value of ancestors and Shark Search also keeps in consideration the Anchor text of a web page while assigning it any relevancy value.

4.4 Chapter Summary

A number of methods for crawling are currently used by the search engines. The two approaches of searching and crawling are - Basic Blind Search approach which uses Breadth First Algorithm. Any good crawling algorithm should address certain issues. The other more powerful heuristic Approach is currently used by most of the search engines. The algorithms using this approach are – Best First, Fish Search and Shark Search Algorithms.

Chapter 5

Problem Statement

In the previous chapter, basic web-crawling algorithms were discussed. Also there are some issues related with the design of any web-crawler. Thus a good Crawling algorithm should possess certain essential features that can solve issues related with the capabilities of a good crawler.

A Web repository is a large special-purpose collection of Web pages and associated indexes [16]. Many useful queries and computations over such repositories involve traversal and navigation of the Web graph. However, efficient traversal of huge Web graphs containing several hundred million vertices and a few billion edges is a challenging problem. The structure of the web can be seen as Graphical. Every page contains links to many new pages. For example- consider the following scenario.

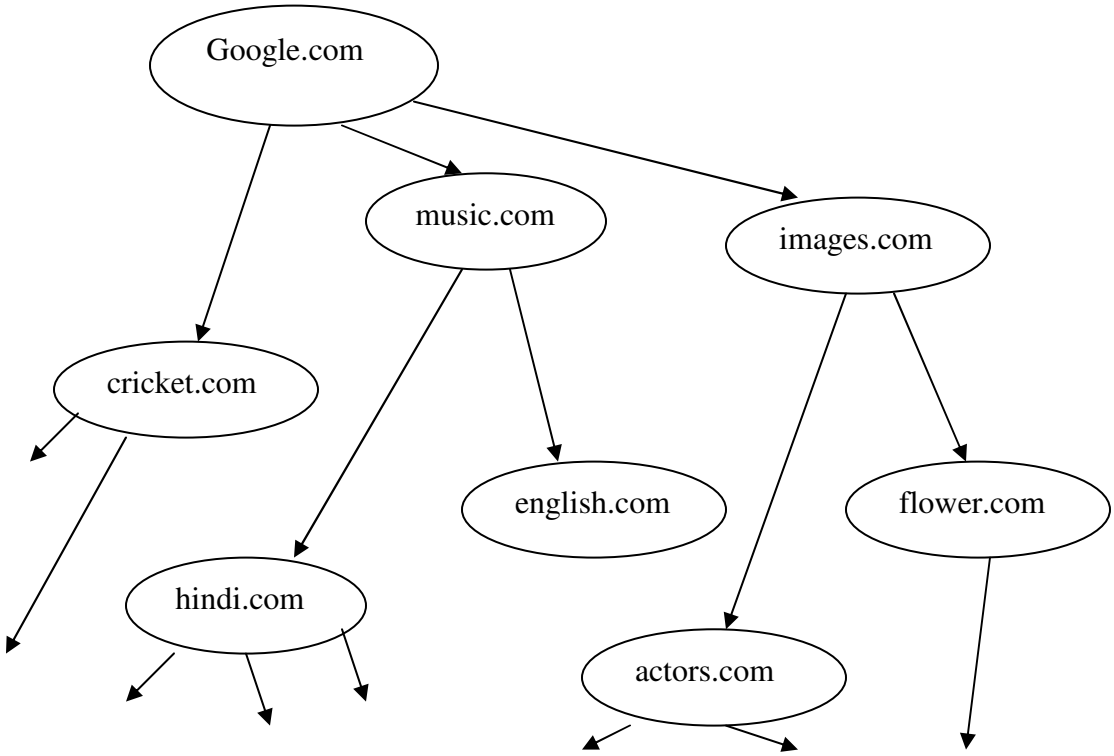


Figure 5.1 Graphical Structure of Web

We start with “www.google.com” webpage. Suppose it has links to 3 new pages. These pages again have links opening other web-pages and so on. Thus this forms a graphical structure.

As cleared from the example in figure 5.1, the structure of web is huge and each page explores a set of new web pages. So the most critical objective of any web crawling algorithm is to traverse more and more pages by following the links [17].

All the discussed algorithms fulfill the traversing purpose of any web-crawler. The Breadth-First algorithm simply do the blind search, it doesn’t consider the relevance or prioritize the web-pages according to relevancy. So the algorithms that perform Heuristic Traversing like – Best First, Fish-Search and Shark-Search are used to achieve better results.

5.1 Designing Algorithms and Programs

The major problem is to design proper Pseudo-codes of the algorithms and then converting these pseudo-codes into some programming language. Every Pseudo-code should depict the procedures involved in specific algorithm. If the pseudo code approach is accurate, only then it can be converted into programming language

5.2 Modifying Naive Best First Algorithm

In the simple Best First Approach, the relevancy of current page is calculated and that relevancy value is assigned to every children of that page. In our modified approach, we traverse each child of the current page, and check every page for relevancy. The relevancy of each page is associated with them and according to the relevancy; these pages are entered into the Frontier. Thus every child doesn’t get same relevance as of its parent page.

In the next chapter, we have discussed the designing and implementation issues in details. The pseudo codes of algorithms are given and assumptions made during implementation are also discussed.

