Nick Benevento

Computer Architecture

December 5, 2018

Project 5 Report

a) If all the words were put into one large linked list, then given N documents with each document containing m words, then searching for a query of K words would take O(m*N*K). If we let m = the number of words in a document, there would be m*N words in the list. Each word search would take O(m*N) amount of time, so the entire process would take O(m*N*k).

b) Using a more efficient approach of a hashmap, then each bucket would contain a linked list of the same amount of words, m*N/b where b = the number of buckets. Then, the time complexity would be O(1*K), because the program is able to directly jump to any bucket, and only have to search through a constant few nodes to find each word in the query.

c) Removing stop words leads to a more efficient solution because there will be less nodes overall in the hashmap. So, when searching for a word, the linked list for a bucket will have less words, meaning it will take less operations to search through them.

The program starts by asking the user to enter the number of buckets for the hashmap. I have an error test here to make sure that the user input is a number, and not a string or some other character. Something I do here and also whenever I get user input is to make sure to clear

the stdin. Whenever a character is gotten from the terminal, any characters after that entry, including the newline character, are not read (they are still in the buffer). So, these are cleared out so that there is not a problem getting further input. Next, the program prompts the user to search (S) or exit (X). If the user enters X, then the program frees the memory used for the hashmap and the program ends. Again, there are error checks in place so the user has to enter either s or x.

The next part of the program is the training phase. Here, I have implemented the extra credit portion of the project so that all the files in a certain directory matching a search string can be read in. I have simply put all 3 text files in a directory called documents/, so to read all the .txt files in this directory, the user can enter 'documents/*.txt'. Next, the program either returns -1 if this fails, or goes through and reads all the words/document pairs into the hashmap. In this phase, the words read in from the file are also converted to lowercase so that two words with different cases are counted as the same word.

I have reworked my hashmap implementation from homework 7 to the example in figure 2.

d) The second approach shown is the more efficient solution. Using approach 1, in order to calculate the df, the program has to go through the list and count how many documents it appears in. In approach 2, the head node contains this information. When removing the stop words, approach 1 has to manually search through the list to find each occurance of the word, and then delete it from the list. Approach 2 only has to find the word (which appears once at the head), and then it can delete the entire list at once.

I decided to use approach 2 because it is not only more efficient, but made more sense to me and was easier to implement than approach 1.

After the training phase, the program calls the stop_word() function that removes any stop words from the hashmap. It does this by going through each node and checking if it appears in all the documents. If it does, then the node (and it's document list) are removed from the map.

Next, the program enters a loop that asks the user for the search query. Since the project design specifies that this must be of any length, this string must be gotten and allocated dynamically at run time. The way I implemented this was by starting with a string of length 5. I get 1 character at a time from the user, and add this character to a string. Then, if the size of the string is equal to the initial length, I increase the initial length and reallocate memory for extra space. Then, the process continues until the user presses enter.

I also implemented a separate linked list data structure, doclist, to hold the document ID and the tf-idf score for that document in relation to the search query. A data structure is needed because there are 2 separate fields, a string and a double, that need to be stored together. It is also easy to use this structure at the end to print out the list in order of highest to lowest scores when determining the ranking.

After getting the query, the program ranks the documents by relevance using the tf-idf ranking system described in the project description. It splits the query string that was read in into tokens, which gets the search words of the query, and searches for that word in the hashmap. If the word is not found, the loop continues to the next search word. Once the word is found, it loops through that word's document list and for each document, adds the total score to the corresponding document in the doclist. At the end of this process, the doclist is printed out in order of high to low relevance to the search query. As specified in the project description, if the document score is 0, then the document is not printed out, because it has no relevance to the

search query. Finally, the document list values are reset, so the user can enter a different search query string and receive accurate results.

After this, the search query is freed and another search query is prompted for the user to input. If they enter a certain character, '#', then the loop breaks, the memory for the document list and hashmap are freed, and the program ends.