

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών
ΗΥ463 Συστήματα Ανάκτησης Πληροφοριών
Εξάμηνο: Άνοιξη 2022

Γραπτή Αναφορά Έργου

BioMedicEngine Phase A

Στοιχεία Φοιτητών

Όνοματωπόνυμο	Manos Chatzakis
AM	4238
Email	csd4238@csd.uoc.gr

1 Introduction

This project is the implementation of a BioMedical Search Engine over a biomedical document collection of 5GB.

How to run:

It is mandatory to execute the **exejar** file, located at **target/** folder.

- **Indexer:** An example is
java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar -mode indexer -input ../sample/ -output /mnt/c/Users/manos/Desktop/simple_example -gr ../stopwords/stopwordsGr.txt -en ../stopwords/stopwordsEn.txt

```
C:\Users\manos\Desktop> java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar -mode indexer -input ../sample/ -output /mnt/c/Users/manos/Desktop/simple_example -gr ../stopwords/stopwordsGr.txt -en ../stopwords/stopwordsEn.txt
>>BioMedic Indexer started creating the partial files
```

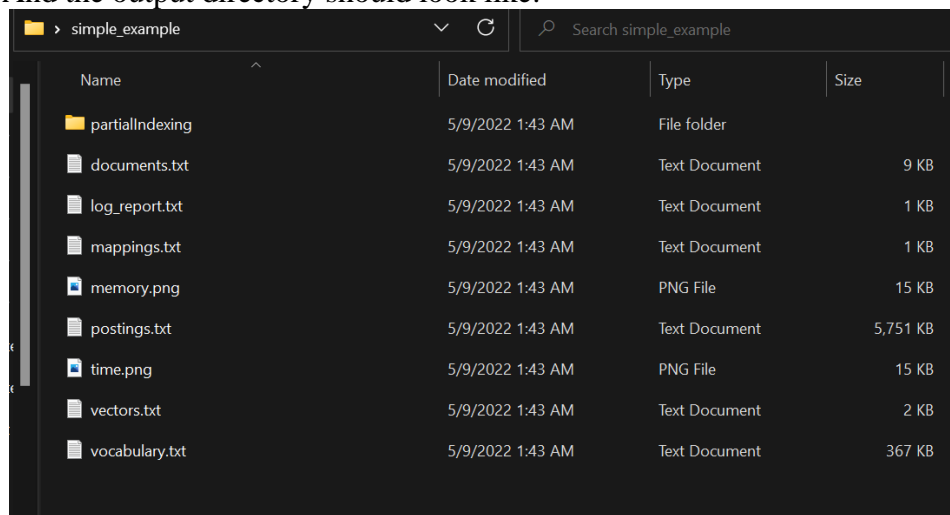
Important Note 1: It is mandatory that the output directory has a folder named: “partialIndexing/”, in which the partial files will be stored.

Important Note 2: Always put “/” at the end of the directories.

The output of the program should look like:

```
>>BioMedic Indexer started creating the partial files
>>BioMedic Indexer created the partial files. Proceeding to merging phase.
>>Merging pair /mnt/c/Users/manos/Desktop/simple_example/partialIndexing/vocab0.txt and /mnt/c/Users/manos/Desktop/simple_example/partialIndexing/vocab1.txt
===== BioMedic Indexer Results =====
Threshold of terms: 15000
Directory of documents indexed: ../sample/
Directory of indexer output: /mnt/c/Users/manos/Desktop/simple_example/
Total Documents Indexed: 54
Time Elapsed for Partitioning Phase (seconds): 13.491488
Time Elapsed for Merging Phase (seconds): 1.6927992
Time Elapsed for Norm Calculation phase (seconds): 1.2248832
Total Time Elapsed (seconds): 16.40966025
=====
```

And the output directory should look like:



Name	Date modified	Type	Size
partialIndexing	5/9/2022 1:43 AM	File folder	
documents.txt	5/9/2022 1:43 AM	Text Document	9 KB
log_report.txt	5/9/2022 1:43 AM	Text Document	1 KB
mappings.txt	5/9/2022 1:43 AM	Text Document	1 KB
memory.png	5/9/2022 1:43 AM	PNG File	15 KB
postings.txt	5/9/2022 1:43 AM	Text Document	5,751 KB
time.png	5/9/2022 1:43 AM	PNG File	15 KB
vectors.txt	5/9/2022 1:43 AM	Text Document	2 KB
vocabulary.txt	5/9/2022 1:43 AM	Text Document	367 KB

- **Retriever:** BioMedicRetriever should run from CLI using
java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar -mode retriever -collection /mnt/c/Users/manos/Desktop/collectionIndex/

for the simple retriever which uses the vector model and using

java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar -mode topicRetriever -collection /mnt/c/Users/manos/Desktop/collectionIndex/

The output of the program should look like this:

```

chatzakis@DESKTOP-C9U6MUG:/mnt/c/Users/manos/Documents/GitHub/BioMedicEngine/BioMedicEngine/target$ java -jar BioMedicEngine
-1.0-SNAPSHOT-exejar.jar -mode retriever -collection /mnt/c/Users/manos/Desktop/collectionIndex/
>>Total terms loaded: 3027032
>>> Type a query

```

After this, you can start querying the collection. The initialization of the tool might require some time.

