San Jose State University

SJSU ScholarWorks

A Smart Web Crawler for a Concept Based Semantic Search Engine

Vinay Kancherla
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Databases and Information Systems Commons

A Smart Web Crawler for a Concept Based Semantic Search Engine

Presented to

The Faculty of Department of computer Science

San Jose State University

In Partial Fulfillment of the

Requirements for the

Degree Master of Computer Science

By

Vinay Kancherla

Fall 2014

SAN JOSE STATE UNIVERSITY

The Designated Thesis Committee Approves the Thesis Titled

A Smart Web Crawler for a Concept Based Semantic Search Engine

by

Vinay Kancherla

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2014

_____

Dr. T. Y. Lin, Department of Computer Science   Date

_____

Dr. Suneuy Kim, Department of Computer Science   Date

_____

Mr. Eric Louie, DBA at IBM Corporation   Date

ABSTRACT

A Smart Web Crawler for a Concept Based Semantic Search Engine

By Vinay Kancherla

The internet is a vast collection of billions of web pages containing terabytes of information arranged in thousands of servers using HTML. The size of this collection itself is a formidable obstacle in retrieving information necessary and relevant. This made search engines an important part of our lives. Search engines strive to retrieve information as relevant as possible to the user. One of the building blocks of search engines is the Web Crawler. A web crawler is a bot that goes around the internet collecting and storing it in a database for further analysis and arrangement of the data.

The project aims to create a smart web crawler for a concept based semantic based search engine. The crawler not only aims to crawl the World Wide Web and bring back data but also aims to perform an initial data analysis of unnecessary data before it stores the data. We aim to improve the efficiency of the Concept Based Semantic Search Engine by using the Smart crawler.

Contents:

List of Figures:

List Of Tables:

1. Introduction

The internet is a vast collection of billions of web pages containing terabytes of information arranged in thousands of servers using HTML[1]. The size of this collection itself is a formidable obstacle in retrieving necessary and relevant information. This made search engines an important part of our lives. Search engines strive to retrieve information as relevant as possible. One of the building blocks of search engines is the Web Crawler.

A web crawler is a program that goes around the internet collecting and storing data in a database for further analysis and arrangement. The process of web crawling involves gathering pages from the web and arranging them in such a way that the search engine can retrieve then efficiently. The critical objective is to do so efficiently and quickly without much interference with the functioning of the remote server. A web crawler begins with a URL or a list of URLs, called seeds. The crawler visits the URL at the top of the list. On the web page it looks for hyperlinks to other web pages, it adds them to the existing list of URLs in the list. This methodology of the crawler visiting URLs depends on the rules set for the crawler. In general crawlers incrementally crawl URLs in the list. In addition to collecting URLs the main function of the crawler, is to collect data from the page. The data collected is sent back to the home server for storage and further analysis.

The Smart Crawler for the Concept based Semantic Search Engine described here crawls the internet collecting web pages and storing them in the form of text files. This is because the Concept based Semantic Search engine will take inputs only in the form of text files. In order to improve the efficiency of the engine the Smart crawler filters the text before storing them. In addition to filtering, the Smart crawler skips crawling URLs of files like image or mp3 files that contain non-textual data. The main contribution of the Smart crawler in comparison to the existing crawler is that the Smart Crawler performs an advanced level of data analysis on the data extracted from the web. It also follows a breadth first mode of link traversal unlike the existing crawler. The Smart crawler also has the ability to filter out URLs like image and multimedia that do not contain any useful text information. The Smart crawler also effectively manages the metadata by systematically storing it. In this section we describe characteristics and classification of crawlers. We also give some background information about the programming language used to build the crawler. The second section describes the architecture of the Smart crawler. The third section describes the implementation details of the crawler. The results section compares the performance of the Concept Based semantic Engine when input from the Smart Crawler is given and from the existing crawler is given. This proves how the techniques used to analyze and extract data from HTML pages improve the efficiency of the Concept based Semantic Search engine.

1.1 Characteristic features of Crawlers

Crawlers that crawl the internet must have the following basic features so that they serve their purpose, the well being of the servers that hold data and also the web as a whole. [2]

1.1.1 Robustness

The web contains loops called spider traps, which are meant to mislead the crawler to recursively crawl a particular domain and get stuck in one single domain. They generate an infinite loop of web pages that lead to nowhere. The crawler needs to be resilient to such traps. These traps may not always be designed to mislead the crawler but may be a result of faulty website development.

1.1.2 Politeness

Web servers have policies regulating when a crawler can visit them. These politeness policies must be respected. A server is meant to serve other requests that it is originally designed to serve. Hindering the server may lead to blocking of the crawler by the server altogether. So it is better to respect the policies of the server.

1.1.3 Distributed

The crawler should be able to function in a distributed fashion. It could have multiple images of itself working parallelly in proper coordination to crawl the internet as quickly as possible.

1.1.4 Scalable

The crawler should be scalable. It should have the flexibility to add new machines and extra bandwidth whenever necessary.

1.1.5 Performance and Efficiency

The use of system resources like processing power, network bandwidth and storage should be judicious. These factors determine how efficient the crawler is.

1.1.6 Quality

The crawler should be able to differentiate between information that is useful and information that is not. As servers mainly serve other requests that contain a lot of information that may not be useful. Crawlers should filter out this content.

1.1.7 Freshness

In many situations, crawlers will need to crawl the same pages again in order to get new content from the old page. For this reason, crawlers should be able to crawl the same page at a rate that is approximately equal to the rate of change of information on the page. Thus, the crawler will be able to make sure that the concepts on the search engine are the latest and relevant to the present context.

1.1.8 Extensible.

The crawlers should be able to adapt to the growing number of data formats that it will encounter on web sites. It also needs to cope up with the new protocols that may be used on some servers.

