CSCE 5533 Advanced Information Retrieval
Homework 2
By: Kyle Morley

Objectives

The objectives of this homework are to build on the html document tokenizer from assignment 1 and use it to create an inverted index from a directory of html documents. The inverted index will consist of three files, a dictionary file, a posting file, and a mapping file.

Approach

My approach was to start with the results from homework #1, which was programmed in python and used python dictionaries (the python in-built hashmap) to keep track of the count of tokens within a file and also the total occurrences of each token.

The changes I made to my homework #1 code were to modify the tokenizeFile() method in my HTMLLexer class. The method tokenizerfile() previously tokenized a file and output the tokenized file to a text file. This function already contained code that build a python dictionary of the form (token: count), which was used in homework #1 to count occurrences of a tokens within the file. I modified the function to now return the dictionary.

Next a class was made to build the inverted index, called Indexer. The Indexer class contains a dictionary which is used to represent the inverted index in memory. This dictionary has the form of (token: posting_list) where posting_list is a list of tuples of the form (document number, occurrences of the token in the document). This class has two methods: The first is called addToIndex(), which takes as input a token, and a posting list. This method appends the input posting list to the posting list which already exists in memory for the token. The second method is called parse(), which takes as input a document and calls tokenizeFile() to tokenize the document, which as mentioned returns a dictionary of (token: count), then calls addToIndex() for each token in the dictionary.

In the main function, we loop implement the memory based algorithm by looping over all files, and calling Indexer.parse on each file.

At the end, we just need to write out the three files. The mappings and postings file already exist in memory as dictionaries, that we just loop over and write to a file. We create the dictionary file while we are writing out the posting file. When we get to a new token, we save the token, number of occurrences, and the current offset of the file to a python dictionary of the form (token: (occurrences, offset)). We can then loop over the dictionary to make the dictionary.txt file.

Results

(if working in pairs, each set of results should be described separately)
The results of indexing are a trio of files (dict, post, map). Describe the records each contains, how many records, and total size of the files.

The initial set of files that was processed contained 955 files, with 534,919 total words and 100,269 unique words. The final inverted file has 180,990 postings. The dictionary file has 29,221 unique tokens.

The file dictionary.txt is formatted as: token, total occurrences of the token, offset of the first record of token in the postings.txt file. The file has 29,221 records and takes up 568,226 bytes.

The file postings.txt is formatted as: document number, occurrences of token in the document. This file has 180990 records and takes up 1,068,683 bytes.

The file mappings.txt is formatted as: document number, filename. This file has 955 records and takes up 46,586 bytes.

Efficiency

(if working in pairs, each approach's efficiency should be described separately)

Complexity Analysis

The complexity of this approach is O(some function of n) based on the number of documents? Unique tokens? Total tokens? Files? Characters?

Recall the complexity of my tokenizer from homework #1 is O(c + n log n) where c is the number of characters in the documents and n is the number of unique tokens.

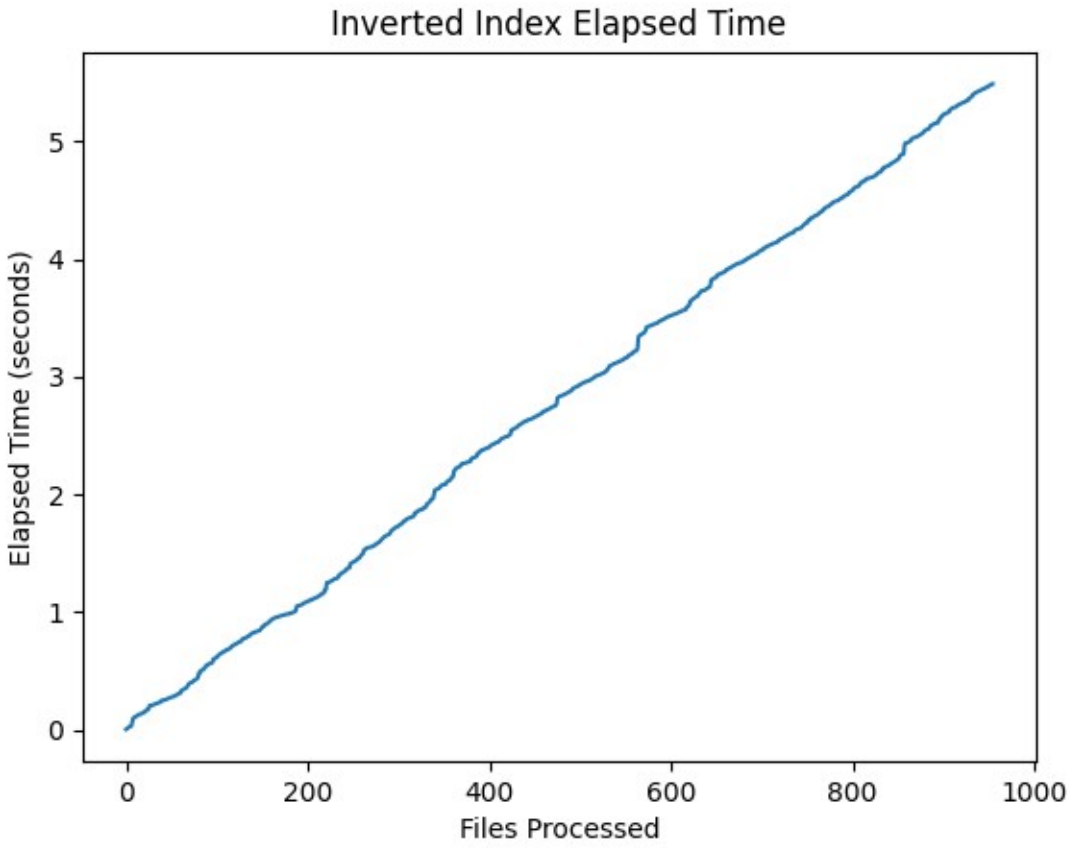
The additional work that is done in this assignment is for each tokenized document we must update the inverted index, and at the very end we must write the three files. Our inverted index update is just a hash table insertion which runs in average time O(1), but has a worst case complexity of O(n) if the insertion has many collisions.

