Navigating the government website network using parallel breadth-first crawler

Web Information Extraction and Retrieval 2018/19, Faculty of Computer and Information Science, University of Ljubljana

Matej Klemen, Andraž Povše, Jaka Stavanja

***Abstract*—In this work we present our implementation of a crawler that navigates the government websites. It starts from a few given seed pages and crawls pages that are part of the “*.gov.si*” network, meanwhile storing binary data and the text on crawled websites. Using a starting seed of 2 URLs, our crawler visits 544 unique URLs, gathers 254 images and 265 files. Most frequent file is in PDF format and maximum number of images on a specific website is 15.**

I. Introduction

A web crawler is a program that visits web pages and downloads them for some purpose. It begins with some amount of seed URLs, which are put into a frontier. The crawler then selects a web page from frontier according to some strategy, visits the web page and extracts the web page’s content and outgoing links. The outgoing links are put into the frontier and the content is used for some purpose or stored. This process is then repeated until a certain stopping criteria is reached (e.g. the frontier being empty) [1]. The crawling can also be a continuous process (“without end”), but we do not study these in this project.

In this project we implement a web crawler that crawls Slovenian government websites. We are given a set of 4 government websites and choose additional 5 of them and proclaim them as seed websites, from which we start the crawling. While crawling these websites, our crawler only follows links to other Slovenian government websites. The crawler crawls these websites, downloads images and files from them and stores them into a database. The crawler visits new links in a breadth-first fashion. We also implement a method to detect near-duplicate websites based on their content. For this, we use a locality-sensitive hashing algorithm based on min-hashing [2].

The rest of the report is structured as follows.

- In chapter II we present general architecture of the built crawler.
- In chapter III we present how we got content from Robots and Sitemaps.
- In chapter IV we present how the database is built.
- In chapter V we present how we took care of link sanitizing and base urls.
- In chapter VI we present how images and files are stored.
- In chapter VII we present how Locality sensitive hashing deduplication works.
- In chapters VIII and IX we present the results of the crawler runs.
- In chapter X we conclude our work and present possible future improvements.

II. Crawler

In this section we present the architecture of the crawler. The full source code is available online [1].

The crawler consists of multiple components: link, sitemap and robots parser, database manager (and the database itself), content deduplicator and a core unit which uses the other components and adds on top of them to produce a parallel crawler that uses breadth first search to navigate the websites.

It takes 4 parameters in total. These are seed pages, maximum number of workers, maximum crawling depth and a flag, determining whether to get files. The first two parameters are self-explanatory. The maximum crawling depth is optional and specifies at what depth we want our crawler to stop. If this parameter is not provided, the crawler will stop when there are no more links being produced.

We used Selenium to make requests, because it also renders the Javascript code on the website. It does, however, not include HTTP status code, which is why we also had to make another request to get that information.

The crawler uses (parallel) breadth-first traversal to crawl the websites. The master (the initial process) keeps a structure with visited links and links that still need to be visited (frontier). On each level, the frontier is emptied and the links are divided among workers in the following way. First, the links are grouped by their base domain. The base domains are split among workers as evenly as possible and the workers then crawl links that were found in previous level for the particular base link. This means that a certain worker is responsible for crawling links of 1 or more domains. Each worker keeps a local copy of its visited websites and a local copy of its newly produced links. Each worker also checks if a newly visited website is present in the global visited structure, but it does not add new links into it as we want to keep the synchronization between workers to a minimum. We must wait for all the workers to finish so that breadth-first strategy is ensured. After they are all done, the master collects newly collected links from workers and checks for duplicate links. It also marks the links from current level as visited. This process is repeated with the newly collected links, which represent the next level of search process.

While our implementation does not use parallelism as effectively as possible, it does fully respect the crawled sites’ rules (for example, the specified crawl delay). Prior to the current implementation, our strategy was to distribute links as evenly as possible among workers, ignoring any information about the domains they belong to. Although this was a much faster strategy, it did not fully respect the sites’ rules. The information about the time of last request to a website was not synchronized fast enough in order for all the workers to respect it.

