Web Information Extraction and Retrieval - Assignment 1 Report

Roman Komac, Dik Medveščček Murovec, Žan Ožbot

April 2019

1 Introduction

The goal of this programming assignment was to build a standalone crawler that will crawl only .gov.si web sites.

We approached this problem by creating a web crawler according to the definition in section 2. This crawler ran for **INSERT TIME** hours. In that time it covered **INSERT NUMBER** sites. Further data description and visualization can be found in section 3.

2 Crawler implementation

We took a look at the proposed structure, as seen in figure 1, for a simple web crawler and adapted it a bit. All the proposed functionalities are still present, but may be included in some other components.

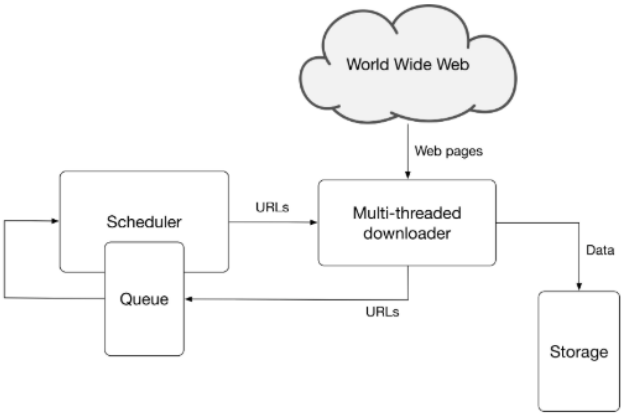

Figure 1: Proposed structure for a web cralwer

2.1 HTTP Downloader & Data Extractor

HTTP Downloader & Extractor is in essence composed of two components. 'Get binary file' component and 'Get website content' component.

Get binary file

This component is used only when the URL points directly to a binary file. We use it to quickly retrieve PDF, PPT, PPTX, DOC and DOCX files. If the robots.txt allows connection we use the requests library to retrieve contents, headers and the status code.

Get website content

To retrieve website content we opted to always use Selenium with the Chrome headless browser. If robots.txt allows connection we try to get page contents as HTML source. From the HTML source we extract all links ending in .gov.si and return them to be added to the frontier.

If the tasks requires it, the component has support for downloading of images. We look through all the ¡img¿ tags within the HTML source and download respective images. The component as a whole returns links to be added to the frontier, raw HTML source, status code, page type, images and time accessed.

Both components also have a timeout exception - if a website doesn't respond in 30 or 10 seconds, depending on the component, we ignore it.

2.2 Crawler

Our crawler has a predefined seed list of 9 base URLs. We use the crawler to create the process pool and after get the number of URLs in the frontier.

After that we get all the sites and create the robotsparser directory, and bind a dictionary of robotparsers.

We finally load the database or start from the seed list and finally get the site data.

2.3 Tools

In some cases the default libraries were not enough. In order to get the required functionality we had to recreate some tools.

- **Get mime type tool**: Accepts HTTP request header type and decodes and returns its MIME type
- **Ending to datatype tool**: Returns site datatype base on its ending
- **Get sitemap locations tool**: Finds all locations within sitemap and returns their canonized URLs
- **Get domain tool**: Returns domain of an URL

- **Robots tool**: Gets robots data, parses it and returns it.
- **Canonize URL tool**: Canonizes URL

2.4 Pool

To connect together document retrieval with database management we have created a pool class. Its purpose is to assigns tasks dynamically depending on the links available in the frontier. Our first iteration of the pool class had a major drawback. Since the frontier was a shared queue the items could only be taken out in the order governed by the sequence at the time of insertion. Most websites usually contain links that belong to the same domain space. When taking into account crawl delay the whole implementation therefore offered speed-ups only when sequential links belonged to different domains. The implementation as a whole therefore proved to be only as efficient as a single-threaded crawler.

The second iteration of the pool object was implemented using only basic processes forking and shared states. By selecting this method over the pool manager we gained more control over the states of each process. Instead of a single queue we divided the frontier into several queues, each corresponding to a unique subdomain of the *gov.si* domain space. This enabled us to be more selective in the processing order.

For crawling through the domains we have used eight concurrent workers. Their sole cooperation is done through a shared queue collection. During initialization every process starts a connection to the database and a selenium driver session. After that it enters a loop which terminates only when the whole collection of queues has been emptied. After a process downloads a website together with its corresponding components and inserts it into the database it acquires an exclusive lock over the shared collection of queues. From that collection it chooses the queue which timer has exceeded the crawl delay time. If there are none, it sleeps for a certain amount of time before checking again. Each process must also contain knowledge of whether it should crawl a specific webpage in a domain. To alleviate synchronization difficulties and the need to constantly check if an appropriate *robots.txt* has been parsed we have created a helper function named *get_make_robotsparser*. We share preprocessed *robots.txt* in another synchronized collection. Each time a process requires *robots.txt* of a specific webpage this function is called with a given webpage url. If the domain of the webpage already exists in the collection the appropriate object is returned. Otherwise we check for the file in the database. If no record is found we download and preprocess the file which in turn gets inserted into the database. If available, we also read and set the crawl delay limit for that domain. Finally we return the appropriate robotparser object.

