Project Report on

Implementation of Web Crawler in Python

Submitted to Manipal University, Jaipur

Towards the partial fulfillment for the Award of the Degree of

BACHELORS OF TECHNOLOGY

In Computer and communication Engineering

2017-2021

By Neha Sheth

179303093

Lakshay Agarwal

179303081

Mehul Gupta

179303085

CCE 6 B

MANIPAL UNIVERSITY
JAIPUR

Under the guidance of

Deepak Sinwar

Department of Computer and communication Engineering

School of Computing and Information Technology

Manipal University Jaipur

Jaipur, Rajasthan

ABSTRACT

Information Retrieval deals with searching and retrieving information within the documents and it also searches the online databases and internet. A web crawler (sometimes known as a robot, a spider, or a screen scraper) is a piece of software that automatically gathers and traverses documents on the web. Web crawlers are a fundamental component of today's web. For example, Googlebot is Google's web crawler. Googlebot is constantly scouring the web, downloading pages in search of new and updated content. All of this data forms the backbone of Google's search engine infrastructure.

Web scraping replaces the need for manual data entry and more easily reveals trends among data collected. It can also aggregate information from multiple sources into one central location. While this application provides three specific examples of web crawling/scraping, it could be easily altered to better suit additional markets and/or needs.

In our project, we will be developing a web crawler in Python using BeautifulSoup and Requests module in order to crawl a website.

INTRODUCTION

The World Wide Web (WWW) is internet client server architecture. It is a powerful system based on complete autonomy to the server for serving information available on the internet. The information is arranged as a large, distributed, and non-linear text system known as Hypertext Document system. These systems define part of a document as being hypertext pieces of text or images which are linked to other documents via anchor tags. HTTP and HTML present a standard way of retrieving and presenting the hyperlinked documents. Internet browsers, use search engines to explore the servers for required pages of information. The pages send by the servers are processed at the client side. Now days it has become an important part of human life to use Internet to gain access the information from WWW. The current population of the world is about 7.049

billion out of which 2.40 billion people (34.3%) use Internet. From .36 billion in 2000, the amount of Internet users has increased to 2.40 billion in 2012 i.e., an increase of 566.4% from 2000 to 2012. In Asia out of 3.92 billion people, 1.076 billion (i.e.27.5%) use Internet, whereas in India out of 1.2 billion, .137 billion (11.4%) use Internet. Same growth rate is expected in future too and it is not far away when one will start thinking that life is incomplete without Internet. A web crawler is a piece of software that automatically gathers and traverses documents on the web. It will first download a page, then it will parse the HTML and locate all hyperlinks (i.e. anchor tags) embedded in the page. The crawler then downloads all the HTML pages specified by the URLs on the homepage, and parses them looking for more hyperlinks. This process continues until all of the pages on the page to be crawled are downloaded and parsed..

In our project, we will be gathering data from a website in order to familiarize ourselves with the HTTP protocol. HTTP is (arguably) the most important application level protocol on the Internet today: the Web runs on HTTP, and increasingly other applications use HTTP as well (including Bittorrent, streaming video, Facebook and Twitter's social APIs, etc.).

PROBLEM STATEMENT

In this project, we first study currently developed web crawlers and their architectural algorithms. The main purpose is to increase the speed of web crawling process by implementing an algorithm which can be helpful to increase the effectiveness of Search Engine. We also put some improvements in web directory to retrieve more information of web applications which will be helpful for knowing the content as well as the area of web applications i.e. social network, educational, business, online shopping etc. so that web applications can be categorized more close to particular topic and focus on a specific subjects. The basic web crawling algorithm is simple: Given a set of seed Uniform Resource Locators (URLs), a crawler downloads all the web pages addressed by the URLs, extracts the hyperlinks contained in the pages, and iteratively downloads

the web pages addressed by these hyperlinks. Despite the apparent simplicity of this basic algorithm, web crawling has many inherent challenges like :

1. Scale : The web is very large and continually evolving. Crawlers that seek broad coverage and good freshness must achieve extremely high throughput, which poses many difficult engineering problems.

2. Content selection tradeoffs. : Even the highest-throughput crawlers do not purport to crawl the whole web, or keep up with all the changes. Instead, crawling is performed selectively and in a carefully controlled order. The goals are to acquire high-value content quickly, ensure eventual coverage of all reasonable content, and bypass lowquality, irrelevant, redundant, and malicious content.