III. Robots and Sitemaps

Key element of the crawler is also reading the “robots.txt” file and sitemap. Based on the robots file, we know whether we can visit a certain link or whether it is not allowed for a crawler to visit that link. We make the simplification of only checking the robots file for the wildcard agent (“*”) as our crawler is not known to the public and therefore there are likely no special rules for our crawler. What the possibly existent robots file

[1] https://github.com/matejklemen/govrilovic

also tells us is the request delay which specifies how much time we have to wait between crawling a page from the same url. For that reason, we check if either the request delay specified in the “robots.txt” file or the default delay of 3 seconds has passed to not overload the website we are crawling. Another thing we can read from the same file is the location of the sitemap. What we do is we check if there is a sitemap property in the “robots.txt” file and if there is one, we save its contents to our sitemap’s dictionary and the database. Otherwise, we try and do the same by going to the site’s URL and checking the default sitemap location (“/sitemap.xml”). When we read a sitemap, we add all the URLs listed in it into the frontier.

IV. Database

The database is built inside a Docker container. We also mount the database on our local file system, so we have access to the saved files even when the container is shut down. We use the schema, provided in the instructions, with one minor change, which is adding a column ’lsh_hash’ into the page table. The column stores the value that our content deduplication method (locality sensitive hashing) produced for the website. Access to database is made by each worker when he visits a new page and saves its content. We figured this will not be an issue, since the bottleneck of the entire crawler is the network and time that requests take. The database can handle multiple workers making read and write operations by proper locking of the operations. For observing the results and current status of the database, we use Adminer, which is also running as a Docker container.

V. Link

To work with links correctly, we need to normalize them. The first thing we do is we contruct an absolute path for each link, so that we get rid of issues with relative paths. To do that correctly, we take into consideration the “<base href="somelink">” tags in the webpages, which set the base URL for all the relative paths used in the HTML on the currently crawled webpage. If there is no base URL on a website, we treat the URL we are currently crawling as the base one. What is left is to append the relative path to the base URL.

The next thing we have to take care of is to get rid of deeply nested URLs to avoid spider traps. We pass every link through a sanitizer which removes anything after the tenth slash in the URL to avoid looping of URLs. Then, we sanitize the links further to get rid of any positional arguments after the hash symbol in the url. We also URL encode the query parameters to prevent different issues which can arise by maybe injecting some malicious code or SQL queries in the URLs. This encoding also normalizes the capitalization for each URL (the part that is not case sensitive). It is also somewhat useful in optimization terms for our specific case to remove the ’www’ subdomain prefixes from links as they generally lead to the same page. Some servers do not redirect links correctly and there might be some inconsistencies, thus we optimize the crawling and reduce duplicates by removing that part from our URLs (in our case, that was useful to treat ’www.e-prostor.gov.si’ the same as ’e-prostor.gov.si’).

VI. Image and files extracting

We store images and files into the file system, since it gives a nicer overview of the gathered information. We have created a subfolder for each new site we visited, that had any images. Files were stored in a simmilar way, but we also had different folders based on file type (PDF, DOC, ...). We stored the location of the file/image in the database under the provided data field.

VII. Locality sensitive hashing deduplication

In order to not waste resources on websites we have already visited it is essential that we have some form of a deduplication mechanism. Its job is to find duplicates and near-duplicates of webpages based on their content. For this purpose, we implement the locality sensitive hashing algorithm [2] with min-hashing.

