Programming Assignment 1

Guilherme Soeiro de Carvalho Caporali (2021031955)

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

guilhermesoeiro@dcc.ufmg.br

1 Introduction

This project aims to implement a web crawler capable of efficiently collecting a mid-sized corpus of webpages while strictly adhering to established crawling policies. The main objectives include fetching 100,000 unique HTML pages, respecting site-specific politeness rules (such as robots.txt and request delays), and storing the retrieved data in compressed WARC files.

The crawler is developed in Python 3, running in a controlled virtual environment to ensure compatibility and reproducibility. It supports parallel crawling using multiple threads to maximize efficiency and includes a debugging mode for detailed tracking of the crawling process. This document provides an overview of the crawler's design, key data structures, implemented algorithms, and an analysis of the collected corpus.

2 Implementation Details

The crawler is written in Python 3 and organized under `crawler/` in the repository[1]. Key components:

2.1 Frontier and URL Management

A thread-safe min-heap stores $\langle \text{priority}, \text{URL} \rangle$. Priorities are set using a uniform random function to scatter requests across domains, smoothing politeness enforcement. Operations run in $O(\log N)$ time and the heap is capped at 1 000 entries to bound memory.

2.2 Visited Set

Instead of full URLs, we store 64-bit Python hashes in the `visited` set, reducing per-URL storage from $O(L)$ to 8 bytes. Collision risk at 100 K entries is negligible ($< 10^{-11}$).

2.3 Domain Controller Cache

A dict maps each domain to a `DomainController` that tracks robots.txt rules and last-fetch times. To conserve memory, the cache is flushed every 1 000 pages, trading a minor re-fetch delay for lower footprint.

[1] `https://github.com/guilherme13c/simple-search-engine/tree/main/crawler`

2.4 Parallel Crawling

Using T `threading.Thread` workers, each thread repeatedly:

1. Dequeues a URL (with lock).
2. Enforces delay via its `DomainController`.
3. Downloads and parses HTML (BeautifulSoup).
4. Appends raw content to the current WARC (1 000 pages per file, gzip).
5. Extracts and enqueues new HTML links.

Ideal speedup is $O(T)$ until network or shared memory structures (e.g. Frontier) dominate.

2.5 WARC Storage and Debugging

Pages are streamed into gzip-compressed WARC files (1 000 records each). In debug mode (`-d`), each fetch logs a JSON record (URL, title, first 20 words, timestamp), adding $O(P)$ output cost per page without altering core flow.

These choices ensure bounded memory, politeness, and scalable parallelism for crawling 100 000 pages.

3 How to use & Command-Line Arguments

The crawler's execution is configured via the following command-line options:

`-s, --seeds` ***(required)*** Path to the file containing one seed URL per line. Stored in `seed_file`.

`-n, --number` Maximum number of unique pages to crawl. Type: `int`, default: `100 000`. Stored in `max_page_count`.

`-d, --debug` Enable debug mode. When set, each fetched page emits a JSON record to stdout. Stored in `debug` (`bool`).

`-p, --show-progress` Display a progress and average speed so far during crawling (printed afeter every 50 pages). Stored in `show_progress` (`bool`).

`-c, --max-concurrency` Maximum number of worker threads.
Type: `int`, default: `16`.
Stored in `max_concurrency`.

`--domain-concurrency` Default maximum simultaneous requests per domain.
Type: `int`, default: `5`.
Stored in `default_max_concurrent_requests_per_domain`.

`--craw-delay` Default delay (in seconds) between requests to the same domain.
Type: `float`, default: `100ms`.
Stored in `default_crawl_delay`.

`--save-interval` Number of pages per WARC file before rotating to a new one.
Type: `int`, default: `1 000`.
Stored in `save_interval`.