1.2 Challenges for a Crawler

The basic algorithm for a crawler will look simple at first glance. It simply involves going to a URL, getting URLs from the web page and iteratively going to each of those URLs, get data from them and store the data. Despite the simplicity, there are some intense challenges that we need to tackle to make a crawler. The following are the challenges for a crawler. [3]

1.2.1   Scale

The internet is an enormous evolving and ever growing system. The crawlers need to have good freshness; seek broad coverage and extremely high throughput. It is difficult to achieve all of these at the same time. Modern companies use complex high-speed network links and processing power but expert programmers still fall short of a perfect web crawling bot.

1.2.2   Content selection tradeoff

Even the best crawlers do not intend to crawl the whole internet, or keep up with all the changes that happen on the web. Instead, crawling is performed selectively and in a controlled order. The goal is to acquire valuable content quickly, bypass low-quality, irrelevant, redundant, and malicious content and ensure eventual coverage of all reasonable content. The crawler must obey constraints such as per-site rate limitations while balancing competing objectives such as coverage and freshness. A trade-off must be achieved between exploitation of content already known to be useful and exploration of potentially useful content.

1.2.3 Social Obligation

Crawlers should not impose too much burden on the web to crawl web pages. They should be good bots. They should not over burden the servers hosting the data in times when the load is heavy and lead to crashing of services. They should follow the rules of the servers. Crawlers if used without the right safety mechanisms can even cause denial-of-service attacks.

1.2.4 Adversaries

Some content providers seek to inject misleading content or useless information into the data assembled by the crawler. This is often done for monetary benefits like commercial advertisements. This may also be done by other competing data miners to outsmart other aggregators.

1.3 Types of Web Crawlers.

Crawlers are broadly classified into four types. The following are the general classification of crawlers[4].

1.3.1 Focused Crawler

Focused Crawler is a web crawler for downloading pages that are related to a specific area of interest. It collects the documents that are focused and relevant to a given topic. It is also called as a Topic Crawler because of the way it works. The focused crawler determines the relevance of the document before crawling the page. It estimates if the given page is relevant to a particular topic and how to proceeds. The main advantage of

this kind of crawler is that it requires less hardware and network resources and so it costs less. It also keeps a check on network traffic.

1.3.2 Traditional Crawler

A traditional crawler periodically crawls the already crawled URLs and replaces the old documents with the newly downloaded documents to refresh its collection. On the contrary, an incremental crawler refreshes incrementally the already existing collection of pages by visiting them frequently. This is based upon an estimation of the rate at how often pages change. It also replaces old and less important pages by new and more relevant pages. It resolves the problem of freshness of the data. The advantage of incremental crawler is that only valuable data is provided to the user. Thus we save network bandwidth and also achieve data enrichment.

1.3.3 Distributed Crawler

Distributed computing technique is the main foundation for distributed web crawling. Many crawlers are working at the same time in tandem and distribute the work load of crawling the web in order to have maximum coverage of the internet. A central server manages the communication, synchronization of nodes and communicates between the different bots. It is also geographically distributed. It primarily uses Page rank algorithm to increase efficiency and quality of search. The advantage of distributed web crawler is that it is robust. It is resistant to system crashes and other events, and can adopt to various crawling requirements.

1.3.4   Parallel Crawlers

Parallel crawlers are multiple crawlers running at the same time. It consists of multiple crawling processes called as C-procs which can run on network of workstations. The Parallel crawlers depend on page selection and page freshness. A Parallel crawler can be distributed at geographically distant locations or be on a local network. Parallelization of crawling system is vital from the point of view of downloading documents in a reasonable amount of time.

1.4   Related Work

There are many crawlers written in every programming and scripting language to serve a variety of purposes depending on the requirement, purpose and functionality for which the crawler is built. The first ever web crawler to be built to fully function is the WebCrawler in 1994. Subsequently a lot of other better and more efficient crawlers were built over the years. [5] The most notable of the crawlers currently in operation are as follows.

- Googlebot: The Google search uses this crawling bot. It is integrated with indexing process as parsing is done for URL extraction and also full text indexing. It has a URL server that exclusively handles URLs. It checks if the URLs have previously been crawled. It they are not crawled they are added to the queue.

- Bingbot: The Bingbot is the crawler that the Microsoft owned search engine Bing Search uses to crawl the web and collect data. It was previously known as Msnbot.

- FAST Crawl: This is the web crawler that the Norway based Fast Search and Transfer uses. It focuses on data search technologies. It was first developed in 1997 and is periodically re-developed based on latest technologies.

- WebRACE: It is a Java based crawler. It acts in part as a proxy server as it gets requests from users to download pages. When pages change they are crawled again and the subscriber is notified. The feature of this bot is it does not need a set of seeds to start crawling.

- WebFountain: It is a distributed crawler written in C++. It has a controller and ant machines that repeatedly download pages. A non-linear programming method is used to solve freshness maximizing equations.

We also have a lot of open source crawlers that are available online and can be used according to needs for non commercial purposes.

- DataparkSearch: This is a search engine with a crawler under GNU.

- GNU Wget: It is a command line operated crawler in C language. It is usually used to mirror web and ftp sites.

- GRUB: Wiki search uses this to crawl the web.

- Heritrix: This is a Java based crawler used for archiving large portions of the web.

- HTTrack: It is a C language based crawler that creates an image of web pages to access then offline.

- ICDL Crawler: A cross-platform web crawler in C++

- mnoGoSearch: A c based crawler and indexer with a search engine

- NOrconex HTTP Collector: A java based crawler that helps Enterprise Search Integrators.

- Nutch: A Java based crawler used in with a text-indexing package.

