CS6913 Web Search Engines

Homework 3

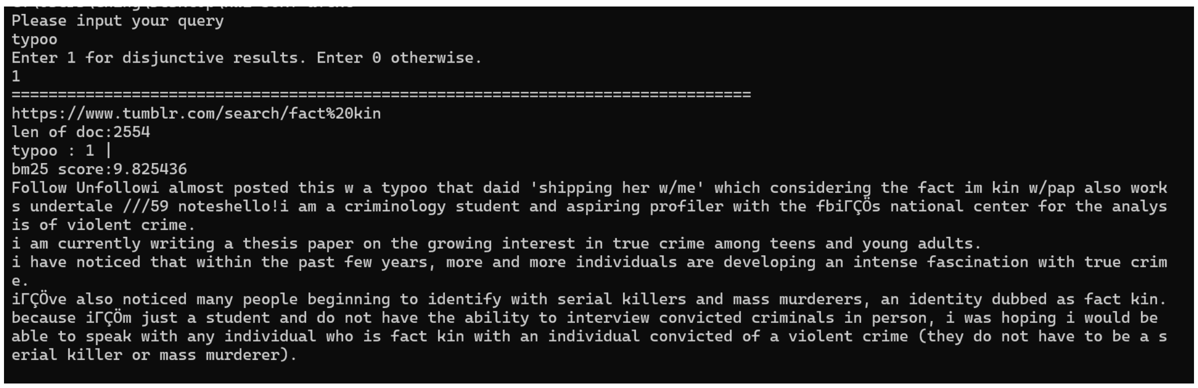
Ching Huang
N17375561 (ch4802@nyu.edu)

Overview

The goal of this assignment is to create a program that mimics the functionality of a search engine. A search engine takes in a query from the user and returns a list of web pages that are relevant to the query. This assignment uses a fixed set of web pages as the data set, and each web page is parsed into a document. The program takes in queries from the user and returns snippets of relevant documents, which are defined as documents that contain query terms. There are two main tasks in this assignment. The first task is to build an index structure based on the given data set. This index structure will allow the program to efficiently look up documents and check the frequency of a term inside the document. The second task is to perform query processing. Query processing mainly entails parsing a query into terms and getting query results from the index structure.

Functionality of the program

The program is written in C++ and runs on a command-line interface. To run the program, compile the C++ program file with g++ and then execute the executable file created. When the program is started, it will first take around 1 to 2 minutes to load. Once the program is ready, it will ask the user for a query input. It will also allow the user to choose whether he wants a conjunctive or disjunctive result. The user can indicate whether he wants a conjunctive or disjunctive result by typing 1 for disjunctive and 0 for conjunctive when he is prompted. If the user wants a conjunctive result, the program will return a list of documents, where each document contains all the terms in the query. If the user wants a disjunctive result, the program will return a list of documents, where each document contains any of the terms in the query. Each document is ranked by its BM25 score computed on the terms inside the query. Only the top 10 documents are returned as the result of the query. After returning the result of one query, the program will be available to take in more queries from the user. A sample query and result are shown below:



```
Please input your query
typoo
Enter 1 for disjunctive results. Enter 0 otherwise.
1
================================================================================
https://www.tumblr.com/search/fact%20kin
len of doc:2554
typoo : 1 |
bm25 score:9.825436
Follow Unfollowi almost posted this w a typoo that daid 'shipping her w/me' which considering the fact im kin w/pap also work
s undertale ///59 noteshello!i am a criminology student and aspiring profiler with the fbiΓÇÖs national center for the analys
is of violent crime.
i am currently writing a thesis paper on the growing interest in true crime among teens and young adults.
i have noticed that within the past few years, more and more individuals are developing an intense fascination with true crim
e.
iΓÇÖve also noticed many people beginning to identify with serial killers and mass murderers, an identity dubbed as fact kin.
because iΓÇÖm just a student and do not have the ability to interview convicted criminals in person, i was hoping i would be
able to speak with any individual who is fact kin with an individual convicted of a violent crime (they do not have to be a s
erial killer or mass murderer).
```

First task: Creating the index structure

The Index Structure

The index structure contains 3 files:

1.	DocMeta.txt	(~280MB)
2.	Lexicon.txt	(~804MB)
3.	InvertedIndex.bin	(~3.1GB)

DocMeta

- DocMeta contains general information about the documents. For each document in the given dataset, DocMeta stores its docID, the URL the document is from, the number of terms inside the document, and the offset to the document's text in the original data set file. The documents are sorted by their docIDs, which starts from 0 to the number of documents in the data set minus 1. Each document occupies a line in DocMeta.

