Information Retrieval - Assignment 2

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

MESTRADO INTEGRADO EM ENG. DE COMPUTADORES E TELEMÁTICA

ANO 2018/2019

Beatriz Coronha 92210

Pedro Santos 76532

1. Introduction

This report will explain the development of the first two exercises of the assignment number two. In this assignment was extended the indexer to apply tf-idf weights to terms using the improved tokenizer and also it was created a class that implements a ranked retrieval method.

2. Class Diagram

The class diagram for the project is presented below (Figure 1) and it is in the folder of the assignment to be easier to analyse it.

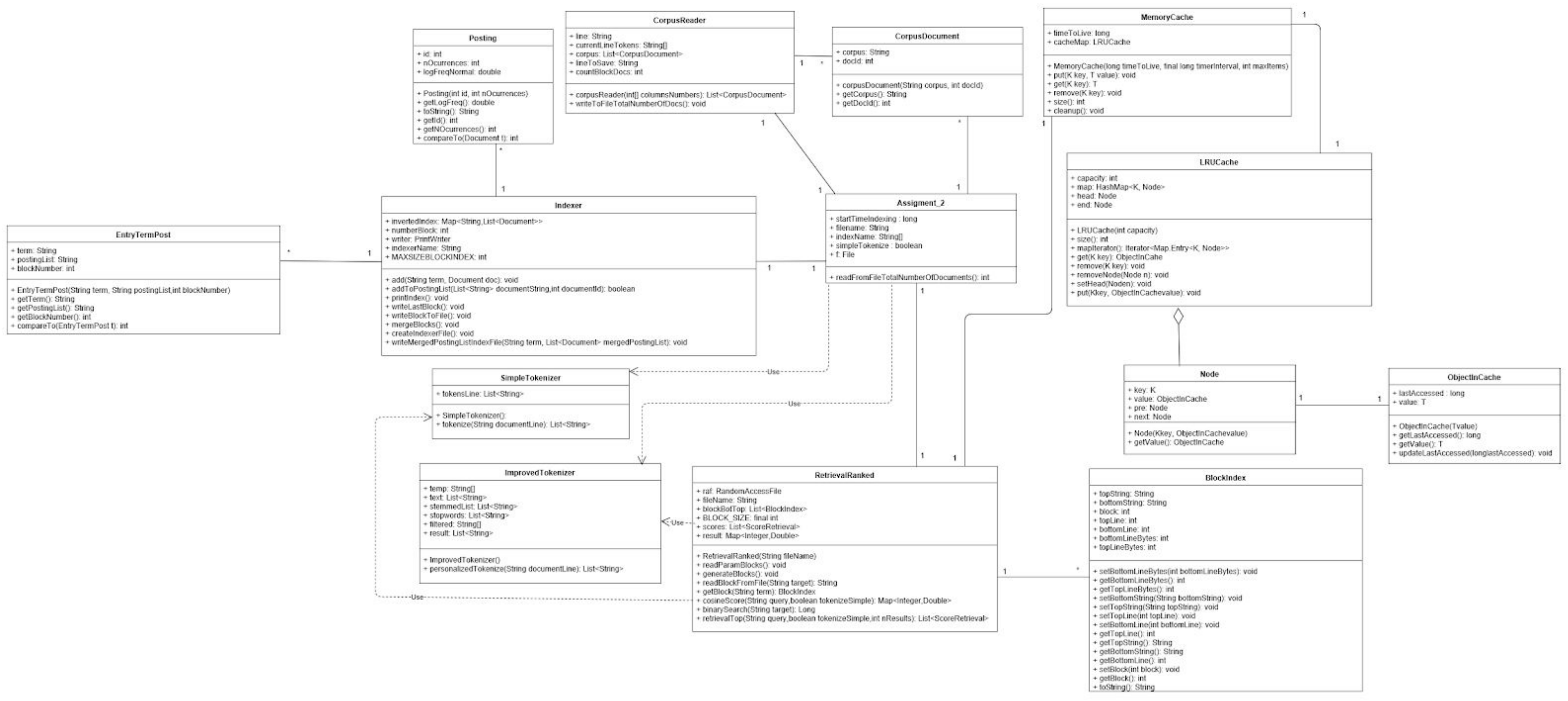
Posting
+ id: int
+ nOcurrences: int
+ logFreqNormal: double
+ Posting(int id, int nOcurrences)
+ getLogFreq(): double
+ toString(): String
+ getId(): int
+ getNOcurrences(): int
+ compareTo(Document t): int
CorpusReader
+ line: String
+ currentLineTokens: String[]
+ corpus: List<CorpusDocument>
+ lineToSave: String
+ countBlockDocs: int
+ corpusReader(int[] columnsNumbers): List<CorpusDocument>
+ writeToFileTotalNumberOfDocs(): void
CorpusDocument
+ corpus: String
+ docId: int
+ corpusDocument(String corpus, int docId)
+ getCorpus(): String
+ getDocId(): int
MemoryCache
+ timeToLive: long
+ cacheMap: LRUCache
+ MemoryCache(long timeToLive, final long timerInterval, int maxItems)
+ put(K key, T value): void
+ get(K key): T
+ remove(K key): void
+ size(): int
+ cleanup(): void
LRUCache
+ capacity: int
+ map: HashMap<K, Node>
+ head: Node
+ end: Node
+ LRUCache(int capacity)
+ size(): int
+ mapIterator(): Iterator<Map.Entry<K, Node>>
+ get(K key): ObjectInCahe
+ remove(K key): void
+ removeNode(Node n): void
+ setHead(Noden): void
+ put(Kkey, ObjectInCachevalue): void
EntryTermPost
+ term: String
+ postingList: String
+ blockNumber: int
+ EntryTermPost(String term, String postingList,int blockNumber)
+ getTerm(): String
+ getPostingList(): String
+ getBlockNumber(): int
+ compareTo(EntryTermPost t): int
Indexer
+ invertedIndex: Map<String,List<Document>>
+ numberBlock: int
+ writer: PrintWriter
+ indexerName: String
+ MAXSIZEBLOCKINDEX: int
+ add(String term, Document doc): void
