Web Information Extraction and Retrieval Programming Assignment 1

Jure Bevc, Luka Tavčer, Grega Dvoršak
University of Ljubljana, Faculty for Computer and Information Science
bevc.jure@gmail.com, lt7339@student.uni-lj.si, gd4667@student.uni-lj.si

1 Introduction

This report presents the details and results of solving an assignment on the subject of web information extraction and retrieval. The goal of the assignment was to implement a standalone crawler which crawls on gov.si websites. The architecture of the crawler consists of a HTTP downloader and renderer, a data extractor, a duplicate detector, a URL frontier and a database. In the report we present our solution and its specifics in detail and present some general statistics of the results.

2 The Implementation

One of the main components of the crawler implementation is the database, which stores data about the crawled sites and pages. The basic schema was predefined along with a prepared SQL script [1] which was used in our implementation to create the database. The database we used is a PostgreSQL database. The basic schema was extended with 4 new page type codes:

- **DISALLOWED** for pages that were found in robots.txt and shoudln't be fetched,
- **DOMAIN_ERROR** for pages whose domain couldn't be resolved to an IP address,
- **TIMEOUT** pages that were unreachable (20 seconds),
- **WEBDRIVER_ERROR** for any other exceptions that occurred during page loading in the webdriver (browser).

We added a **html_content_hash** field to the **crawldb.page** table in order to detect page duplicates by their HTML content. We used Python programming language and created a database driver to abstract SQL queries from the Python crawler with methods like `create site`, `create page`, `update page`.

2.1 Crawler and frontier initialization

Our crawler is created in a way, that it can stop at any time and continue from where it has stopped. We start by fetching every site's `robots.txt` content from the database and parsing it with urllib's robotsparser [2]. Every site's robotsparser is then stored in the dictionary of all robots parsers and serves for the purpose of robots content caching, which is then passed to the frontier manager.
Next, we exclude all starting URLs that are already processed, i.e. are already in the database and fetch all pages of the FRONTIER type, stored in the database, and pass them to the frontier as starting urls. The number of crawlers is set as a parameter to the program and each crawler is started as a parallel process in a separate thread.
Crawlers first preform basic configuration. where they set parameters like the user agent header and a page loading timeout (**TIMEOUT**), which is set to 15 seconds. Every crawler instance also creates its own database connection, so that threads don't interfere with each other.

2.2 Crawling

After the initialization, crawlers start iterating trough the frontier's URL queue which is implemented as the FIFO list, first URL that comes to the list is also the first that leaves it, this also guarantees that we have a breadth first searching strategy. However this is not really 100% true, because we can crawl with more than 1 crawler in multiple threads. And for example we have 4 URLs from the first level in the queue and we crawl with 5 crawlers. The first 4 URLs will be taken by the first four crawlers, but the fifth crawler will receive the URL from the next level, if it exists.

Next, crawlers extract the domain and base URL information from the received URL and check if the site with that domain already exists in the database. If it doesn't, they create a new site, fetch its `robots.txt` content and update frontier's site robots dictionary. At this step it is also checked if the URL is allowed (can be fetched) according to the site's `robots.txt` content. If URL is marked as disallowed, it is discarded. Otherwise if all checks have passed, we check if the page with the same URL already exists in the database, if it doesn't, it means that it was passed as one of the 4 starting URLs and it gets created. If the page exists, it is fetched from the database.

When a URL is assigned to a page, as is the case at page creation, the URL is canonicalized using the urllib library [2]. Relative URLs are joined with the base URL and normalized with the urltools library [3], which formats URLs in a generalized way to ensure all URLs are correctly canonicalized. In the process we also remove URL fragment identifiers (#fragment). If we don't remove fragment identifiers, we will detect a lot of duplicates by HTML content later, when we fetch these pages, but this would waste our resources, so we remove them.

The requirement of a five second delay between consecutive requests to the same IP address is checked after the above described page handling. Crawler updates request time history for every IP when request to that IP is made. A difference between the current time and the time from the frontier's request history is computed to check if more than five seconds have passed. If the difference is too small the URL is returned to the start of the frontier queue and another one is taken.

After the appropriate URL is chosen from the frontier, we start by creating a HEAD request to check the response headers. First we check if the `Location` header is present, as it means that the server redirected as to another resource. We follow redirects until we get to the last one. We then check `Content-Type` header and if it is not `text/html` we update the page with the BINARY data type and skip all of the following processing. We also remember the last request status code and the content length of the returned resource. The content length is used to see if the resource is bigger than 10 MB as we mark those resources as BINARY files too.

If the HEAD request failed or didn't show that the resource is a binary file, we proceed to retrieve the resource with a GET request. If the request was successful, a HTML content hash is calculated to detect duplicates. If the hash matches any other in the database, it means that we have found a duplicate page by HTML content and we mark it with a DUPLICATE page type.

Then we traverse the DOM to find links in the href and other tags. A HTML parser is used to detect email links, which are ignored, and the beautiful soup library [4] is used to detect links in **onclick** elements. Image link URLs are extracted from the **img** tags where the URL is in the image source.

