Programming Assignment #1 - Web Crawler

Daniel Freire
daniel.carneiro.freire@gmail.com
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

ABSTRACT

In this assignment it was implemented a crawler capable of fetching a mid-sized corpus of webpages in a short time frame while respecting politeness constraints defined by each crawled website. The corpus was then characterized by a number of statistics. The collected corpus can be accessed at https://drive.google.com/drive/folders/1rxzYReSFWOaZVV-mLrI5mS1l36NLZKxP?usp=sharing.

ACM Reference Format:
Daniel Freire. 2022. Programming Assignment #1 - Web Crawler. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Crawling the web and building a corpus is the first step of information retrieval, in this assignment we crawled 100,000 webpages and stored them in WARC format, with gzip compression.

First we'll explain the architecture of the algorithm that was implemented, then we'll point out challenges and current issues with the implementation, thirdly we'll expose the results, followed by the final considerations.

2 ARCHITECTURE

In this project we chose to design and implement an architecture largely inspired by the Elixir programming language, that tries to maximize concurrency and minimize the problems that come with it.

A diagram of the implemented architecture can be seen in figure 1. The idea is that a Crawler object spawns a $2 * \text{cpu\_count} + 3$ processes each dedicated to only one task, where cpu_count is the number of cpu cores (virtual and physical) of the machine, $2 * \text{cpu\_count}$ processes are Crawler.workers, one is a Manager, one a Logger, and one a WARCWriter.

In the next session we'll explain in more detail how each of these components work.

2.1 Crawler.workers

The Crawler.workers are the actual crawlers, they request and parse the webpages, then it adds the new links it found to the frontier. Algorithm 1 further explains what the worker does. It is worth noting that in the actual implementation, after the worker gets a new url from the frontier it spawns a new thread to do the rest.

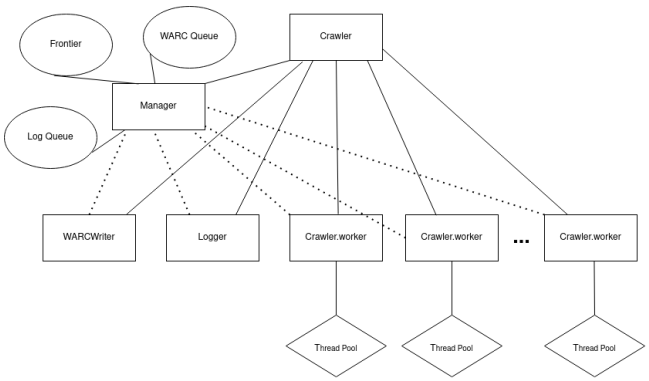

Figure 1: The proposed architecture

Algorithm 1 Crawler.worker

```
while True do
    url ← get new url from frontier
    wait_politely(url)
    head ← HEAD(url)
    if head response is successful and document type is html
then
        wait_politely(url)
        resp ← GET(url)
        add resp to WARCWriter queue
        for all links in resp do
            Send the link to the Frontier
        end for
    end if
end while
```

2.2 WARCWriter

The WARCWriter is the process that writes the WARC files. It's good to have one process dedicated to this, since having multiple processes write multiple files can get complicated. Another advantage of doing it this way is that it is easy to closely monitor how many pages we've saved.

2.3 Logger

The Logger is just used when the debug flag is set, and it just prints a summary of the fetched webpages sequentially.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.4 Manager

And finally the Manager, this is an object from the multiprocessing library in python that is capable of storing objects and exposing via proxies their methods for any process to use. Every method call that is sent to the manager is ran a new thread.

In this implementation the manager is storing the shared Queues, the Frontier as a whole and a few counters.

The most interesting object here is the frontier, in particular the enqueueing function, instead of checking if we should crawl the url at the crawler, we check if we will want to crawl it when we enqueue the url. If the url meets the criteria (this involves if the url has already been visited, robots.txt restrictions and depth restrictions) for being visited in the future we add it to the queue, otherwise we don't.

The usage of a Manager is simple and intuitive, however it does produce it's own set of problems as we will discuss in section 3.

2.5 Complexity

The overall time complexity for n documents is $O(n)$, since each used function is has a time complexity of $O(1)$.

The usage of shared objects via the manager makes it so the space complexity is $O(n)$, however this can be reduced by setting limits on the objects size so that the space complexity is $O(1)$.

3 CHALLENGES AND ISSUES

The first problem that arose from this implementation was the memory usage, the longer a worker process ran the more resources it consumed. The exact reason for this behavior could not be determined, however the issue was solved by restarting a worker after a certain number of urls crawled.

Another problem has to do with the Frontier and its Manager. As said previously the manager creates a new thread for each request it receives, this means that its computations are not sequential. From one side this is a good thing, since it means that if something were to block (eg. the Frontier.enqueue function requesting the robots file), it does not compromise the execution of the program as a whole. However it introduces two problems, one with politeness, and another with revisiting webpages.

First with politeness, if two workers call the Frontier.wait_politely method at the same time, and they have to wait since one worker visited the page recently, they'll both exit the function at roughly the same time, and make a request at roughly the same time violating politeness. Another issue is if they don't have to wait, two workers can possibly make requests to the same host at the same time (or almost at the same time).