+ addToPostingList(List<String> documentString,int documentId): boolean
+ printIndex(): void
+ writeLastBlock(): void
+ writeBlockToFile(): void
+ mergeBlocks(): void
+ createIndexerFile(): void
+ writeMergedPostingListIndexFile(String term, List<Document> mergedPostingList): void
Assigment_2
+ startTimeIndexing : long
+ filename: String
+ indexName: String[]
+ simpleTokenize : boolean
+ f: File
+ readFromFileTotalNumberOfDocuments(): int
SimpleTokenizer
+ tokensLine: List<String>
+ SimpleTokenizer():
+ tokenize(String documentLine): List<String>
ImprovedTokenizer
+ temp: String[]
+ text: List<String>
+ stemmedList: List<String>
+ stopwords: List<String>
+ filtered: String[]
+ result: List<String>
+ ImprovedTokenizer()
+ personalizedTokenize(String documentLine): List<String>
RetrievalRanked
+ raf: RandomAccessFile
+ fileName: String
+ blockBotTop: List<BlockIndex>
+ BLOCK_SIZE: final int
+ scores: List<ScoreRetrieval>
+ result: Map<Integer,Double>
+ RetrievalRanked(String fileName)
+ readParamBlocks(): void
+ generateBlocks(): void
+ readBlockFromFile(String target): String
+ getBlock(String term): BlockIndex
+ cosineScore(String query,boolean tokenizeSimple): Map<Integer,Double>
+ binarySearch(String target): Long
+ retrievalTop(String query,boolean tokenizeSimple,int nResults): List<ScoreRetrieval>
Node
+ key: K
+ value: ObjectInCache
+ pre: Node
+ next: Node
+ Node(Kkey, ObjectInCachevalue)
+ getValue(): ObjectInCache
ObjectInCache
+ lastAccessed : long
+ value: T
+ ObjectInCache(Tvalue)
+ getLastAccessed(): long
+ getValue(): T
+ updateLastAccessed(longlastAccessed): void
BlockIndex
+ topString: String
+ bottomString: String
+ block: int
+ topLine: int
+ bottomLine: int
+ bottomLineBytes: int
+ topLineBytes: int
+ setBottomLineBytes(int bottomLineBytes): void
+ getBottomLineBytes(): int
+ getTopLineBytes(): int
+ setBottomString(String bottomString): void
+ setTopString(String topString): void
+ setTopLine(int topLine): void
+ setBottomLine(int bottomLine): void
+ getTopLine(): int
+ getTopString(): String
+ getBottomString(): String
+ getBottomLine(): int
+ setBlock(int block): void
+ getBlock(): int
+ toString(): String
Use
Use
Use
Use
1
*