Example:

```

chatzakis@DESKTOP-C9U6MUG:/mnt/c/Users/manos/Documents/GitHub/BioMedicEngine/BioMedicEngine/target$ java -jar BioMedicEngine-1.0-SNAP
SHOT-exejar.jar -mode retriever -collection /mnt/c/Users/manos/Desktop/newCollection/
>>Total terms loaded: 3027032
>>> Type a query
help
{91471,C:\MedicalCollection\27\2697138.xml} score: 0.011639375083962218
{91474,C:\MedicalCollection\27\2697141.xml} score: 0.011051521474260775
{91476,C:\MedicalCollection\27\2697143.xml} score: 0.010358743288986484
{91494,C:\MedicalCollection\27\2697161.xml} score: 0.00954185659160358
{91472,C:\MedicalCollection\27\2697139.xml} score: 0.006926535930435605
{91456,C:\MedicalCollection\27\2697018.xml} score: 0.005591830967790626
{91496,C:\MedicalCollection\27\2697163.xml} score: 0.005363657083666051
{91451,C:\MedicalCollection\27\2696939.xml} score: 0.004913221708170295
{91486,C:\MedicalCollection\27\2697153.xml} score: 0.004843079575481763
{91468,C:\MedicalCollection\27\2697135.xml} score: 0.003911882709974891
{91499,C:\MedicalCollection\27\2697166.xml} score: 0.0029349943920014845
{91500,C:\MedicalCollection\27\2697170.xml} score: 0.0026738537764491466
Response Time: 0.1234199
>>> Type a query

```

```

>>> Type a query
medical examination
{91488,C:\MedicalCollection\27\2697155.xml} score: 0.30104223291052123
{91463,C:\MedicalCollection\27\2697058.xml} score: 0.28967830801249217
{91491,C:\MedicalCollection\27\2697158.xml} score: 0.25370247724183126
{91479,C:\MedicalCollection\27\2697146.xml} score: 0.10454594058783535
{91462,C:\MedicalCollection\27\2697057.xml} score: 0.055499593441249974
{91490,C:\MedicalCollection\27\2697157.xml} score: 0.04730279877153752
{91469,C:\MedicalCollection\27\2697136.xml} score: 0.046089989977593636
{91474,C:\MedicalCollection\27\2697141.xml} score: 0.04154358245307059
{91493,C:\MedicalCollection\27\2697160.xml} score: 0.03329790362870334
{91498,C:\MedicalCollection\27\2697165.xml} score: 0.031291621087382746
{91452,C:\MedicalCollection\27\2696940.xml} score: 0.02825536377117772
{91497,C:\MedicalCollection\27\2697164.xml} score: 0.026782327245678177
{91467,C:\MedicalCollection\27\2697134.xml} score: 0.02659995250539185
{91461,C:\MedicalCollection\27\2697056.xml} score: 0.019953148430708965
{91487,C:\MedicalCollection\27\2697154.xml} score: 0.0189937463681741
{91486,C:\MedicalCollection\27\2697153.xml} score: 0.018405141459545474
{91457,C:\MedicalCollection\27\2697027.xml} score: 0.017144743966430197
{91464,C:\MedicalCollection\27\2697115.xml} score: 0.01606394294606189
{91473,C:\MedicalCollection\27\2697140.xml} score: 0.014221187895066462
{91466,C:\MedicalCollection\27\2697133.xml} score: 0.012541826877474462
{91489,C:\MedicalCollection\27\2697156.xml} score: 0.011328746207359396
{91475,C:\MedicalCollection\27\2697142.xml} score: 0.010294034505854793
{91459,C:\MedicalCollection\27\2697051.xml} score: 0.009712208318868225
{91451,C:\MedicalCollection\27\2696939.xml} score: 0.00933585120289796
{91482,C:\MedicalCollection\27\2697149.xml} score: 0.00806302054851832
{91465,C:\MedicalCollection\27\2697132.xml} score: 0.007898111158393239
{91468,C:\MedicalCollection\27\2697135.xml} score: 0.007433158337386592

