WIER 2021: Programming assignment 1, Report

Klemen Stanič, 63150267
Email: ks4211@student.uni-lj.si
Luka Kavčič, 63150139
Email: lk2633@student.uni-lj.si

I. Introduction

The goal of this assignment was to implement and run a web crawler on **.gov.si* sites. Crawler needs to follow certain requirements: it should use a predetermined database schema (with modifications allowed), respect crawling ethics specified in the *robots.txt* file, throttling the speed of crawling and running it in multiple threads. The crawler was implemented in the programming language *Python3*, without the usage of libraries that already implement the functionalities of crawling.

II. Implementation

Crawler was implemented in the programming language *Python3*. It is composed of several separate entities that work together. These include the scheduler, frontier, page handler, crawler as a thread manager and a database. The overview of the crawler is presented in the figure 1. All crawled data as well as the next pages to crawl are stored in a *Postgres* database, for which a *Docker* container was used.

A. Database extensions

We used the predetermined database schema given to us as a part of the assignment, which we then extended to satisfy our needs of the implementation. We added several attributes in certain tables and few new page type codes.

The database consists of seven tables, which includes *site*, *page*, *page_type*, *page_data*, *data_type*, *image* and *link*. We extended the table *site* with three additional attributes: *site_ip*, *timestamp* and *done*. Attribute *site_ip* is used for storing a site's IP address (in order to limit the number of requests per certain IP address in a time period), *timestamp* is used to schedule the number of requests on a certain site and the *done* attribute is used to mark a certain site as finished with crawling.

Table *page* was extended with the *content_hash* attribute, which is used for storing the hashed content of the crawled page and helps determine whether a given page is a duplicate of another page, already stored in the database.

Codes that determine the current status of a given page are stored in the table *page_type*. A site's code can be *FRONTIER*, *HTML*, *BINARY* or a *DUPLICATE*. We added an *ERROR* code, which marks a page that encountered an error whilst crawling. A *DISSALOWED* code was also added, which marks the pages that are not allowed to be crawled by the *robots.txt* specification.

To make our connection with database easier, we used object-relation mapping (ORM) with library *SQLAlchemy*.

B. Overview of the crawling process

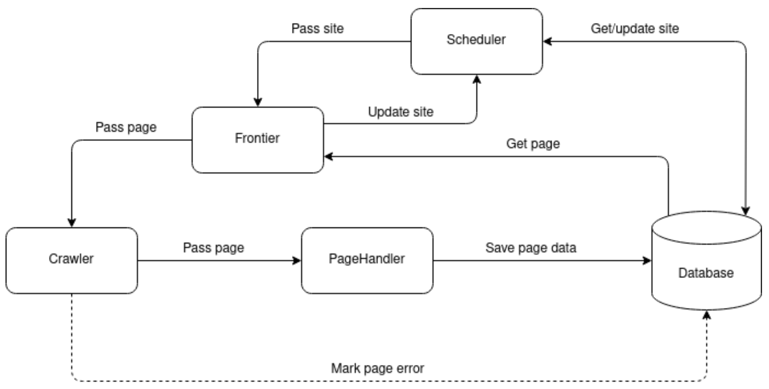

Fig. 1: Crawler schema

Conceptually, the crawling process goes as follows:
The configuration, specified in the *app/config.py* is loaded. This is where the list of seed sites, the database configuration, the number of threads to run on and other settings are set. After that, the **Frontier** module is initialized, which then initializes the **Scheduler** module. Next, the **Scheduler** prepares the seed sites. After, **Crawler** initializes the number of threads and starts querying the **Frontier** for the next available free page. The **Frontier** prompts **Scheduler** for the site, which hasn't been accessed in the last 5 seconds. At this time, the **Scheduler** updates all sites with their IP addresses (if not present) and returns the selected site. Next, the **Frontier** checks, whether the site has a *robots.txt* and the *sitemap* already specified and downloads them if necessary. Then it checks for the next page for the given site, marked as *FRONTIER*. In case there is no pages, marked as a *FRONTIER*, and the current site has not yet been visited, a new root page is created with an *url*, equal to the site's domain. In case of no pages marked as *FRONTIER* and the site has already been visited, it calls **Scheduler** to mark the site as *done* and again starts the process of selecting a new page. The **Frontier** checks, whether the selected page is allowed to be crawled, based on *robots.txt* rules. If not, the page is marked as *DISALLOWED*, and the page selection process starts again. In the other case, the page is returned back to the **Crawler**, where a new thread is initialized. The thread creates a new **PageHandler** object, which is responsible for retrieving, processing and storing the page content. In case of a **PageHander** failure, thread marks the page as *ERROR*.