3. Social obligations : Without the right safety mechanisms a highthroughput crawler can inadvertently carry out a denial-of-service attack.

4. Adversaries. Some content providers seek to inject useless or misleading content into the corpus assembled by the crawler.

OBJECTIVES

The main objectives of the study are:

1. Building an efficient, robust and scalable crawler.

2. Selecting a traversal order of the web graph, assuming content is well-behaved and is interconnected via HTML hyperlinks.

3. Scheduling revisitation of previously crawled content .

4. Avoiding problematic and undesirable content .

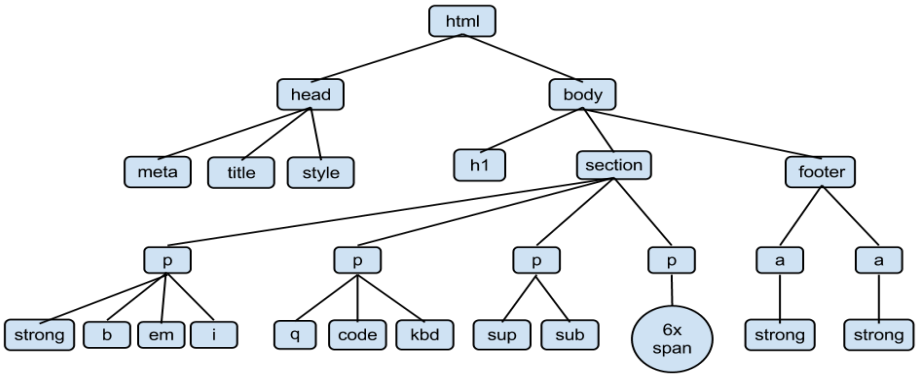
5. Crawling so-called “deep web” content, which must be accessed via HTML forms rather than hyperlinks.

DESCRIPTION

We create a simple program that provides a framework for handling the repetitive details of web crawling such as loading pages, finding the links, keeping track of what's been called, and so on. Then one can plug custom processing code into this framework for all the specific functions the web crawler must perform. In order to achieve this goal, we have used two popular Python modules – BeautifulSoup and Requests module.

BeautifulSoup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work. It creates a parse tree for parsed pages that can be used to extract data from HTML, which is useful for web scraping.

The Requests module allows us to send HTTP requests using Python. The HTTP request returns a Response Object with all the response data (content, encoding, status, etc). It focuses on the task of interacting with web sites. It can download a web page's HTML given its URL. It can submit data as if filled out in a form on a web page. It can manage cookies, keeping track of a logged-in session. And it helps handling cases where the web site is down or takes a long time to respond.

