**CSCE 5533 Advanced Information Retrieval**

**Homework 3**

**Due: Sunday, Oct. 21 at 11:59pm**

**By: Chenglong Lin**

**Objectives**

The objective of this homework is to implement a ranking algorithm on top of the file inverter (homework 2). This allows the user to obtain the most relevant documents in the document collection by querying the inverted files.

**Approach**

Part 1: Indexing

To deal with words of length 1, in all my regex rules, I added an if statement to check the length of the matched token. Only matched token with length larger than 1 are inserted into the local hash table.



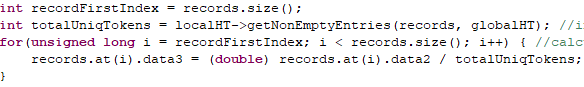
To deal with the stop list word, I first obtained the most frequent occurring stop list word in the tokenizer result from homework 1. I added stop list words in to an array. For every token matched by the regex rule, it will check if the token is in the stop list array. Another way of implementing the stop list is to add regex rules that match the stop words. This method would be more efficient than using an array. I’m using an array because my stop list has only 18 stop words, so it’s not very costly to iterate through.





My term weighting/ranking formula is rtf\*idf. Rtf is freq of token i in document j / totalfreq of all tokens in document j. idf is Log10(# of doc in collection / # of doc containing the token).

To calculate the rtf, in pass1, for every document processed by the lexer, I grab the non-empty entries from the local hash table and calculate the rtf. The rtf values are saved in the 4th column of the temporary file.



To calculate the idf, in pass2, for every alphabetically first token/buffer, I calculate the idf for that token. When writing the posting file, I calculated the weight by multiplying rtf value stored in the alphabetically first token/buffer with idf value calculated.





Part 2: Querying

To query the inverted files, use following format:



Command line arguments do not have to be tokens. You can also query the document that contains linux command line operators, but you have to wrap your query within the single quotation mark.



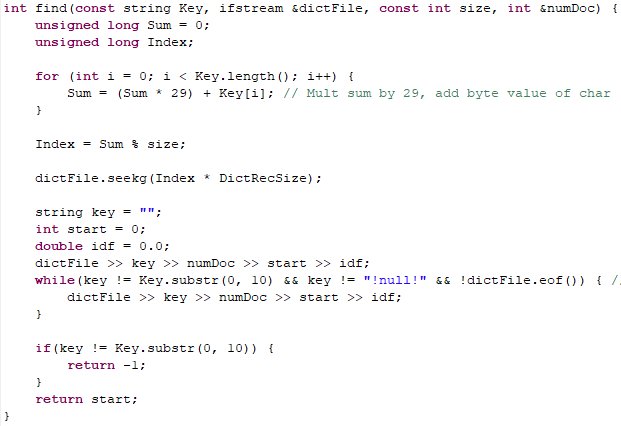
The data structure of my accumulator is an array. It’s fast to access, because the index corresponds to the document id. However, the down side is it consumes a lot of space and most of the space will be empty. Since the size of the array is determined by the size of the collection and the size of the collection is 1769, using an array as the data structure of my accumulator isn’t too bad.

My query processing algorithm can be broken down to tokenization of query string, weight accumulating, and sort/displaying stage.

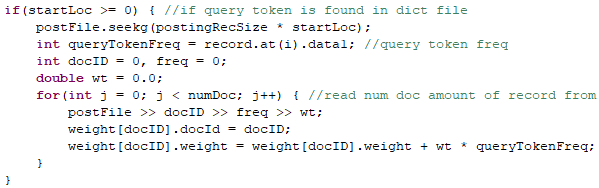
To preprocess the query string, I used the same flex lexer rules from homework 1. The only difference this time is we are preprocessing something in memory instead files in the disk.



To count weights for each unique query token, I wrote a function that can determine the record position of the token in the dictionary file. This function uses the same hashing function as the 2-pass algorithm.



If the query token is in the dictionary file, it will seek to posting file using the start value in the dictionary file.



Once all query tokens are processed, I used the C++ sort function to sort the accumulators in descending order. After the accumulators are sorted, I display the first 10 items in the accumulator.



**Results**

Part 1: Indexing

My map file record size is 17 characters. The document id field uses 4 characters, because the total number of documents in the collection have 4 digits. The file name field uses 11 characters, because the largest document name “simple.html” has 11 characters. There is 1 space character in between the fields. There is a new line character at the end of the record.

My dictionary file record size is 32 characters. The token filed takes up 10 characters. The number of documents containing the token field takes up 4 characters. The start position takes up 6 characters. The idf field takes up 8 characters.

My posting file record size is 18 characters. The document id takes up 4 characters. The frequency of the token field takes up 3 characters. The weight of the token takes up 8 characters.

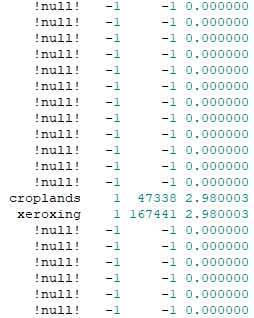
The number of records in the dictionary file is 3 times of the number of unique tokens in my collection.