Lexicon

- Lexicon contains information needed to find a specific term in the inverted index file. For each term in the given dataset, Lexicon stores the term, the number of docID blocks in its inverted list, the number of docIDs in the last docID block of its inverted list, the position(offset) of its inverted list in the inverted index file, and the size of the metadata in its inverted list. The terms are sorted in alphabetical order, and each term occupies a line in Lexicon.

InvertedIndex

- InvertedIndex contains inverted lists for each term found in the dataset. Each inverted list starts with some metadata and is followed by docID blocks and frequency blocks. The metadata contains the last docID in each docID block, the block size of each docID block, and the block size of each frequency block. All docID block contains 64 docIDs, with the exception of the last docID block in the inverted list. All frequency block contains 64 frequencies, with the exception of the last frequency block in the inverted list. The whole inverted list, including the metadata, is encoded with var-byte. The docIDs and frequencies are sorted in the order of the docID. Only the first docID in each inverted list is stored by its actual value. The rest of the docIDs are stored by their differences with the docID preceding them. Frequencies start from 0, so the actual frequency of a term in a doc is the frequency found in the inverted list + 1.

Constructing the index structure

The construction of the index structure can be split into three parts:

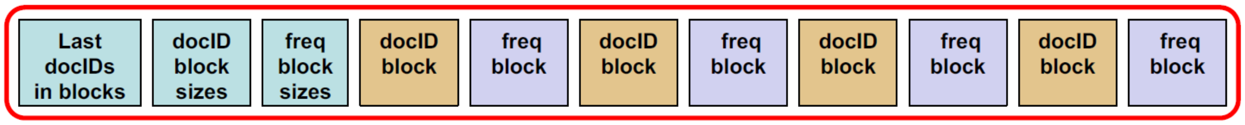
- Parsing data set, creating raw postings, and creating a DocMeta file to store information about each document
- Sorting the raw postings

- Consolidating raw postings into final inverted lists. This includes compressing and writing inverted lists to an InvertedIndex file and creating a Lexicon file to store the metadata of the inverted lists

The data set was first uncompressed then read and parsed line by line. By finding specific tags in the lines, such as <TEXT> and </DOC>, the start and end and the source(URL) of each document were determined. Document IDs (docID) were assigned according to the order in which the documents were parsed. For the inverted index, a term is defined as a string of characters that do not contain whitespaces or punctuation. All characters were converted to lowercase characters during parsing and punctuations were changed into whitespaces. The stringstream library in C++ was used to get all the terms separated by spaces. Raw postings of the form (term, docID) were generated in the process and written to a RawPostings file directly. After parsing a document, the information of the document, such as its docID, URL, number of terms it contains, and the offset of the document in the original data set file was updated inside a DocMeta file. By the end of this phase, 2 files were generated. One file contained all the raw postings and the other contained all the information about the documents. This part of the program took around 20 minutes to run.

In the second phase, Unix sort was used to sort the raw postings primarily by their term and then by their docIDs. The command used was `sort -k1,1 -k2,2n -o sortedPostings.txt rawPostings.txt`. -k specifies the key for sorting. In this case, the command means that rawPostings.txt should first be sorted by the primary key, column 1 in the document, which contains the terms. Then it should be sorted by the second key, column 2 in the document, which should be read as numbers, and which are the docIDs. This took almost 2 hours to run.

In the last phase, the sorted postings were consolidated. By reading the sorted posting file line by line and checking whether the current posting was the same as the previous one, the program could get the frequency of each term inside each document in the data set. This information was stored in the form of inverted lists in the InvertedIndex file. Each inverted list contains information about the occurrences of a term inside the data set. The inverted lists were created block by block, each containing a fixed amount of docIDs or frequencies. For each docID in an inverted list, the frequency of the term in the document corresponding to the docID is stored in the next frequency block. The structure of an inverted list is shown below.



