University of Leeds **School of Computing**

COMP3011, 2024-2025

Web Services and Web Data

A Search Tool for a Simple Website

By

Toby Hutchinson

201530569

sc21t2hh@leeds.ac.uk

Date: 02/05/2025

1. Introduction

I have implemented the *build*, *load*, *print* and *find* features, as well as additional *exit* and *help* functions for exiting and displaying the features available respectively. All features are implemented in the *Crawler* class which encapsulates all important data and logic. In the main function a simple terminal client takes in commands from the user and invokes the appropriate functions. The Python framework, *Rich*, is used to handle terminal output (styling and tables).

2. The crawler

The crawler was implemented using the *Requests* library to fetch web content, and *BeautifulSoup* to parse the fetched pages into Python objects. The Crawler class performs a breadth-first search of the website, starting from the base URL (https://quotes.toscrape.com/). The crawler holds a Python list of URLs that are waiting to be crawled which acts as the frontier. It also holds a list of URLs that have already been visited.

To traverse the website, the crawler repeats the following steps:

1. Dequeue the first URL of the frontier. This will be the next URL the crawler scrapes.
2. If the elapsed time since the last request is less than the politeness window, wait for the remainder of the politeness window. Otherwise, fetch the content using the *get()* function from the *Requests* library. If this returns with no errors, add the fetched URL to the set of visited URLs.
3. The content of the URL page is then parsed using *BeautifulSoup*'s HTML parser.
4. Anchor elements (link elements) are looped through using the parsed content, if the URL that the link points to is not an external website, hasn't been visited already, and isn't already in the frontier, it gets enqueued in the frontier.

Using this method, all links contained in each web page that gets visited is added to the *end* of the frontier queue, thus creating a breadth-first traversal of the website.

3. The inverted index

The *Crawler* instance keeps two indexes: the main inverted index and the URL index, which maps URLs to their own respective indices. Both the inverted index and the URL index are stored in memory as Python dictionaries. In order to create the URL index, the crawler keeps track of the number of the number of pages it has fetched. Once a URL's content is successfully fetched, this URL count is used as the URL's index.

To create the inverted index, all text is extracted from the parsed HTML content and turned into lowercase. Next, a regular expression matches all words in the lowercase text. A word here can consist of letters, digits and underscore characters. Once all words have been collected from the text, the program iterates through all words. If the word is less than 3 letters long, ignore it. Otherwise, add the word to the inverted index as a key which points to a dictionary of URL index - word position pairs, where the word positions are stored as a list of positions where the word appears in the text. Essentially, every word stores a dictionary of

URLs in which the word occurs, and every URL in that dictionary stores a list of all the positions of the word within that document.

The inverted index is stored to the filesystem as JSON. Python's JSON module is used to save and load the inverted index and URL index to the filesystem. Since Python dictionaries and the JSON format are similar, this method allows for simple and intuitive saving and loading, using the *json.dump()* and *json.load()* functions.

4. The ranking method

Results are ranked in the following order:

1. Exact matches of the query.
2. All words in the query exist in the document in any order.
3. Some words exist in the document.

Within these three categories of results, all matches are ordered by occurrence. For example, if URL1 has 2 exact matches, and URL2 has 3 exact matches, URL2 will be higher in the results than URL1. For pages that contain all words in any order, the "score" of the document is the sum of the occurrences of each word in the document. Similarly, for documents that contain some words, documents that contain words from the query 5 times will score higher than documents that contain 4 or less.

As the position of each word is saved in the inverted index, finding exact matches is straightforward:

1. Find the subset of URLs that contain all words (intersection).
2. For each URL in this subset, loop through the positions of the first word of the query.
3. If all subsequent words in the query appear in their relative positions after the first word within the current URL, it is an exact match, otherwise it's not.

Using this method, exact matches are counted per-document, and then added to the list of results, sorted by the number of matches.

Since we have already calculated the subset of URLs that contain all words, we can remove all URLs that have been found to contain exact matches. This gives us the set of URLs that contain all words in any order. Ranking them is simple. For each URL in the new subset, find the number of times each query word appears in the URL and add them all together. This number is the URLs score. The URLs are then added to the results - ordered by score.

Finally, all URLs that are already in the results are removed from the query words' inverted index entries. The number of occurrences of each word in each remaining URL is the URL's score. The URLs are then added to the list of results - ordered by score.

It's important to note that 2 or 1 letter words are treated like stopwords. Therefore, any query that contains any 2 letter words will never be an exact match.

5. Using the tool

Usage

1. Make a virtual environment and activate it (Mac or Linux):
   ```
   $    python -m venv venv
   $    source venv/bin/activate
   ```
2. Install dependencies:
   ```
   $    python -m pip install requirements.txt
   ```
3. Run the command line app:
   ```
   $    python scraper.py
   ```

Features

Once the command line app is running, type:

build - Build the inverted index and URL index

load - Load the inverted index JSON file to memory (inverted index must be built)

print <word> - Print the inverted index for `<word>` (inverted index must be loaded to memory)

find <query> - Return all pages containing the word(s) in the query, ranked by relevance (inverted index must be loaded to memory)

exit - Quit the app

help - Print the available commands.