Daniel Choo
CMSC476 — Information Retrieval
Phase 3: Program Documentation

For Phase Three of the project, I implemented an index to look up which terms appear in the postings file first and the number of documents this word can be found in. I also created the postings file, which contains the document id and the normalized weight of the word in the document. I created a new function called *phase_three*( ) that had some similarities to my *tfidf*( ) function, but was also very different to the latter function as well. We will now examine what changes and additions to the code I have made on this phase of the project.

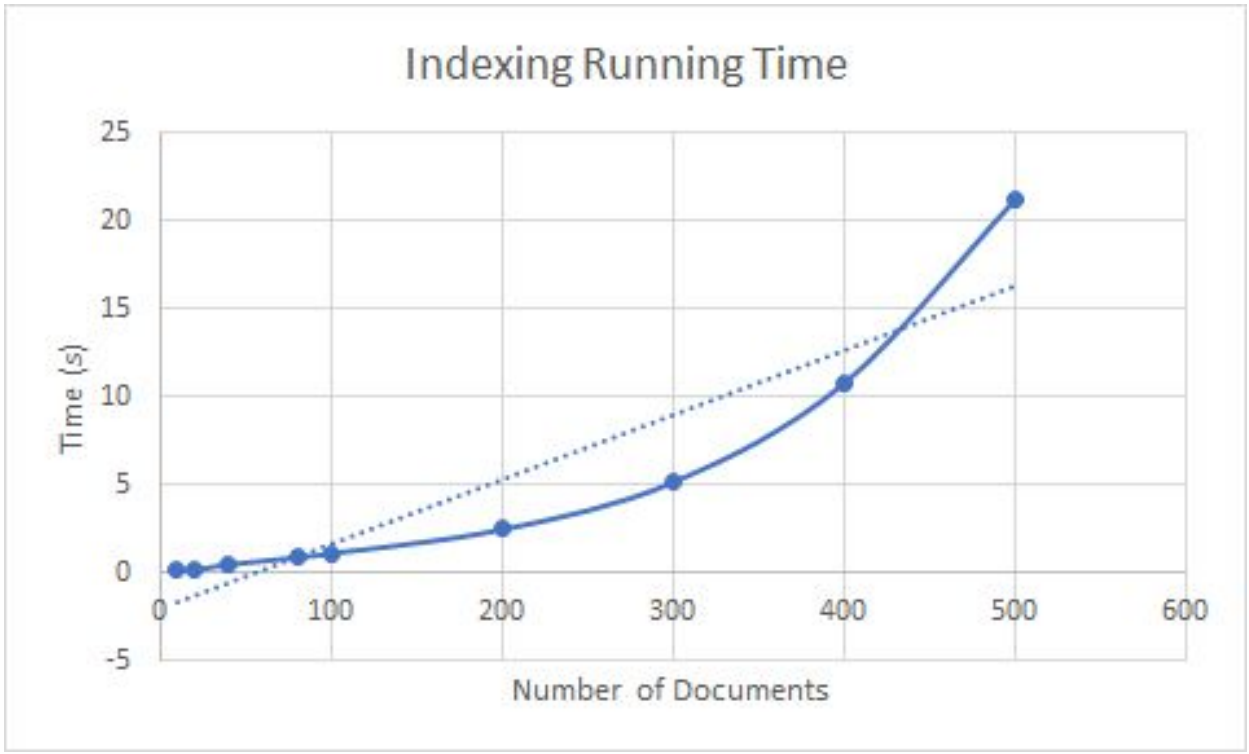
Regarding the changes in the codebase, the *write* function now has a new parameter called *type*. This parameter indicates what type of write we will do, namely UTF-8 versus ASCII. I attempted using json.dumps for the corpus and postings dictionaries; however, I encountered some formatting issues during the file writing process. Thus, the file is not encoded in ASCII, but was just for naming convention sake as it was recommended to be encoded in ASCII for debugging purposes. So hopefully, the ASCII portion of the *write* function reformats the dictionaries in a way that is actually readable to the user.

There are some similarities between Phase 2's function, *tfidf*( ), and Phase 3's function, *phase_three*( ), as we are comparing the runtime between a varying number of documents, i.e., in this case 10, 20, 40, 80, 100, 200, 300, 400, and 500 documents, and utilizing tf for our postings list. Although the two share some similarities, they are completely different functions as the *phase_three* function contains two major dictionaries, corpus and postings. The corpus dictionary contains the word, the number of documents that contain the word, and the location the word was first found in the postings file. The postings dictionary contains the document ID and the weight of the word in that document.

The approach I took to create the corpus was to iterate through the tf_array that was created by my *tf* function and then check whether or not the term is in our corpus yet and if we have seen it in our current document—handled by another dictionary. If we have not seen it yet, then we will insert an array value of [1, 0] to the term in our corpus. If we have seen it and it is not the same document, then we will increment corpus[term][0] by one. If we have the term and we have seen it already, then go to the next term. The approach used to create the postings occurred simultaneously with the corpus, so when the term is in the corpus and is not in the check or if the term is in our corpus and not in our check, then we will append the document counter and the tf value to the postings[term] value.

Following this, the corpus is populated with the first occurence of this term in a document by iterating through all of the postings and assigning the current counter's value to the corpus[terms][1] index. We then sort the corpus by alphabetical order—formatted as an array—and create an array of key-pair values in postings. Finally, we write the file to our designated location and calculate how long it took to do all of this.

The running time of this algorithm appears to be $\theta(2n)$ according to the time produced and this is further corroborated by the line of best fit on the chart at the end of this paper. However, theoretically the worst possible running time of this algorithm should be $O(n^3)$ as we have two nested for-loops underneath our measurement of time the index function takes to process a varying number of documents. We should disregard the for-loop that is regulating the number of documents we are processing, thus the algorithm runs in $O(n^2)$ time. This will occur if there is no repetition of words in all of the documents we processed.


Indexing Running Time
Time (s)
25
20
15
10
5
0
-5
0
100
200
300
400
500
600
Number of Documents