Web Crawler

Alen Kurtagić (vpisna), Brin Colnar (63190077), Mitja Kocjančič (63200482)

I. Crawler implementation

The crawler was developed with Python and PostgreSQL, utilizing a Docker container for the database. Additionally, a Conda environment was utilized to streamline Python environment setup. We also set up a Linux server and used it to run our crawler.

A. Frontier

The implementation of the frontier within the database is a pivotal component, utilizing the structured framework of the crawldb.page table. Pages queued within the frontier are distinctly identified by the type code FRONTIER, denoting their pending status. Upon activation for processing, the column *in_progress* is activated, indicating ongoing retrieval and analysis. Notably, the frontier employs a timestamp-based sorting mechanism, strategically organizing pages to facilitate a systematic breadth-first crawling strategy.

An additional layer of complexity is introduced by restricting the inclusion of URLs into the frontier solely to those bearing the *.gov.si domain. This deliberate restriction ensures that only relevant governmental Slovenian web pages are eligible for integration, aligning with specific project requirements or objectives. This meticulous curation of URLs helps maintain the integrity and focus of the crawling operation, prioritizing authoritative sources within the target domain.

Overall, the integration of the frontier within the database architecture, coupled with stringent URL filtering criteria, underscores the meticulous planning and execution inherent in the crawling process. Our crawlers adhers to breadth-first strategy where order of the processing is based on descending access time of the pages.

B. Robots Exclusion Standard

In compliance with the robots exclusion standard our crawler ensures adherence to directives outlined in robots.txt files, including User-agent, Allow, Disallow, Crawl-delay, and Sitemap.

In instances where a specific crawl delay is unspecified, a default minimum delay of 5 seconds is enforced as a precautionary measure.

To effectively manage crawl delays, the crawler meticulously tracks the last visitation time for each domain. If a page is scheduled for processing before the crawl delay period elapses, it is judiciously deferred to the rear of the frontier queue. This approach ensures that crawl operations align with designated delay intervals, thereby maintaining compliance with established standards while optimizing resource utilization.

C. Adding links from sitemap files

When reading robot.txt files, we also use the sitemap XML file to add additional links to frontier.

D. Multiple workers

Through Python's multi-threading library we also support crawling using multiple workers. We ran our crawler with 4 workers, which significantly speeds it up. Each worker used its own Chrome driver instance. Number of workers is a settable parameter in *settings.py* file.

E. Getting pages

Initially, the crawler initiates a HEAD request to acquire the response code and ascertain the page's content type. Should the page not be identified as a binary file, Selenium is employed to fetch and visualize the HTML using Geckodriver.

F. Duplicate Detection

Duplicate detection is a crucial aspect of web crawling to prevent the crawler from revisiting pages that have already been processed. Here's how duplicate detection is implemented:

- **Hashing HTML Content**: When processing HTML pages, the crawler generates a hash value from the HTML content. This hash serves as a unique identifier for the page's content.
- **Checking for Duplicate Hashes**: Before updating the database with a newly fetched page, the crawler checks if the generated hash already exists in the database. If a duplicate hash is found, it indicates that the page content is identical to an existing page, and the crawler marks the page as a duplicate.
- **Updating Page Status**: Duplicate pages are marked as such in the database, and the crawler proceeds to the next frontier without further processing the duplicate page content.

G. Link Detection

Link detection involves identifying and extracting URLs embedded within HTML pages. Here's how link detection is implemented:

- **Parsing HTML Content**: The crawler parses the HTML content of fetched pages to extract hyperlinks (URLs) present within anchor tags (`<a>`).
- **Extracting URLs**: Using HTML parsing techniques or regular expressions, the crawler identifies and extracts URLs from the anchor tags.
- **Filtering URLs**: Extracted URLs may undergo filtering based on predefined rules such as domain restrictions, URL patterns, or directives from robots.txt files.
- **Queueing New URLs**: Valid URLs that pass the filtering criteria are queued for further exploration by adding them to the frontier, ready for subsequent crawling.

H. Image Detection

Image detection involves identifying and processing images embedded within HTML pages. Here's how image detection is implemented:

- **Fetching HTML Content**: When processing HTML pages, the crawler retrieves the HTML content, which may contain embedded image tags (`<img>`).
- **Extracting Image URLs**: Using HTML parsing techniques, the crawler identifies image tags and extracts the URLs of the referenced images.
- **Fetching Image Resources**: After extracting image URLs, the crawler fetches the corresponding image resources from the web server.

- **Storing Image Metadata**: Metadata about the images, such as filename, content type, and data, are stored in the database for future reference.
- **Updating Database**: The crawler updates the database with information about the processed images, including their associated page IDs, metadata, and timestamps.

II. Problems during development

During development we had a lot of problems with duplicate entries, mainly from Github. Also we had problems because multiple treads used the same db handle so the program crashed as soon multiple threads tried to use the same handle at the same time.

III. Results

A. Sites and pages

Our crawler found 506,854 pages across 2,148 sites. There were 5,232 failed, 508 disallowed by robots.txt, so the total number of processed pages is 23,815.

Table I
Number of pages by type.

Page type	Count
HTML	23,815
Binary	434
Duplicate	1,559
Unavailable	501
Disallowed	508
Frontier	479,994
Total	506,854

B. Documents and images

We found a total of 431 documents which is 0.01 documents on average per page.

Table II
Number of documents by type.

Document type	File extensions	Count
PDF	.pdf	318
Word	.doc, .docx	105
PowerPoint	.ppt, .pptx, .pptm	11
CSV	.csv	478
Total		**431**

There were 177,181 images, which adds up to 7.7 average images per page.

Table III
Number of images by type.

Image type	File extensions	Count
JPG	.jpg, .jpeg, .pjpeg	50,998
PNG	.png	69,994
GIF	.gif	5,638
SVG	.svg	1862
Others		
Total		**177,181**

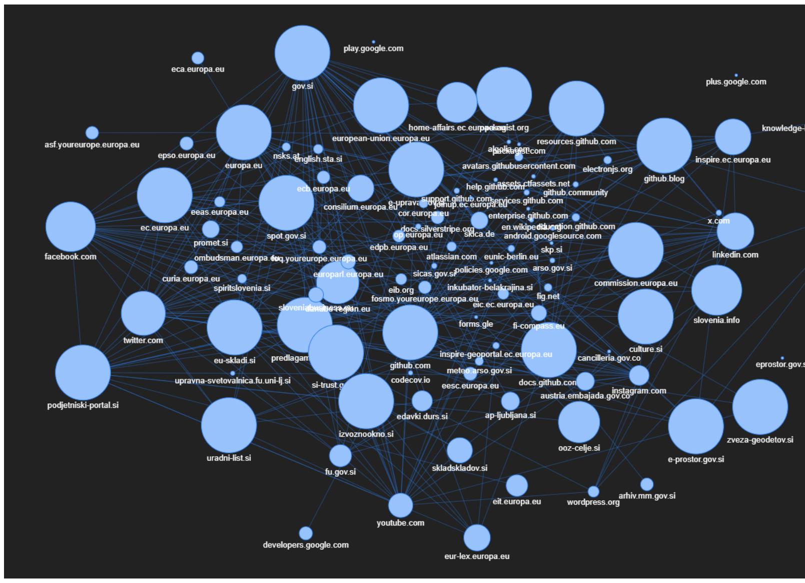

Figure 1. Interconnectedness of sites (100 most frequent)