```

```

>>> Type a query
hypertension
{24185,C:\MedicalCollection\06\3180239.xml} score: 0.2424403408622787
Response Time: 0.0089879
>>> Type a query
|
>>> Type a query
!exit
chatzakis@DESKTOP-C9U6MUG:/mnt/c/Users/manos/Documents/GitHub/BioMedicEngine/BioMedicEngine/target$

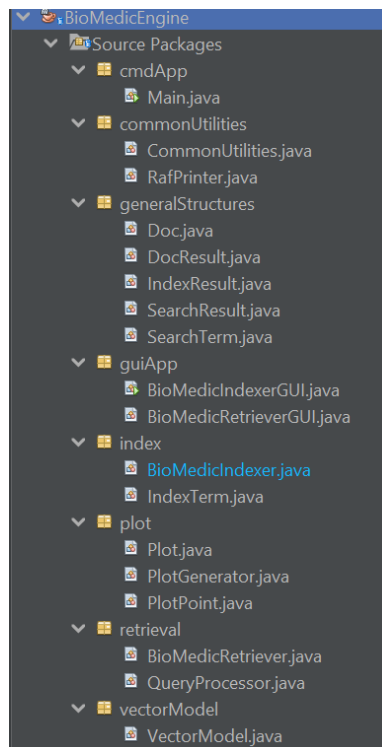
```

```

chatzakis@DESKTOP-C9U6MUG:/mnt/c/Users/manos/Documents/GitHub/BioMedicEngine/BioMedicEngine/target$ java -jar BioMedicEngine-1.0-SNAPSHOT-exejar.jar -mode topicRetriever -collection /mnt/c/Users/manos/Desktop/newCollection/
>>Total terms loaded: 3027032
>>> Type a query
58-year-old woman with hypertension and obesity presents with exercise-related episodic chest pain radiating to the back.
>>> Type a medical type (diagnosis, test, treatment)
diagnosis
{91462,C:\MedicalCollection\27\2697057.xml} score: 0.05965217193681754
{91491,C:\MedicalCollection\27\2697158.xml} score: 0.01764250490639709
{91461,C:\MedicalCollection\27\2697056.xml} score: 0.033378639751156104
{91475,C:\MedicalCollection\27\2697142.xml} score: 0.0840518662786474
{91494,C:\MedicalCollection\27\2697161.xml} score: 0.02925512949572195
{91497,C:\MedicalCollection\27\2697164.xml} score: 0.05245093178211357
{91493,C:\MedicalCollection\27\2697160.xml} score: 0.026004092975442584
{91452,C:\MedicalCollection\27\2696940.xml} score: 0.013854477907309903
{91459,C:\MedicalCollection\27\2697051.xml} score: 0.02080376563247204
{91488,C:\MedicalCollection\27\2697155.xml} score: 0.021299351868214505
{91490,C:\MedicalCollection\27\2697157.xml} score: 0.04870054919240994
Response Time: 0.204379
>>> Type a query

```

Project Layout:



About the architecture:

Package index: This package contains all classes need to Index a directory.

Package retrieval: This package contains all classes needed to perfrom query answering (simple or topic).

Package cmdApp: This package contains the CLI application of the BioMedicIndexer.

Package generalStructures: This package contains the general structures needed by (most of) all other packages, such as “Document” etc.

Package guiApp: This package contains the GUI applications (not finalized).

Package plot: This package contains methods to create the plots (used in the report of the index results).

Package vectorModel: This package contains all methods needed to calculate norms, TF*iDF arrays etc.

Package commonUtilities: All utility functions.

2 Implementation

In this section, the basic methods used to implemented both parts of BioMedic Indexer (Index Creation and Query Answering are described).

2.1 Index Creation

BioMedic Indexer index a selected directory in three steps: (a) Partial Indexing (b) Partial Merging and (c) Document Norm Calculation.

Partial Indexing.

BioMedic Indexer uses a sorted <String, Term> map to store the terms. During the first phase of indexing, the terms and related information are stored into this map. This map stores their df, their occurrences per tag etc. The document collection is read sequentially and the contents of each document are added to the map.

When the size of this map gets greater than a threshold TH, the contents of the map are flushed to the disk, creating a pair of partial files, with names “vocabX” and “postX”. These files are stored in a list. Every time these files are flushed to the disc, this list is cleared.

BioMedic Indexer uses a relatively small TH, to be able to run in systems with small memory capacity.

Merging.

After the partial indexing phase is completed, the files need to be merged. To merge the files, the tool removes two files from the list, and adds the output of merging to the list, till the list size is 1.

This list contains the names of the partial vocabularies files, and the corresponding posting file is found by replacing the part “vocab” with “post”. This is why the directory path should not contain words such as “vocab” and “post” to avoid such mistakes.

Then the remaining file is the vocabulary file, and the corresponding posting is the posting file.

All of this files are maintained, traversed etc. using the Java RandomAccessFile API.