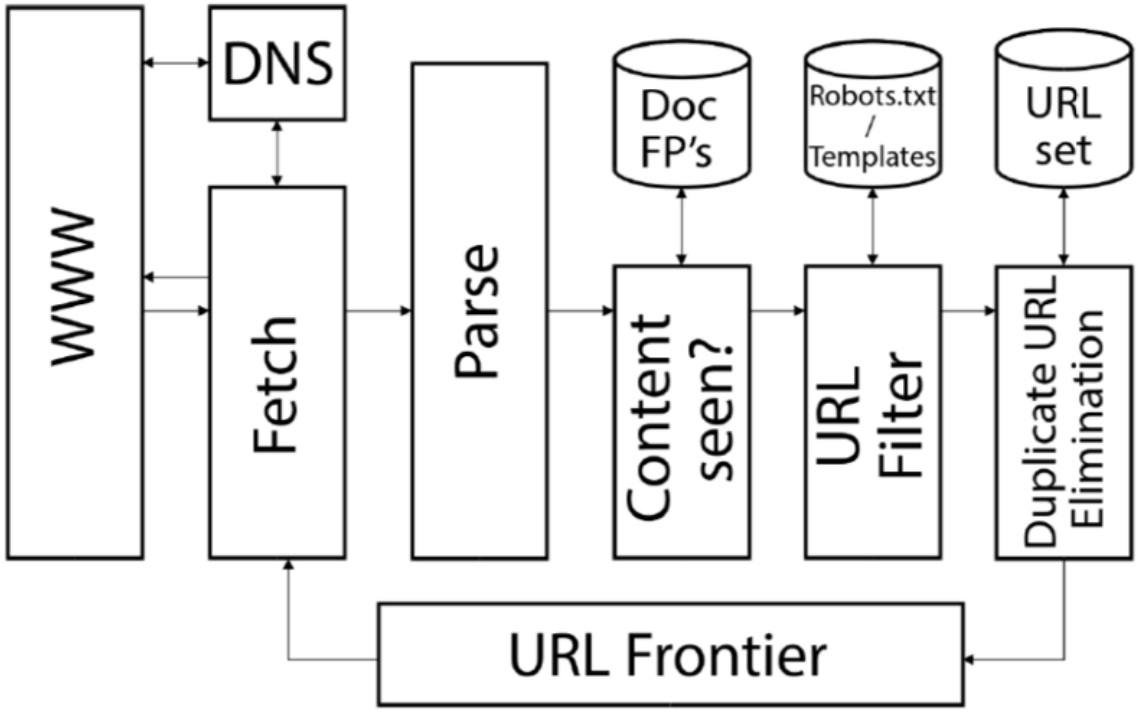


HTML Element Tree

Architecture of Web Crawler –

The architecture of the system is can be divided as follows :

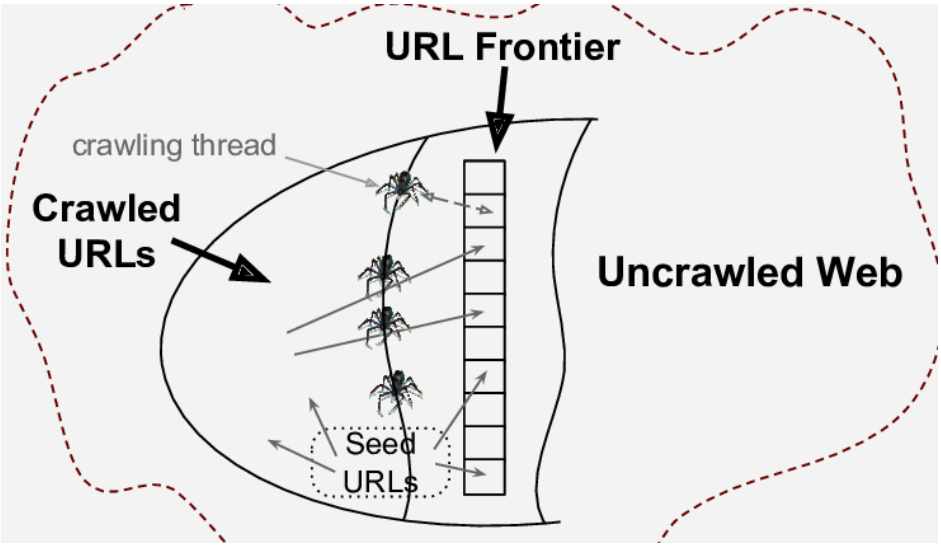
1. The URL frontier, containing URLs yet to be fetched in the current crawl (in the case of continuous crawling, a URL may have been fetched previously but is back in the frontier for re-fetching.)
2. A DNS resolution module that determines the web server from which to fetch the page specified by a URL.
3. A fetch module that uses the http protocol to retrieve the web page at a URL.
4. A parsing module that extracts the text and set of links from a fetched web page.
5. A duplicate elimination module that determines whether an extracted link is already in the URL frontier or has recently been fetched.



URL Frontier – The URL frontier at a node is given a URL by its crawl process. It maintains the URLs in the frontier and regurgitates them in some order whenever a crawler thread seeks a URL. Two important