(source: lecture slide for CS6913 CSE Department NYU Tandon School of Engineering)

Four temporary files were used during the process of creating an inverted list. These files kept parts of an inverted list during the creation process before the whole inverted list was written in the final InvertedIndex file. This ensured that the inverted lists didn't have to fit in the memory. Inverted lists and metadata were compressed using var-byte encoding and written in binary form. In addition to the InvertedIndex file that contained all the inverted lists, the Lexicon file, which contained information about each inverted list, was also generated in this phase. This part of the program took around 25min to run.

Major functions for constructing the index structure
parseDataSet(dataSetFile, rawPostingFile, docMetaFile): this function parses the data set and generates the DocMeta text file, which contains information about each document inside the dataset, and the RawPosting text file, which contains raw postings of the form (term, docID).

operationSort(): this function runs UNIX sort on the system it is running on and is used to sort the raw postings.

For my homework, I used a WSL terminal and input the sorting command directly because my computer's system was Windows and didn't have a sort command that had the same function as the UNIX sort.

consolidatePostings(sortedPostings, invertedIndexFile, lexiconFile): this function consolidates term frequency in documents from the sorted raw postings and writes them out into an InvertedIndex binary file in the form of inverted lists. The function also creates a Lexicon file that stores metadata for the inverted lists.

encodeVarBytes(numbers, size, buffer): takes in a vector of integers and encodes each integer into variable-length characters.

Design decisions and limitations

- In the lexicon, the number of docs contained in the last block of the inverted list is stored. This allows the total amount of docs that are contained in the inverted list to be computed directly.
- In the lexicon, the size of the metadata of each inverted list is stored. This allows the metadata to be separated from the inverted list without extra computation, making it faster to load the data during query processing.
- Unix sort is used because it is an i/o efficient sorting mechanism. The limitation is that there needs to be enough memory in the computer for the sorting to be completed successfully. Since Unix sorti can only be done on specific systems, and my computer, which is in Windows, does not support UNIX sort by default and does not have a similar Windows command, the sorting had to be done separately in another terminal, creating a gap in the program.

- The postings were sorted not only by the term but also by the docID during Unix sort, which made consolidating them into inverted lists much easier in the next step.
- For compression, VarByte was chosen to be used as the encoding method due to its simplicity and effectiveness in compressing data. Furthermore, techniques such as storing frequencies starting from 0 instead of 1 and sorting and storing the differences between docIDs instead of the actual docIDs also helped compress the inverted index.
- By storing the content of an inverted list in blocks and storing the block sizes and last IDs of each docID block at the beginning of the inverted list, parts of the inverted list can be skipped during query processing.
- Documents are sorted in the DocMeta file by their document ID to make lookup faster. Terms are also sorted in the InvertedIndex file.
- The consolidation of sorted posting to inverted index could be faster with the assumption that the compressed form of an inverted list always fit in memory. Since this assumption is made during query processing, where inverted lists for query terms are loaded into memory at the start of query processing, the assumption could and should also be made here. With this assumption, temporary files that store parts of an inverted list before the whole list is created would be unnecessary, and there will be fewer reads and writes into files.

Second task: Query processing

In the second task, the program is written to take in queries from users and return query results based on the index structure generated in the first task. Query results are essentially snippets of documents that are deemed relevant to the query. This task can be split into loading data from the index structure, query intake and parsing, candidate and final results generation, and snippet generation.

Loading of data from the index structure

Before the start of query processing, the program loads the information stored in the DocMeta file and the Lexicon file into memory. Each document data inside the DocMeta file is parsed and stored as a vector<string>. The average length of all documents inside the data set is also calculated along the process. Each inverted list data from the Lexicon file is also stored as a vector<string>. After the program takes in and parses a query, the program finds the location of each query term's inverted list by performing a binary search on the lexicon vector and reading the information stored inside (the offset of the inverted list inside the InvertedIndex file). The inverted list for each query term is loaded from the InvertedIndex file and assigned a char pointer. Furthermore, the metadata of each inverted list, which is stored at the beginning of each inverted list, is extracted, decompressed, and stored as a vector<int>. The rest of the inverted list remains in compressed form until it is used. By loading the metadata and inverted lists at the beginning of query processing, the program can get query results more efficiently.

Taking and parsing queries
The program takes in a query from the command line and parses the query by converting every character into lowercase and removing any punctuations in the query. It then separates the query into query terms based on the whitespaces in the query. The program also takes in an integer, which tells the program whether it should generate conjunctive or disjunctive results. A conjunctive result means that all the terms that appear in the query must also appear in all the documents returned by the program. Meanwhile, a disjunctive result means that any document containing at least one of the query terms can be returned as a valid result. The query processing function then uses the query terms and the information of whether the result should be conjunctive to find documents that are relevant to the query.