Figure 1: Class Diagram

3. Description of Class/Methods

This section will explain the goal of each class, the methods in them and how they're connected. The classes that were created for this assignment will be fully explained and the changes made to the class from the assignment number one will be explained.

a. Assigment_2.java

This is the main class of the program. Like in the previous assignment there is a variable filename where it is put the path of the file that has the documents to index and a boolean variable to choose if it is to use the simple tokenizer or the improved one. It was added the code to verify if the indexer is created. If the indexer is not created it will be created like explained in the previous report, however now it will apply tf-idf weights to terms. If the indexer is already created it will skip that part and the next step will be the part of ranked retrieval method of the indexer. In this part the configurations that could be changed are the size of the cache that will be used, the time interval that the cache will verify for objects that are in the cache for more than a certain time and it is also possible to define that. If the objects are in the cache more than the define time, they will be erased. So, in this part it is instantiated the class RetrievalRanked and after the instantiation the function to generate the sub blocks of the indexer will be called, after that the retrievals can be made using the function *cosineScore* or the function *retrievalTop*. The function cosineScore will return all the results in an hashmap. The function retrievalTop will return the results in an array and it will be sorted by the high score, also this function receives an argument to define the maximum number of results.

b. CorpusReader.java

This class has only a difference from the last assignment. Now, it has a function that will be called after reading all the documents and this function will write in a file named *indexer_number_of_docs.txt* the total number of documents. This file is read in a function in the main class and the returned value will be used to instantiate the RetrivalRanked class.

c. Indexer.java

The goal of the first exercise on this assignment was to use tf-idf weights to terms. So, to accomplish this task in the function that adds to the indexer the term and the correspondent posting list it is necessary to do some extra operations. In the assignment number one, was only necessary to count the times that each term appears in the document and them add that information to the indexer. Now, it was necessary to count also the number of times that the term appears in the document. Then, it is used that result to calculate the Log-frequency weighting for each term that has a different count. Doing like this it is not necessary perform operations that we already know the result. So, basically it is saved in a hashmap the result where each count will have an result associated that corresponds to the 1+log10(tf). The next step is to calculate the value that will be used to normalize the results. To do that it is calculated the square root of the sum of the squares of the Log-frequency weight of each term in the document. The last step is to calculated the normalized values. Here it is done the same thing to not repeat operations and a lot of the values are the same, so this will optimize significantly the calculation. Therefore, basically for each different count will be calculated the division between the Log-frequency weight and the normalization value. After that the process is the same, however it is saved the tf-idf weights and not only the frequency of each term. So, for accomplish beyond this changes were made some other minor things, such as the Posting class was changed to have an attribute to save the weight.

d. Posting.java

To this class was added the attribute to save the normalized value of log frequency and the function to get the respective value. Also, it was added a different constructor to be able to instantiate an object using the document id and the weight. So, the changes made didn't compromise the use of this class in older versions of the project.

e. BlockIndex.java

This class is used for a more efficient search, the general idea is that the indexer is divided in blocks and this class is used to keep the number and the first and last word of it.