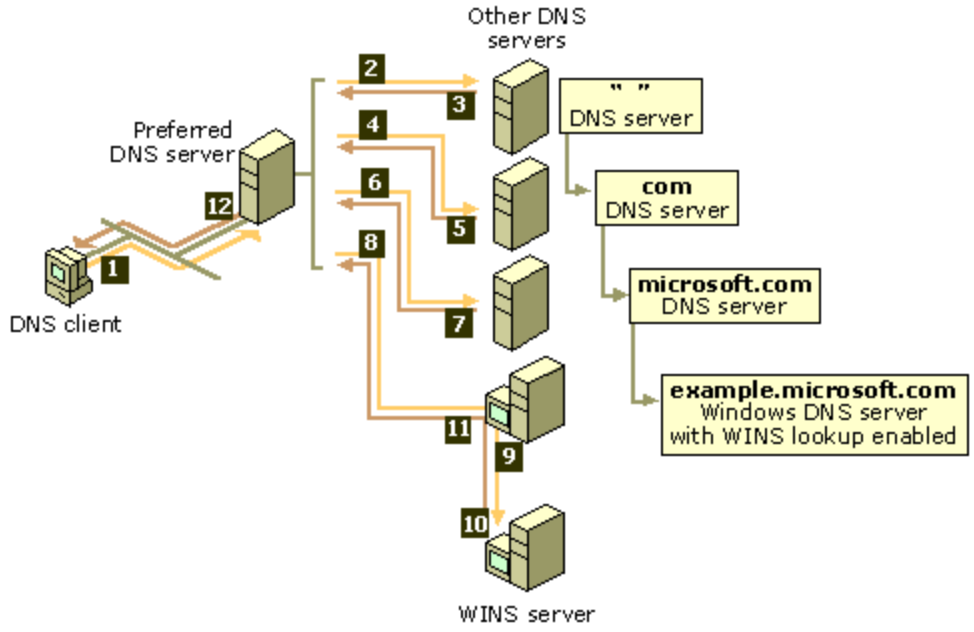
considerations govern the order in which URLs are returned by the frontier. First, the priority of a page should be a function of both its change rate and its quality (using some reasonable quality estimate). The combination is necessary because a large number of spam pages change completely on every fetch.

The second consideration is politeness: we must avoid repeated fetch requests to a host within a short time span. A URL frontier implemented as a simple priority queue might result in a burst of fetch requests to a host.



DNS Resolution – Each web server has a unique *IP address* in textual form, translating it to an IP address is a process known as *DNS (Domain Name Service) resolution* or DNS lookup. During DNS resolution, the program that wishes to perform this translation (in our case, a component of the web crawler) contacts a *DNS server* that returns the translated IP address. For a more complex URL, the crawler component responsible for DNS resolution extracts the host name - in this case en.wikipedia.org - and looks up the IP address for the host en.wikipedia.org.

DNS resolution is a well-known bottleneck in web crawling. DNS resolution may entail multiple requests and round-trips across the internet, requiring seconds and sometimes even longer., avoiding the need to go to the DNS servers on the internet. We solve this by implementing a timed-wait.



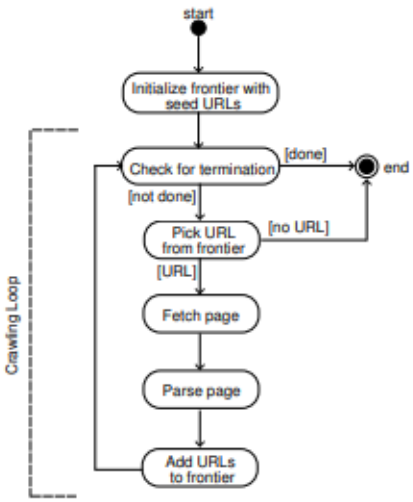
We follow the progress of a single URL through the cycle of being fetched, passing through various checks and filters, then finally (for continuous crawling) being returned to the URL frontier. Hence, we are using double ended queues.

The basic crawling infrastructure is as follows :

The crawler maintains a list of unvisited URLs called the frontier. The frontier is the to-do list of a crawler that contains the URLs of unvisited pages. In graph search terminology the frontier is an open list of unexpanded (unvisited) nodes. The list is initialized with seed URLs which may be provided by a user or another program. Each crawling loop involves picking the next URL to crawl from the frontier, fetching the page corresponding to the URL through HTTP, parsing the retrieved page to extract the URLs and application specific information, and finally adding the unvisited URLs to the frontier. Before the URLs are added to the frontier they may be assigned a score that represents the estimated benefit of visiting the page corresponding to the URL. The crawling process may be terminated when a certain number of pages have been crawled. If the crawler is ready to crawl another page and the frontier is empty, the situation signals a dead-end for the crawler. The crawler has no new page to fetch and hence it stops.

Crawling can be viewed as a graph search problem. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes. The process of fetching a page and extracting the links within it is analogous to expanding a node in graph search. A topical crawler tries to follow edges that are expected to lead to portions of the graph that are relevant to a topic.

Flow of a basic sequential crawler –



WORKING

The main steps involved in the basic structure of the simple web crawler are :

1. **Creating the queue :** Instead of using Python's built-in *list* data type for this, we are using double ended queue, because lists don't provide good performance when repeatedly pulling items off the front of them (because the entire list needs to be re-written in memory each time). So we used deque from the *collections* module, which is designed for this scenario and provides fast and predictable performance.