1.5 Python Programming Language

Python is currently one of the world's most used programming languages. Its syntax allows programmers to write code in far fewer lines than languages like C or Java. Code readability is the main focus of its design philosophy. Its constructs enable clean programming. Python supports a wide range of paradigms like imperative and functional programming, object-orientation and procedural styles. It has automatic memory management, dynamic type system and a comprehensive standard library. [6] Python can be installed on many operating systems allowing python to be executed on a majority of systems. Third party installers are also available for every popular operating system for python which can be executed to install python.

Python is both an object-oriented programming and structured programming. It also has a large number of features that support aspect-oriented and functional programming.

1.6 Features of Python Language

The following are the features of python programming language that sets it apart from other languages. [7]

1.6.1 Simple

Python is a surprisingly simple language. Python code is easy to understand almost like English. One of the greatest strengths is its pseudo code nature, it allows programmer to concentrate on finding the solution of the problem without worrying much about the language specifics.

1.6.2 Portable

As Python is open-source, it has been ported to many platforms. It is made to work on many platforms. Python programs do not need any changes on different platforms if we can avoid system-dependent features. We can use Python on Windows, Linux, PlayStation, Sharp Zaurus, Windows CE, AROS, AS/400, BeOS, OS/390, z/OS or even PocketPC.

1.6.3 Free and Open Source

Python is a FLOSS (Free/Librê and Open Source Software). You can freely read its source code, distribute copies of this software, and use pieces of it in new free programs, make changes to it. This is one reason python has been constantly improving over the years. Free access has allowed people to develop python into a language with full fledged libraries.

1.6.4 **Interpreted**

Python, unlike C or C++, does not need compilation to binary. Compilation is a process of conversion of source language to a language understandable by the computer. Usually a compiler does this job for languages like C or C++. But python runs the program directly from source code. It internally converts the code in byte code before running it converts it into the native language of the computer. So we don't have to worry about compiling, linking libraries, etc. This also makes python potable.

1.6.5 **High level Language**

We don't have to worry about the low level details like memory management of the programs as Python is a high level language. It handles memory by itself.

1.6.6 **Extensive Libraries**

Python has a huge Standard Library. We can do various things involving unit testing, documentation generation, regular expressions, HTML, WAV files, XML, XML-RPC, HTML, cryptography, GUI (graphical user interfaces) WAV files, threading, databases, web browsers, CGI, FTP, email,, Tk, and other system-dependent stuff. This is called the 'Batteries Included' Python philosophy. All of this is installed when Python is installed. Apart from this there are many high quality libraries like Python Imaging Library, Twisted etc.

1.6.7 Object Oriented

Python supports object- oriented as well procedure-oriented programming. The program is built around functions which are nothing but reusable pieces of code in procedure oriented language. And in object-oriented languages, the program is built

around objects that combine data and functionality. Python has a very simple but powerful way of doing object oriented programming when compared to languages like C++ or Java.

1.6.8 Easy to Learn

Python is a language that is really simple to learn. Even beginners will find it simple to start learning.

1.6.9 Extensible

Python is extensible. If there is any part of code or algorithm that is available in other languages, that part of code can be extended and used in python.

1.6.10 Embeddable

Python code can be embedded in other languages like C or C++ and the scripting capabilities of python can be used in those languages.

1.7 Scope and requirements of the Project

The primary purpose of the project is to make a crawler that can provide text files to the Concept Based Semantic Search Engine [8]. The text files are meant to be the input for the Search engine which will try to analyze the data and extract meaningful concepts from the data and store them in an SQL Database. The crawler will extract text data from the data obtained from crawling and create text files with the data. The crawler also aims to systematically store metadata in a different set of files for future use. The crawler thus aims to improve the efficiency of the Concept Based Semantic Search engine.

2. Architecture of the Smart Crawler

The architecture of the smart crawler as shown in the figure below consists of the following components:

1. Data Extractor
2. Extracted URL stack
3. Domain filter
4. URL classifier
5. Valid URL list
6. HTML Analyzer
7. Script Pruning
8. Text extractor
9. Metadata formatter
10. Data Files

2.1 Data Extractor

The data extractor is the component of the crawler that in the first iteration gets the URL from the user and uses the URL to access the remote server on which the URL is hosted. This module sends a http request to the remote server just like any other http request to the server. The server responds to the request by sending back the requested information. In this case the requested information is the page located in the URL. Now it is the job of the data extractor to scan through the data and find all the URLs that are in the data. It searches the obtained data for the links to other pages and supplies them to the