On revisiting webpages the issue is similar, if more than one worker tries to add the same url at the same time, possibly they will both add the url to the queue. This issue had to be mitigated in a less than ideal manner by having the WARCWriter keep the set of written urls, and make it so it does not write html files with a repeating url. This means that we do repeat requests on certain urls, however we do not do repeat writes.

These issues related to the Frontier are still present in the submitted version of the assignment, since those were only noticed one day before the deadline.

4 RESULTS

Running the crawler in 32 processes with 40 threads each it collected 100 thousand web pages in a little over 2.3 hours, if the worker processes are not restarted it takes about 1.3 hours. The perceived bottleneck was a limit on the amount of concurrent requests, when too many requests are made at the same time new requests start taking a long time to get a response. Of the 100Mb available bandwidth at most the crawler used at most 20Mb but usually stayed around 15Mb.

The crawler was configured so that after 10 urls from the same host were added to the frontier it did not add anymore urls from the same host (However because of the issue described previously, this did not work as intended).

The produced corpus contained 100 thousand unique urls, from 57 thousand unique hostnames. Meaning that on average there were approximately 1.74 documents for each hostname. However the hostname with the most documents crawled had a whopping 820 pages (This hostname was www.tumblr.com). How many documents were fetched from each hostname is show in figure 2.

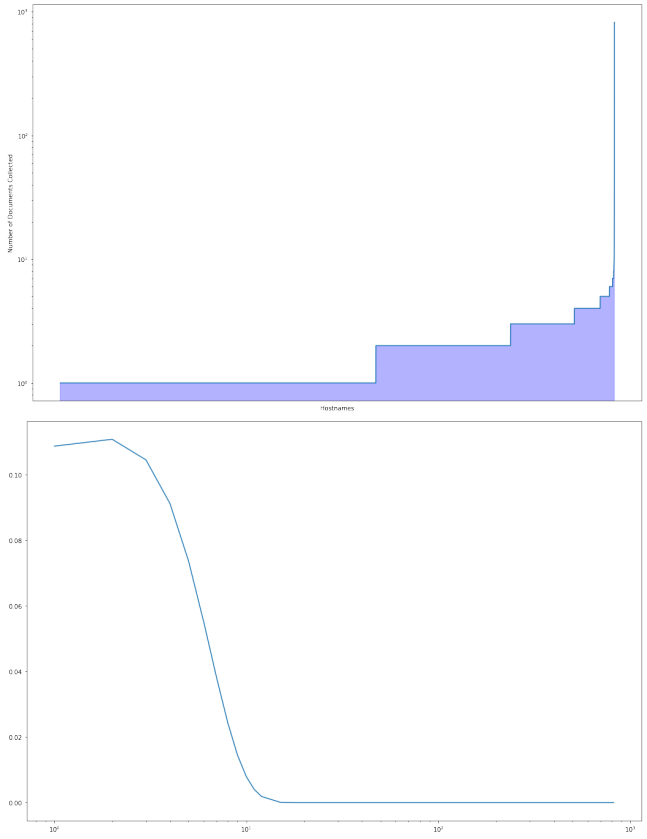

Figure 2: Documents Fetched from each Hostname, and its Distribution

Furthermore the distribution of the document sizes in terms of tokens can be seen in figure 3. As we can see most documents have less than 10^5 tokens, and very few have more than 10^6.

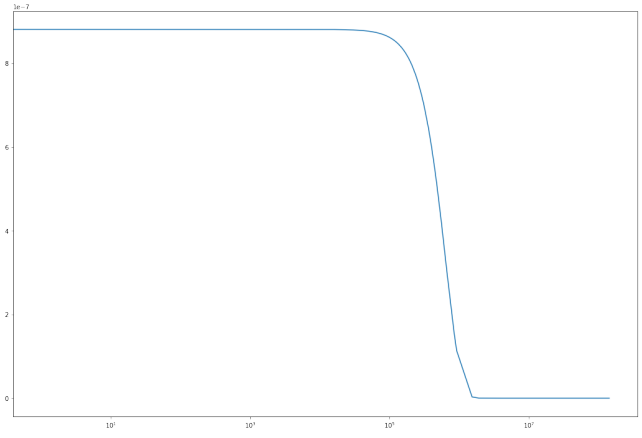

Figure 3: Size Distribution per Hostname

5 CONCLUSION

Although the implementation had some unfixed issues, caused by concurrent requests not being synchronized when waiting politely to visit a website again, and by repeated urls being enqueued at the same time, the implementation did successfully fetch and write 100 thousand webpages in a timely manner.

Further work is needed however to fix the aforementioned issues. Both issues have relatively simple fixes. For the politeness issue, we could calculate when the next visit would be allowed, instead of how many seconds the crawler has to sleep for. Then any calls to the politeness function would update that next visit time by the amount of time specified in the robots.txt.

For the enqueueing issue some refactoring is needed. The enqueueing function is sufficiently fast overall, but has a bottleneck when it collects robots.txt files, therefore running it in one thread in its current state would be inviable. A possible solution would be having only the part that requests the robots file be concurrent, and have the rest of the function be synchronized in a single thread.

As said before, the structure of the crawler was inspired by the Elixir language, however Elixir provides a infinitely more concurrent environment than python's standard libraries, and also easy ways to communicate between processes in a plethora of manners, also it provides great tooling for monitoring running processes and its resources. So it is no surprise that trying to create a (very) simplified version of it in a completely different environment would come with its own set of challenges.