Institute for Advanced Studies in Basic Sciences
Gava Zang, Zanjan, Iran

# Information Retrieval Final Project

Web Crawler

Prof. Ebrahim Ansari

**By**

Mahsa Radinmehr

Institute for Advanced Studies in Basic Sciences

Summer 2017

## Table of Contents

## 1- Statement of the problem

A Search Engine for all pages in the IASBS Domain Crawl, Index and search all pages in the IASBS domain.

## 2- Introduction

In order to solve this problem, we have to do the following steps:

1) Crawl using a seed URL in the domain of IASBS website.
2) Parse the web page and extract all the hyperlinks in it and if it's not duplicated and is part of IASBS domain, add them back to a front queue.
3) Store and Index web pages that has been visited in order to prevent duplication and use them for further search engine implementation.
4) Grab a link from front queue and continue from step 1 assuming the link is our new seed URL until all the web pages visited and front queue is empty.
5) After this we have crawled all the web pages in this domain. Now we can do the search.

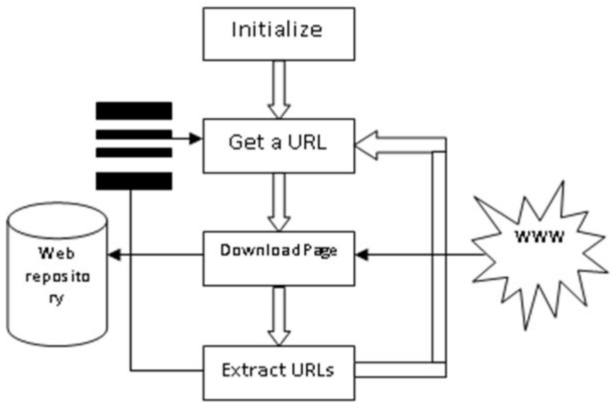

*Figure 1- Flow of a crawling process*

### 2-1 Web Crawler

A Web crawler, sometimes called a spider (or an ant, an automatic indexer, or a Web scutter), is an Internet bot that systematically browses the Word Wide Web, typically for the purpose of Web indexing (web spidering).

Web search engines and some other sites use Web crawling or spidering software to update their web content or indices of other sites' web content. Web crawlers can copy all the pages they visit for later processing by a search engine which indexes the downloaded pages so the users can search much more efficiently.

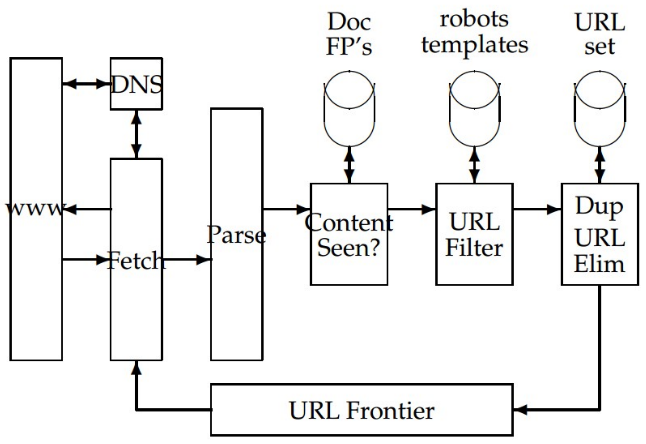

*Figure 2 – The basic Crawler Architecture*

The objective of crawling is to quickly and efficiently gather as many useful web pages as possible, together with the link structure that interconnects them.

### 2-2 cURL and libcurl

In order to read a web page from the internet we will use this library.

cURL is an Open Source project and its primary purpose and focus is to make two products:

- curl, the command-line tool
- libcurl the transfer library with a C API

It can be downloaded from this address: https://curl.haxx.se/

Both the tool and the library do Internet transfers for resources specified as URLs using Internet protocols. libcurl is designed and meant to be available for anyone who wants to add client-side file transfer capabilities to their software, on any platform, any architecture and any purpose.

