Programming assignment 1

Viktor Petreski and Lodi Dodevska

I. Introduction

For the purpose of this assignment we implemented a crawler that crawls **.gov.si* pages. The crawler is implemented in Python and for storing the information from the crawled pages we used PostgreSQL database with the provided *crawldb* model.

II. Methods

The HTML page was extracted using *Selenium* and the content was separated from the HTML/Javascript tags using *BeautifulSoup*. Furthermore, we used *requests* library to get the header of the response which was needed to fill in the data for status code and redirected URL. Since we expected duplicates to appear, we implemented a version of MinHash algorithm for distinguishing duplicates from the originals.

A. Implementation

The crawler was initialized with the seed sites that were provided. For each seed site, its *robots.txt* was retrieved and partially parsed using *robotparser* library (part of native urllib). We used this library to find out if the current URL was allowed to be visited by crawlers and to get the timeout, if provided. A dictionary that was used as a map between the URL and its RobotParser object obtained from the parsed robots.txt. Since the version of python that we used did not provide sitemap retrieval, we used requests to get the sitemap content.

As frontier, we used OrderedSet which provided BFS approach to the crawling, while ensuring that we will not have undetected duplicate URLs in it. We also maintained binaries and seeds set for this types of URLs. Besides the original sets, we had lists for the duplicates that were found during the crawling. Keeping duplicate lists made it easier to avoid trying to add new URLs in the page table and causing exceptions.

Before an URL was crawled for its content, it was checked if its domain was part of the seed sites and if not, first we parsed its robots.txt and added the domain to the seed sites set. **A dictionary was kept for each seed site URL and its ID in the database. That is explained further below**.

REPLACE javascript with JavaScript later

B. MinHash

To identify duplicate content we used MinHash algorithm, since comparing the whole content did not seem efficient and it would not find similar contents. The first step in implementing the algorithm was to preprocess the extracted content, meaning remove the punctuation, make everything lowercase and divide it in words. We decided to use three-grams (three consecutive words) and used 64-bit MurmurHash hashing function to encode the "sub-sentences" into 64-bit integers, to avoid collisions. We chose MurmurHash because it returned consistent hashes unlike native python *hash* function which was random. Since the function returned negative numbers, we used and operation between the generated integer and the largest 64-bit integer.

$$h(x) = (a * x + b) \mod c \tag{1}$$

That ensured positive integers. These integers are needed in the final hashing function (Eq. 1), since it is needed to randomize their order. The coefficients a and b are randomly chosen ($a, b \in [0, 2^{64}]$) and stayed the same for the whole time and $c = 2^{64} + 1$. We generated 25 hash functions, of which the algorithm takes the minimal one, since the name MinHash. This means that each HTML is encoded with 25 hashes and they form its signature. The signatures are kept in a list and in the same time we kept a dictionary that maps the URL to the its signature for easier lookup when a duplicate came. The comparison is done using Jaccard similarity. If the similarity was larger than 60%, the HTML was classified as duplicate. Comparison times were negligent and most of the time less than 0.5 seconds.

C. Crawler

The HTML content of each page was parsed to find the next links from the *a* tags and the javascript's *onclick* event that contained *location* as part of the code. The HTML tags was easier to obtain, since Selenium provides this option. For the JavaScript part, besides Selenium, we used regex expressions to extract the next location/URL. The image sources were also extracted using Selenium, from the *src* attribute of each *img* tag. The next URLs were combined in one list and processed further. If the URLs were relative, we appended the domain from which they came from. We also made sure to remove the *www* from the URLs and only use *https* protocol. Furthermore, we checked if the URL contained *gov.si*, because we did not need to crawl other domains. Before adding them to the frontier, first we checked if there are ones that are binaries, using regex expressions. If the observed URL was not in the frontier set, it was added to both the frontier and the duplicates. The same process was applied to the binary URLs. After iterating over the new links and binaries and adding them to the database, the content from the HTML was extracted. The *script*, *style*, *[document]*, *head* and *title* tags were removed, to try to save only the unique content for that page. Then we extracted the text from the remaining HTML tags, removed trailing spaces from each line and combined them in one string. This content was saved in the database if it passed the MinHash criteria, otherwise it was set to None. We ran the crawler multithreaded using 12 threads at once. To prevent simultaneous access of different threads to the database, we used one lock at each access point. Furthermore, we used different Selenium driver and database connection for each thread.

D. Communication with database

The crawler uses the *psycopg2* driver for communicating with the database. For each page there are multiple accesses to the corresponding tables, depending on whether we need to execute INSERT, UPDATE or SELECT query. Every query execution is surrounded with appropriate try/except block, which deals with potential problems that can occur (e.g. non-existing *page_id*, insertion errors, duplicate link etc.).

