Information Retrieval – Programming Assignment 1

Alexander Thomas Mol Holmquist
alexthomasmol@gmail.com

ACM Reference Format:
Alexander Thomas Mol Holmquist. 2022. Information Retrieval – Programming Assignment 1. In *Proceedings of PA 1 (Information Retrieval).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Information Retrieval is concerned with information extraction and processing. The most common place to extract information from is the Web. In this programming assignment, we have developed a Web crawler: a program that recursively searches pages by following hyperlinks they contain. The set of downloaded web pages is referred to as "corpus". We have gathered a corpus of one hundred thousand pages.

2 CONSTRAINTS

2.1 The Constraints

The goal of this project is to download one hundred thousand pages. In order to do that, there is a set of constraints which must be taken into account:

(1) We must only download pages which contain HTML content.
(2) We must not revisit a web page.
(3) We must perform requests concurrently to make crawling time feasible.
(4) We must not "storm" the web page's host machine with requests.
(5) We must store the downloaded pages in the WARC, compressed, format.

2.2 Solution

At a high level, this is how we solved the constraints:

(1) After downloading some content, we check if the page header contains the right "content type" of "text/html".
(2) We keep a global cache of the crawled URLs, to check if a URL has been crawled in the past. If it has, we do not include it in the list of URL to be crawled in the future.
(3) We used the module "concurrent.futures" from the Python Standard Library to generate a pool of threads. These threads are not distributed across different CPU cores, but still avoid delays from IO operations.
(4) This is done by extracting a minimum crawling delay from the "robots" policy, which can be fetched from the URL */robots.txt*. If the policy does not exist or does not specify a delay, we use the default wait time of 0.1 seconds.
(5) We use the *warcio* Python library (see [1]), which does all the work of transforming the obtained HTML document into WARC format, and compressing the resulting file.

3 IMPLEMENTATION

The algorithm used for crawling can be summarized in the following steps. The algorithm receives as parameter a set of *seed* URLs. In our case, the seeds used were: *http://news.google.com.br/*, *https://exame.com/*, and *https://boaforma.abril.com.br/*. The crawling starts from these seeds, and proceeds by following the hyperlinks found therein.

```
Data: page_limit
Result: WARC files containing page_limit total pages
register seeds in global cache
for (each seed)
    crawl(http_pool, host, seed)
get results of crawl from completed worker threads
while (page_limit pages have not been crawled)
    get results of crawl from completed worker threads
    register the all new unique URLs in global cache
    for (host in the global cache)
        create dedicated HTTP pool
        crawl(http_pool, host, max of 25 URLs from host)
aggregate WARC files into 100 1000-sized files
```

Algorithm 1: Dispatcher (master thread).

Here, notice that the last step involves "aggregating" the WARC files into chunks with the right size. This is because we do not let each thread have a chunk of 1000 pages reserved to itself, because load balancing could be affected when crawling a smaller number of pages. Therefore, we let each thread lock files that will contain only 50 pages. After all crawling ends, we take these 50 pages-sized files and put them together to form files with 1000 pages each.

There are many details that were omitted from the description of the algorithms given here. These include specific details such as thread synchronization, imposing upper bounds to number of jobs, number of hosts, number of cached URLs, number of cached URLs per host, etc.

These fine-tuned limits turned to be critical to ensure the right functioning of the crawler. For example, if we don't limit the number of URLs per host, we might have a cache where one host contains 100,000 URLs all by itself, so that little space is left for other hosts. Another example is the total number of cached URLs. If we don't limit the total number of cached URLs, we might overflow the memory of the machine that is executing the program, and this list grows exponentially fast.

Details on the environment in which the program has been tested can be found in Appendix A.

```
Data: http_pool, host, urls
Result: host, num_failures, crawled_urls, new_urls
Find out my output WARC File and lock it
Find out robots.txt policy robots
for (url in urls)
    download page at url
    if (page does not have HTML content)
        skip
    else
        fetch all URLs in the page
        normalize all URLs and discard invalid ones
        add page to crawled_urls
        add normalized URLs to new_urls
    wait for robots.delay seconds
Release WARC file.
```

Algorithm 2: *crawl* procedure (worker threads).

4 CHALLENGES

4.1 Timeouts

Downloading some pages takes too long. We must impose a timeout to each request we perform. This timeout was defined to be 2 seconds for normal pages, and 3 seconds for the robots.txt policy.

Additionally, when the procedure *crawl* fails to download 5 of the 25 given URLs, it signals failure to the master thread, and the master thread will mark that host as unreachable for the rest of the program's execution. This goes to avoid retrying this download too many times. It took some time to figure out this heuristic.

4.2 Number of threads

It was very hard to determine which is the right number of threads. Obviously, the ideal number varies according to how many page one wishes to crawl. However, always towards the end of the program's execution, we want less threads, otherwise there is high race for few left over non-full warc files.

4.3 URL format

It was also a challenge to properlly normalize the URLs before attempting to crawl any of them. This is because the "href" attribute the links of the crawled pages can come in many different formats, including things like *mailto:* or *tel:*. It took some time to tune our parsing for all of these corner cases.

4.4 Printing debug information

The program has an option *-d* which turns on "debug mode". In this mode, for every page crawled, the program will emit a JSON-formatted output containing specific fields. Even though this is a simple task, this by itself required many hours of work to get right.

5 STATISTICS

We were not able to extract the statistics required in the assignment, due to time constraints. We do know that the total number of unique domains (hosts) crawled was around 10,000.

6 OTHER CONSIDERATIONS

Below we show some adjustments that might improve the overall performance of the program. We did not have time to test these.

- Assign a TTL for each URL, so that the cache is periodically cleaned.
- Do not download "minor sites". That is, don't download websites that don't contain much relevant information.
- Use different processor cores to perform computations (we only use one).

A TESTING ENVIRONMENT

The crawler was written in Python, and runs over the Python Interpreter. The crawler has been tested with Python version 3.8, in a machine with Ubuntu 20.04.4 LTS, Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz with 4 cores, 16 GB of RAM.

REFERENCES

[1] *The warcio Python library.* https://pypi.org/project/warcio/.