Initial URL stack. It is also the responsibility of the Data Extractor to supply data to the Data Analyzer for further analysis of the data. The Data Extractor runs iteratively for each URL that the Valid URL list supplies to it. The Data Extractor is the main component of the crawler.
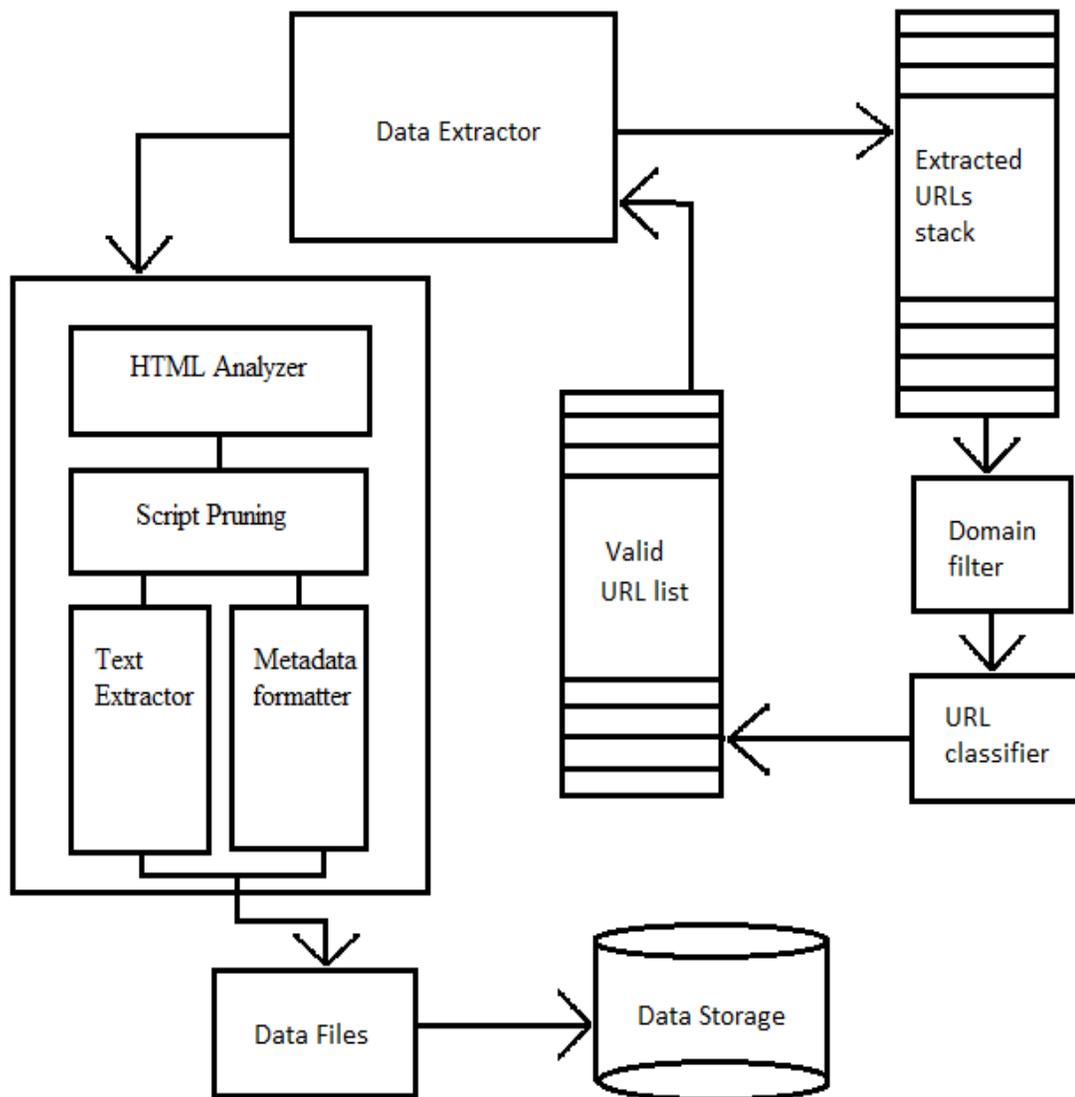

Figure 2.1: Architecture of the Smart crawler

2.2 Extracted URL Stack

The Extracted URL Stack stores the URLs that the Data Extractor finds on the pages that it extracts. The URL stack is actually a queue data structure. So it has a first in first out (FIFO) mechanism. The job of the Extracted URL stack is to supply URLs for further processing. It sends each URL if a first in first out order to the Domain Filter. So the URL Stack first stores the links from each page and forwards them for processing.

2.3 Domain Filter

Domain Filter checks whether the URL it gets from the URL stack belongs to the given domain or not. This is necessary to restrict the crawler to the specified domain. The links that the data extractor extracts will belong to servers all over the world. But if we need data of only the given domain, this Classifier filters out all the other URLs. But if we want data from all the links in the page we can also bypass this filter. The user will be given an option to choose if he wants to crawl the entire web or only the given domain. If he does not specify that he wants to crawl only the given domain this filter will be bypassed. After filtering out the other domain URLs the Domain Filter sends the remaining URLs to the URL Classifier for further processing. This component handles one URL at a time.

2.4 URL Classifier

The URL Classifier is the component that checks whether the URL is a useful URL or not. The URLs that the extractor extracts will contain all kinds of files like jpeg, css, img, js, etc. These files are useless to crawl as they will not contain any text data. They are files that are meant for styles or adding functionality to the pages. They won't have any data and need to be pruned. This classifier checks the MIME type of the file in the URL. If the MIME type is a HTML or txt, the Classifier considers the url for next stage. If the URL belongs to any other multimedia data type, it simply filters the URL out of the stack. After this stage the URLs that have been screened and have proved to be useful will be stored in another URL holding list called Valid URL list. It is the responsibility of the URL Classifier to populate the Valid URL List.

2.5 Valid URL List

The Valid URL list contains urls that have been processed and are ready to be given to the extractor. The urls in the Valid URL List have been checked if they belong to the same domain, if they contain useful text information and if they have already been crawler. It a URL passes all the above tests it if a URL that is fit to be crawled and it is stored in this URL list. Another feature on this list is that it avoids redundancy of URLs. It makes sure that the content in the list, in this case the URLs are unique. It is this list that supplies valid URLs to the data extractor.

2.6 Data Analyzer

The Data Analyzer is the component that is responsible for processing the data extracted by the data extractor. This Data Analyzer gets raw data from the extractor and processes it to get some meaningful text information. This has four important sub components The HTML tag Analyzer, the Java Script pruner, the Text Extractor and the Metadata Formatter . This component also writes the processed data to the .txt files on the hard disk.

2.8 HTML Tag Analyzer

An html page contains a lot of html tags which contain information that is necessary to format the html page and makes it render properly on the browser. But all those tags contain information contain tags that are unnecessary for the data analyzer. Only some tags like the paragraph tag or the header tag contain useful and meaningful information. So in this component we remove all that tags from the raw data that the extractor receives. At this stage most of the data is polished but still some traces of garbage data still remain and need to be removed. So this component sends the semi-pruned data to the Java Script Remover for get rid of all the java script in the data.

