Exploring Information Retrieval Techniques Through Programming Assignment 1

João Vítor Fernandes Dias
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
joaovitorfd2000@ufmg.br

Abstract

This article is summary of the implementation of the first programming assignment of the Information Retrieval course at the Federal University of Minas Gerais (UFMG). The assignment consists of programming a web crawler using Python 3 to scrape 100.000 unique pages efficiently, respecting certaing policies. The crawler must be distributed, using the multiprocessing library to take advantage of multiple CPU cores. After crawling, the pages must be stored into a WARC file, which is a standard format for archiving web pages, and zipped to save space.

CCS Concepts

• **Information systems** → **Information retrieval**; *Information extraction*; *Retrieval effectiveness*; *Retrieval efficiency*; *Distributed retrieval*; Data structures; • **Social and professional topics** → Acceptable use policy restrictions; Student assessment.

Keywords

Information Retrieval, Web Crawler, Python, WARC, Multiprocessing

ACM Reference Format:
João Vítor Fernandes Dias. 2025. Exploring Information Retrieval Techniques Through Programming Assignment 1. In *Proceedings of Information Retrieval (IR '25).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The task at hand involves developing an efficient parallel web crawler capable of exploring and collecting web data at scale. Throughout this project, a variety of strategies and optimizations were considered to balance computational performance with robustness. In this report, we document the implemented data structures and algorithms, analyze their theoretical and empirical performance, and characterize the crawled corpus. Special attention was given to how threading could be leveraged to maximize URL collection rates without compromising data integrity or server politeness policies.

IR '25, April 01–28, 2025, Belo Horizonte, MG

ACM ISBN 978-1-4503-XXXX-X/2025/04
https://doi.org/XXXXXXX.XXXXXXX

2 Brainstorming

Considering the project extent, many ideas came to mind. The initial brainstorming phase involved exploring various approaches to web scraping, including the use of different data structures and threading techniques. The goal was to consider the most effective methods for efficiently crawling and storing web data.

2.1 Expanding Frontier

Well, the simples would be to do a **Breadth-First Search** (BFS) on the website, starting from a seed URL and exploring all its links before moving on to the next level. This approach is simple and effective, but it can be slow and inefficient, especially for large websites with many links.

To avoid visiting the same website, we could think of a **data structure** such that this issue is avoided. Using a **list** to store the frontier, would not remove duplicates, and then may end up with a lot of repeated URLs. For that matter, we could use a **set** to store the URLs that have already been crawled. This would allow us to avoid crawling the same URL multiple times, but now we lose the ability to crawl URLs in a queue fashion, since we could only *pop* the top URL from the set. But the looping structures doesn't handle well when its iterable is being modified. So, we would have to use a different approach. Another possibility is to use a **deque** (double-ended queue) to store the URLs. This would allow us to efficiently pop URLs from both ends of the queue, but this just solves the last problem, returning us to the first one.

Instead of doing a BFS through the scraped outlinks of a page, a more idealistic approach would be to traverse the sitemap of the website. This allows for a more structured approach to crawling, as sitemaps often provide a clear hierarchy of URLs and their relationships. Considering the quantity of URLs to be crawled, and, since the sitemaps results in thousands of URLs usually beginning with the same domain, a way os reducing the storage cost is to store it into a **trie**, a tree-like data structure that is used to store prefixes of strings. This allows for efficient storage and retrieval of URLs, as well as the ability to quickly check for duplicates.

2.2 Knuth's optimization principle

In adherence to Donald Knuth's famous dictum, "**Premature optimization is the root of all evil,**" we consciously delayed performance tuning and creation of new features until a basic but functional version of the crawler (MVP) was operational. This decision allowed for a more deliverable product instead of a recursive loop of feature development and optimization.

2.3 GitHub Projects

Project management was facilitated through GitHub Projects, which helped track the crawler's features, bug fixes, and performance improvements systematically. Tasks were clearly delineated and prioritized based on their impact on crawler efficiency and stability. Although it helped streamline the development process, it was not as effective as initially anticipated, many desired features were not even written, and the project was not as organized as it could have been. The main reason for this was the lack of time to dedicate to the project, which led to a more ad-hoc approach to development.

3 MVP

The concept for the **Minimum Viable Product (MVP)** was to create a pipeline that would create the needed flow for the application to work. It was basically consisting of a few steps: **1.** Adding seeds to frontier, **2.** Crawling the frontier, **3.** Parsing the crawled pages, 4. Storing the crawled data.

Considering the need for **argument parsing**, and the desire for the easy reproducibility of the code, a few steps were taken to ensure that the code arguments were set as default values, then, not needing to be passed as arguments.