The number of total tokens in my preprocessed files is the same as the number of entries in my postings file.

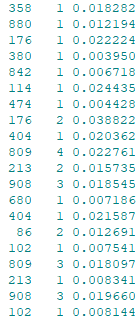
The size of my dictionary file is 2 times larger than the one from homework 2. This is caused by the space characters generated by the fixed record size.

In theory, after implementing stop word list and removing words with length of 1, the posting file size should decrease. The posting file I got from this assignment is also 2 times larger than the posting file from homework 2. This is also caused by the space characters generated by the fixed record size.

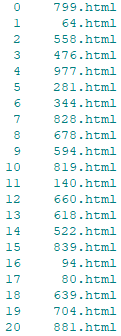
First 20 records of dictionary file:



First 20 records of posting file:

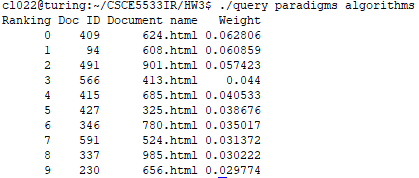


First 20 records of mapping file:



Part 2: Querying

The query result has 4 columns. The first column is the ranking. The smaller the ranking number entails higher relevance. The second column is the document id. The third column is the document name. The fourth column is the document weight of the query string.



**Efficiency**

Complexity Analysis of Querying

Because I used the lexer to preprocess the query string, the tokenization is O(N) | N: number of characters in the query string.

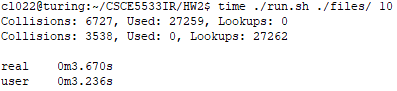
I used an array as my weight accumulators, the weighting process is O(N) | N: # of unique query tokens.

Accumulators are sorted using the C++ sort function. This sorting function has an average time complexity of O(NlogN) | N: # of accumulators in the array, because it’s a data comparison-based sorting.

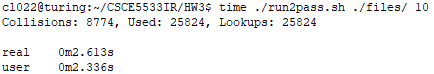
Timings

The inverting/indexing process time in this assignment is about the same compared to homework 2, because I did not make any major change to my 2 pass algorithm.

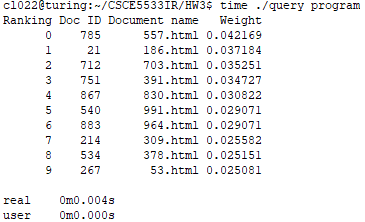
HW2:



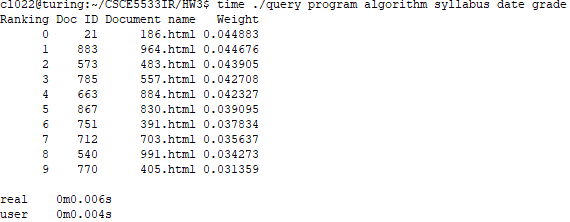
HW3:



program:

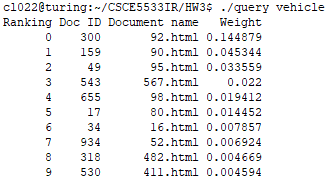


program algorithm syllabus date grade:



**Testing**

Sample Run



Show docids and weights, in order for the following queries:

Gauch:



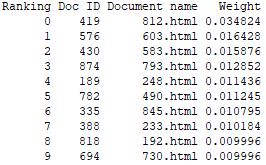
the:



c:



20:



20.07:



30332-0280:



[sgauch@uark.edu](mailto:sgauch@uark.edu):



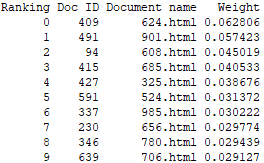
[www.cc.gatech.edu](http://www.cc.gatech.edu):



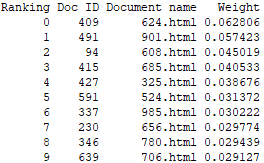
<http://www.cc.gatech.edu/systems/projects/Catamaran>:



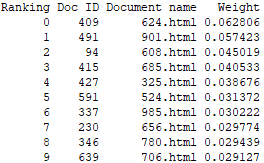
<HTML>algorithms<HTML>:



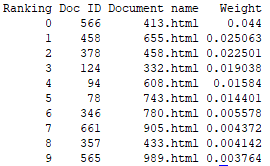
Algorithms:



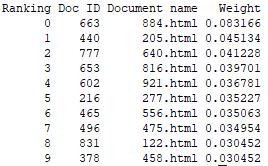
algorithms:



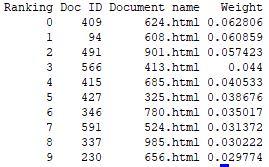
paradigms:



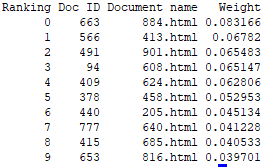
programming:



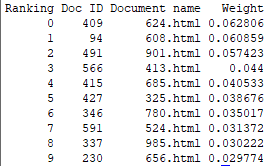
paradigms algorithms:



paradigms algorithms programming:



paradigms-algorithms:



paradigms!! algorithms?:

