CMSC 476 – Homework 1 Executive Summary

Stephen Policelli

HTML Tokenizer and Indexer

My program, compiled into the JAR executable "cs476hw1SP.jar" is a HTML file tokenizer and indexer on the content of the text inside HTML tags. To run the tokenizer, use the following command:

java -jar cs476hw1SP.jar <input_dir> <output_dir>

Where *<input_dir>* is the directory corresponding to all inputted HTML files, and *<output_dir>* is the output directory for all outputted information; relative paths are allowed for both arguments. The output directory will be created if it does not exist, but the input directory of course needs to exist and contain only HTML files. The files outputted into the output directory are tokensFreqSorted.txt and tokensTokenSorted.txt, which are the list of all tokens in the file sorted by frequency and by token string, respectively. The *tokenized* directory inside the output directory holds all of the tokenized HTML files, with the name convention "tokenized<original_name>.txt", where <original_name> is the original filename of the corresponding HTML file.

Implementation

I chose to implement this HTML tokenizer and indexer in Java, as I have already previously worked with Jsoup, an HTML parser library for Java. My Main.java file took the stream of text outputted by Jsoup's Element.text() method, which obtains the text content inside of the given HTML element, and sent it to an instance of my Tokenizer.java class. The Element I decided to perform this method on was the <html> tag, so in all cases of valid HTML syntax, this would gather all text content in the document.

In my Tokenizer class, I decided to split tokens was by the regex string "[^a-zA-Z]", which would obtain all non-letter characters between any letter; whitespace, numbers, punctuation, and special characters are all ignored, and all are considered the separators between tokens. Therefore, the String.split() method with this regex string would get me an unprocessed array of token strings, which I would then convert to lower-case and traverse to check for empty strings or one-letter string tokens that are not "a" or "I", since those are the only one-letter words. Then, the final token array would be returned back to Main.java.

Main would then print all of these tokens to the tokenized output file under the tokenized directory, then add all of these tokens to the instance of my Index.java class, where for each element in the token array, it would check if that token is a key in the index hash map, and if exists, increments the existing value, the number of occurrences of that token, otherwise insert it with an occurrence value of 1. After this whole process has been done for all inputted HTML files, the Index class outputs an Entry array, where each Entry is a pair of tokens and their frequencies, and this is sorted twice on two different sorting rankings, both by frequency and by alphabetical order of the token string, and outputted to two files, tokensFreqSorted.txt and tokensTokenSorted.txt.

Tokenization Problems

My tokenizer did not seem to parse as many total words as most of the other classmates, from what I could tell. My best guess as to why is that I used Jsoup, the Java HTML parser, to obtain only the text inside any HTML tags, so any properties of tags, or any comments, with words inside them were ignored.

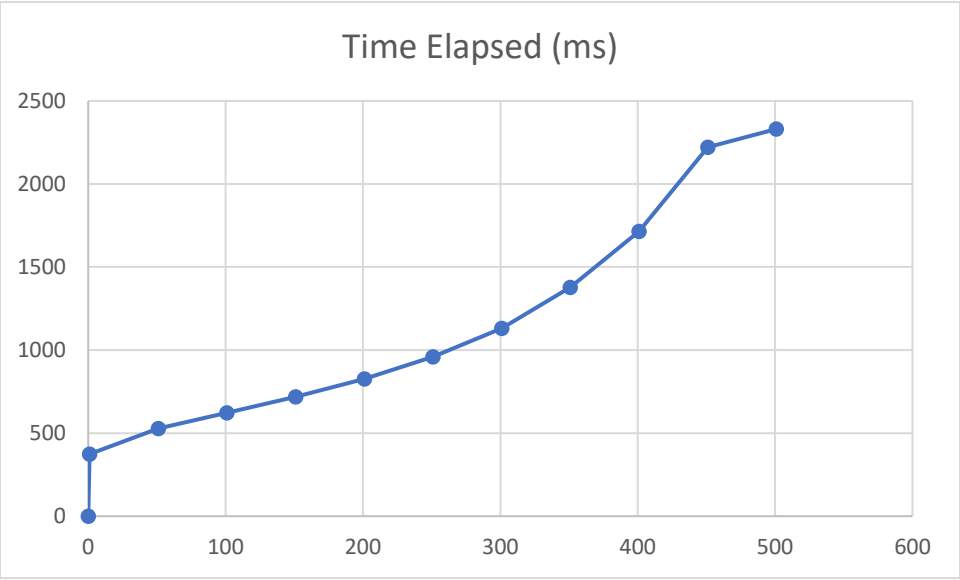
My tokenizer did not parse URLs very friendly. As it simply split words between instances of groups of letters of the alphabet and non-letters, a URL like www.google.com would be split into the tokens “www”, “google”, and “com”. As well, other strange non-word strings like “aaaaaaaaa” were present in the tokens as well, and this comes down to the fact that a tokenizer that wanted only words would have to include the entire English dictionary, which is not a simple feat given that language changes every day and the entire dictionary would take up a lot of memory, and either searching or updating it would likely be inefficient for the runtime of the program.

Some parsed HTML files caused issues when running through Jsoup: these files would not have all HTML tags properly removed from the text content of the files. I then had to ensure that these tags were ran again through the Jsoup parser to completely scrub them. It appears that two times was all that was needed for all test files, but I’m not sure why Jsoup wasn’t able to remove all tags, other than the HTML syntax for specific files were poor, and if more passes would be needed for other files that have similar issues.

Program Efficiency

Some values for the number of inputted HTML files and the time it takes to index all of them is given in the table and graph below.

Number of files	0	1	51	101	151	201	251	301	351	401	451	501
Time elapsed (ms)	1	373	528	622	718	826	959	1131	1377	1715	2221	2331



The program took 1 ms at the beginning to get the list of all HTML files provided and setup the data structures necessary for indexing. The first file took 372 ms since the cache lookups likely all failed, and the system had to open and parse the file the slow way. 50 more files only took 155 more ms, as the files are likely now being found through the cache rather than the slower lookups, and the data structures for tokenizing and indexing are also in constant use, so my computer also kept them readily available. The total shape of the graph is approximately linear or O(n), but there are certainly some outliers, and deviation from the expected runtime, and this is likely due to the individual complexity of each HTML file parsed; perhaps when more time is taken than expected, large files are being processed, whereas when less time is taken than expected, a lot of small files are being processed.

Differences Between My Implementation and Jason Garcia's

As mentioned previously, Jason was one of the students who had more total tokens indexed than me, and again that was likely due to the strategy I employed with Jsoup to eliminate all properties of HTML tags and all comments, and focus only on the text content of each tag. For example, his count for the token "the" was 33,531, whereas mine was 33,426. This is a small difference for this specific token, but for some of the tokens like "az", I had about 400 less instances than Jason.

Despite not communicating about this, Jason and I ended up both eliminating all one-character tokens except for "a" and "I", since these are the only one-letter words. I did this to eliminate undesired tokens in instances of possessive nouns or contractions; "man's" would be split into the tokens "man" and "s", but "s" is not particularly a useful token, so I decided it should be discarded. I did not do a similar thing for any two-letter tokens, but I figured one-letter would contain most cases of these contractions or possessives.

I ran Jason's code on a Jupyter Notebook instance, so it took a lot longer than he told me it ran for on his computer through the python command line interpreter. On Jupyter, it took my computer about 12700 ms to tokenize and index all HTML files, however he told me on his computer it took his program about 3400 ms to run. I then executed it on the command line interpreter, and found that it took my computer about 8650 ms to complete its execution of his program, and the graph his program outputted, of number of files processed vs time elapsed, was completely linear, far straighter than mine.