University *of Ljubljana*
Faculty *of Computer and Information Science*

Programming Assignment 1: Web Crawler

Tim Vučina

Abstract

We present our implementation for the course project and roughly outline the problems and solutions during the development. We analyze and show general statistics of the created *crawldb* database and visualize the generated network of crawled sites and pages.

Keywords
Web Crawler, Multi-threaded, Frontier

Advisors: Slavko Žitnik

Introduction

A Web crawler, sometimes called a spider, is an Internet bot that systematically browses the World Wide Web and that is typically operated by search engines for the purpose of Web indexing. Web search engines and some other websites use Web crawling to update their web content or indices of other sites' web content. Web crawlers copy pages for processing by a search engine, which indexes the downloaded pages so that users can search more efficiently.

A Web crawler starts with a list of URLs to visit. Those first URLs are called the seeds. As the crawler visits these URLs, by communicating with web servers that respond to those URLs, it identifies all the hyperlinks in the retrieved web pages and adds them to the list of URLs to visit, called the crawl frontier. URLs from the frontier are recursively visited according to a set of policies which in our case was just a breath-first search strategy. Our crawler is also performing archiving of websites, which means that it copies and saves the information as it goes.

Implementation

As part of this project assignment we implemented a multi-threaded crawler (python) that collects and stores information about the sites, pages, files and network topology. The general architecture of the implemented crawler followed a common seperation of logic into a **repository.py** file which was responsible only for communication with the database, **entities.py** file which contains classes for each of the tables in the database and a main logic component in **crawler.ipynb** with some helper functions separated into another file for readability. In order to successfully crawl pages that require execution of JavaScript code to generate content, our crawler uses the Selenium library and headless browsers that run on multiple threads. For downloading binary content such as images and different filet types it uses the requests library. To store the collected data it uses the PostgreSQL database with the schema shown in Figure 1.

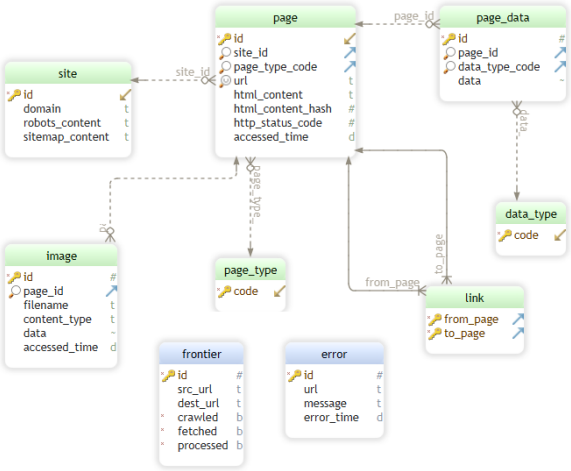

Figure 1. Model of our crawldb database.

The crawler can be limited to a certain domain such as **.gov.si*, as requested in the assignment instructions, but can also be set to crawl all types of URLs. All crawler parameters are described in Table 1. In order to respect the instructions in *robots.txt* files, we used the *robotparser* class from the urllib library.

Name	Type	Description
USER_AGENT	*string*	The name of the User-agent for robots.txt
LIMIT_DOMAIN	*string*	Domain part of URL for which crawling is allowed
SEED_URLS	*list[string]*	Starting URLs from which the crawler starts crawling
BINARY_CONTENT	*list[string]*	Names of file types that are considered binary content
ALLOWED_LINK_TYPES	*list[string]*	Page content types that are allowed (from URLs)
NUMBER_OF_WORKERS	*int*	The number of workers/threads
TIMEOUT	*float*	Default wait time between accessing pages (seconds)
START_CLEAN	*bool*	If TRUE the database is wiped and the crawler starts from scratch
STORE_BINARY	*bool*	If TRUE the binary data will be stored in the database
RESPECT_CRAWL_DELAY	*bool*	IF TRUE the crawler will respect the crawl_delay from robots.txt

Table 1. List of crawler parameters

Development

During the development of the crawler we encountered a number of problems. We started with developing the basic crawler logic with selenium. In the beginning we stored all the gathered data as well as the frontier in memory. This is obviously problematic as we could run out of memory and in the case of an error we would loose our progress. To combat this we created classes for each of the entities and wrote some code to act as a repository for the database which we ran in a docker container. We also added the **Frontier** and **Error** tables to our database with the help of *crawldb.sql* script. This allowed us to store gathered URLs into frontier in the database which prevented the loss of progress in the case of an error.

We also encountered a number of problems during the parsing of some robots.txt files as they were often poorly written or the sitemaps had a non-standard xml structure and the user agent rules sometimes contained unexpected characters. In one instance during the crawling this led to the crawler getting caught in a loop as it could not parse the robots.txt file correctly and was therefore crawling on disallowed pages. We partially solved the problem with the use of a parser library which we had to modify in order for it to correctly parse robot files that had these problems.

After implementing the crawler to work as a single process or rather on a single thread, we added the multi-threading functionality with concurrent.futures library. This presented a challenge because some processes were checking if an entry already exists in the database before another process completed adding that exact entry to the database.

Results

The general statistics of the filled *crawldb* database are presented in Table 2. Even though it is expected, it is still interesting how there are many more links than there are pages. This means that the network contains hubs or very high degree nodes which makes sense as websites are structured to have a main page that leads to most of the other pages on that site.

From the gathered links we generated a visual representation of the network for the first 10.000 links shown in Figure 2.

#	**Seed Sites**	**Whole Database**
Sites	4	227
Links	71489	119910
Pages	14616	44931
Duplicates	3053	12148
Images	14288	156946
Binary files	796	2373
pdf	566	1550
doc	76	383
docx	154	460
ppt	0	0
pptx	0	0

Table 2. General database statistics

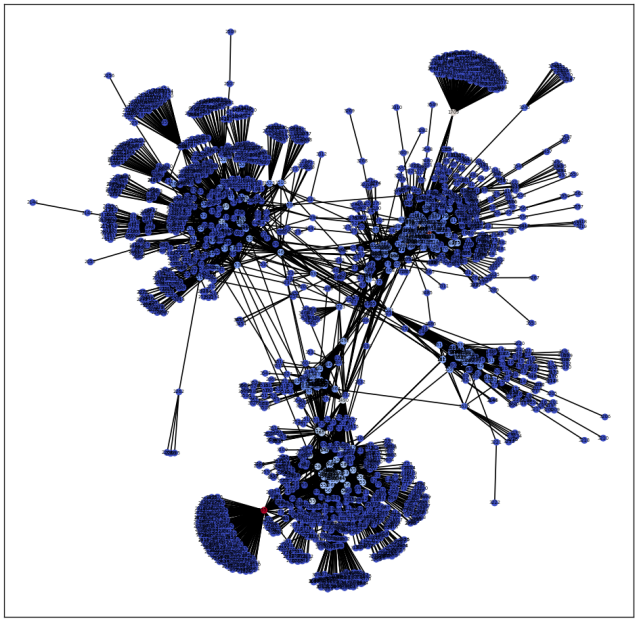

Figure 2. Network visualization for first 10K links.