- ✓ To make this library works in Visual Studio, we need to modify dependency of the project in its properties.
- ✓ The callbacks should be static class member functions in C++ as it mentioned in its official website.

## 3- Implementation Code

### 3-1 Prerequisites

We use "htmlstreamparser" to extract html code (to be precise <a href=…). As we mentioned before we use libcurl as internet communication library.

### 3-2 Data Structure

We use:

- STL queue _which is a FIFO_ to store extracted URLs from web pages.
- User defined WEBPAGE struct to store visited URL within their IDs.
- Vector of WEBPAGEs for visited web pages.

```
struct WEBPAGE
{
	int webID; // Also file ID
	string url;
};

vector<WEBPAGE> visited;
```

### 3-3 Dive into codes

Everything is in the class called Crawler. We start with a private domain URL which in our case is "`http://www.iasbs.ac.ir/`". Methods are:

#### 3-3-1 class constructor: Crawler()

it begins with predefined domain URL.

```
Crawler::Crawler()
```

```
{
	this->URLs.push(domain);
}
```

### 3-3-2 Init_crawler():

It is for initialization of crawler. In the beginning we read domain page (or seed URL) and extract all the links that is in it. Then we write what we read to a file so we can use it for indexing part later. Also it will be added to visited URLs so we can check for duplicate URLs:

```
// Init with seed url (domain)
queue<string> Crawler::init_crawler()
{
	string source = getPageSource(domain);
	queue<string> initQ = extractLinks(source);
	// Write source to the file
	_mkdir(root_dir.c_str());
	writeWebSource(source);
	addVisitedUrl(0, domain);
	return initQ;
}
```

### 3-3-3 getPageSource()

In this method, we use libcurl to read a webpage. A lot of these lines are for configuration of our connection. Among them it is ‘write_function’ that will get the output of curl_easy_perform and make a buffer of memory and copy it into that. So After executing this part we have a buffer of string that has all of the (html) source code.

```
string Crawler::getPageSource(string url)
{
	CURL *curl;
	CURLcode res;
	string buffer;
	curl = curl_easy_init();
	if (curl)
	{
		curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
		curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1L);
		curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_function);
		curl_easy_setopt(curl, CURLOPT_WRITEDATA, &buffer);
		curl_easy_setopt(curl, CURLOPT_TIMEOUT, 5000);
		curl_easy_setopt(curl, CURLOPT_USERAGENT, "Mozilla/5.0 (Windows NT 
6.1; Win64; x64; rv:47.0) Gecko/20100101 Firefox/47.0");
		curl_easy_setopt(curl, CURLOPT_NOPROGRESS, 1L);

		res = curl_easy_perform(curl);
	}
	curl_easy_cleanup(curl);
	return buffer;
}
```

### 3-3-4 extractLinks()

This method uses ‘htmlstreamparser’ and will extract anchor links (<a href=…). We should mention that the mentioned library is a little buggy. For example, it will give us something like `‘//example.com/’` or `‘/subexample/sub/’`. So we need to deal with them later.

```
queue<string> Crawler::extractLinks(string source)
{
	char tag[1], attr[4], val[128];
	queue<string> linkQ;
	HTMLSTREAMPARSER *hsp;
	hsp = html_parser_init();

	html_parser_set_tag_to_lower(hsp, 1);
	html_parser_set_attr_to_lower(hsp, 1);
	html_parser_set_tag_buffer(hsp, tag, sizeof(tag));
	html_parser_set_attr_buffer(hsp, attr, sizeof(attr));
	html_parser_set_val_buffer(hsp, val, sizeof(val) - 1);
	size_t pageSize = size(source), p;
	for (p = 0; p < pageSize; p++)
	{
		html_parser_char_parse(hsp, source[p]);
		if (html_parser_cmp_tag(hsp, "a", 1))
			if (html_parser_cmp_attr(hsp, "href", 4))
				if (html_parser_is_in(hsp, HTML_VALUE_ENDED)) {
					html_parser_val(hsp)
[html_parser_val_length(hsp)] = '\0';
					linkQ.push(html_parser_val(hsp));
				}
	}
	html_parser_cleanup(hsp);
	return linkQ;
}
```