It takes a document D_{input} as input and returns its signature, which is a d-dimensional vector. First, the document is represented with a set, which contains the triples of a document which are present in our used vocabulary. This sparse representation is then converted into a dense representation by using a range of hash functions on the sparse representation. To increase the robustness of this method, the dense representation (vector) is divided into equally sized bands and the contents of each band are transformed using another hash function. The result is the document’s signature. To find *candidates* for duplicates of the current query document, the signature for the query document is obtained and compared against the signatures of other documents, stored in the database. This way, we obtain a small amount of candidates, which are then compared with the query document again, this time using Python SequenceMatcher similarity measure. The query document is marked as duplicate if the similarity is higher than 0.9. For setting the parameters of the LSH, we tried different vocabularies and hashing methods. In the end, we use a single hash method, because it performs just like tests with multiple ones. The one we used is the built in Python $hash()$ function. The vocabularies we tried were top 1000 Slovenian words, triplets from such words, triplets from HTML content of visited websites and finally what we are using in the final version, triplets made from most frequently used Slovenian words. Due to time constraints, we could not configure excellent LSH parameters, which would probably require an even more task-specific corpus, from which we would extract a vocabulary and tweak the possibly tweak the parameters on. Because of that, we are getting more equal hashes, than we would in a perfect scenario. Nonetheless, using the LSH deduplication method provides a major improvement compared to checking HTML content of every single website in the database and making similarity comparison.

VIII. Results - 2 seed URLs

For the first run of the crawler, we used 2 seed URLs, which were "http://evem.gov.si"and "http://e-prostor.gov.si". We set the maximum crawl depth to None, which means it will run untill there are no more links produced. The maximum number of workers/threads we used was 20. The crawling process took 30 minutes in total and retrieved results as seen in table III. Total number of links our crawler had visited in this run was about 1300. These include HTML websites and also other data types, such as PDF or DOC files.

Table I
Results of the crawling 2 initial seed urls.

Pages	544
Images	254
Duplicates (Links)	5
Files	265
Sites	3
Average image number (sites that contain at least 1)	2.91
Maximum images on a website	15
Frequency of file type: PDF	86 %
Frequency of file type: DOC	12 %

IX. Results - 9 seed URLs

The second run consisted of 9 seed urls. For the extra 5 urls of our choice, we have choosen "http://www.mz.gov.si/", "http://www.mnz.gov.si/","http://www.up.gov.si/", "http://www.ti.gov.si/" and "http://www.mf.gov.si/".

We have firstly done a run up to 3000 crawled links and enabled file and image downloading, which we have then terminated due to issues with space on the local machine and slower performance. We received around $1GB$ of binary files, of which their specific sizes are shown in Table II.

Table II
File sizes retrieved from the 3000 links run.

Images	34.3 MB
PDF documents	870.9 MB
Word documents	84.1 MB
PowerPoint presentations	57.7 MB

Most of the images were actually thumbnails of certain posts, found on the site, which had pictures that resembled things that should be on a certain page (judging by the url). For example, images from 'e-prostor.gov.si' were mostly related to maps of Slovenia and its nature and landscape while 'mnz.gov.si' contained more images from ministers' meetings and police related issues.

The PDF and Microsoft Word documents were mostly application forms for different purposes (e.g. applications for registering companies from 'evem.gov.si') and periodical reports from different governmental institutions (e.g. reports for years 2007-2016 on Slovenia's national real estate market from 'e-prostor.gov.si').

In the next test run, we used same 9 seed URLs as above, and disabled image and file downloading. The data we have gathered from it is displayed in table III.

Table III
Results of the crawling 9 initial seed urls. Second run.

Pages	2966
Duplicates (Links)	7
Sites	22

We can use graphs or networks to visualize crawled websites and extract some information from them as well. We can first retrieve some general graph statistics such as the number of links in the graph, which is 2962 while the number of nodes 2982 which clearly shows that some URLs didn't produce any out links. The average degree for nodes in the graph is 1.99. To visualize the graphs, we can represent crawled sites as nodes and link one to the other, if the crawler took that path between them as well. We can then plot the network using different layouts. In our case we will use force-directed algorithms for layouts, taken from the OpenOrd software [3] and an algorithm from Yifan Hu [4]. As this was a simple crawling procedure, we should expect get many hubs (meaning nodes with high degree) and a number of pages attached to them. From that we should see which site produced the most links in our frontier and which site had 'Disallow: /' in its 'robots.txt' file. To prove that, we use Gephi to plot the graphs using seen on Figures 1 and 2 and come to the expected results.

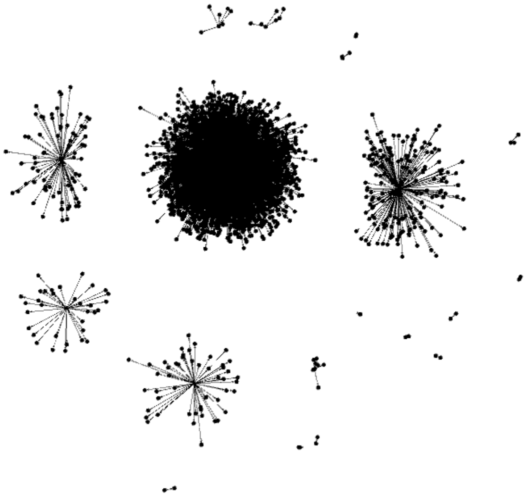

Figure 1. Page network plotted using OpenOrd's force-directed algorithm.

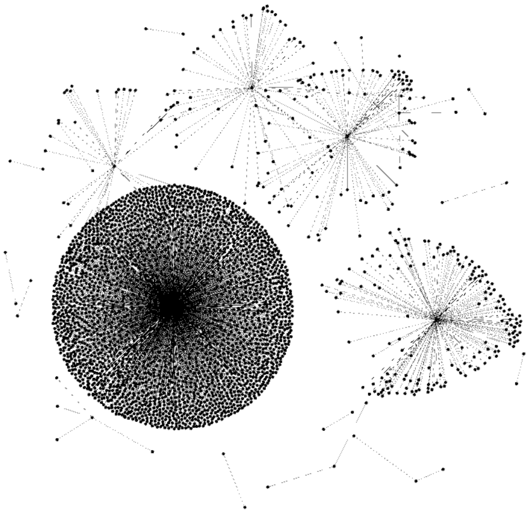

Figure 2. Page network plotted using Hu's force-directed algorithm.

We can also retrieve the URL of the biggest hub node and see that it is 'mz.gov.si', the page that we have chosen as an additional seed for our crawling task. It produced around 88 percent of all links that we have crawled.

X. Conclusion

We presented the overall architecture of our crawler that navigates the government websites and provided some implementation challenges that we have had to deal with. Although our crawler works relatively well, there are many thing that could be improved. First thing that could be improved is better use of concurrency to speed up the crawler. A possible way could be to use our previous strategy of implementing parallel breadth-first traversal, but making it more sophisticated by taking care of communication between workers. This way, better efficiency could be achieved while also following the crawling rules.

Another thing that could be tweaked further is the ability of the crawler to handle large amounts of data. As an example, in our implementation, we keep the frontier in memory at all times, which could become problematic when encountering a large amount of different domains. The obvious solution would be to connect the frontier with the database as well, while also thinking about what proportion of the frontier one could keep in memory in order to gain a performance boost.

The last improvement we mention is the possibility of task-specific parameter tuning of locality sensitive hashing. As already mentioned, the performance of our deduplication method could be improved by using a part of the crawled data for parameter tuning. More specifically, one could combine the data of known duplicate pages with known non-duplicate pages into a dataset, on which parameter tuning could be performed with the goal of maximizing classification accuracy (documents are classified as duplicates or non-duplicates with the help of locality-sensitive hashing).

References

[1] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[2] A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimensions via hashing,” in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB ’99. Morgan Kaufmann Publishers Inc., 1999, pp. 518–529.

[3] S. Martin, W. M. Brown, R. Klavans, and K. W. Boyack, “Openord: an open-source toolbox for large graph layout,” in *Visualization and Data Analysis 2011*, vol. 7868. International Society for Optics and Photonics, 2011, p. 786806.

[4] Y. Hu, “Efficient, high-quality force-directed graph drawing,” *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.