Another new learned feature was the use of the **WARC files**. The Web ARChive (WARC) file format is a standard for storing web crawls and other web-related data. It is widely used in the web archiving community and is supported by many web crawlers and archiving tools. The WARC format allows for efficient storage and retrieval of web content, as well as the ability to include metadata about the crawled pages. After storing the crawled data in WARC files, it was analyzed and many apparent unuseful data was being stored, so, it was decided to remove some header information and compress the files to save space. Some other problems encountered were: 1. Some URLs were missing, making it so that the file had only a few URLs, it was caused by the fact that there was a missing entry in the dictionary regarding the full HTML page; 2. Batch saving the whole 1000 pages into one file at once seemed more efficient than append a new page every time it was crawled, but, surprisingly enough, it was even simpler doing the seemingly more inefficient way.

3.1 Robots.txt and Sitemap

One of the needed tasks was to check the **robots.txt** file of the website. Apparently parsing this file was the focus of the *Protego* library, but, since not all libraries were known, it was decided to use the *requests* library to fetch and manually parse the file. For that matter, all lines were read and split into an Python Dictionary, where the user-agents were into a list of another dictionary with its allow and disallow policies, another key was the sitemap list, the expected crawl delay and also a miscellaneous key for any other information that were eventually found. Since, for now, we are focusing on the MVP, let's just stick to the imposed 100 ms delay per domain.

As for the **Sitemap** parsing, it was a bit more tricky, but ended up being event more effective than expected. After the BFS traversal of the sitemap was implemented using *BeautifulSoup*, it was found that the sitemap was not always in the same format. Some sitemaps were in XML format, while others were in HTML table format. This made it a bit more challenging to parse the sitemaps, but it ended up being actually just the same process, so, good to go. After parsing one of the seeds' sitemaps tree, it was found around 50000 URLs, which is most of the needed URLs for the MVP. Since one of the policies included a limited delay for requesting URLs from the same domain, it was decided to not take so much time getting this huge mass of URLs, and just get them from the usual *href* tags and make the frontier spread further.

3.2 Frontier

Frontier as a *list* of URLs to be crawled could have many advantages, such as allowing for a *queue* of URLs to be crawled. But, it would also allow the existance of a list of many duplicates. To avoid this, we could use a *set* to store the URLs that have already been crawled. This would allow us to avoid crawling the same URL multiple times, but now we lose the ability to crawl URLs in a queue fashion, since we could only pop the top URL from the set. We could also go for a deque, since I could pop from both ends, but it goes down the same path as the list without any processing for duplicates, so, for the MVP sake, let's make it work with the set for now.

Another point to be considered is the revisiting policy. The crawler should not revisit the same URL multiple times, so we need to implement a mechanism to track visited URLs. Many approaches could deal with that, but the feel that abusing of the operation **not in** would make the complexity skyrocket, so we kept it at the **set** that naturally handles this.

Other major consideration that took place was the decision to use only the domain name for the scraped URLs. By following that approach, we could set the main point for crawling these websites later on. And, considering that this is a whole new domain, there is no need to worry about the domain revisiting delay. Another limit used was to not scrape URLs that refers to the same domain that it was scraped from, and by doing so, the crawler is forced to wander around the web.

4 Parallelism

After implementing the MVP, it was time to consider parallelism. The initial approach was to use the *threading* library, which allows for the creation of multiple threads to handle different tasks concurrently. This was a good starting point, but after some testing, and LLM suggestions, it was decided to use the *ThreadPoolExecutor* class from the *concurrent.futures* module. This class provides a high-level interface for asynchronously executing callables using threads. It ended up being a more efficient choice time-wise.

But, even after implementing the ThreadPoolExecutor, it was still not as efficient as expected. The max-thread count was increased but no significant change was noticed. It kept at that until it was decided to print the number of active threads, and it was found that the number of active threads was equal to the number of the initial seeds. So, after some tweaking, as soon as the frontier received more URLs, the number of active threads increased until it reached its cap. Now the speed was much better.

5 Results

Considering everything that was done, the results were quite satisfactory. The crawler was able to crawl a large number of URLs in a relatively short amount of time. The use of parallelism made a huge difference in the crawling speed, and, surprisingly enough, the use of WARC files for storing the crawled data as it is parsed was less error prone than expected.

