University of Leeds **School of Computing**

COMP3011, 2023-2024

Web Services and Web Data

A Search Tool for a Simple Website

By

Leandro Russo

201444966

Date: 30/04/2024

1. Introduction

The submitted project fully implements all the features specified in the coursework requirements file. The project has been written and manually tested in Python 3.10.

2. The crawler

To crawl the given website, a class named Spider was implemented.

When the spider class is initialised, it checks whether the base URL has been crawled already. If it has not, it makes a get request to the base URL. The text response is then stored to create the inverted index and extract all the links using BeautifulSoup.

Before storing the new links, a set is created to make sure that the same link does not get stored twice. Also, the set stored only the links with the base URL equal to https://quotes.toscrape.com/.

The crawled link is then added to a file containing all the crawled URLs while the list of new links is added to a file containing URLs that need to be crawled.

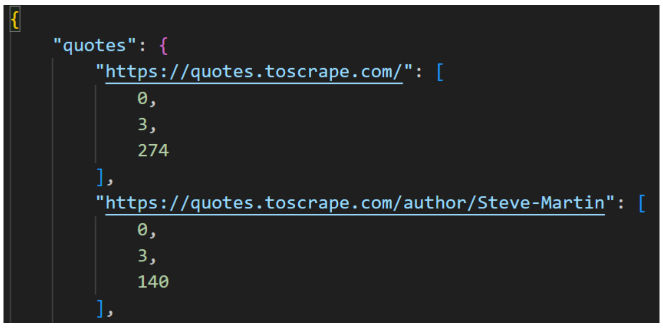
The crawler then waits 6 seconds and checks the file containing the not crawled links, extracts the first one and repeats the cycle until no links are in the file.

3. The inverted index

Before the inverted index is updated, the text response from successful get requests is passed to a BeautifulSoup instance to extract the text. After that, the text is sanitised from special characters using regular expressions and tokenised using word_tokenizer from the nltk library. Also, the "stopwords" corpus from nltk.corpus is used to filter out stop words from the text. Finally, each token is then added to the inverted index.

The index is a JSON file and is loaded and saved using json.load and json.dump.

The structure of the inverted index is as follows:



```
{
    "quotes": {
        "https://quotes.toscrape.com/": [
            0,
            3,
            274
        ],
        "https://quotes.toscrape.com/author/Steve-Martin": [
            0,
            3,
            140
        ],
```

Fig 1 snippet from the implemented inverted index.

A dictionary is used to store the tokens as keys. The value of each key is a dictionary containing URL pages as keys and a list as value. The list contains the positions of where on the page the token was found.

This structure can be expensive in terms of memory, especially if there are millions of tokens and each token appears millions of times. However, it also allows for easy usage, and it can give a deep inside of the structure of a page and where the tokens are located.

4. The ranking method

The ranking method was implemented by combining the score from counting all the query tokens entries on each page with the average distance between pairs of tokens per page (when the query tokens were more than one).

Count of the query tokens entries

After breaking down the query string into tokens, each token's URLs are put into a set. A bitwise OR operation filters this set with URLs containing the second token. This process is repeated for each token to filter out the URLs that do not contain any of the targeted tokens.

Then, all the URLs are added to a dictionary as keys and a counter is set as values. The function “create_dict_with_query_tokens_count” increments the URL counts at every encounter of a target token. The result is a dictionary that has URLs of all the pages containing any of the query tokens at least once and the total number of query tokens per page.

The counts are then normalised and used to build up a score along with the score from the average distance between pairs of tokens per page and the number of unique query tokens per page

$$page\ score = normalised\ count + \frac{1}{average\ proximity + 1} + unique\ query\ tokens + are\ all\ tokens\ concatenated$$

Average distance between pairs of tokens per page

The function “proximity_score” creates a score for each page based on the query tokens. It takes in input query tokens, a single URL, and the inverted index.

The tokens are iterated through in pairs to retrieve the associated list (for simplicity they will be referred to as list A and list B) of positions from the single URL in the index dictionary. Then, the distance between each element of list A and list B is calculated and added up to “total_distance”. After that, “total_distance” is divided by “total_pairs” (total number of pairs) to extrapolate the average distance between pairs. Finally, the resulting “average_distance” is converted into a score using the following formula, where the shorter the distance, the higher the score:

$$\frac{1}{average\ distance\ + 1}$$

The scores produced by “counting the query tokens entry” and “average distance between pairs of tokens per page” are floating numbers between 0 and 1. Hence, they are added together to generate a robust scoring system.

Count of unique query tokens

The function “number_of_diff_query_tokens_per_page” counts how many unique tokens from the query tokens are present on the page. This method is essential for page ranking because it ranks pages where all the query tokens are present higher than the others.

E.g., if the query has 3 tokens, the scoring will be as follows:

Ranking	Score
1	3 + tokens count + average distance + …
2	3 + tokens count + average distance + …
3	3 + tokens count + average distance + …
.	.
.	.
10	2 + tokens count + average distance + …
.	.
.	.
.	.
25	1 + tokens count + average distance + …

Fig 2 shows an example of how the ranking score works

This method works well because the score from the tokens counts and the ones from the average distance are normalised and cannot be greater than 1.

Concatenated tokens

The method “score_for_consecutive” adds to the page score a fixed value of 3 if all the query tokens are found one after the other on the page. E.g., if the query is “love albert bottle” then a page containing “today, I was love albert bottle” receives 3 points that will be added to the score.

5. Using the tool

Once the program Is executed, a while loop is initialized. A list of available commands and the related descriptions are printed out. The user can then write the name of the command along with associated tokens if needed (“print” accepts only one word, while “find” accepts 1 or more words).

the first 2 commands (“build” and “load”) need to be executed to be able to execute “print” and “find”. If the user tries to execute them before the system will not recognise the command.

Two additional commands (available always, with no precondition required) can be used. “show” can be used to print the list of available commands and “quit” can be used to quit the program.