2.9 Java Script Pruning

The Java Script that is left over in the data after all the tags are removed is filtered from the data in this component. This component scans the script elements from the data and removes them. The residual data sent to the text extractor

2.10 Text Extractor

This component extracts important text data from the body part of the HTML page. We internally parse the data to a parse tree using the BeautifulSoup python module. From this parse tree we extract all the paragraph tags that are in the body of the HTML page. We filter texts from all the remaining tags that contain texts. This is because we consider that the paragraph tags in the body contain text that is unique. Rest of the tags like title tag could be same throughout a given domain. We then write this data to text files, one text file for each URL, that will be given to the Concept based engine as input.

2.11 Metadata Formatter

This component also utilizes the parse tree of the html tags and tries to convert the HTLM page into a fixed format which is a requirement of the project. It re-formats the contents of the HTML page. This component will create a separate text file for each URL. This text file will contain information about the data in the first text file and will be stored in a different directory but with the same file name. All of these files like will contain the URL of the page the crawler crawls in the first line. The second line will contain the title of the page. This will be followed by the data from the meta tags that

contain information about the data in the page. From the metadata section we use a five star delimiter at the starting of each section. The metadata section is followed by the important keyword section followed by the image information section. The multimedia information section comes after the image information part. The links information section, the table information and the summary follow the image information section. If a page does not contain any of the sections the crawler simple skips that section and goes to the next section. This component handles every HTML tag to place the content from that tag in the appropriate section.

The main purpose of this re structuring is to provide the Concept based semantic engine with information about the data in the input text files. For example if the engine wants information regarding multimedia from a particular file that it processed, the engine goes to the metadata directory, open the file with the same file name and searches for the five star delimiter followed by multimedia information(*****MULTIMEDIA INFORMATION) to quickly get the information. If it wants the URL of the file then it goes to the first line of the file. However, the Concept based engine, in its present state, will not be able to use these files as it is built only to take input text files and process them. These metadata files are a requirement of the project and will be used in the future.

2.12 Text Files

The text files contain data from each URL from the text extractor. These text files serve as input to the Concept Based Semantic Search Engine. They are stored on the hard disk at a location known to the Semantic Search Engine.

3. Implementation of the Smart Crawler

3.1 System Models: Unified Modeling Language

The Unified Modeling Language (UML) is a general purpose graphic language used by software professionals for specifying, visualizing, constructing, and documenting the artifacts of a software intensive system. It is the standard language for writing software blueprints. [9]

The UML has three main models:

1. The User Model

2. The Object Model

3. The Dynamic Model

3.1.1 The User Model

The user model consists of Use Case Diagrams which are used to graphically describe the interaction of the user with the system. They represent actors, activities and their relations. They are primarily used to gather the requirements of the system to be built. These include both internal and external requirements and mostly design related. So when we analyze a system we identify functionalities and actors.

In our Smart crawler system we have one use case as the user here is ususlly a programmer. The following diagram shows the use cases of the Smart crawler.
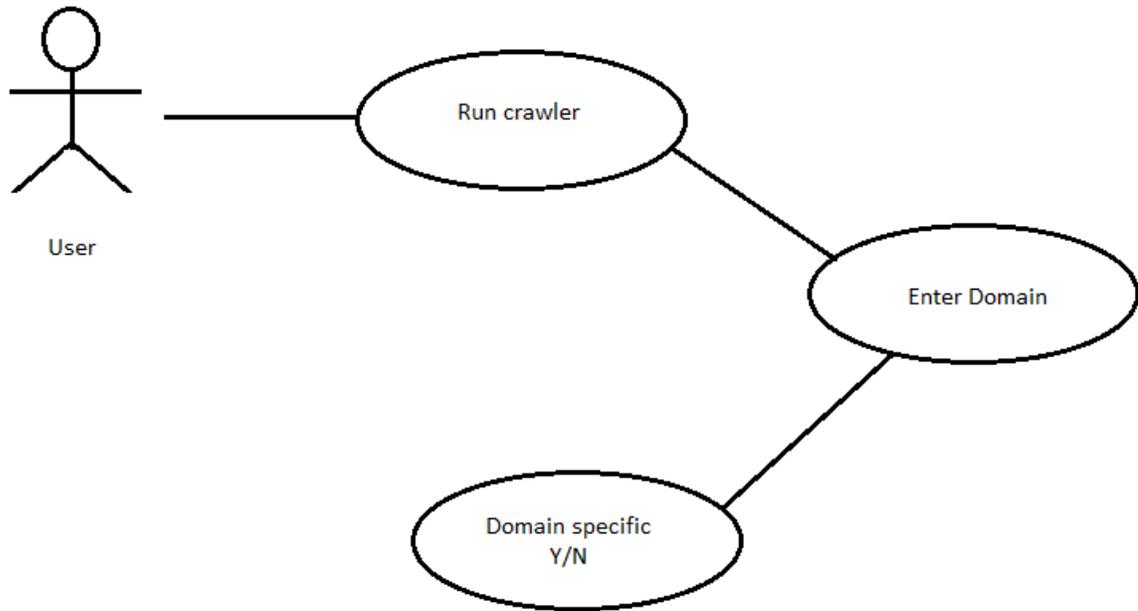
Figure 3.1: Smart Crawler Use case diagram.

The Use case diagram shown above describes the interactions the user will have with the system. In the first step the user will be required to execute the crawler on his machine. This will be a python file that needs to be run using the python interpreter. He will then be asked to enter the domain he wants to start crawling. After he enters the domain the user will be asked if he wants the crawler to crawl links only in the given domain or he wants to crawl links outside of the given domain also. He will be prompted to enter Y/N. After he enters his choice, the crawler starts crawling the links. The user will be able to see the proper text files in the directory specified.

3.1.2 The Object Model

The object model in the unified modeling language is represented by the class diagrams. A class diagram is used to represents the static view of an application. In addition to visualizing, describing and documenting different aspects of a system they also used for constructing executable code of the software application. These diagrams describe attributes and operations of a class. They also describe the constraints imposed on the system. They are the only diagrams that map object orientation, so they are extensively used in objects oriented systems. A class diagram will show a collection of classes, interfaces, associations, collaborations and constraints.