Chapter 6

Algorithms & Implementation Results

A number of crawling strategies are illustrated in the previous chapters. We have implemented some of the crawling algorithms using programming in C and the results are obtained by running these algorithms on a same graph structure.
We have implemented following algorithms:

- **Breadth-First Algorithm**
- **Naïve Best First Algorithm**
- **Fish Search Algorithm**
- **Modified Best First Algorithm**

6.1 Data Structures and Programming Language used

In this section, we will describe the data structures used and the programming language used in the implementation of algorithms. We start with giving introduction to C language which is used as main language in our implementation.

6.1.1 Introduction to C language

C is a general-purpose, block structured, procedural, imperative computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the UNIX operating system. It has since spread to many other platforms. Although C was designed as a system implementation language, it is also widely used for applications. C has also greatly influenced many other popular languages, especially C++, which was originally designed as an extension to C.

6.1.2 Linked List Representation of Graph

A linked list is a collection of objects linked to one another like a string of pearls. As a minimum, a linked list knows about its first element, and each element knows about its successor. The structure of linked list is shown in figure 6.1

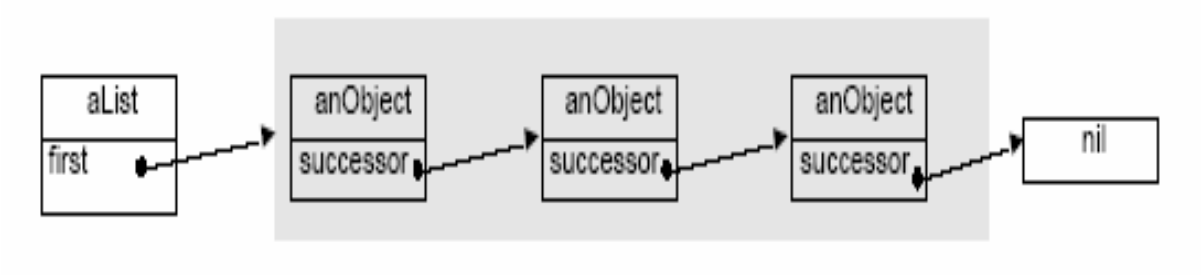


Figure 6.1: Linked List Representation

As mentioned in the previous chapters, the structure of World Wide Web is Graphical in nature. Web graphs contain a few hundred million vertices and a few billion edges. With each vertices representing a web page and each edge representing link from one page to another web-pages.

So, to implement these algorithms, we have used the techniques of representing Graph in memory and for this representation, and one technique of representing graph in memory is to use the Linked List Structure.

Linked list is a collection of nodes and in C language; the node structure can be given as below:

```
                    struct node
{
                    int vertex;
                    float weight;
                    node *link;
};
```

Figure 6.2: Node Structure of Graph in C

6.2 Assumptions

During the implementation process, we have taken some assumptions in to account just for simplifying algorithms implementation and results.

As mentioned in Chapter 4, the basic procedure executed by any web crawling algorithm takes a list of ***seed*** **URLs** as its input and repeatedly executes the following steps:

a) **Remove a URL from the URL list**
b) **Determine the IP address of its host name**
c) **Download the corresponding document**
d) **Check the Relevance of Document**
e) **Extract any links contained in it**
f) **Add these links back to the URL list**

In this implementation of algorithms, we have made some assumptions related to steps (d) and (e) of the above procedure.

6.2.1 Relevancy Calculation

In the algorithms, the relevancy is calculated by parsing the web page and matching its contents with the desired key-words. After matching, a Relevancy Value corresponding to that page is generated by the Relevancy Calculator component.

In our implementation, we mainly check how the traversing is done by various algorithms, we have not implemented concept of parsing as it in itself is a different module. The parsing module can be simply replaced in our implementation source code. For the time being, to calculate relevancy of different documents, we have generated relevancy values randomly using random values between **0 and 1**. Whenever a page is traversed, a random relevancy is generated corresponding to it.

6.2.2 Link Extraction

As the implementation of Parser is not available, the link extraction from a page is performed by traversing its Adjacency list, and adding the pages corresponding to that list into the unvisited URL list.

6.2.3 Input Graph for Implementation

Since the entire web can't be selected for implementation, we have used a graph structure as shown below [18] to implement all the algorithms on it.

