**University of Leeds School of Computing**

**COMP3011, 2023-2024**

**Web Services and Web Data**

A Search Tool for a Simple Website

By

Morgan Craig Lewis

201426038 sc20mcl@leeds.ac.uk

**Date:** 06/05/2024

# Introduction

# I have incorporated all the specified features outlined in the brief. Furthermore, I have invested additional effort to enhance the visual appeal and readability of the search tool's command-line interface (CLI). The page ranking mechanism extends beyond mere word frequency, adding complexity for more accurate results. To enhance user experience, the CLI output is configured to utilize different colours, effectively highlighting key information. Search and index results are neatly presented in a table format tailored for the command line, while a progress bar visually tracks the crawler's progress.

# The crawler

The crawler is a class that takes an indexer object and a web page URL. To crawl a page, a method is called to fetch the site’s content using the ‘Requests’ library. This method records when the last request was made and ensures that the politeness window of 6 seconds is observed. Next, the crawler parses the HTML content using the ‘Beautiful Soup’ library and the raw text is extracted. This text is indexed using the indexer which is detailed in the next section. Then, using Beautiful Soup, the links of that page are extracted, filtered to remove previously crawled links, and formatted to add the main website to any relative links found. Then these links are crawled recursively until all links have been crawled. The status of the crawler is shown via a progress bar to indicate how long is left.

# The inverted index

|  |  |
| --- | --- |
| Name | Data Structure |
| URL-to-page ID map | {url: page\_id, ...} |
| Page ID-to-URL map | {page\_id: url, ...} |
| Word-to-page index | {word: {page\_id: [word positions], …}, ...} |

Table 1: Table to show the data structures of the maps and indexes used.

The indexer is a class that generates an inverted index. The class contains 3 main data structures the URL-to-page ID map, the page ID-to-URL map, and the word-to-page index. These are stored as Python dictionaries in the class and their structure is shown in Table 1.

Firstly, the class is given a page URL and its text. The page is assigned a unique integer ID which is used to reference the page in the word-to-page index. Using an ID, rather than the full URL, reduces the size of the main index. The mapping from URL to ID and ID to URL is stored in the URL-to-page ID and page ID-to-URL dictionaries. The text is processed by splitting it into a list of words with any punctuation being removed. This creates a clean list of words to be indexed. The positions of each word in the page are stored in the word-to-page dictionary which is used to find and rank pages during search. The positions are specifically used to identify the proximity between multiple words on a page and thus identify if a phrase is present on a page.

Each map and index dictionary is stored in an ‘index.json’ JSON file. The keys of this file are the dictionary’s names, and the values are the dictionaries themselves. JSON was chosen since it allows Python dictionaries to be stored and loaded easily. To load the mapping and index dictionaries from the file, the file is loaded to one dictionary using the ‘json’ library, and each dictionary is extracted.

# The ranking method

The search result ranking algorithm prioritizes pages based on three key factors: the presence of the exact query phrase, the inclusion of unique query words, and the overall frequency of query words on the page. Implementation involves first consulting the index to identify pages containing at least one query word. These pages are then divided into two categories: those containing all query words and those that do not. These categories are labelled as 'all\_words' and 'other', respectively. Within the subset of pages containing all query words, those with the exact query phrase are identified by verifying that the positions of each query word are consecutive. These pages are moved from the 'all\_words' bucket to a separate 'phrase' bucket. Subsequently, both the 'all\_words' and 'phrase' buckets are sorted based on query word frequency, while the 'other' bucket is sorted by the number of unique query words and then by query word frequency. Finally, the sorted buckets are merged in the order of 'phrase', 'all\_words', and 'other', ensuring the correct ordering of search results.

The ranking algorithm ensures more relevant search results by prioritizing pages with the highest number of unique query terms. This approach ensures that a page containing all the words of a query holds more weight than one with multiple instances of a single word. Moreover, pages featuring the exact query phrase are consistently displayed first, as they directly match the search criteria and prevent other pages from surpassing them solely based on word frequency.

# Using the tool

To use this tool, firstly, create a virtual environment and install the requirements specified in the requirements.txt file. Then, in the command line, run ‘python3 search.py’ in the root directory of the project to start the CLI. A prebuilt index is already included in the project, so using the ‘load’ command, this index can be loaded, and search queries can be run. However, if you wish to wipe the index and rebuild it, the ‘build’ command can be used to crawl <https://quotes.toscrape.com/> and build the index. A progress bar will be displayed to show the progress of the crawler since, due to the politeness window, it can take up to 20 minutes to crawl the site. To view the index of any word, the ‘print’ command can be used. Then the ‘exit’ command can be used to close the CLI.