Figure 3.2 shows the class diagram for the smart crawler project. The class diagram has four main components. The main class is the class that calls all other classes whenever necessary and gets things done. It has functions like sendurls() to the web and fetch data from the internet. It sends the data obtained from the web for processing to the dataProcess class and receives the data after processing. It sends this data to the database to create files and store the data
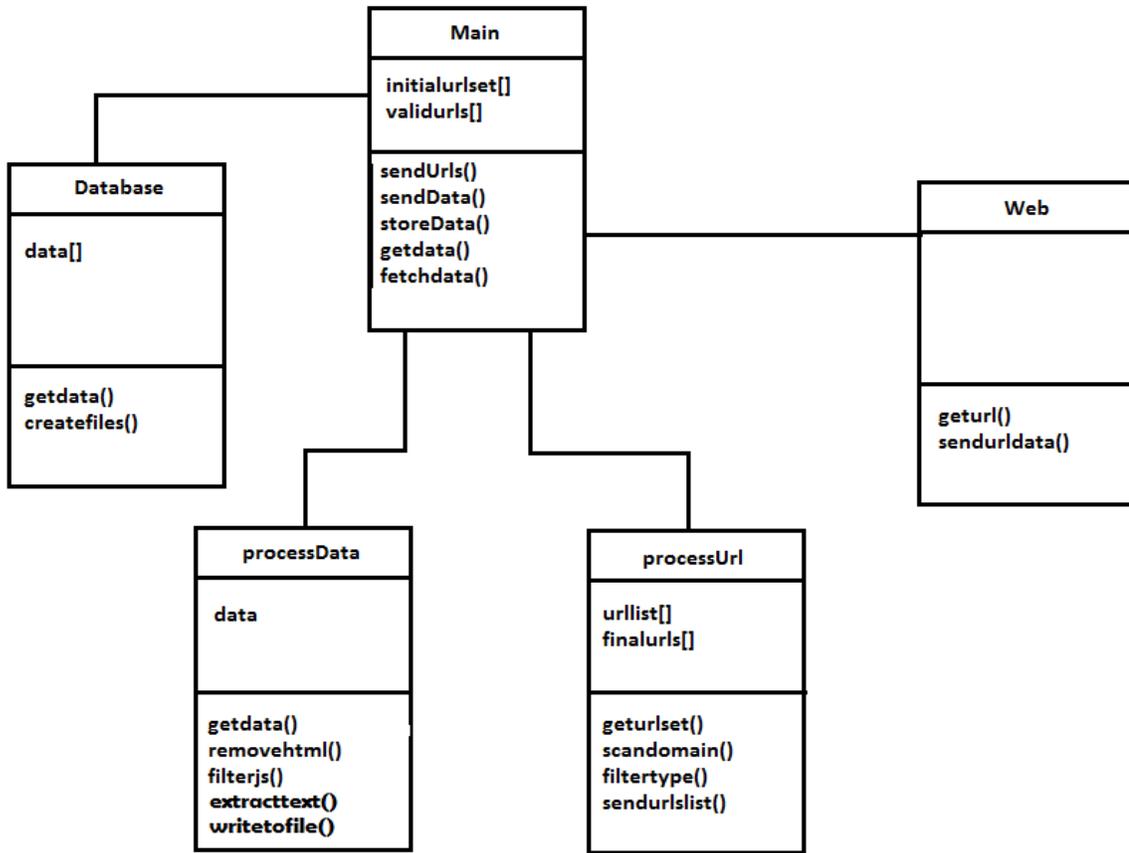
Figure 3.2: Class Diagram

.

The processData class gets raw data from the main class and processes it to get useful data. It has functions to remove all the html tags in the data and also functions to remove all the java script data. It also has functions to extract the required text and format meta information. Thus the residual data is meaningful text data that it sends back

to the main class. The processUrl class deals with urls. It gets URLs from the main class and has functions to process then .The scandomain() function checks if the current URL belongs to the given domain or not. The filterurl() function checks if the URL points to a useful file. After this processing the validUrlList is sent back to the main class.

3.1.3 The Dynamic Model

The Dynamic Model in the unified modeling language is represented by sequence diagrams. Sequence diagram are meant for emphasizing the time sequence of interactions between classes. The main purpose is to visualize the interactions between the components of the system.