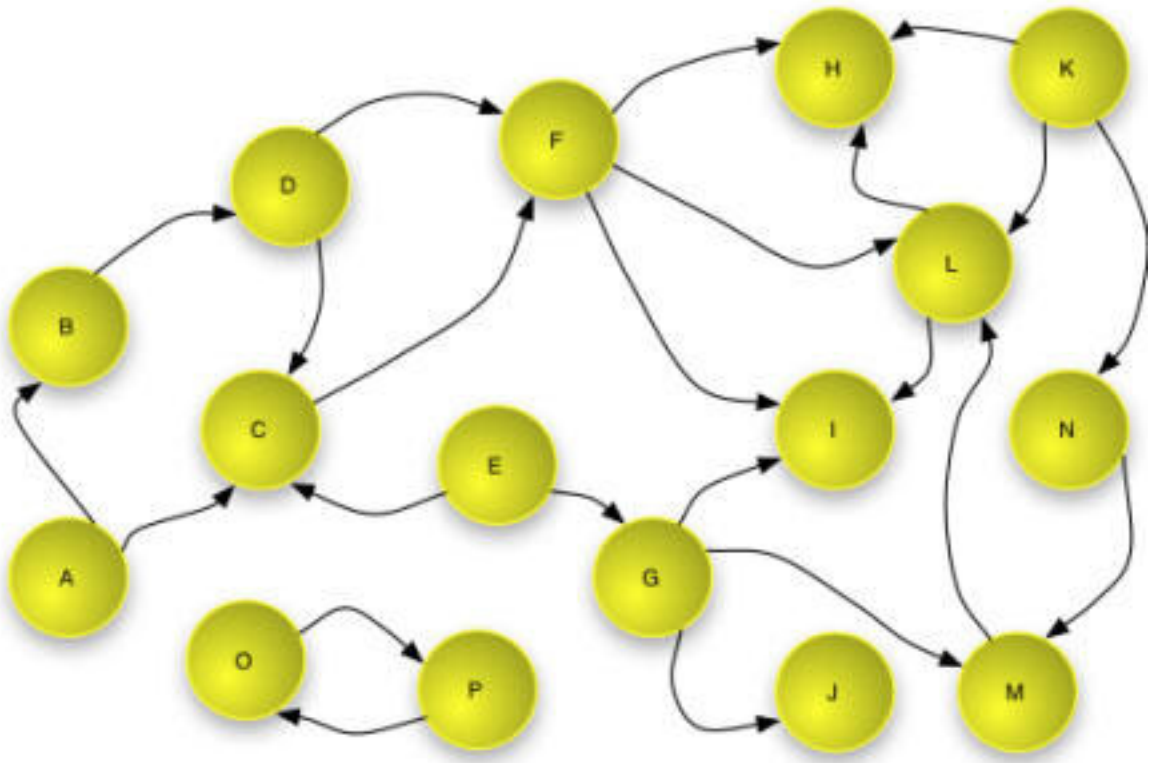


Figure 6.3: Input Graph for Implementation

As for example, we have used above graph of 16 nodes. i.e. these 16 nodes represent 16 web-pages, while each edge represents link between two web-pages. As in real web, some web-pages are not having links with others; nodes O and P represent this case.

We will execute programs made in **C language** of different algorithms on this dummy web structure and corresponding results will be noted.

6.3 Pseudo-Code of Algorithms

We have implemented 4 crawling algorithms in C language. The Pseudo-code of these crawling algorithms is given below:

6.3.1 Pseudo Code of Breadth First Algorithm

The pseudo-code of a Breadth First Approach is as follows:

```
Breadth_First_Algo(seed URLs)                    //  Start with given Seed URLs as input

Insert_Frontier (seed URLs);                     // Insert seed URLs into the Frontier
While (Frontier! = Empty)                        // Crawling Loop

Link: =Remove_Frontier (URL);                    // Pick  new link from the frontier
Webpage: = Fetch (Link);                         // Open Webpage corresponding to selected link
Insert_Frontier (newly_extracted_links);         // Add new links into frontier

End While Loop
```

Figure 6.4: Pseudo-code of Breadth-First Approach

The working of code is simple. We start with a set of URLs, called Seed and insert these seed into Queue, also called Frontier. Crawling process continues while Queue is not empty, every time a new URL is removed from the Front of Queue and is traversed. All new links found in this page are inserted into the queue.

6.3.2 Pseudo- Code of Naive Best First Algorithm

Pseudo-code for a Best First Approach can stated as follows:

```
Best_First_Algo(seed URLs)              //  Start with given Seed URLs as input
{
Insert_Frontier (seed URLs, 1);                    // Insert seed URLs
Repeat While (Frontier! = Empty)                   //  Perform Recursively following steps
Link: =Remove_Highest_Frontier (URL);    // Pick link with highest rel_val from frontier
Webpage: = Fetch (Link);                 // Open Webpage corresponding to selected link
Repeat For each child_node of Webpage
{
Rel_val (child_node):= Relevancy(desired_topic, webpage);   // Calculate Rel_val
Insert_Frontier (child_node, Rel_val);   // Add new links with Rel Val into frontier
}
End While Loop
```

Figure 6.5: Pseudo- Code of Naive Best First Approach

We start with a set of Seed URLs; initially the relevancy value is 1 for all seed URLs. These URLs are added to the Frontier. Priority Queue is used to implement the Frontier of any Best First Algorithm. Page is fetched and new links (child nodes are fetched) are assigned a relevancy value according to their relevance with desired keywords. This relevancy value is assigned to the newly found links in it, and these newly found URLs are inserted into the Frontier along with their respective relevancy values. Again the URL with highest relevancy is picked from the Frontier and this procedure goes on adding more and more new pages into the Frontier.