When we write the dictionary file, we sort the hashtable containing the dictionary in memory so that our dictionary.txt file will be sorted in alphabetical order of the tokens. This sort is done with the python default TimSort which takes O(n log n) time where n is the number of unique tokens in the set of files.

So in total our complexity for building the inverted index is O(c + n log n), which is the same complexity of my homework #1.

Timings

Graph of time taken **on turing** as a function of the number of documents. Please provide 2 charts (or one chart with two lines) showing the "real" time (elapsed time from start to end) and "user" time (CPU time used by the program).



To see the cpu time used for this assignment I used the python module cProfile which lists the cpu time spent in each function call of your program. The run time of the program is 7.504 seconds with the added overhead of cProfile. Some of the output of cProfile is below:

ncalls tottime percall cumtime percall filename:lineno(function)
955 0.103 0.000 6.995 0.007 Indexer.py:19(parse)
955 0.626 0.001 6.427 0.007 HTMLLexer.py:96(tokenizeFile)
180990 0.322 0.000 0.464 0.000 Indexer.py:7(addToIndex)

The main logic of the program happens in the Indexer.parse() function where it makes a function call to HTMLLexer.tokenizeFile() and a call to Indexer.addToIndex(). We can see that it takes 6.427 seconds to tokenize the documents, and only 0.464 seconds in total to build the inverted index. Each additional document takes on average 0.007 seconds of cpu time to process and add to the index.

Testing
Sample Run

Resulting Files: dict.html

0, 537, 779751
00, 30, 232052
0000, 2, 16211
000000, 26, 25845
000020, 1, 990887
000033, 1, 196565
000050, 1, 792031
000080, 1, 826543
000090, 1, 433097
000099, 2, 984856

Resulting Files: post.html

120 1
362 1
152 2
411 1
131 1
723 1
214 1
384 1
418 1
671 1

Resulting Files: map.html

1 /home/sgauch/public_html/5533/files/799.html
2 /home/sgauch/public_html/5533/files/64.html
3 /home/sgauch/public_html/5533/files/558.html
4 /home/sgauch/public_html/5533/files/476.html
5 /home/sgauch/public_html/5533/files/977.html
6 /home/sgauch/public_html/5533/files/281.html
7 /home/sgauch/public_html/5533/files/344.html

8 /home/sgauch/public_html/5533/files/828.html
9 /home/sgauch/public_html/5533/files/678.html
10 /home/sgauch/public_html/5533/files/594.html

Tracing one “query”

Dictionary.txt entry: ['watchdog', ' 1', ' 1061765']
Postings.txt entry at offset 1061765: ['880', '1']
Mappings.txt entry:['880', '/home/sgauch/public_html/5533/files/299.html']

grep -i "watchdog" /home/sgauch/public_html/5533/files/299.html returns:
nonmonotone watchdog strategy is employed in applying the path search;

Here is another example with a word that has 2 occurrences:
Dict record: ['token', ' 2', ' 314124']
Postings at offset 314124 :['308', '1']
['662', '1']
Mappings:['308', '/home/sgauch/public_html/5533/files/240.html']
['662', '/home/sgauch/public_html/5533/files/905.html']

grep -i "token" /home/sgauch/public_html/5533/files/240.html returns:
 LAN and WAN technologies (Ethernets, Token rings, ATM, FDDI)

grep -i "token" /home/sgauch/public_html/5533/files/905.html returns:
filters, token passing etc.