2. **Loading the page and finding the links :** This is where requests and Beautiful Soup come into play, and they make the code extremely simple compared to the alternatives. The program reads the web page, creates a DOM of the page, and extracts a list of the targets of all links on the page.

3. **Operations on the crawled pages :** Specific functionality needed for each use of the crawler. In the specific case, finding all the images on each page and analyzing their use of EXIF metadata. Beautiful Soup made that very easy to do.

4. **Adding the page's links to the queue :** Relative URLs to avoid re-crawling the same page multiple times. Python's list comprehensions make it easy to modify the list of links.

We are crawling www.microsoft.com just for demo purposes, but this link can be changed in the source code in order to enable crawling of other websites aswell. There is also a timer set up in order to count the number of seconds the crawler takes. It also displays the total number of pages it could crawl, and the number of pages that it failed to crawl.

Some *important functions* from the program are :

crawler - crawls the web starting from specified page.

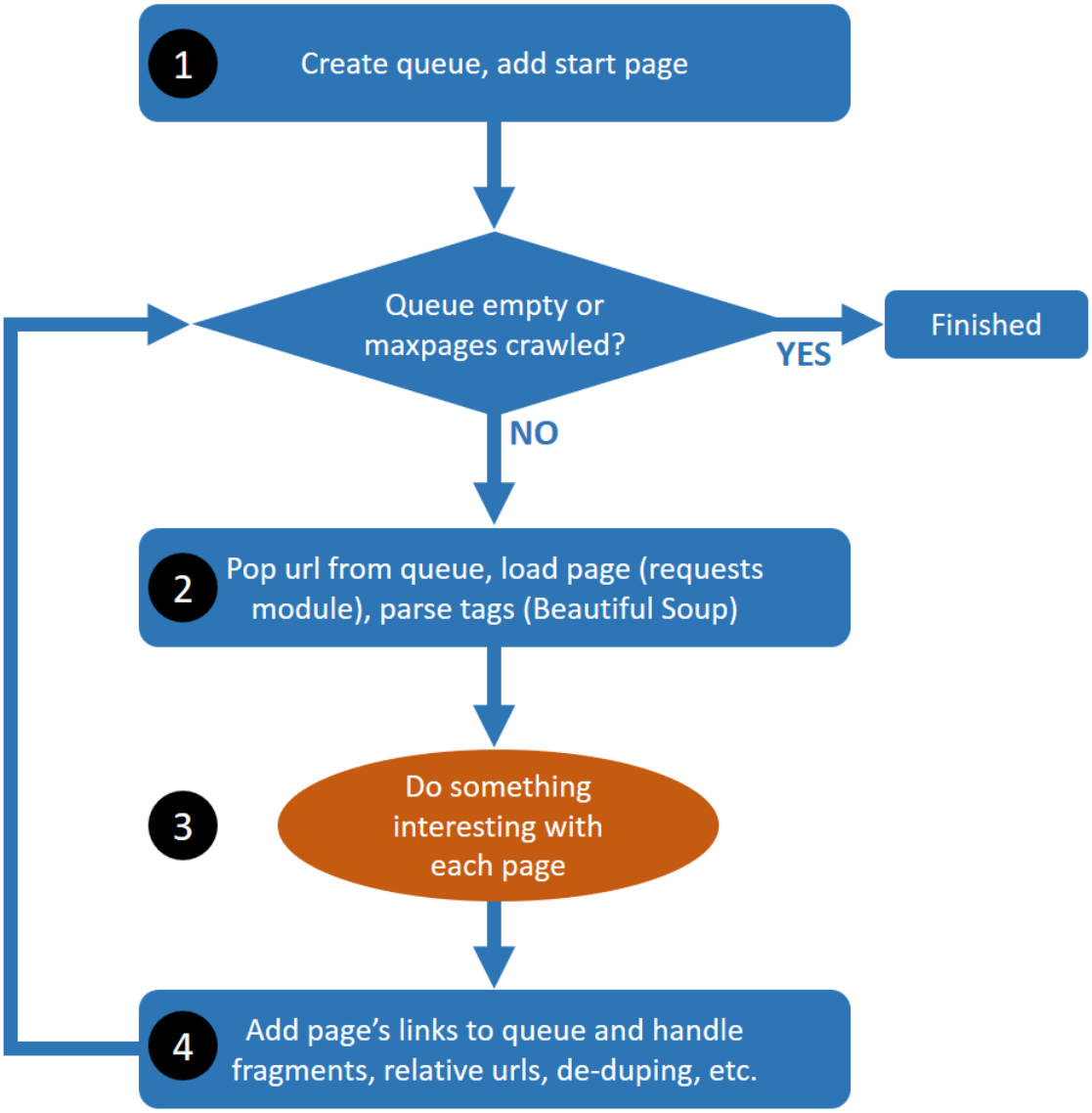
getcounts - convert a list of words into a dictionary of word/count pairs.

getlinks - returns a list of links from from this page to be crawled.

pagehandler – function which can be customized according to user needs.

samedomain – determines whether two netloc values are in same domain.

url_in_lists – determines whether a URL is in a list of URLs.



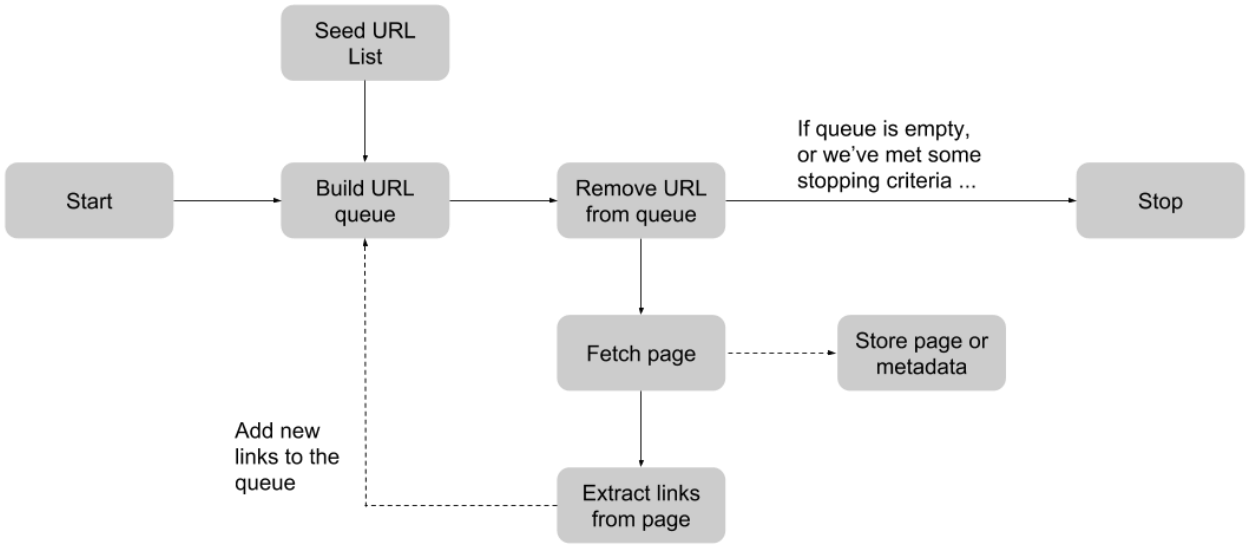
Flowchart of Web Crawler

For the web scrapping part, these are the steps involved.

1. Send an HTTP request to the URL of the webpage we want to access. The server responds to the request by returning the HTML content of the webpage. For this task, we will use a third-party HTTP library for python-requests.

2. Once we have accessed the HTML content, we are left with the task of parsing the data. Since most of the HTML data is nested, we cannot extract data simply through string processing. One needs a parser which can create a nested/tree structure of the HTML data.
3. Now, all we need to do is navigating and searching the parse tree that we created, i.e. tree traversal. For this task, we use Beautiful Soup.

BLOCK DIAGRAM FOR WORKING MODEL –



REFERENCES –

1. An Efficient Approach for Web Indexing of Big Data through Hyperlinks in Web Crawling by R. Suganya Devi, D.Manjula.

2. Web Crawler Architecture by Marc Najork in Encyclopedia of Database Systems.
3. Shestakov, Denis. (2013). Intelligent Web Crawling (WI-IAT 2013 Tutorial). IEEE Intelligent Informatics Bulletin. 14. 5-7.
4. https://en.wikipedia.org/wiki/Web_crawler

INDIVIDUAL CONTRIBUTION –

1. Neha Sheth : Developed the crawler function based on lists of URLs yet to crawled and whether a URL in in an existing list of URLs.
2. Lakshay Agarawal : Developed word counts functions after converting a list of words into a dictionary of word/count pairs.
3. Mehul Gupta : Developed page handler and function to determine whether the links are in the same domain.