6.3.3 Pseudo-Code of Fish Search Algorithm [19]

```
Fish Search ( Initial node(seed) , depth (D) , the time limit, search query)
{
Set Depth of the initial node = D, and Insert into an empty list;
Repeat While (Frontier! = Empty)
current_node:= First_Frontier(URL);
Webpage: = Fetch (Current_Node);     // Open Webpage corresponding to selected link
Rel_val := Relevancy(desired_topic, current_node);    // calculate relevancy(0 or 1)
If depth(current_node) > O
{
If (current_node is relevant)          // i.e. relevancy of current_node is 1

Repeat For (each child_node of current_node)
{
Insert_Frontier (child_node);
Set potential_score(child_node):=1;
Set depth(child_node)=D;
}

Else                                          // if current_node is irrelevant
 Repeat For (each child_node of current_node)
{
Insert_Frontier (child_node, last);        // insert all children into the last
Set potential_score(child_node)= 0;
depth(child node)=depth(current_node)- 1;     // decrease the searching depth
}
End While Loop
```

Figure 6.6: Pseudo-code of Fish Search Approach

6.3.4 Modified Best First Algorithm

The Pseudo-code of Modified Best First approach can be stated as below:

```
Modified_Best_First_Algo(seed URLs)      //  Start with given Seed URLs as input

Insert_Frontier (seed URLs, 0);    // Insert seed URLs along with their relevance Val=0

While (Frontier! = Empty)                        //  Perform Recursively following steps

Link: =Remove_Highest_Frontier (URL);    // Pick link with highest rel_val from frontier
Webpage: = Fetch (Link);                    // Open Webpage corresponding to selected link
Rel_val (page) := Relevancy(desired_topic, webpage);
Repeat For each child_node of Webpage
{
Rel_val (child_node):= Relevancy(Parent);          // Calculate Rel_val
Insert_Frontier (child_node, Rel_val);              // Add child with val into frontier
}

End While Loop
```

Figure 6.7: Pseudo-code of modified Best Approach

As cleared from the pseudo-code above, the initial SEED URLs are assigned relevancy value 0 and are inserted into the Frontier. Until the Frontier gets empty, the URL with highest relevancy value is fetched. Each URL is traversed and checked for relevancy. The relevancy score of the node is assigned to each of its child nodes. Along with respective relevancy values inherited from parent node, new children URLs are inserted into the Frontier.

6.4 Implementation Results

After implementing the Pseudo-codes given in the previous section, we have achieved the following results. Referring to **Figure 6.3**, the input graph is used and various results are noted according to our input.

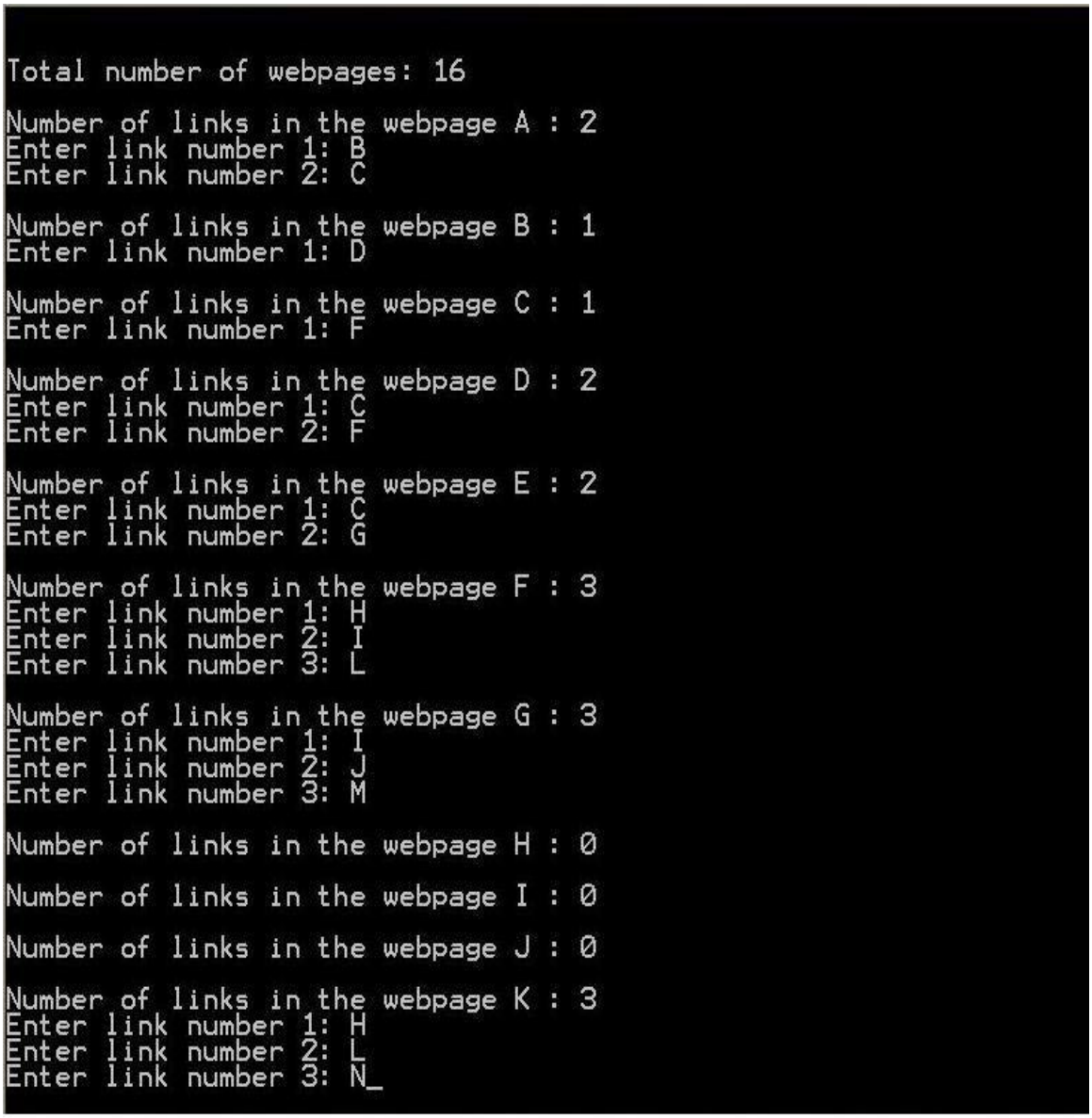
Page number	Node Label	Approachable Pages
1	A	B , C
2	B	D
3	C	F
4	D	C , F
5	E	C , G
6	F	H , I , L
7	G	I , J , M
8	H	-
9	I	-
10	J	-
11	K	H , L , N
12	L	H , I
13	M	L
14	N	M
15	O	P
16	P	O