### 3-3-5 getWebIDStr()

As mentioned before a vector stores visited webpages so we use the size of this vector to find out how many pages we have crawled and also it will be used for storing filename:

```
string Crawler::getWebIDStr()
{
	int id = visited.size()-1;
	return to_string(id);
}
```

### 3-3-6 getWebID()

Like the previous method it gives us a ID but this time in an integer.

```
int Crawler::getWebID()
{
	int id = visited.size()-1;
	return id;
}
```

### 3-3-7 writeWebSource()

It writes a string of source into a file that we get its name from the previous method.

```
int Crawler::writeWebSource(string source)
{
	fstream weboutput;
	string filename = getWebIDStr();
	string fullpath = root_dir + filename;
	weboutput.open(fullpath, fstream::out);
	weboutput << source;
	weboutput.close();
	return 0;
}
```

### 3-3-8 addVisitedUrl()

This will add visited URL to the vector.

```
int Crawler::addVisitedUrl(int webID, string url)
{
	WEBPAGE visitedWeb;
	visitedWeb.webID = webID;
	visitedWeb.url = url;
	visited.push_back(visitedWeb);
	return 0;
}
```

### 3-3-9 isFindUrl()

In order to avoid duplication in our process we use this function:

```
bool Crawler::isFindUrl(string url)
{
	vector<WEBPAGE>::iterator it;
	it = find_if(visited.begin(), visited.end(), FindUrl(url));
	if (it != visited.end()) // If it's already exist in visited
		return true;
	else
		return false;
}
```

FindUrl which is a function that is used in find_if function, is defined to contrast URLs in two different vector of WEBPAGE struct and it is defined as the following:

```
struct FindUrl
{
	const string url;
	FindUrl(const string& url):url(url){}
	bool operator()(const WEBPAGE& w)const {
		return w.url == url;
	}
};
```

### 3-3-10 isPartOfDomain()

Because in our situation we only want to crawl every webpages in a particular domain (IASBS domain in our case), we use this to avoid other website to be crawled:

```
bool Crawler::isPartOfDomain(string url)
{
	if (url.find(domain) != string::npos)
		return true;
	else
		return false;
}
```

### 3-3-11 main()

First we define an object of Crawler kind and after initialization, we use the front URL of extracted URLs and check if it doesn't exist in visited webpages and it is in the desired domain, we crawl that webpage and extract its URLs and add them to the queue.

We continue this scenario until there wouldn’t be any other URL in the queue.

```
int main(int argc, char *argv[])
{
	Crawler crawler = Crawler();
	queue<string> initQ;
	initQ = crawler.init_crawler();
	while (!initQ.empty())
	{
		string currentUrl = initQ.front();
		string currentPageSource;
		currentPageSource= crawler.getPageSource(currentUrl);
		if (currentPageSource != "")
		{
			crawler.addVisitedUrl(crawler.getWebID(), currentUrl);
			crawler.writeWebSource(currentPageSource);
		}
		queue<string> newUrl = crawler.extractLinks(currentPageSource);
		initQ.pop();
		for (size_t i = 0; i < newUrl.size(); i++)
		{
			string tempUrl = newUrl.front();
			// Check for duplicate and is part of default domain, then 
add to front queue
			if((!crawler.isFindUrl(tempUrl)) && 
(crawler.isPartOfDomain(tempUrl)))
				initQ.push(tempUrl);
			newUrl.pop();
		}
	}
	getchar();
}
```

## 4- Future Work

We need to add some extra filtering to avoid downloading files of some kind of extensions like: *.pdf, *doc, *.mp4, …

## 5- References

[1] https://en.wikipedia.org/wiki/Web_crawler
[2] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schutze, An Introduction to Information Retrieval, Cambridge University Press, 2008.
[3] Daniel Stenberg, Everything curl.