Functions *process_website* and *process_documents* also defined in the same python file are merely callbacks for when a document has been downloaded and processed. Each function inserts appropriate data into the database. Which function will be called is decided inside of *HTTPDownloader.py*. The selection is done depending on the mime-type of the retrieved document. Documents with

mime-types corresponding to *xhtml* and *html* are passed to *process_website* callback while documents with mime-types specified for binary files are sent to *process_documents*.

2.5 Database

The given database was extended with another table called *frontier* which works as a storage for the frontier in case the application suddenly crashes. The whole database diagram is shown in figure 2.

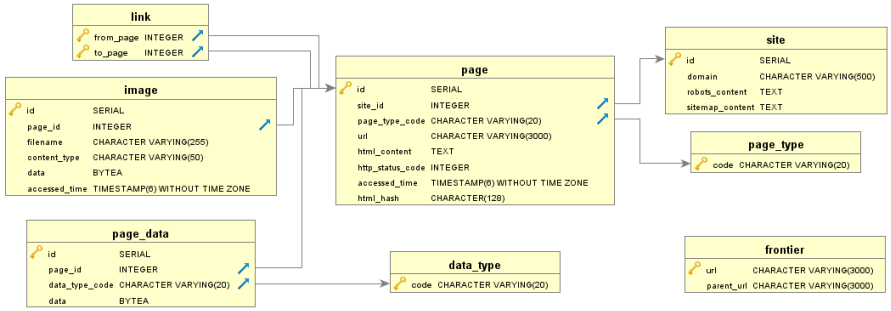

Figure 2: The extended database diagram.

The data from the database was retrieved with SQLAlchemy library which acts as an ORM (Object Relational Mapper). Reflecting each datatable we need to use is pretty simple. The following example below demonstrates the creation of an object which is mapped to its corresponding datatable.

```
class Page(Base):
    __table__ = Table('page', metadata, autoload=True, schema='crawldb')
```

3 Results

The network on figure 3 consists only of crawled seed urls, since the visualization of the whole network would freeze or crash.

The figure 4 shows the crawled seed pages per page type. The procentage of the binary file types can be seen in the figure 5. When crawling through the seed url list the average number of images per page was 2.6, the average number of binary files per page was 1 and each site had an average of 213.7 pages.

The mentioned stats can be found in figure 6. The crawling of the seed url list resulted in getting 13 sites, 641 pages and 168 images, while the whole crawling process resulted in getting 282 sites and 14443 pages. The statistics are shown in figures 7 and 8 accordingly.

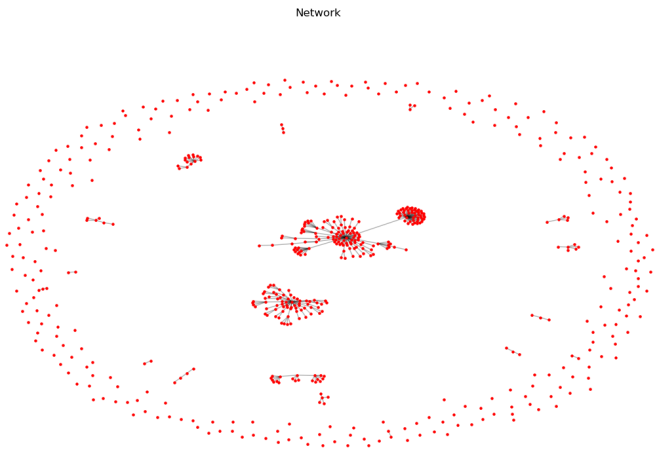

Figure 3: Representation of the network consisting of seed urls only.

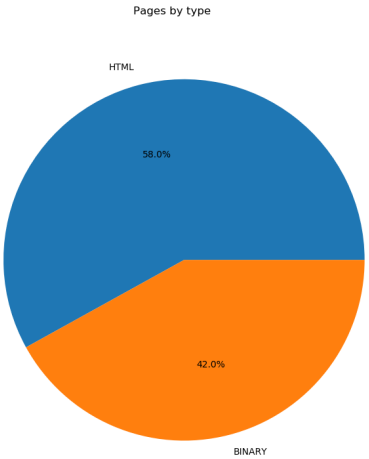

Figure 4: Pages according to their page type.