5.1 Algorithms

As the algorithm implemented, we can summarize the following steps:

- **Crawling:** the crawler uses the *ThreadPoolExecutor* to crawl multiple URLs concurrently. Each thread is responsible for crawling a single URL and parsing its content.
- **Frontier Expansion:** after the initial seeds are added to the frontier, the crawler pops one URLs, checks if it was already visited, parses it, and stores its main data into a dictionary and the domain of the parsed outlinks are added to the frontier.
- **WARC Storage:** the parsed data is then appended to the correspondent WARC file. The WARC file is compressed to save space and the needed information is passed to it.

5.1.1 *Data Structures.*

5.2 Computational Complexity

Most of the computational complexity relies on the underlying implementation of the libraries used. The most notable ones are the `requests` and `BeautifulSoup` libraries, which are used for making HTTP requests and parsing HTML content, respectively. The complexity of these operations can vary based on the size of the HTML content being processed and the number of URLs being crawled.

So, diving deeper into the mathematical complexity of the crawler, we can analyze the following components:

- **Crawling URLs:** The time complexity of crawling a single URL is $O(n)$, where n is the size of the HTML content. This is because we need to download the entire page and parse it to extract links.
- **Parsing HTML:** The time complexity of parsing HTML content using BeautifulSoup is also $O(c)$, where c is the number of elements in the HTML document. This is because we need to traverse the entire DOM tree to extract links and other relevant information.
-

5.3 Empirical Efficiency

After tweaking a lot with the threads, some notable improvements were made. The Table 1 below shows the performance of the crawler in terms of time taken to crawl a certain number of URLs, as well as the expected time for crawling 100k URLs based on the mean time per URL.

Table 1: Crawling performance

[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	100	1447	60.94	0,61	16:56:40	
1	1000	3000	859.00	0,86	23:53:20	
1	1000	3000	859.00	0,86	23:53:20	
1	1647	5767	1904.00	1,16	32:13:20	
8	100	1590	25.00	0,25	06:56:40	
10	1002	3268	584.00	0,58	16:06:40	Thread Pool
16	100	1191	14.00	0,14	03:53:20	
32	100	1009	10.00	0,10	02:46:40	
40	537	1204	47.35	0,09	02:30:00	Thread Pool (Corrected)
40	539	598	58.98	0,11	03:03:20	No WARC Batch
40	100	417	5.95	0,06	**01:40:00**	
48	100	693	16.81	0,17	04:43:20	Domain only
48	100	417	5.68	0,06	**01:40:00**	
50	100	558	6.26	0,06	**01:40:00**	
50	100	502	7.03	0,07	01:56:40	
50	100	848	12.97	0,13	03:36:40	Domain only
50	1048	3189	82.60	0,08	02:13:20	Thread Pool
50	502	2048	109.63	0,22	06:06:40	Thread Pool: broken warc
50	502	2092	187.36	0,37	10:16:40	Thread Pool: batch 100
50	517	1948	34.66	0,07	01:56:40	No WARC Batch
50	520	1827	47.69	0,09	02:30:00	No WARC Batch
50	527	1810	41.63	0,08	02:13:20	No WARC Batch
50	534	2563	44.40	0,08	02:13:20	No WARC Batch.gz
50	538	2079	37.76	0,07	01:56:40	Thread Pool
56	100	824	7.41	0,07	01:56:40	
60	543	1332	31.13	0,06	**01:40:00**	Thread Pool
60	559	2252	32.63	0,06	**01:40:00**	Thread Pool
60	1200	2113	82.67	0,07	01:56:40	Converted to py
64	100	613	7.86	0,08	02:13:20	
70	547	1721	51.55	0,09	02:30:00	Thread Pool: Worsened

5.4 Characterization of the crawled corpus

- **Total number of unique domains:** approximately 100,000
- Size distribution:
  - **Webpages per domain:** Exactly one per (sub) domain
  - **Tokens per webpage:** Ranges from 0 to thousands

5.5 Crawling Policies

- **Selection:** The crawler was configured to select URLs based on a set of rules, including domain filtering and URL depth limits.
- **Revisitation:** The crawler implemented a revisitation policy to avoid crawling the same URL multiple times, using a hash set to track visited URLs.
- **Politeness:** The crawler adhered to politeness policies by respecting the robots.txt file and implementing delays between requests to avoid overwhelming servers.
- **Storage:** The crawler stored the crawled data in WARC files, ensuring efficient storage and retrieval of web content. Compression was enabled to save space.

6 Conclusion

This project successfully implemented a parallel crawler using a variety of techniques to improve performance, culminating in notable speedups and scalability. The main challenges were related to the dinamic nature of the web and the need to balance efficiency

with politeness, also, using a new file extension (WARC) for storing the crawled data also posed some challenges.

7 Future Work

As explored in the previous sections, there are many areas for improvement and further exploration.

References

Received 31 March 2025; revised 28 April 2025