**University of Leeds School of Computing**

**COMP3011, 2023-2024**

**Web Services and Web Data**

A Search Tool for a Simple Website

By

Robert Helps

Sc21rh – sc21rh@leeds.ac.uk

**Date:** 09/05/2024

**Introduction**

My implementation of a search tool for a simple website has implemented the following modules:

* A web crawler
* Construction of an inverted index for a set of URLs.
* A print function which prints the occurrence of a word with the URL index
* A find function which returns a list of pages for a phrase/word

With all these modules we have been able to make a search tool for a simple website. To make these modules we have used the following libraries:

* BeautifulSoup to parse HTML pages
* Requests to request a URL
* re for processing the words from the html page
* Time to allow for a politeness policy of 6 seconds

# The crawler

The first part of my implementation of a search tool was to make a web crawler. The goal of the web crawler was to take a URL as an input, and return a list of other URLs on the current HTML page. As we know, we can use the HTML <a> tag to find other URLs. Upon finding a URL we append it to an array (url). Moreover, upon exploring a URL we remove this URL from the url array and append it to another array called url\_searched. This was so that we could track which pages we have visited so we do not search it again.

Furthermore, the web crawler follows a politeness policy of 6 seconds, as recommended by fellow professionals. We have decided to follow this politeness policy so that we do not overload the web server with requests which would not be expected, so that the website can function as the developer intends.

As well as crawling URLs, we also get all the words on the page by parsing the response from the request by using the BeautifulSoup library. With parsing the page, we can easily extract all the words and store them in an array for indexing. We make sure that all words are put into lowercase, as we are not concerned about cases sensitivity while also putting words containing an apostrophe as a singular word.

# The Inverted Index

As well as making a web crawler, an inverted index data structure was implemented to track the count and position of a word within the URLs we have explored. Within the web crawler, we also get all the words on the page (excluding the <head> tag) and store it as a dictionary. The format of this dictionary is word:(count1, url\_index1),…,(countn, indexn), where n is an integer.

Upon getting the words on the page we count the occurrence of each word and the URL index for the page we are exploring. If the word has not been found previously, we create the word as a new key in the dictionary, else we append it to the word. When we have explored all the URLs found, the url array is null, we then write the dictionary to a file called “Inverted\_Index.txt”. Each line in this file is a new word from the dictionary with a colon separating it from the rest of the data. I believe for this format to be human readable as it is presented similar to how a dictionary would be printed in Python.

# The Ranking Method

In the implementation of Print and Find, we have created a ranking function which ranks the page for a given word/phrase.

The print module ranking function ranks a word by the amount of time it appears within an index. To rank in this way, we first search the inverted index file for the word then extract the data. If the word does not exist, we return to the user with this information. Upon finding the word and the data for it, we have to format the data string into a 2D array in the form [(count1, index1),…,(countn,indexn)]. This array is then sorted by count in descending order, representing which index has a greater count of the word, a more accurate result. We then print this information in an example such as figure 1.

A black screen with white text

Description automatically generatedFigure 1- Example out for print where word = Harper

Furthermore, ranking the webpages returned for a phrase when using the find command we also use the count of words and if the words appear together. Before we rank a phrase, consisting of one or more words, we make sure that the words exist in the inverted index, extracting the data as we have implemented in the print function. Upon finding a set of words that match, we see how many words that were found, and inform the user of this.

After finding the words and their associated counts and indexes, we combine all the indexes into one dictionary and count the occurrence of each index. The higher the value, the more the words appear within a certain index, potentially showing a better search result. In addition, we have further ranked the indexes by if the words appear together. To do this, we check for the given index if each word appears together, within that HTML document, and count the number of times these words appear together. Upon getting this data, we further sort the dictionary of indexes, by if the words appear together and how many times the words occur. The final sorted array is returned to the user as shown in figure 2.

A computer screen shot of a black background with white text

Description automatically generated  
Figure 2 – Example output for find where words = Harper,Lee

# Using the Tool

There are 5 commands for this program: build, load, print, find, exit.

The first command to be used is build which crawls the web and saves the resulting inverted index to the file “inverted\_index.txt”. This command must be used first as we need to build the index before we use any kind of searching features.

The load command loads the inverted index and saves it in the global dictionary, words.

The print command is followed by an input which takes a word from the user. We then print the occurrence of this word for a given document index (ordered as described) if it exists.

The find command is followed by taking a word or phrase of words. These word(s) are then used to return a list of pages, printed to the command line, in the order of best results (as described).

The exit command is used to exit the program.