To do that, the indexer is read line by line and copied to another file that will be the block, this files are identified by a number, the function responsible for doing that is *generateBlocks*. After the blocks are generated, the objects of "blockIndex" are saved in an array, this objects will have the number of the block and its first and last words. That way, when a search is been made and the term is not in cache it's possible to know in which block the term is and only this block will be read to search for the query term.

f. RetrievalRanked.java

This is the class that will actually do the retrieval and the block operations mentioned previously. This class reads the indexer and then does the *generateBlocks* and the searches for the term and then calculates the scores.

This class has an constructor that accepts the name of the indexer, the total number of docs that were read to index the terms and the three settings for the cache that already were explained in the final of the section of the class assigment_2.java. So, the constructor will instantiate the cache. After that the main class calls the function *generateBlocks*. This function will check if the sub blocks of the indexer were created. If the blocks were not created the function will create the blocks, like was described in the previous class BlockIndex. Then, after generating all the blocks it will serialize using the function *serializeBlocksArray* the array that has the collection of objects BlockIndex. By doing this is not necessary to read the blocks again to know the first and last word that each block contains. Therefore, if the blocks were already created this function will call *deserializeBlocksArray function* and it will deserialize the array that contains the objects BlockIndex. The main function of this class is *cosineScore.* This function has two arguments and it is the method to make the ranked retrieval, so it will return a hashmap where the key is the id of the document and the value the score. The first argument that the function has is the query and the second argument is the boolean value to such the tokenizer that it is supposed to use. The first step is to tokenize the query. Then, for each term of the query will be seen if it is in cache the value. If the value is in cache it is received immediately the posting list of that term. Otherwise, it will call the function *readBlockFromFile.* This function will call the function *getBlock* that will return the object BlockIndex that belongs to the respective term. Therefore, the function *readBlockFromFile* will read line by line until find the term and then it will return the line. The next step in the cosineScore function is to add the received line to the cache and then for it will calculate the score of each document. Then, it was create a extra function called retrievalTop that has the same arguments of the cosineScore and the maximum numbers of results. This function will call the cosineScore and it will order the results by the score and it will return a maximum of results.

g. ObjectInCache.java

The cache is generic, so it was not only developed to this project. This class is the object that will be each entry of the cache. This object has two attributes. One attribute is the value of the last time that was accessed and the other is the object to save, in our case will be a string.

h. LRUCache.java

A LRU cache is is a hash table of keys and double linked nodes. This class has the methods to put, remove and get elements in the cache. The cache will have a hashmap where the key will be associated to Node. Also, it will save the head and the end of the added nodes to the cache. This Class Node has three attributes, one is the object of the class ObjectInCache and the others two are the before and next node. In this way it is very fast to remove the older elements in the cache.

i. MemoryCache.java

This class is the “top level” of the cache and it is the class instantiated by the RetrievalRanked class. The constructor of this class receives three arguments. One of them is the size of the cache and this will be used to instantiate the LRUCache. The other one is the timer interval to use to sleep the thread that will clean the memory by calling the function *cleanup*. Then, this function will check for all the objects in memory that are in the memory more time than supposed. This time is defined by the first argument in the constructor. Then, the class has the functions to remove, add and get from the LRU Cache. So, It was created a cache that has a limit of objects that can save, if it is full and it is necessary to put more objects will delete the older ones and it will preserve in the cache the most used values and it will delete the values that are not usually used.

j. ScoreRetrieval.java

This class was created to use to save the results of the scores of each document in a list and them order the results. So, the class implements the Comparable interface to be possible to order the elements easily and it has two attributes the id of the document and the score.