We created a dictionary that stores the seed site IDs from the *site* table as keys and the corresponding URLs as values. Because of this, we do not have to access the database every time we are adding a new page and we need the appropriate

site ID. When there is a new seed page in the frontier we add it to the *site* table and we update the dictionary.

Every new page coming from the frontier needs to be crawled. First, we find the *site_id* from the dictionary with seed site IDs. Then we add the page to the *page* table, but the *html_content* column is not filled yet. The content is added after the page is parsed. During the parsing we check for a couple of things: binary files, images, duplicates and links to other pages. At the beginning of each loop the *add_link_flag* is set to False. If we find anything that needs to be added to the database, the flag will be set to True.

First we check every potential URL. If there is a new binary page found in the current page it is added to the *page* table first, with the *page_type_code* set to 'BINARY' and *html_content* set to None. Then, we check if the binary page is an image (because it may represent a link to some image) or a document (.pdf, .doc, .docx, .ppt and .pptx). If it some kind of document, it is added to the *page_data* table. If it is a link to an image, it is added to the corresponding *image* table. The *add_link_flag* is set to True in both of these cases.

If the crawler finds an URL to another page, first we check if it is a duplicate. If it is not a duplicate we add the new URL to the *page* table and we set the *page_type_code* to 'FRONTIER'. This page will be processed and its content will be updated once it is visited. Again, we set the *add_link_flag* to True.

Finally, if the found URL is duplicate, it is added to the *page* table but the content is set to 'DUPLICATE'. The *add_link_flag* is set to True.

If the *add_link_flag* is True at the end of the loop, we update the *link* table because we need to add link from the current page (*from_page*) to the found duplicate, image, binary page or the newly found page (*to_page*).

Next, we check if there are any *img* tags on the page. For each *img* tag we check its *src*. If the image URL is not duplicate we insert into *page*, *image* and *link* table, just like we did with the other binary pages and image URLs.

After we have finished with the processing of potential URLs, we finally update the content of the page. We use MinHash algorithm (explained in the previous section) for identifying the duplicate content. If the algorithm recognizes the current page as a duplicate, we find the *page_id* of the original page from the *page* table and we insert new link from the original page to the current page into the *link* table. If the content is not identified as a duplicate, we extract the content and we update the *html_content* of the current page in the *page* table.

E. Problems and solutions

Initially we had a problem with the duplicate URLs where the database threw a lot of constraint errors, because we forgot to add the unique URL to the duplicates when adding it to the frontier. While trying to run the crawler on multiple threads, we forgot to initialize new Selenium driver for each thread and a lot of pages were not crawled since the content was replaced with the last page requested. We also faced some problems with unresponsive pages, where the timeout would expire and the crawler would stop since it threw an exception. That was handled with try and except blocks. Also, some of the sites had an expired certificate, which prevented the crawler from accessing them, but we added a flag to the requests call and it was fixed. The crawler also leaked memory, because while handling the exceptions, we forgot to check the database connections and close them if any were active and same thing with the Selenium drivers.

III. Results

We retrieved 40.000 HTML pages. Tables III and III show the general statistics for *crawldb* database. Figure 1 shows the number of crawled pages on each 30 minutes and Figure 2 shows the number of pages by type that are crawled each day. On the github repository there is a network graph for *e-prostor.gov.si*. We did not attach the graph to the report because it was be pointless to attach such a complex image as a very small picture here.

Total numbers				
sites	**htmls**	**duplicate**	**binary**	**images**
0	38343	38389	17600	4900

Table I

doc/docx	**ppt/pptx**	**pdf**
2782	55	9765

Table II

Figure 1. Number of crawled pages each 30 minutes

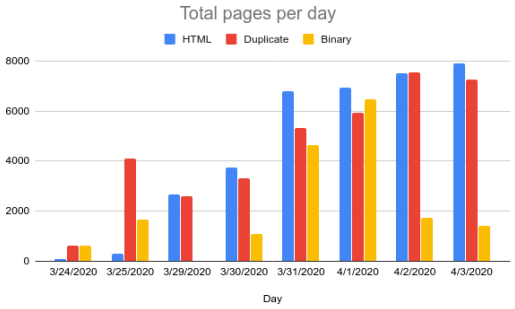

Figure 2. Number of pages by type crawled each day

IV. Conclusion

The assignment was very demanding and detailed, but also very interesting. We learned a lot of new things and we are looking forward to more work in the future.