Figure 6.8: Adjacency Table of input Graph

6.4.1 Input of Graph

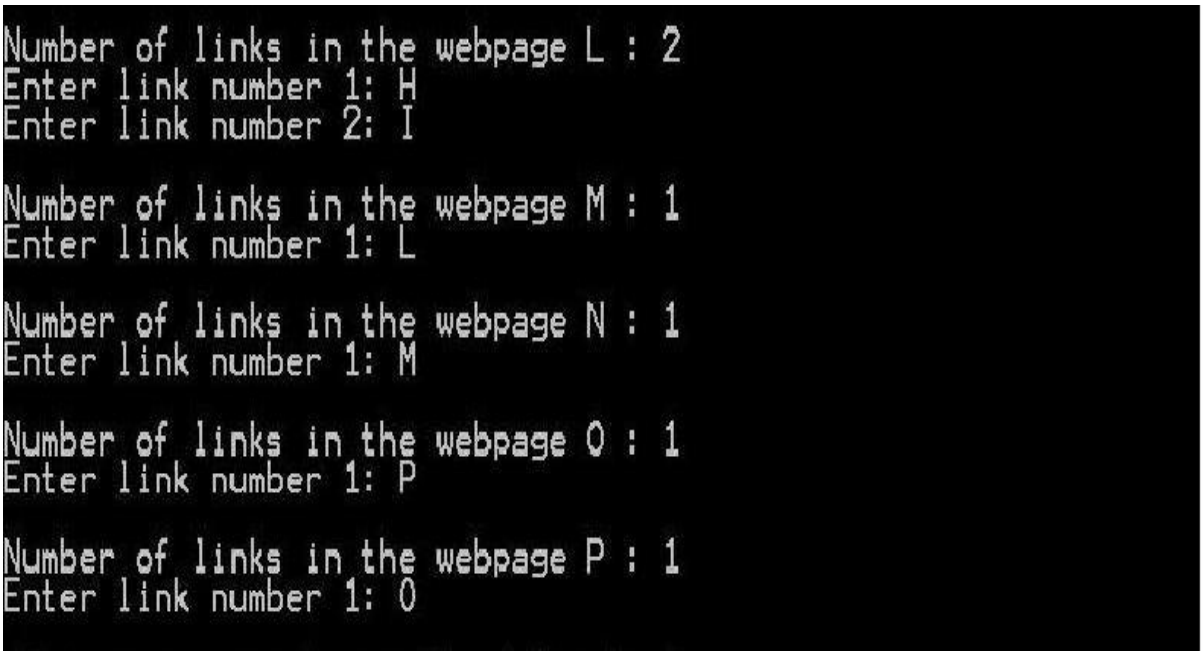
The given graph is input into the memory using its Adjacency Table as shown in Figure 6.8.

The input of the graph is shown in the following screenshot:



Screenshot 6.1: Input of Graph along with Adjacency Table

Input of Graph (continued…)



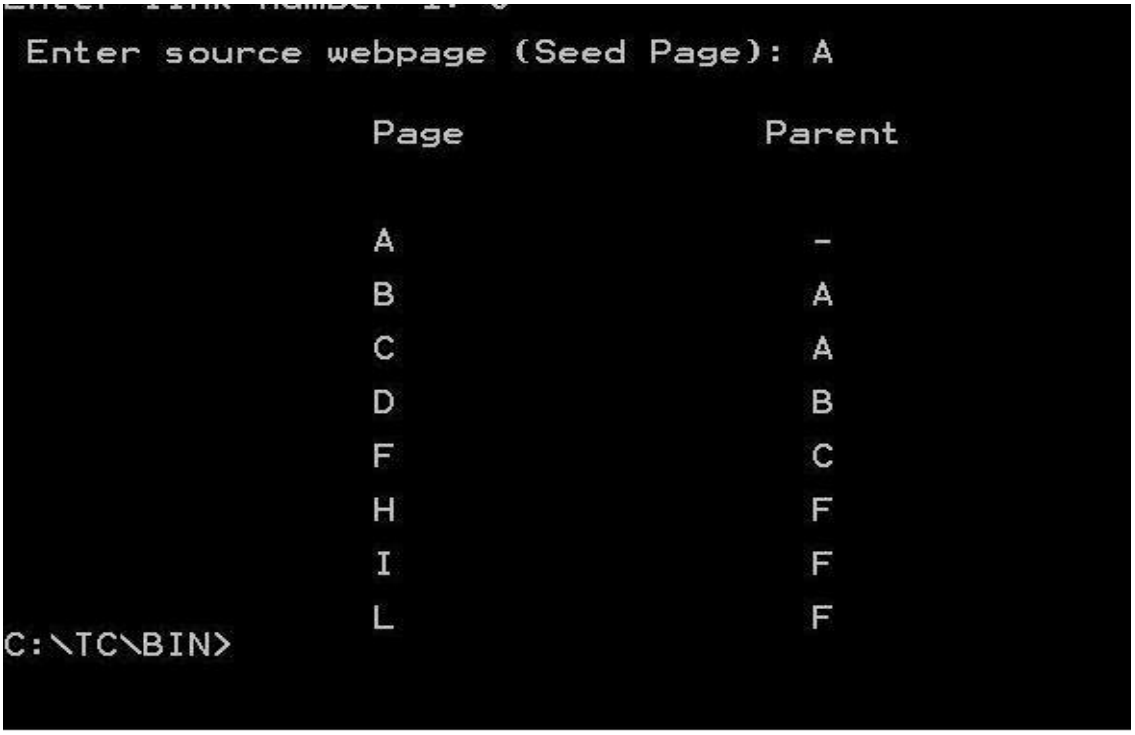
Screenshot 6.2: Input of Graph (continued...)