The following is the sequence diagram for the crawler.
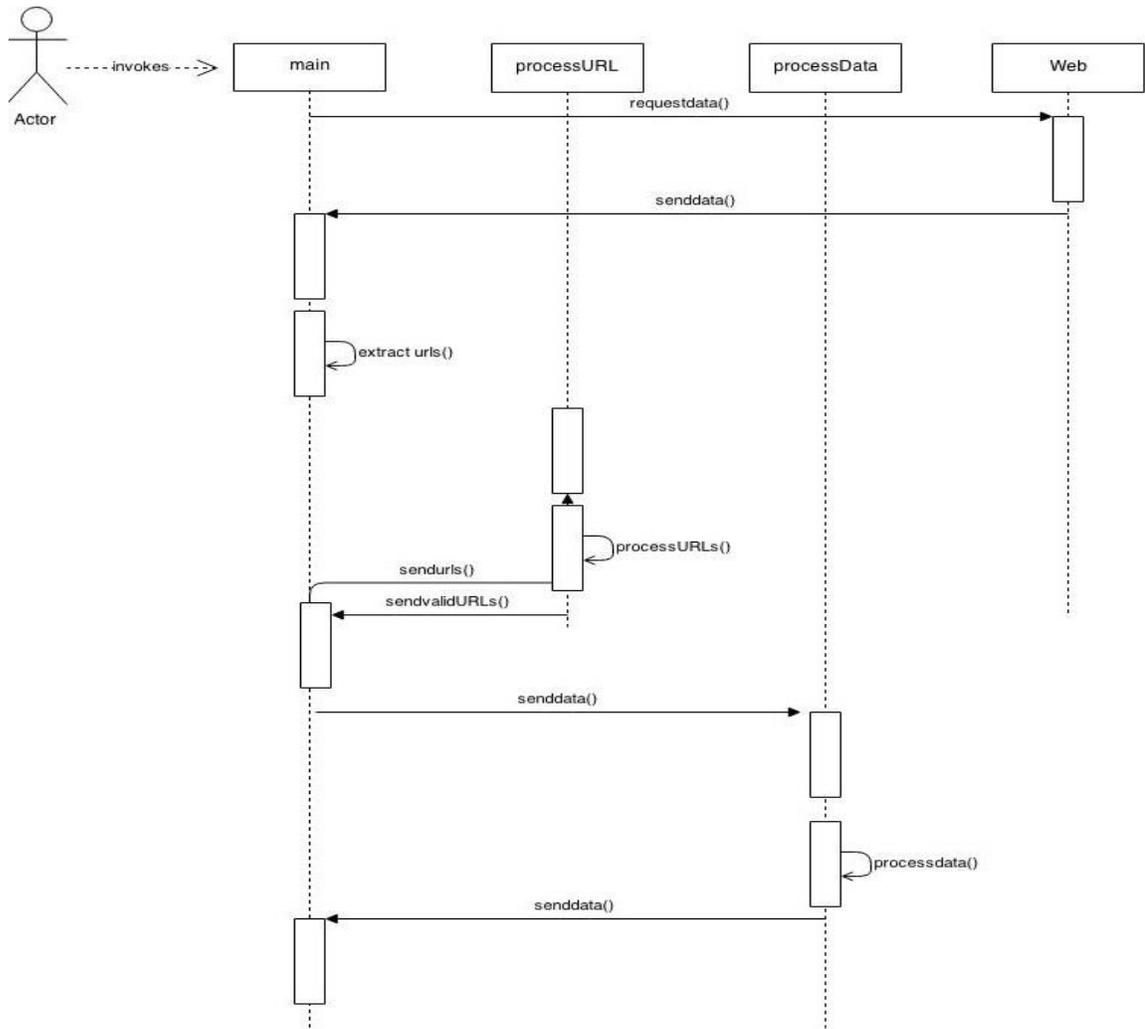
Figure 3.3 Sequence Diagram

3.2 Algorithm for Smart Crawler

The algorithm used for the Smart crawler is shown below.

```
For(each url in urllist)
{
        If (MIME type == html or txt)
        {
                If(url == given domain)
                {
                Extract data();
                Extract urls();
                Cleaner.script()
                Extract text();
                Extract metadata();
                Save to disk();
                }
        }
}
```

This algorithm precisely describes the working of the Smart Crawler. All the URLs are stored in a list called URL list. We begin by taking each URL from the URL list iteratively and check if the URL has a MIME type that is will hold data or not. If the MIME type proves to be anything else rather that html or txt type, we ignore the URL and proceed to the next URL. If the URL goes through we then check it again to see if it belongs to the domain specified by the user. This helps the crawler to be domain specific. If the crawler is domain specific we then supply it to the Data Extractor. The Data Extractor extracts data from the URL and also extracts all the URLs in the data. These URLs are stored in the URL list. The data extracted is now filtered to remove the java

script and then the html tags are also removed.We then extract the required text data from this data and also handle the metadata appropriately. The residual data is saved to the disk appropriately.

```python
def process_url(url,text_num):
#Processes the url
    result = []

    mime = getPageMimeType(url)
    #print mime
    if (mime == "application/xhtml+xml" or mime == "text/html"):
        result = getAndProcessData(url,text_num)
    for item in result:
        #print item
        if (option == "y" or option == "Y"):
            parsed = urlparse(item)
            if(input == parsed.netloc):
                urllist.add(item)
        else:
            urllist.add(item)


class HeadRequest(urllib2.Request):
## Custom Headrequest Request class for urllib2 to get page headers.
    def get_method(self):
        return "HEAD"

def getPageMimeType(url):

    try:
        response = urllib2.urlopen(HeadRequest(url), timeout = 20)
        content = response.info()["content-type"]
        contents = content.split(";")
        return contents[0]
    except:
        return None
```

Figure 3.4 Code for processing URL

The above code snippet shows the processing of URLs. The function process_url() is called to process a URL from urllist[]. For every url except the given url the crawler uses the parsed.netloc function available in the urlParse library of python to

35

extract the domain of the url under processing. It then compares it with the given domain name and if they match the URL is added to the valid urllist []. It then proceeds to check the MIME type of the URL before requesting data from the URL. It gets the MIME type from the getMimeType(). This function in turn calls the Headrequest class to return the MIME type of the URL. We now check if the URL is useful by comparing the MIME type to the predefined types. If they match they are useful. The URL is crawled.

```python
def searchforURLs(page):
    validurls = []
    links = re.findall('"((http|ftp)s?://.*?)"', page)
    for item in links:
        url = item[0]
        validurls.append(url)
    #print validurls
    return validurls
```

