Austin Bailey

Dr. Claudia Pearce

CMSC 476

8 April 2018

Homework 3: Inverted File/Index

In this homework, the objective was to create an inverted file search. This is also known as an index. The idea is searching the files for the term rather searching the terms for the file. To do this, the preprocessor made previously needed to be extended to properly handle this kind of functionality.

Using a memory-based algorithm, I extended the preprocessor by using hash tables to store the lists of weights for certain terms across the entire corpus. This was done in my second loop that goes through the list of files. Having already calculated the weights and the number of documents that all terms in corpus appeared in, I would add to the dictionary for each weight of each document in a list.

After doing all of this, I exited the loop and went into creating both the dictionary and postings list files. Creating the dictionary file first, I used the dictionary I created earlier in the loop to access the lists of weights for each term and appended to a list in order: the word, the number of documents the word appeared in (in a hash table), and the first position in the postings list file. Following that, I then went and appended the list created for the postings file. For each term, it would go and append the pair of the document id and the weight. It would

make sure to skip any documents that it wasn't in by seeing the weight of it in advance. This was done for each key in the dictionary created for the two files. It would then write to the two files to the directory specified.

As for efficiency, it didn't take much longer than the original preprocessor from HW2. This was helped by essentially performing half the calculations within the original loop that did the weighting. Unfortunately, I had to have the other for loop that created the dictionary and postings list files outside because of how it was structured. This added a little bit more time to the overall runtime, but it was still relatively quick. From the graph, you can see that even with 320 files, it doesn't go beyond 15 seconds. When moving up to 500, it increases sharply to 53.84 seconds, which could be improved. Overall, well made but there are possible optimizations that can be made to make it even faster.

Comparing the file sizes, it was interesting. The size of the first 10 html files was 135 KB, which compared to the 22 KB size of the dictionary file and the 53 KB of the posting file is about what I expected, considering the terms are being trimmed. Same with the first 320 html files, which is 3.4 MB in size but compared to the 268 KB dictionary file and 2.637 MB of the posting file is considerably smaller. The first 500 files were 11.3 MB but the dictionary file was 999 KB and the posting file was 7.455 MB, which is still considerably smaller. So it shows that the preprocessor is trimming the files tokens down and making them more manageable to read and gain data from.

So overall, I think what went well with this was the preprocessor extension and the data that was gained. The only negative aspect that I feel I could do a better job in is the actual

efficiency of the program. It could run faster than it is right now. But this was largely successful despite the less than efficient approach.