4. Data Flow

The data flow for this project is presented below (Figure 2). In it is possible to see the most important steps of the code, first there is the “Initialization”, where the “CorpusReader” and the “Indexer” are initialized. Then, the “Processing” happens, where the Indexer is created, as explained in the last assignment. After this part, it is necessary to merge all the different blocks, calculate the score and then return the rank. Firstly, it is verified if the indexer is created. If the indexer is not created the CorpusReader reads 10000 documents and then the tokenize will process that documents and the tokens will be consumed by the indexer. Then, another 10000 documents will be processed and this process happens until there are no more documents. After this part, it is necessary to merge all the different blocks and create the final indexer. If the indexer is already created the RetrievalRanked will instantiated and the function generateBlocks will be called. If the blocks are already created it is only necessary to deserialize the array that contains the information, like was explained before. If the blocks are not create, the blocks will be created using the indexer and the array that saves the information will be serialized to not be necessary read all the times the blocks. Then, it is possible to do the retrievals. It is used the function cosineScore to receive all the results in a hashmap or the function retrievalTop to receive the results in an ordered list and it is possible to choose the maximum number of elements that will be returned. When the search for the term is being made before searching the terms in the block, it will be checked if the term is in the memory cache.

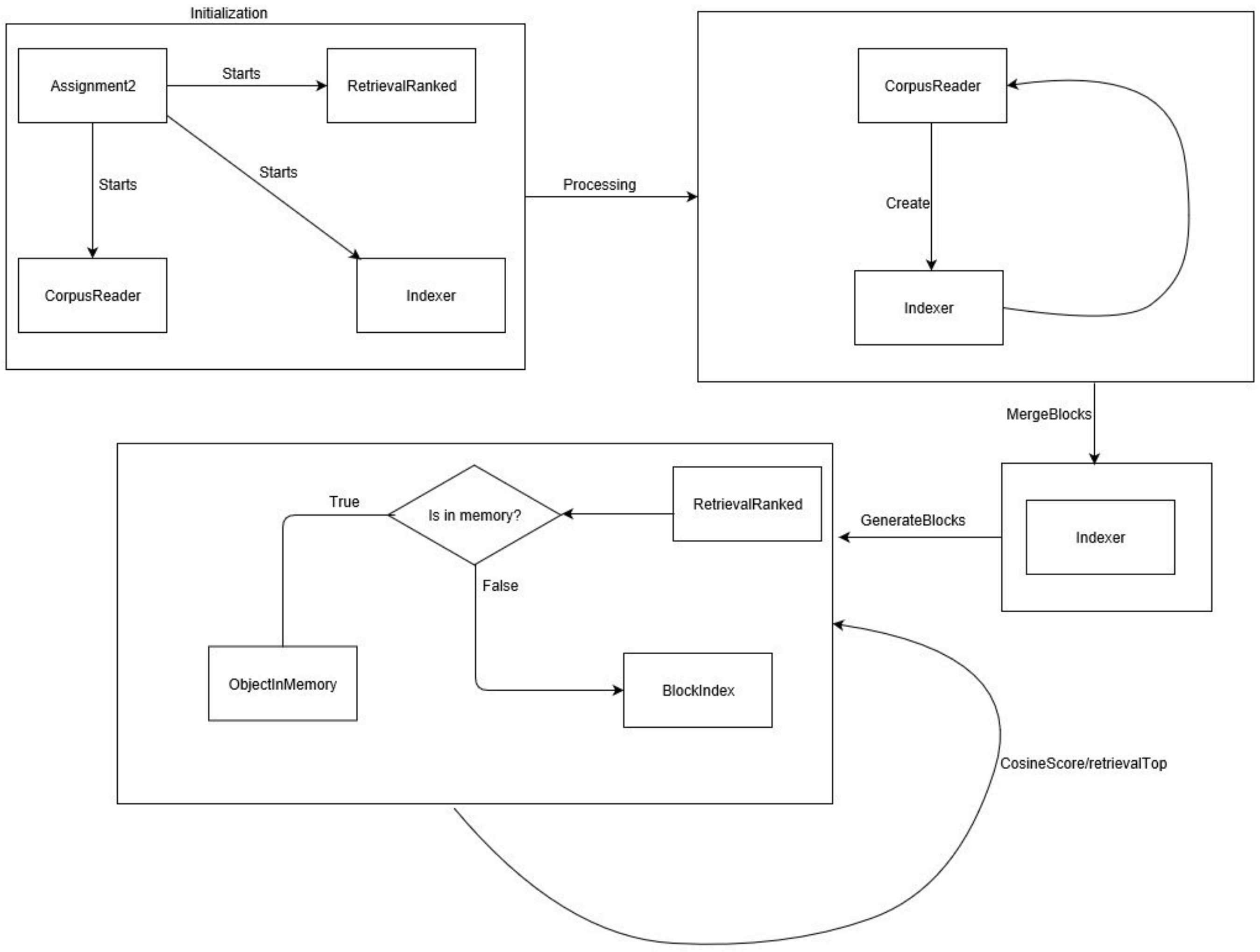

Figure 2: Data Flow

5. Results/Tests

a. Indexing Results

Total Indexing time = 33:37.771s

Maximum memory used in Indexing = 1249738832 bytes ~ 1.16 Gb

Total Indexer Size on disk = 3980859642 bytes ~ 4.00 Gb

b. Retrieval Results

To test the search of one term it was done 21 tests using 21 different random terms from the indexer, so the cache will not have any of the terms and it will be necessary to search on the index. The average time for each retrieval was 178 milliseconds. If all the terms were in cache the average time is 108 milliseconds. One thing to have in mind is if we take of the prints of the results the time without any term in the cache is 57 milliseconds and the average time when the terms are in cache is 14 milliseconds. It is possible to conclude that the retrieval is so fast that the prints increase a lot the time, also because we are printing all the results and some terms appear in a lot of documents. In all of the next tests the time of the prints will not be considered.