Thus in this way, the entire graph is input in to the memory along with its adjacency Table.

Now we will execute 4 crawling programs on this input and the corresponding results for each program are noted.

6.4.2 Output Results for Breadth-First Search

When we select **node A** as starting SEED, the following nodes are approached by the crawler along with their sequence.



Screenshot 6.3: Output result for BFS

The working of any Breadth-First algorithm is very simple. It simply works of first come first serve. We start with page A. After processing A, its child nodes B and C are inserted into the Frontier. Again the next page is fetched from the Frontier and is processed, its children inserted into Frontier and so on. This procedure continues until the Frontier gets empty.

Breadth-First is used here as a baseline crawler; since it does not use any knowledge about the topic, and its performance is considered to provide a lower bound for any of the more sophisticated algorithms.

6.4.3 Output Results for Naïve Best First Search

The same input graph when uses the Naïve Best First approach for crawling, the following output results:

```
Enter source webpage (Seed Page): A

      Page              Relevancy of Page          Parent Page
       A                 2.000000                   -
       C                 1.000000                   A
       F                 1.000000                   C
       L                 1.000000                   F
       H                 0.800000                   F
       I                 0.800000                   F
       B                 0.100000                   A
       D                 1.000000                   B
```

Screenshot 6.4: Output Results for Naïve Best First approach

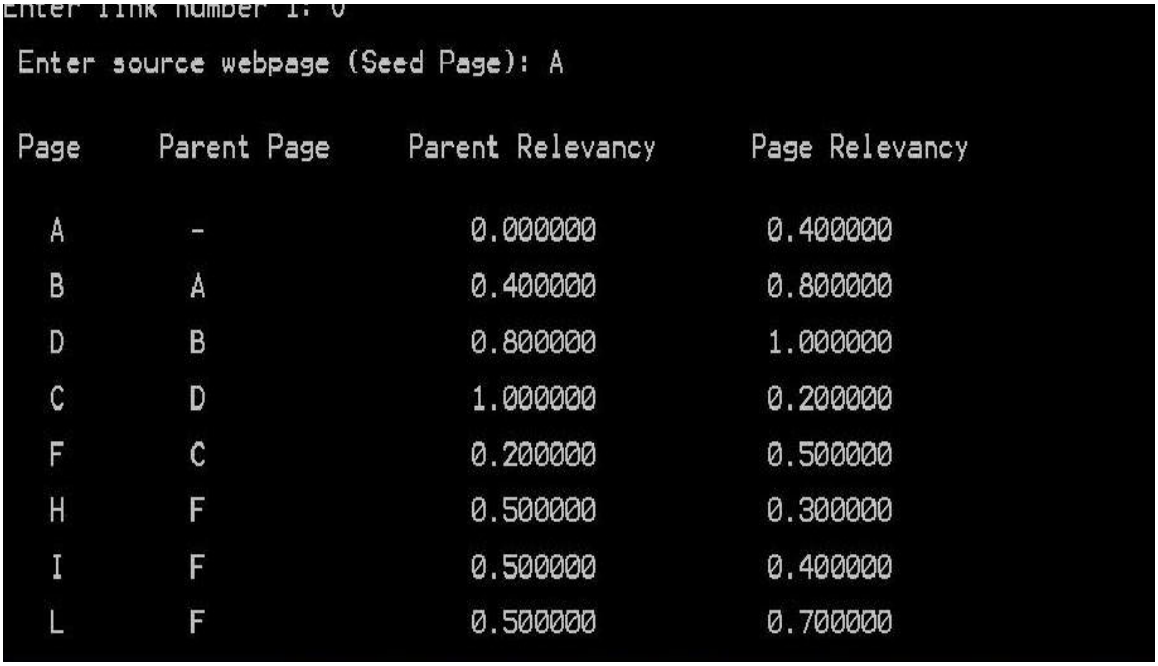
Note that in naïve best first, the preference of next page to be approached depends upon the relevancy of that page.

