WEIR - Report on the first assigment

Matej Bevec
Nejc Hirci
Rok Nikolič

I. Introduction

A web crawler is a program that automatically and systematically browses and indexes the World Wide Web. In this assignment, we implement a simple web crawler, which, starting at a given set of seed URLs, continuously visits outgoing links in breadth-first fashion and stores the visited pages' metadata in a relational database. In the following sections we first outline the key aspects of our implementation, before analyzing and visualizing the collected dataset.

II. Implementation

A. Database and back-end

Our back-end consists of a *PostgreSQL* database and a python interface. The database is based on the supplied schema and runs in a Docker container. The interface, implemented in *backendsql_commands.py* establishes a threaded connection with the database using the *psycopg2* python library. It is runs in the same process as crawler component and interfaces it by exposing simple functions, such as insert_site.

B. Threading architecture

The main web *Crawler* component is implemented as a class, which subclasses python's *threading.Thread*. At initialization, a number of Crawler components are instantiated, each continiously visiting pages from the frontier until termination. Visiting a single page consists of four steps: dequeuing a url from the frontier, requesting the page, parsing the content, and saving metadata to the db via a threaded connection.

C. Frontier

To simplify implementation, we don't store the frontier in the database. Instead, we use python's thread-safe Queue. All running threads globally take or add URLs to this object. Similarly, we use a "write-only" set object to keep track of URLs which had already been visited.

D. Robots file and rate limiting

When crawling, we take steps to ensure to respect the given domain's rules and limitations. In particular, this means following guidelines in robots.txt files and limiting the request rate o a maximum of one per 5s.

We use python's *urllib.robotparser* to handle robots.txt files. Whenever a new domain is fetched, we also fetch it's robots.txt and save its rules in a dictionary. When visiting another URL in the same domain, the rule object is then queried to check privileges, such as whether the URL at hand can be crawled or not.

Rate limiting is handled in a similar manner. After a URL is fetched, its IP address is identified from the response. The timestamp is stored in a dictionary as the time of last call to the IP at hand. We also store a mapping from domains to IP addresses. Before a new request is made, it is checked whether its domain's IP has been accessed within the last 5 seconds. If this is the case, the URL is returned to the frontier and the thread continues. If the rate limit induced by robots.txt is stricter, it overrides the 5s default.

E. Page parsing and dynamic content

We use a combination of static requests and Selenium's rendering to handle both static and (partially) dynamically loaded pages. For every visited URL, we first fetch its static HTML content with the requests library. We then use a simple high-sensitivity, low-specificity heuristic to determine if dynamic content can be expected. If the HTML body is shorter than 25000 characters, we assume dynamic content and run Selenium to render the page.

After rendering we perform several parsing steps to extract necessary metadata. One such example is finding "onclick" tags to extract aditional outbound links.

F. Duplicate detection

To detect duplicates, i.e. pages with identical HTML content, but fetched from different URLs, we rely on SHA256 hashes. A hash of the received page HTML content is computed on the front-end and stored to the DB. When fetching a new page, a database call compares the new hash with existing hashes to identify any duplicates. In such case, the new page is saved but marked as duplicate.

Additionally, we track response history and automatically mark a duplicate if a fetched URL redirected to an existing (stored) URL. The combination of both approaches ensures fast but reliable duplicate detection.

III. Results

Tables I and II depict the statistics of the collected dataset. We track the number of visited domains, pages (urls), duplicate pages (see above), as well as certain categories of data — PDF, DOC, PPT files and images. We also depict the average number of said items per one visited domain and per one visited page.

The program was stopped after it had visited 76151 web pages in total. 2507 (3.3%) of these urls, where part of the initial seed domains. In other cases, the crawler "explored" external sites. About 4.6% of fetched urls in total returned

duplicate pages. In terms of obtained data (files), images were the most common, with a surprising discrepancy between page within and outside seed domains. Seed domain pages had only 0.026 images per page on average, while the global average was over 8.

In terms of performance, the total running time was 8 hours and 6 minutes, which averages at approximately 157 pages per minute.

Table I: Statistics for all visited pages.

	# items	# per domain	# per page
domains	360	x	x
pages	76151	211.5	x
duplicates	3504	9.733	0.046
PDF	8743	24.29	0.115
DOC/DOCX	2403	6.675	0.032
PPT/PPTX	0	0	0
images	637382	1771	8.370

Table II: Statistics for visited pages within the seed domains.

	# items	# per domain	# per page
domains	4	x	x
pages	2507	626.8	x
duplicates	487	121.7	0.194
PDF	8	2	0.003
DOC/DOCX	5	1.25	0.002
PPT/PPTX	0	0	0
images	69	17.25	0.028

IV. Conclusion

In the preceeding assignment, we implemented a python web crawler and deployed it on a seed set of URLs to collect a large network of web pages. Although the program is rather simple in terms of behavior, the task proved to be more challenging than we expected. We repeatedly found ourselves surprised by real-world variations which we hadn't anticipated (e.g different handling of robots.txt by different domains). This often caused unexpected crashes after hours of operation. Still in the final iteration, we have completed our desired objectives.

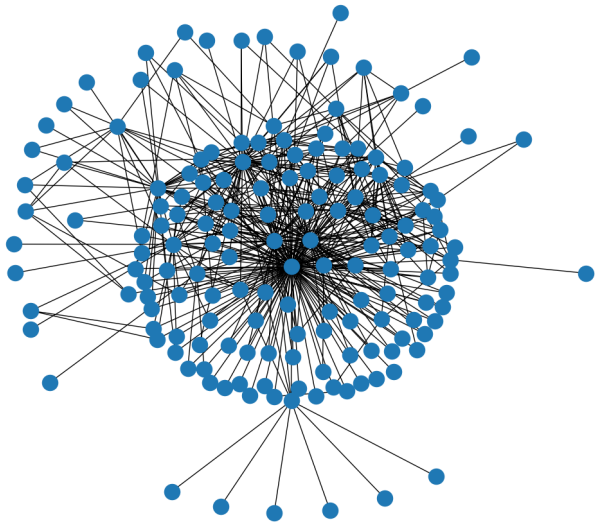

Figure 1: Domain connectivity network. Nodes represent visited domains and edges denote whether two domains are connected (within our dataset) by at least one link. We can observe a more or less tree-like structure, with most links outbound from the central, seed domains.