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 WARCs file, which is a standard format for archiving web pages, and compressed 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, 2 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 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.

A key learning was the use of WARC (Web ARChive) files, a standard format for storing web crawl data with metadata, enabling efficient archival and retrieval. Initial analysis showed excessive unneeded data, prompting header cleanup and compression to save space. Two issues emerged: 1. missing URLs due to absent HTML entries in the crawl dictionary, and 2. although batching 1000 pages into a single file seemed efficient, incremental saving proved simpler and more practical in implementation.

2.1 Robots.txt and Sitemap

To handle `robots.txt`, instead of using specialized libraries like *Protego*, the file was manually fetched and parsed using *requests*. Its contents were structured into a dictionary, mapping user-agents to their allow/disallow rules, along with keys for sitemaps, crawl delays, and miscellaneous data. For the MVP, a fixed 100 ms delay per domain was enforced.

Sitemap parsing proved more effective than expected, despite format inconsistencies, some in XML, others as HTML tables. Using *BeautifulSoup*, both were parsed similarly via BFS. One seed's sitemap alone yielded 50,000 URLs, nearly covering the MVP's needs. However, due to per-domain request delays, only part of this bulk was used; the rest of the frontier was expanded via standard `href` extraction.

2.2 Frontier

The crawler initially uses a **Breadth-First Search (BFS)** strategy, which is simple but inefficient for large websites due to url duplication. To manage the frontier and avoid revisiting URLs, a **set** is used to track visited URLs, though this sacrifices queue behavior. Alternatives like **deque** and **trie** structures are discussed: the deque helps with queue-like operations, and the trie is proposed for efficient storage and duplicate detection, especially when handling thousands of similar URLs from sitemaps.

For the MVP, a set is used for both the frontier and visited tracking to maintain simplicity and performance. Additionally, only domain names are stored for scraped URLs to simplify crawling

and avoid revisiting the same site, encouraging broader exploration across domains.

3 Parallelism

After building the MVP, parallelism was introduced using Python's `threading` library, but it was soon replaced by `ThreadPoolExecutor` from `concurrent.futures` for better efficiency. Initial performance gains were limited until debugging revealed that thread activity matched the number of seed URLs. Once the frontier expanded and more URLs were added, thread usage increased accordingly, significantly improving crawling speed.

4 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 downloaded corpus as it is parsed was less error prone than expected.

4.1 Algorithms, Data Structures and Complexity

The implemented crawler follows a multithreaded architecture using `ThreadPoolExecutor`, where each thread processes a single URL by retrieving and parsing its content. The process begins with a seed set of URLs added to a frontier. URLs are dequeued, checked against a `set` of visited URLs, and if unseen, are parsed and stored in a dictionary. Outlink domains are added to the frontier for further crawling. Parsed results are stored in compressed `WARC` files to save space, with unneeded header data removed. Although initially tested with batch-saving 1000 pages per file, incremental saving proved more reliable due to simpler implementation and fewer data integrity issues.

Core data structures include `set` for visited URLs (to eliminate duplicates), `list` and `deque` for the frontier, and `dict` for storing page data. Additionally, a `trie` was considered for efficient URL prefix storage, especially when working with sitemap-based crawling.

In terms of computational complexity:

- **Crawling:** $O(n)$ - where n is the number of pages, assuming one request per page.
- **HTML Parsing:** $O(c)$ - where c is the average size of the DOM tree for each page.
- **Data Storage:** $O(nm)$ - writing n pages of size m to WARC files.
- **Frontier Expansion:** up to $O(no)$ - where o is the number of outlinks per page, considering set membership checks.

Thus, the estimated total complexity is $O(n + c + nm + no + K)$, where K accounts for overhead from third-party libraries such as `requests` and `BeautifulSoup`, whose internal complexities are abstracted. Actual performance varies depending on page size, server response times, and the crawler's configuration.

4.2 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	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
40	537	1204	47.35	0,09	02:30:00	Thread Pool (Fixed)
40	539	598	58.98	0,11	03:03:20	No WARC Batch
50	100	558	6.26	0,06	**01:40:00**	
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	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
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
70	547	1721	51.55	0,09	02:30:00	Thread Pool: Worsened

(1) Number of threads used in the crawling process
(2) Number of URLs crawled
(3) Number of URLs in the frontier
(4) Time taken to crawl the URLs (in seconds)
(5) Time taken per URL (in seconds)
(6) Expected time to crawl 100,000 URLs (in hours:minutes:seconds)
(7) Description of the crawling process

4.3 Characterization of the crawled corpus

- **Total number of unique domains:** approximately $100,000$
- **Webpages per domain:** Exactly one per (sub) domain
- **Tokens per webpage:** Ranges from 0 to thousands
- **First big crawl size:** $\approx 30,000$

4.4 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 **set** to track visited URLs; **Politeness:** The crawler adhered to politeness policies but not by respecting the robots.txt, but by only request a single url for each subdomain; **Storage:** The crawler stored the crawled data in WARC files, ensuring efficient storage and retrieval of web content. Compression was enabled to save space.

5 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.

Received 31 March 2025