Univerza *v Ljubljani*
Fakulteta *za računalništvo*
in informatiko

Web information extraction and retrieval

Project 1
Webcrawling

Author:
Erik Kristian Janežič

Ljubljana, 6.4.2020

Contents

1 Introduction

In this project we implement a breadth first search web crawler that retrieves data from .gov.si domains.

2 Implementation details

Web crawler was implemented in Python language in a multi threaded fashion. For storage of crawling results we use a Postgresql database. For html rendering we used Selenium in headless mode of operation. This enabled us to correctly render JavaScript based websites.

Two core classes were implemented *CravlManager* and *Cravler*. *CravlManager* is responsible for holding all run time in memory caches regarding visited sites and current ques, starting crawling threads and delegating new pages from top of the ques. *Cravler* holds all necessary logic to create *http* requests with Selenium and parse recieved *html*.

Two support classes were defined for holding domain and page related data, *DomainDeque* and *Page* respectively. For database interactions we create the *PG_Database* class, which also makes sure that potential errors in database insertions are excepted in such way that threads don't get interrupted

2.1 Breadth first crawling

Breadth first crawling was implemented in such way that each domain holds a separate FIFO queue. This decision was made because each domain has a 5 second timeout between individual requests, and with per domain queues we can make sure that individual crawling threads can access top elements fro domains that are of timeout. When one crawler thread completes all processing on a given page, it loop over the per domain queues held by *CravlManager* until it finds one domain that is of timeout and has at least one item in the queue.

2.2 Robots.txt compliance

Whenever *Cravler* encounters a new domain when processing some page, it tries to make a head request to "new_domain/robots.txt". If the response code is 200 we use the library *urllib.robotparser* to parse the robots.txt file. We assign the initialized *RobotFileParser()* to the corresponding domain in the *CravlManager* and from there on we use it to chcek all URL's, that belong to the given domain, if we can visit them.

2.3 Sitemap

To check if sitemap.xml exists for a given domain we first check if head request to "new_domain/sitemap.x has 200 response code. If so we use that for as a sitemap for a given domain. In oposite case, we check if the domain had robots.txt file defined. If so we parse the file and check if it has "sitemap: " entry and extract the sitemap link if so.

2.4 URL canonicalization

For URL canonicalization we use *urllib.parse* library. We reduce all encountered URL's to only its coresponding domain and path sections.

2.5 HTML parsing

For all html and xml parsing task we use BeautifulSoup library. For link extraction we use find all "<a>" html elements that have the "href" attribute. Values in "href" are cleaned and corresponding domains for extracted links are determined. Links are then appeneded to the ends of corresponding domain queues.

For image extraction we search for all "<img>" elements that have "src" attribute. Since pages can have multiple instances of the same image in them we hold evidence of page - "src" combinations that were already added to the database so that we don't duplicate image entries.

3 Results

We managed to extract 7601 pages in 2 hours. Below we present couple of images that represent the statistics of the portion of the network we managed to crawl. To visualize the network graph on 1 we used PyVis library which is a wrapper around Vis.js library.