Generating candidate and final results
Part 1: Document ranking function
For every query that a search engine receives, the search engine generates candidate results, which are web pages relevant to the query. Each candidate result is then assigned some score based on some ranking function or machine learning model. This score ranks how "good" the result is regarding to the query. In this program, the BM25 score of a document and the given query is used to rank candidate results.

The BM25 score is calculated by the following formula:

$$BM25(q,\ d) \ = \ \sum_{t \in q} IDF(t) \ * \ (f_{d,t} \ * \ (k_1 + 1))/(f_{d,t} \ + K)$$
$$IDF(t) \ = \ log\,(\,1 \ + \ (N \ - \ f_t \ + \ 0.5)\,/\,(f_t \ + \ 0.5))$$
$$K \ = \ k_1 \ * \ ((1 \ - \ b) \ + \ b \ * \ docLen/avgDocLen)$$

q = query
d = document
N = total number of documents in the collection
f_t = number of documents that contain the term t
$f_{d,t}$ = frequency of term *t* in document *d*
$docLen$ = length of document *d*
$avgDocLen$ = the average length of documents in the data set
k_1 = constant, set to be 1.2
b = constant, set to be 0.75

The BM25 takes into account the importance of each query term relative to the whole data set and its importance within each document. If a term is considered important in the data set and is also frequently used inside one document, that document will have a high BM25 score.

Part 2: Generating candidate and final results
The program finds documents relevant to the query by utilizing the information about the query terms, their inverted lists, and documents in the data set, which are all loaded into memory at the start of the query processing.

For a conjunctive query, the program uses Document At A Time query processing to get the documents that contain all query terms. In Document At A Time query processing, the program goes through each docID in the shortest query term's inverted list and checks whether the docID exists in all the other query terms' inverted lists. If the docID exists in all the other query terms' inverted lists, the document to which the docID corresponds to will be deemed as a relevant document to the query and stored in a candidate result list. If the docID does not exist in all the other query terms' inverted lists, the program will jump to check for the next possible candidate docID in the shortest inverted list among the query terms' inverted lists. For each document in the candidate result list, the program calculates its BM25 score. The candidate documents are then sorted based on their BM25 score and the top 10 documents with the highest scores are returned as the final result.

For a disjunctive query, the program uses Term At A Time query processing to obtain documents that contain any of the query terms. In Term At A Time query processing, the program goes through each query term inverted list one at a time. For each docID in a query term's inverted list, a partial BM25 score is calculated for the corresponding document. The partial BM25 scores for each document obtained after processing all query terms' inverted lists are summed together to get the final BM25 score for each document. The documents are then sorted by their scores. Documents with BM25 scores of 0 are disregarded and the top 10 highest-scoring documents are returned as the final result.

Snippet generation
After getting the list of documents most relevant to the query, the program generates snippets for each document by reading from the original data set.
For a disjunctive query, the program returns five or fewer continuous lines from the original document where the first line is the first line in the document that contains a query term.
For a conjunctive query, the program returns five or fewer lines from the original document where at least two query terms are contained in the snippet. This is achieved by reading two lines from the document starting from the first line that contains a query term, then skipping to read another two lines from a line that contains a query term that is still absent from the snippet.

Performance
The program takes around 1 to 2 minutes to load the data in the Lexicon and DocMeta files before it starts taking in queries. The time required for query processing is dependent on the query. Usually, conjunctive queries require a shorter query processing time than disjunctive

queries. Short queries with terms that have shorter inverted lists can be completed within a second, while longer queries or queries with terms that have huge inverted lists might take a few seconds to be completed.