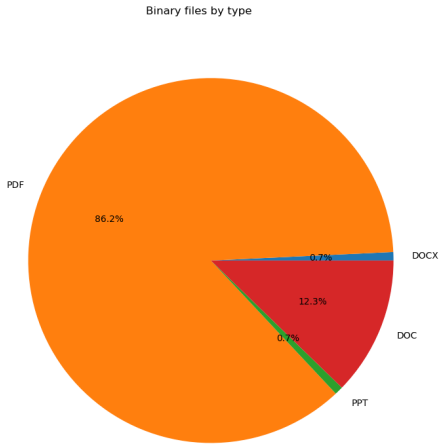

Figure 5: Grouped procentage of binary types.

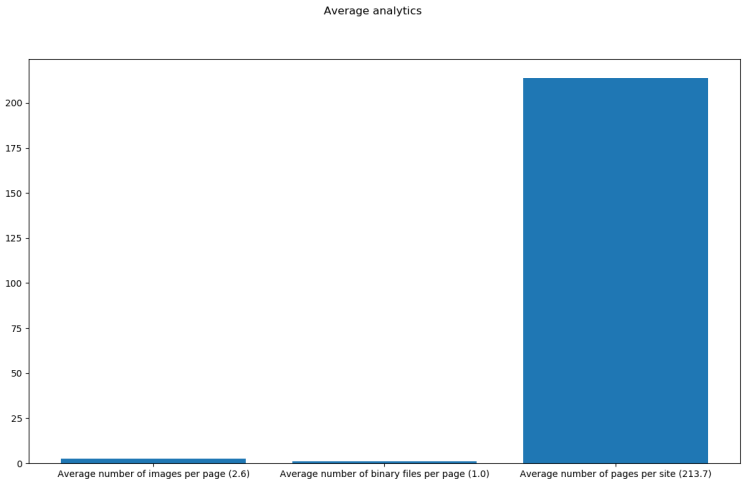

Figure 6: Average analytics of seed url list.

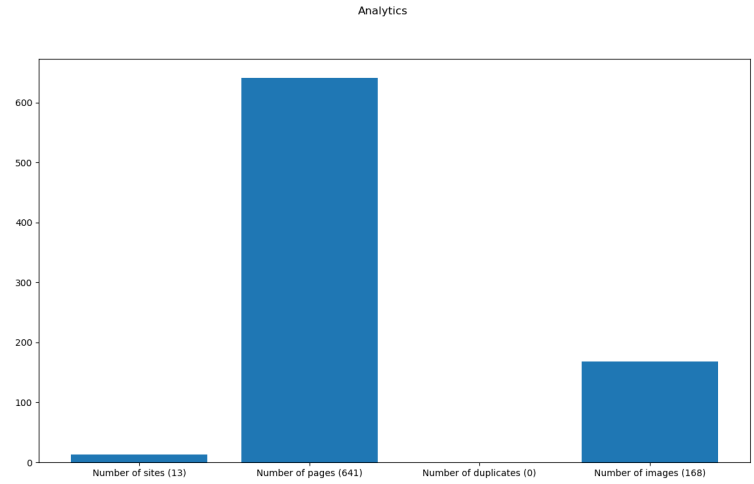

Figure 7: Analytics of seed url list.

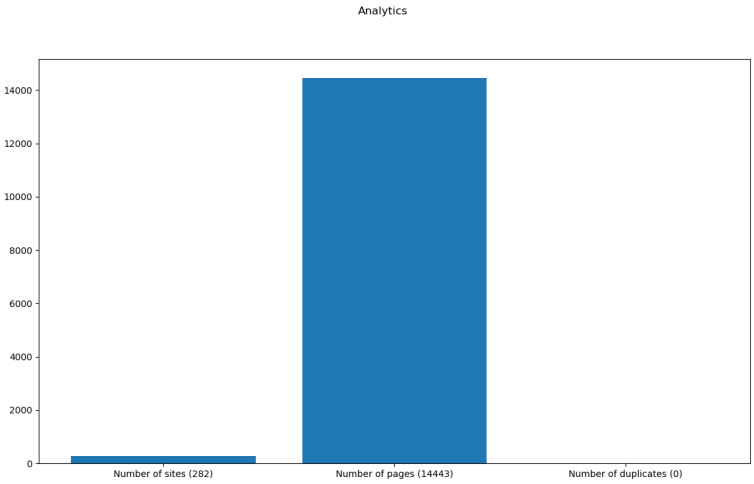

Figure 8: Analytics of the whole database.