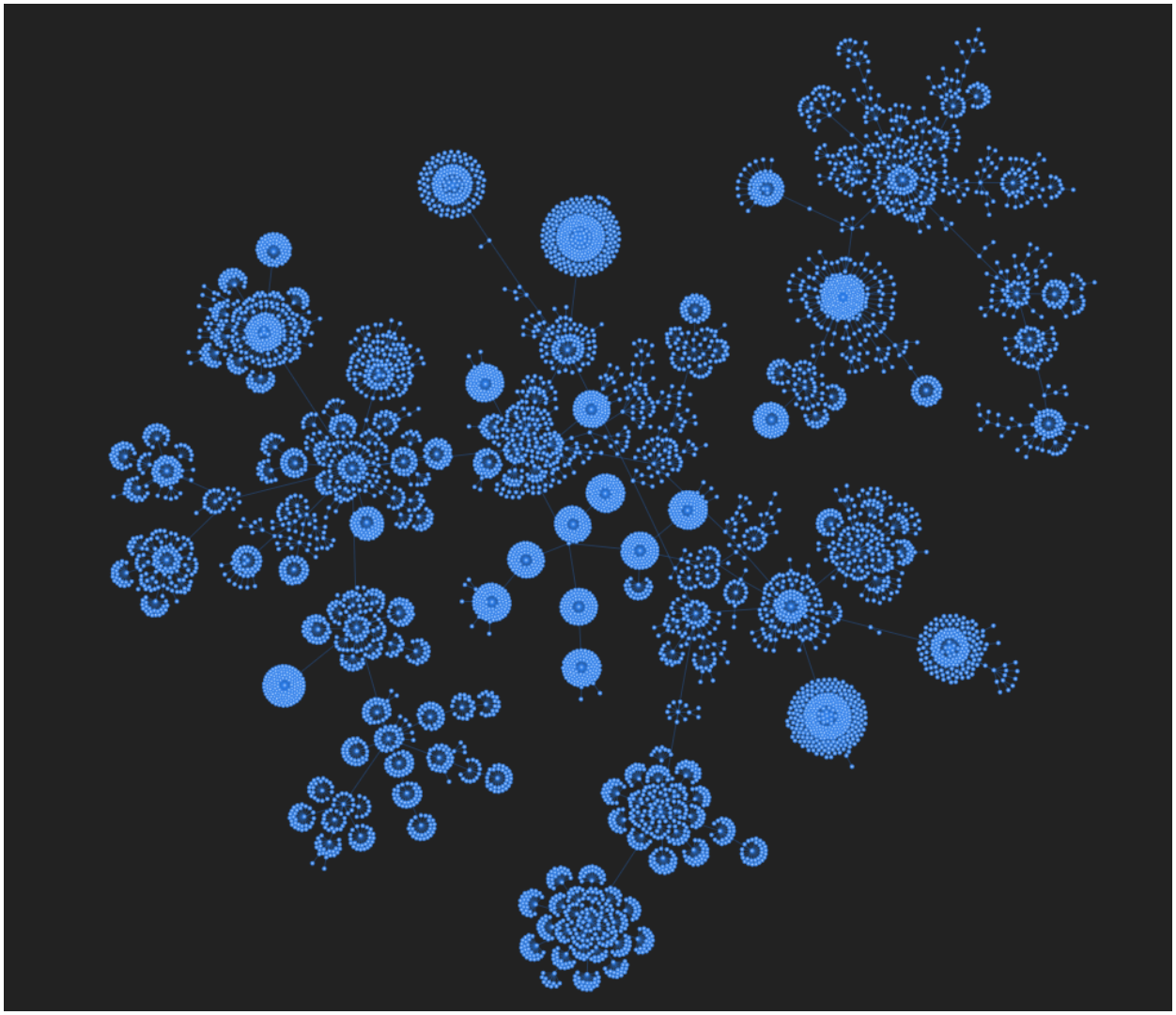

Figure 1: Visualization of the network based on links between sites

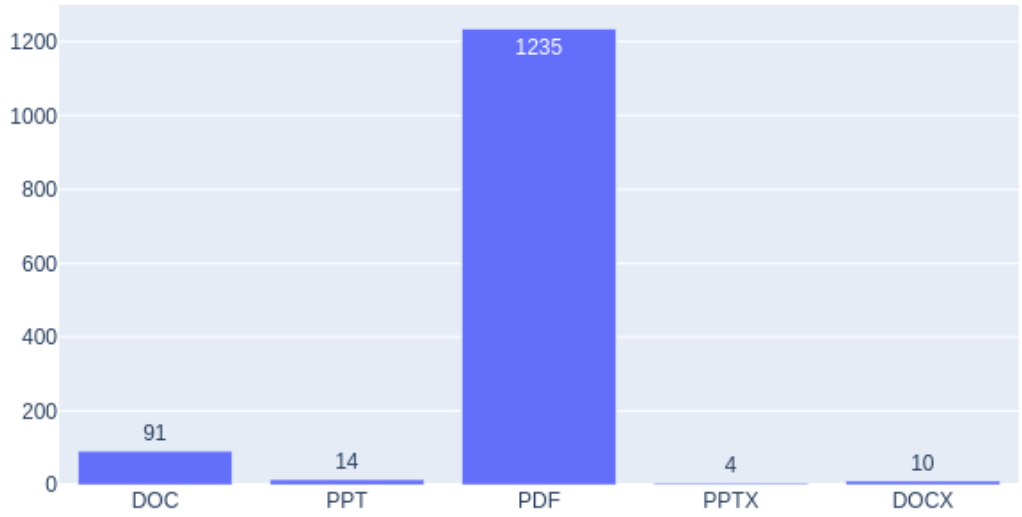

Figure 2: Numbers of encountered binary files

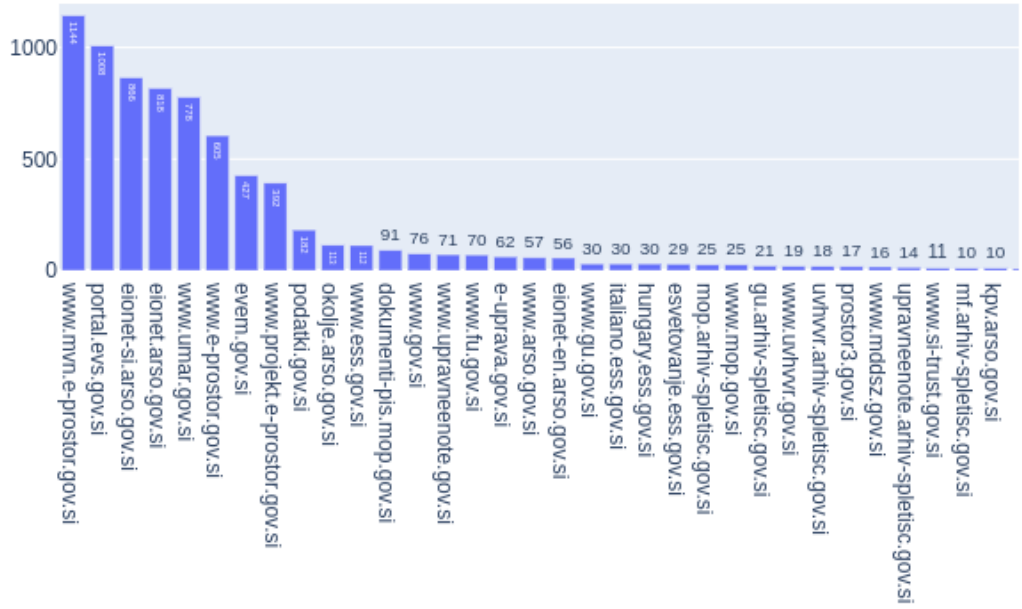

Figure 3: Visualization of number of pages found for specific encountered domains

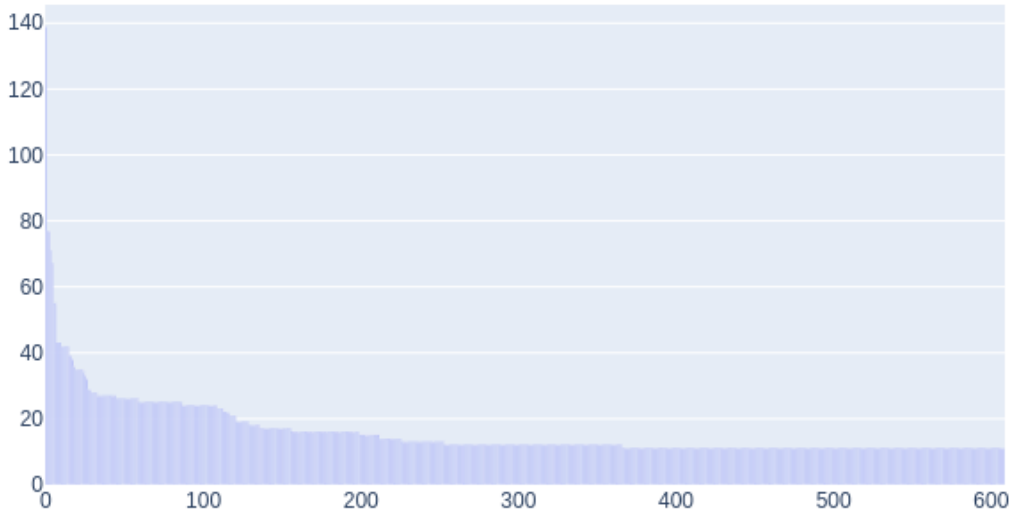

Figure 4: Distribution of number of images among pages. The maximum number of images was 139 and on average pages had 7.46 images

4 Discussion

We implemented a basic version of a web crawler that is capable of parsing majority of websites. Important additions that would improve the capabilities of the crawler would include:

- Better duplicate detection than just extreme url canonicalization
- Implement functionality that makes use of Selenium click and scroll actions to dynamically load new data on web applications. This could be done in such a way that we click all elements that have java script in href attributes and then check if we are still on the same url and the number of <a> html elements changed on the site. This way we can keep clicking the "load more" button and extract additional URL's.

5 Insttructions for running the crawler

Below are listed all necessary steps to run the crawler:

1. Define the "CHROME_DRIVER" parameter in "configs.py" to point to the path of the chrome driver on our computer
2. Define "USERNAME", "PASSWORD", "HOST" parameters in "configs.py" according to your local postgresql server
3. execute "python cravler.py" in the root directory