Major functions
readDocMeta(docMeta): this function reads the DocMeta file for an inverted index and stores information about each document in a string vector. It also calculates the average length of a document in the dataSet.

readLexicon(lexicon): this function reads the Lexicon file for an inverted index and stores the information for each inverted list in a string vector.

runQueries(lexicon, docMeta, invertedIndex, sourceFile): this function accepts and processes queries according to users' input.

processQuery(lexicon, query, isDisjunct, docMeta, invertedIndex, sourceFile, numResults): this function processes and returns the results of a query.

openList(invertedIndex, term, lexicon, termToInvertedList, termToLexicon, termToListMeta): this function reads and stores the inverted list for a term in memory. It also decompresses and stores the metadata and lexicon for the inverted list.

getDisjunctQueryResultsRanked(termToInvertedList, termToLexicon, termToListMeta, docMeta, numDocsToTerm, numResults): this function returns the disjunctive results for a query, ranked by their BM25 score.

getConjunctQueryResultsRanked(termToInvertedList, termToLexicon, termToListMeta, termToCurrentIDBlock, termToCurrentFreqBlock, docMeta, numDocsToTerm, numResults): this function returns the conjunctive results for a query, ranked by their BM25 score.

getDisjunctSnippet(docID, terms, docMeta, source): this function generates a snippet for a document for a disjunctive query by referencing the source file.

getConjunctSnippet(docID, terms, docMeta, source): this function generates a snippet for a document for a conjunctive query by referencing the source file.

nextGEQ(invertedList, k, lexicon, listMeta, currentIDBlock): this function finds the next posting in the given inverted list with docID >= k.

retrieveIDBlock(numBlocks, docsInLastBlock, blockNumber, listMeta, invertedList): this function retrieves the decoded version of an id block given the block number, the inverted list, and some info about the inverted list.

retrieveFreqBlock(numBlocks, docsInLastBlock, blockNumber, listMeta, invertedList): this function retrieves the decoded version of a frequency block given the block number, the inverted list, and some info about the inverted list.

Design decisions and limitations

- At the start of query processing, the Lexicon and DocMeta files are parsed and stored in memory. This allows faster query processing as the program won't have to reaccess and read from those files. However, this might not work for extremely large data sets where the Lexicon and DocMeta files are too large to fit in memory.
- The candidate and final result generation for conjunctive and disjunctive queries use different methods. Document-at-a-time query processing is used for conjunctive queries because it is more efficient. Using document-at-a-time query processing, the program won't have to decompress unnecessary frequency blocks that do not contribute to the queries. However, for disjunctive queries, term-at-a-time query processing is used because document-at-a-time query processing will be slower without an early termination mechanism, such as max-score, which requires additional analysis.
- BM25 is used as the ranking function for query results. However, in a real-world search engine, BM25 is often not enough to determine whether a query result is good or not. As the score is only based on the frequency of query terms in the entire data set and each document, web pages are able to manipulate the score by including more keywords inside their page, even if they do not make sense. Also, the context in which the words are found is not taken into account in the BM25 score calculation, which might be important. For example, if the query terms are used at the beginning of the main text region of a web page or appear in the title or URL of a web page, this might signify these terms are important to the web page, and the web page might be more relevant to the query than another web page that contains the query terms with the same frequency but in less important parts of the page. To better evaluate query results, real search engines might consider the page rank of web pages and use machine learning and other methods to rank the results.
- Snippet generation for both conjunctive and disjunctive queries finds the first occurrence of a query term inside the document and returns the line that it is in and the lines that come after it. The snippet generation is based on the assumption that documents often explain words and the relation of the words relative to the document when they are first used. In conjunctive queries, the program returns a snippet that contains at least two of the query terms, if there is more than one term in the query. This is because, in conjunctive queries, the user might be interested in what the terms mean when they are

used together, which makes sense for the snippet to contain at least two of the query terms. In disjunctive queries, whether or not a document contains every term in the query is not as important, so returning the text around one query term might be sufficient. However, these are based on assumptions made about how terms appear in documents and what users want when they enter a query. More analysis could be done to generate better snippets. Also, for each document in the final results, five lines from the original document were selected for the snippet. This results in snippets of different lengths. One way to solve this problem is to create snippets of a fixed window size instead of a fixed number of lines. Instead of selecting the text around the first occurrence of a query term, we can also generate many candidate snippets and evaluate the quality of each snippet by the number of (distinct) query terms it contains. However, this will also require more computation time.

- Each query is treated separately in this program. Unlike search engines that are used in the real world, no caching is done and information about users' search history is not kept. This makes the program simpler but less efficient.

Conclusion

In summary, creating a search engine is a complex task involving careful consideration of efficiency and various nuances in query processing. Designing an effective index structure requires balancing different factors to handle large amounts of data while ensuring quick retrieval. Additionally, the choice of query parsing and ranking functions may also significantly impact search results.