The **PageHandler** module starts with removing page marking and a request is made to the page to check whether the page is accessible, and to determine the content type. If a page is non-responsive, it is marked as an *ERROR*. If the page contains a *non-HTML* content, it is marked as *BINARY* and its content type is stored in *PageData*. If the page is a valid *HTML* file, a *Selenium driver* is initialized, which retrieves the page's content. In order to check whether a page is a duplicate of an already visited page, the content is hashed and checked for a match in the database. In case of a duplicate, the page is marked as *DUPLICATE* respectively. If not, we collect all the *urls* on the page, including *HTML "a" elements* and *"onclick" javascript* links, check if they point to a **gov.si* site, canonicalize them using the *urlcancon* library and finally save them to the database, marked as *FRONTIER*. We also create new entries in the *link* table in the database. Next, we gather and save all the images on the page. The page is then marked as *HTML*. Finally the thread terminates.

III. Results

We present the results in the following two tables. Table I contains some statistics of the sites, provided in the seed list. Table II contains some statistics of the whole database. The row named avg. images represents the average number of images per page on a given site.

	gov.si	evem.gov.si	e-uprava.gov.si	e-prostor.gov.si
# web pages	652	5	4 202	1
# duplicates	145	2	0	0
# binary	121	0	12	0
# pdf	58	0	0	0
# doc(x)	6	0	0	0
# ppt(x)	0	0	0	0
# other	57	0	12	0
# images	1 267	42	4 376	2
# avg. images	1.94	8.4	1.04	2

TABLE I: Statistics for sites in seed

	All together
# sites	253
# web pages(HTML)	25 382
# duplicates	3 440
# binary	3 629
# pdf	4
# doc(x)	411
# ppt(x)	1 930
# other	1 284
# images	324 018

TABLE II: Statistics of crawled data

IV. Encountered problems

While implementing and solving the assignment, we encountered several problems. During the implementation, we stumbled upon the problem, where page *"gov.si"* could not be retrieved without the *"www"* prepend. We fixed it by manually adding the *"www"* string to the seed list.

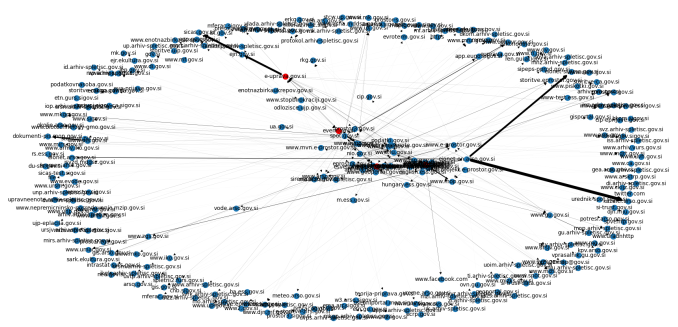

Fig. 2: Crawler schema

While implementing the **PageHandler** module, we discovered that the *Selenium driver* doesn't allow us to get the *http* headers of the request. In order to circumvent this, we made another *http head* request, using the *requests* library. We noticed that some pages do not allow a *http head* request, so we had to use a *http get* request instead. Here, we decided not to wait another 5 seconds, since we believe this doesn't represent a significant load increase on the server, and would only further slow down our crawling process.

During the initial testing, we had some problems with the *Selenium drivers*, where after encountering an error on the page, the driver didn't properly terminate, and the process would hang and clutter the CPU. We noticed that we were only closing the focused window on *Selenium driver*, not the whole driver process in the case of an error with the page.

We also noticed that our threads were not initializing properly and our crawler would actually run sequentially. After some debugging, we discovered that the server, where we were running the crawler, had some sort of limitation on *Python* threading. We then ran the crawler on a different machine, and the multi-threading problem was gone. Since we discovered this problem quite late, the number of the collected data was affected.

We also discovered a problem, where the *reppy* library, responsible for the *robots.txt* parsing, would hang the whole process, if it encountered a non-responsive site. We solved this with a timeout, where we terminate the connection after 3 seconds and allow crawling.

The biggest problem, which is the one we sadly discovered the last, was that our scheduler wasn't prioritizing sites with lower *id*s, which in turn resulted in poorly crawled seed sites.

V. Conclusion

Implementing an efficient and accurate web crawler proved to be a bigger challenge than expected. Many unexpected edge cases only showed themselves once we started the testing period, which meant we had to alter the implementations many times. Our crawler ran on a server for about a week before we noticed the problem with the crawler not running in multiple threads. After we moved it to another machine it ran for about two days. We know that the crawler isn’t bulletproof or super fast, but we are ok with the results we achieved.