This output screenshot depicts the idea behind working of any best first approach. We start traversing from Page A. Its relevancy is set highest (2 in this case). Now the relevancy of A's children B and C comes out to be 0.1 and 1.0 respectively. Along with respective relevancies, B and C are inserted in frontier. Every time, the page with highest relevancy value is picked from the Frontier.

6.4.4 Output Results for Modified Best First Search

In our modified Best First Approach, the Relevancy Score of the Parent Page is also associated with the current page.

The output screen of the results is as shown below:



Screenshot 6.5: Output Results for Modified Best First approach

As cleared from the output screenshot, the parent relevance decides which page will be selected next. The page with the highest parent relevance value is selected from the Frontier every time.

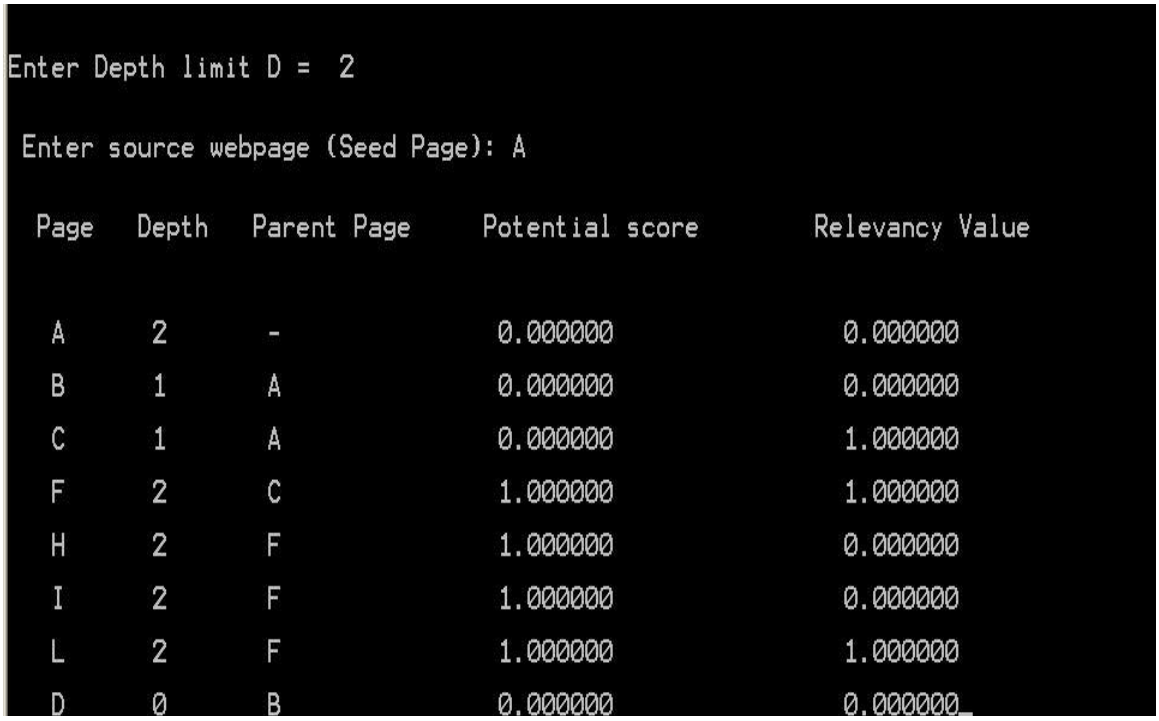
We start with A. since it is the starting node, its parent is nothing and parent relevancy is also 0. Its own relevancy comes out to be 0.4.

Page A has 2 children – B and C. So both B and C are assigned parent relevance 0.4. (Note that the parent relevancy calculation is dynamic. i.e. if a page is traversed again from another parent, its parent value will depend on that new parent page and corresponding parent relevance will also change dynamically)

As here, C is approached again from Page D, since the relevancy of D comes out to be 1.0, t**he parent relevancy of C changes from 0.4 to 1.0.**
Thus our modified program considers the dynamic nature of the web where contents of the pages could change anytime and thus the relevancy also gets affected.

6.4.5 Output Results for Fish Search

A very good and efficient approached as discussed in previous chapters is – Fish Search given by De Bra. It is used to traverse up to some pre-defined depth. Also the relevancy is checked along with going into the depth of the graph. The output screen for Fish Search is shown below:



Screenshot 6.6: Output of Fish Search approach

We start with Depth =2, check relevancy of A, its 0. so the potential score of its children B and C is set to 0 and the depth is set 1 for both. Whenever a page is relevant, the potential score of its children is set to 1 and depth is set to 2. Otherwise, potential score of children is set to 0 and depth is decremented by 1. As soon as depth reaches 0, we stop traversing.

Chapter 7

Conclusion & Future Work

Internet is one of the easiest sources available in present days for searching and accessing any sort of data from the entire world. The structure of the World Wide Web is a graphical structure, and the links given in a page can be used to open other web pages. In this thesis, we have used the graphical structure to process certain traversing algorithms used in the search engines by the Crawlers. Each webpage can be considered as node and hyperlink as edge, so the search operation could be abstracted as a process of traversing directed graph. By following the linked structure of the Web, we can traverse a number of new web-pages starting from a Starting webpage. Web crawlers are the programs or software that uses the graphical structure of the Web to move from page to page.

In this thesis, we have briefly discussed about Internet, Search Engines, Crawlers and then Crawling Algorithms.