Then, it was done the test with query of 3 terms. It was tested with 8 different queries and all the terms were different, so the cache will be empty and all the terms belongs to indexers. The average time without prints was of 189 milliseconds. If all the terms are in cache the average time decreases to 45 milliseconds.

The last test was done with queries of 5 terms that are in the indexer and the total number of tests were 4. The average time with no terms in the cache was 315 milliseconds. If all the terms are in cache the average time was 78 milliseconds. In this test also all the terms were different and they are in the indexer.

6. How to Run

There aren't more dependencies or requirements in relation to the assignment number one. Like before, It is only necessary to install the porter stemmer and in the main class define the file that has the documents to be read.

7. Conclusion

To have this performance and also to achieve this solution it were made a lot of other solutions, however not so good in performance. The first approach was to do binary search to the indexer, to take advantage of the fact that all the terms are sorted, but the indexer was too big to do that with good performance. So, the next test was to not do binary search in all the document, but to make a pre processing to know where are located the terms that begin with a given letter and then, it was taken the first letter of the term that it was to search in the indexer and it would be done the binary search only in that block. After this test, it was noticed that they are for example a lot more terms starting with an ‘a’ than with a ‘z’, so the time to retrieval was not consistent and to search a term starting with ‘a’ was much more slower than a letter by ‘z’. So, the next approach was to divide the indexer in blocks and then do the binary search on that block. This solution was the best until now, but it was not good enough, because to do the binary search it is necessary to use the random access file and this is quite slow. Only after so many attempts it was found the solution that would have a very good performance with low memory use and with the implementation of the cache the performance increased even more.

8. Bibliography

The cache was developed with the support of this tutorial
https://crunchify.com/how-to-create-a-simple-in-memory-cache-in-java-lightweight-cache/
and this page
https://www.programcreek.com/2013/03/leetcode-lru-cache-java/