Figure 3.5 Code for URL extraction

This part of the program deals with extracting all the URLs from the data that is obtained from a given URL. We use the searchforURLs() function . In this function we use a regular expression to match all string patterns that start with http or ftp followed by // which mark all URLs. We then populate the extracted URLs into an array with the name validurls and return the array.

```python
def getAndProcessData(url,text_num):

#Processes data from the url
    try:
        website = urllib2.urlopen(url)
        page=website.read()
        page=unicode(page,errors="ignore")
        soup = BeautifulSoup(page)

        nojs = lxml.html.tostring(cleaner.clean_html(lxml.html.parse(url)))
        clean_data = strip_tags(nojs)
        text_file  =  "C:\ProjectDemo\CrawlerOutput\Data_" + str(text_num)+ ".txt"
        #raw_data = "C:\ProjectDemo\Raw\Data_" + str(text_num)+ ".txt"
        print text_file
        f = open("Address_File.txt","a")
        f.write(text_file)
        f.write("\n")
        p = open(text_file,"w")
        for desc in soup.findAll('p'):
            x = desc.get_text()
            x = x.encode("ASCII", 'ignore')#data = x.decode("windows-1252")
            p.write(x)
        formatPage(url,text_num)
        all_urls = []
        all_urls = searchforURLs(nojs)
        return all_urls
    except:
        r = []
        print "error"
        next_url = urllist.pop()
        print next_url
        text_num = text_num + 1
        r = getAndProcessData(next_url,text_num)
        return r
```

Figure 3.6 Code for the processing data from url.

  This part of the program processes raw data which includes a lot of unnecessary data into meaningful data. It first accesses the internet to get data in the URL. It then passes this data to the strip tags functions in the MLStripper class. This class uses the HTML parser library to parse the data and get rid of all the tags in the data. It then passes this data to cleaner() from the lxml library which removes all the java script from the data. This data is then passed to the Beautifulsoup module to internally create a parse tree. This tree is used to extract the data from the paragraph tags in the pages. It also

helps in formatting the metadata information in the page appropriately.In addition to this the processdata() function opens a text files and writes this data into the files.

4. Results

The Concept Based Semantic Engine is an engine that takes text files as input extracts tokens from the files and stores the tokens in an SQL database. Before storing it also performs some processing on the tokens and tries to get useful tokens from the data. It also gets a token count, the number of words in the token; frequency, the number of times the token occurs across the collection of documents and document frequency, the number of documents containing the token.

Figure 4.1 The Concepts Table in Semantic Database

In order to prove the effectiveness of the Smart Crawler, we used the Smartcrawler to

crawl the www.sjsu.edu website and crawl 500 pages from the site. We then used the

existing crawler that the Concept based Semantic engine uses to crawl the same

www.sjsu.edu site and save the 500 pages. We then feed each of the collection of

documents to the Concept Based Engine and obtain the following results. The two

collections have the same useful data but the files from the existing crawler have other

data in addition to useful information also

  We repeat the experiment three times for each collection and get the averages

results before comparing the results for the two collections.

Total tokens found	243550	243550	243550	**243550**
Time (sec)	5839.64	6016.7	5909.6	**5921.98**

Table 4.1: Results for Existing crawler

Total tokens found	102749	102749	102749	**102749**
Time (sec)	1982.48	2316.59	2261.32	**2186.79**

Table 4.2 Results for Smart Crawler

We now compare the results of both the crawlers and determine which of the crawlers is

more efficient in providing texts to the Concept based Semantic Search Engine.

	Existing Crawler	Smart Crawler
Total tokens found	243550	102749
Time (sec)	5921.98	2186.79

Table 4.3 Comparison of the results from the Concept Based Engine

Since we have crawled the same website www.sjsu.edu we know that the textual information crawled by both the crawlers is the same. Now that the comparison table shows that the total time taken when using the smart crawler is far less than the time taken when using the general crawler. The Smart crawler takes only 35% of the time taken by the existing crawler. The Concept based Sematic Engine was also able to identify better concepts from the Smart Crawler compared to the existing crawler. While using the existing crawler the Concept based engine collected a lot of concepts from the unimportant data. So it wrote a large number of tokens to the database. The smart crawler tremendously increased the efficiency of the Semantic Engine.

The Smart Crawler also helps in saving a lot of disk space. The files saved by the Smart crawler need only 30% of the disk space needed by the files saved by the existing crawler.

5. **Conclusion**

As proposed, we built a smart crawler to serve the needs of the Concept Based Semantic Search Engine. The smart crawler successfully crawls in a breadth first approach. We could build the crawler and equip it with data processing as well as url processing capabilities. We filtered the data obtained from web pages on servers to get text files as needed by the Semantic Search engine. We could also filter out unnecessary URLs before fetching data from the server.

We also formatted metadata from the HTML pages and saved them to a directory so that the metadata can be used in the future. We compared the performance of the existing crawler with that of the smart crawler. With the filtered text files generated by the Smart Crawler the Semantic Search Engine was able to identify concepts from the data quickly and in a much more efficient way. Thus we were able to improve the efficiency of the Concept Based Semantic Search Engine.

6. **References**

1. Gupta, P.; Johari, K., "Implementation of Web Crawler," Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on , vol., no., pp.838,843, 16-18 Dec. 2009
   doi: 10.1109/ICETET.2009.124
   http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6164440&isnumber=6164338
2. Manning, C., Raghavan, P., & Schutze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

3. Olston, C., & Najork, M. (2010). Foundations and Trends in Information Retrieval.

4. Udapure, T., Kale, R., & Dharmik, R. (2014). Study of Web Crawler and its Different Types.IOSR Journal of Computer Engineering, 16(1).

5. Home. (n.d.). Retrieved December 2, 2014, from http://www.thesearchenginelist.com/

6. Zalle, J. (2010). Python Programming: An Introduction to Computer Science (2nd ed.).

7. Features of Python. (n.d.). Retrieved December 2, 2014, from http://www.ibiblio.org/g2swap/byteofpython/read/features-of-python.html

8. Tsau Lin; Hsu, J.-D., "Knowledge Based Search Engine: Granular Computing on the Web," Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08. IEEE/WIC/ACM International Conference on , vol.1, no., pp.9,18, 9-12 Dec. 2008 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4740417&isnumber=4740405

9. Booch, G., Rambaugh, J., & Jacobson, I. (2005). The Unified Modeling Language User Guide(2nd ed.). Addison-Wesley Professional.