There are number of crawling strategies used by various search engines. The basic crawling approach uses simple Breadth First method of graph traversing. But there are certain disadvantages of BFS since it is a blind traversing approach. To make traversing more relevant and fast, some heuristic approaches are followed. The results of all the crawling approaches are giving different results.

These heuristic searches keep a check on the relevancy factor of every page to be traversed. Thus the efficiency of the database of the search engine increases and only relevant pages are stored in it.

The fast and more accurate version of Fish Search is known as – Shark Search, which is probably the best crawling approach. But there are some issues related with the implementation of Shark Search Approach in simple programming environment as there is lot of factors, calculations associated with it.

In future, work can be done to improve the efficiency of algorithms and accuracy and timeliness of the search engines. The work of Shark Search can be extended further to make Web crawling much faster and more accurate.

References

[1] World Internet Usage and population statistics available at-
<< http://www.internetworldstats.com/stats.htm >>

[2] Arvind Arasu, Junghoo Cho, Andreas Paepcke: "Searching the Web**",** Computer Science Department, Stanford University.

[3] David Hawking, "Web Search Engines: Part 1", June 2006, available at-
<< http://ieeexplore.ieee.org/iel5/2/34424/01642621.pdf >>

[4] Web search engine architecture image available at-
<< http://www.ibm.com/developerworks/web/library/wa-lucene2/ >>

[5] Gautam Pant, Padmini Srinivasan, and Filippo Menczer: "Crawling the Web" available at- << http://dollar.biz.uiowa.edu/~pant/Papers/crawling.pdf >>

[6] Marc Najork, Allan Heydon SRC Research Report 173, "High-Performance Web Crawling", published by COMPAQ systems research center on September 26, 2001.

[7] Sergey Brin and Lawrence Page, "The anatomy of a large-scale hyper textual Web search engine", In *Proceedings of the Seventh International World Wide Web Conference*, pages 107–117, April 1998.

[8] Sandeep Sharma and Ravinder Kumar, **"**Web-Crawlers and Recent Crawling Approaches", in International Conference on Challenges and Development on IT - ICCDIT-2008 held in PCTE, Ludhiana (Punjab) on May 30th, 2008

[9] Junghoo Cho, Hector Garcia-Molina: "Parallel Crawlers**",** 7–11 May 2002, Honolulu, Hawaii, USA.

[10] Articles about Web Crawlers available at-
<< http://en.wikipedia.org/wiki/Web_crawler#Examples_of_Web_crawlers >>

[11] Marc Najork, Janet L. Wiener," Breadth-first search crawling yields high-quality pages", WWW10 proceedings in May 2-5, 2001, Hong Kong.

[12] Sergey Brin and Lawrence Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", Computer Science Department, Stanford University, Stanford, CA
Available at- << http://www.his.se/upload/51108/google.pdf >>

[13] Filippo Menczer, Gautam Pant and Padmini Srinivasan, "Topical Web Crawlers: Evaluating Adaptive Algorithms" ACM Transactions on Internet Technology, Vol. 4, No. 4, November 2004, Pages 378–419. Available at-
<< http://www.informatics.indiana.edu/fil/Papers/TOIT.pdf >>

[14] P. De Bra, G. Houben, Y. Kornatzky and R. Post, "Information Retrieval in Distributed Hypertexts", in Proceedings of the 4th RIAO Conference, 481 - 491, New York, 1994.

[15] M. Hersovici, M. Jacovi, Y. Maarek, D. Pelleg, M. Shtalhaim and S. Ur. "The Shark-Search Algorithm – An Application: Tailored Web Site Mapping", In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998. Available at- << http://www7.scu.edu.au/1849/com1849.htm >>

[16] Sriram Raghavan, Hector Garcia-Molina, "Representing Web Graphs", Stanford University, CA – June 2002.
Available at- << http://www.almaden.ibm.com/cs/people/rsriram/pubs/icde03.pdf >>

[17] A. Z. Broder, S. R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener, “Graph structure in the web” in Proc. Of WWW Conf., 2000 Available at < http://www.cis.upenn.edu/~mkearns/teaching/NetworkedLife/broder.pdf>

[18] Input Web Graph available at-
<< http://www.openarchives.org/ore/0.1/datamodel >>

[19] Dr. P.M.E. De Bra, Drs. R.D.J. Post, "Searching for arbitrary information in the WWW: the fish-search for Mosaic." Available at-
<< http://archive.ncsa.uiuc.edu/SDG/IT94/Proceedings/Searching/debra/article.html >>

[20] Blaz Novak, “A survey of focused web crawling algorithms”, Department of Knowledge Technologies, Jozef Stefan Institute, Ljubljana, Slovenia

[21] Edleno S. de Moura, Daniel R. Fernandes, Altigran S. Silva, “Improving Web Search Efficiency via a Locality Based Static Pruning Method”, *WWW 2005*, May 10-14 2005, Chiba, Japan.

[22] “Graph Data structure Introduction “, available at-
<< hamilton.bell.ac.uk/swdev2/notes/notes_18.pdf >>

Paper Published

1. Sandeep Sharma and Ravinder Kumar, **"Web-Crawlers and Recent Crawling Approaches",** Accepted and Published in International Conference on Challenges and Development on IT (ICCDIT-2008) held in PCTE, Ludhiana (Punjab) on May 30th,2008