Links are then on the way to be added to the frontier. URLs are canonicalized and those from non **gov.si** domains are discarded. If they are from any known and already parsed site domain, they are also checked by their `robots` contents and discarded if disallowed. For every URL it is also checked if any page with the same URL already exists and if it does, the URL is discarded as it is a duplicate (on this point, we have encountered a major misunderstanding of duplicate finding instructions and will describe it in the problems section). If the URL is found to be on a new site domain, we just create a new page in the database with the FRONTIER type. An entry in the link table is also created in this step with the source and destination URL added.

The last step is to add new URLs to the frontier queue, set any remaining database values and add retrieved image data such as the source and type to the image table in the database. The next URL from the frontier is then selected and the complete process is repeated until there are no URLs left in the frontier.

3 Problems and solutions

We have encountered many obstacles while coding the crawler. Many of them are related to other topics than crawler itself. A lot of effort was given to the database driver so that in case of failed queries everything goes back to normal state and the following transactions don't cause trouble.

Everything also changed when we implemented multiple threads, although we anticipated problems there and were prepared. Every crawler or thread created its own database connection. One problem was when multiple crawlers discovered the same URL and tried to create a site or a page on the same time. In that case one of them failed because of the unique URL constraint, exception was caught and crawler received a new URL from the frontier.

There was also a lot of irregular URLs which needed to be handled separately. For example `mailto` and email addresses were found among the URLs, wrong domain names and unreachable websites were also found.

Previously, in the Crawling section, we mentioned another problem when trying to find duplicate URLs. As we read the instructions we understood that we need to save every found URL to the database. Therefore when we extracted new URLs, we found a lot of duplicates by URL and we marked all of them correctly as duplicates. In that manner we found almost 1 million URLs, among those there were 20,000 HTML pages and almost 980,000 of them were duplicates. With that information, we could also calculate every page's or site's rank as those with a lot of incoming connections are probably more popular. But then we were comparing the statistics with another group and changed our algorithm to discard all URLs that are duplicates by URL and take into account only those that are duplicates by HTML content.

4 Results and Statistics

In this section, we present a basic overview of the results of the assignment. Some general statistics such as number of sites, number of web pages, number of duplicates, number of binary documents by type, etc. which are measured for the sites that are given in the instructions seed list and for the complete database are presented in Table 1. Figure 1 shows a visualization of links between web pages based on data from the crawler database.

TABLE 1
Statistics of crawling taken from the database.

measured	seed list	database
sites	4	159
web pages	15420	25035
HTML pages	10002	14298
duplicates	9	599
binary files	976	2197
pdf documents	/	1061
doc documents	/	109
docx documents	/	144
ppt documents	/	5
pptx documents	/	4
images	3.75	24001
avg images/page	0	6.45

There we mention that approximately 5000 pages was still in the frontier. And images are only those that were found in the **img** tags, but there were also URLs to images. A lot of pages with binary files were also unreachable. There we see that 44% of binary files come from the initial 4 websites.

5 Conclusions

In conclusion, the assignment was an interesting challenge where we have implemented the requirements and have gradually obtained a working solution.

References

[1] A prepared sql scipt for database initialization. https://szitnik.github.io/wier-labs/data/pa1/crawldb.sql. Accessed: 2020-01-04.

[2] The urllib library. https://github.com/python/cpython/tree/3.8/Lib/urllib/. Accessed: 2020-01-04.

[3] The urltools library. https://github.com/rbaier/python-urltools. Accessed: 2020-01-04.

[4] The beautiful soup library. https://www.crummy.com/software/BeautifulSoup. Accessed: 2020-01-04.

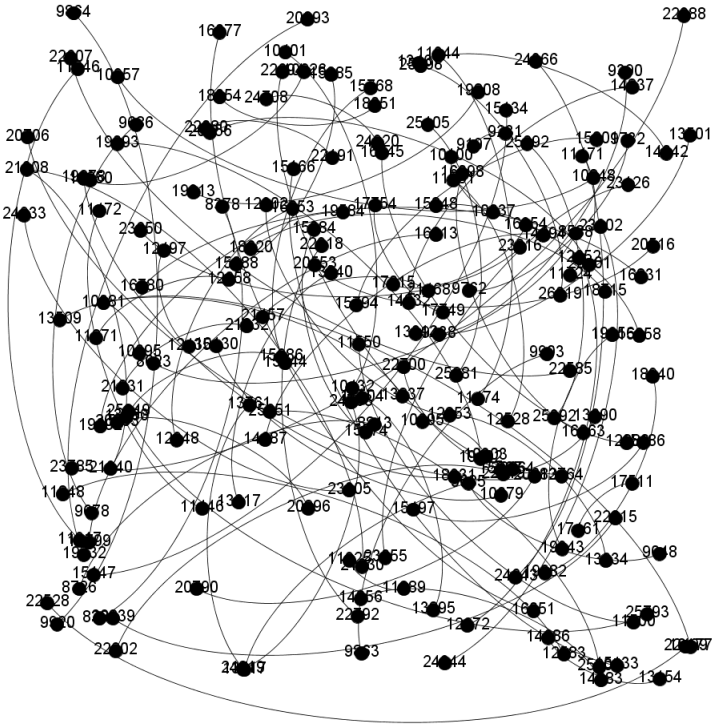

Figure 1. A visualisation of links between 100 most popular web pages in relation to most outgoing links web pages.