Merging Algorithm (Like merging two linked lists😊): We create the new posting and vocab file keeping the the lexicographical order the same. For every term added to the file, we should also merge their postings, in an ordered way using the document IDs, as seen in the algorithms above.

```
int n_counter = 0;
while (vocabFileNames.size() > 1) {
    String vocab1 = partialFilesDirectory + vocabFileNames.remove(index: 0);
    String vocab2 = partialFilesDirectory + vocabFileNames.remove(index: 0);

    mergePartialFiles(vocabFileNames, vocab1, vocab2, dir: partialFilesDirectory, n_counter++);
}
```

```
String lineV1 = voc1.readUTF(), lineV2 = voc2.readUTF();
while (!lineV1.equals(anObject: "#end") && !lineV2.equals(anObject: "#end")) {

    String[] contentsV1 = lineV1.split(regex: " ");
    String[] contentsV2 = lineV2.split(regex: " ");

    String currentTermV1 = contentsV1[0];
    String currentTermV2 = contentsV2[0];

    int comp = currentTermV1.compareTo(anotherString: currentTermV2);
    if (comp == 0) {
        mergeContentsAndCopyToRAF(contentsV1, contentsV2, pos1: post1, pos2: post2, vocNew: newVoc, postNew: newPost);
        lineV1 = voc1.readUTF();
        lineV2 = voc2.readUTF();
    } else if (comp > 0) {
        copyContentsToRAF(vcontents: contentsV1, pos1: post1, vocNew: newVoc, postNew: newPost);
        lineV1 = voc1.readUTF();
    } else {
        copyContentsToRAF(vcontents: contentsV2, pos1: post2, vocNew: newVoc, postNew: newPost);
        lineV2 = voc2.readUTF();
    }
}
```

```

while (!lineV1.equals( anObject: "#end")) {
    String[] contentsV1 = lineV1.split( regex: " ");
    copyContentsToRAF( vcontents: contentsV1, pos1: post1, vocNew: newVoc, postNew: newPost);
    lineV1 = voc1.readUTF();
}

while (!lineV2.equals( anObject: "#end")) {
    String[] contentsV2 = lineV2.split( regex: " ");
    copyContentsToRAF( vcontents: contentsV2, pos1: post2, vocNew: newVoc, postNew: newPost);
    lineV2 = voc2.readUTF();
}

```

Document Norm Calculation.

The norms are calculated in a different file and stored separately. After the completion of the partitioning and merging, we initialize the vocabulary and we keep a map <Integer,Double> which stores the mappings of the document ID with it's norm. We traverse the terms one time and if a document contains the term we add $(TF*IDF)^2$ to the total current norm.

After the traversal, we write the $\sqrt{\text{map}}$ in a new random access file called "norms" and we save the mappings of document IDs and the map pointers.

2.2 Query Answering

Here, the process of query answering is described.

Vector model.

[Step 1 – Initializing BioMedic Retriever]: Given a directory to index, the vocabulary is initialized and kept in memory, while we also load the pointers to the Random Access Files.

[Step 2 – Getting the relevant documents]: Given a query, the query processor parses the query using “ ” and finds its terms. Then, we traverse the terms one by one, and for every term present in the vocabulary, we traverse its postings and retrieve the documents in a list. This list contains the relevant documents.

[Step 3 – Finding the norm of the vector]: The query processor not only parses the query to its terms, but it returns a map of <Term, TF>. Thus, using the TF of the term inside the query and the iDF as it comes from the model, we can calculate the norm of the query the same way we did for the documents. Indeed, terms that are not present in the vocabulary are removed.

[Step 4 – Find the dot product per vector]: For every relevant document, we do the following. For every term in the query, we have the queryTF and iDF. Then, we traverse the postings of this term. If the documentID is found, we return the corresponding TF, and so, we calculate the dot product as the sum of $(\text{queryTF} * \text{iDF}) * (\text{docTF} * \text{iDF})$ of every term.

[Step 5 – Find the score of document]: Given that we have the dot product and the norms available, the score is $(\text{dot product}) / (\text{docNorm} * \text{queryNorm})$.

[Step 6 – Return the results]: The documents are stored in a sorted list based on their score. This list is returned, with the time needed to answer the query.

Vector model with Examination type support.

[Step 7 – Support Examination Type]: The goal of this part is to not only return the documents related to the query, but also try to return documents that also correspond to an examination type. To support this, we do the following. We retrieve the documents related to a query the same way we did previously. Then, we retrieve the documents related to a topic, i.e. using the examination type as the query. As a last step, to return the most related documents, we return the **intersection** of these two sets, using the score of the queries that

was given from the vector model. This way, the engine filters out the documents that are unrelated to the examination type, and keeps the ones that surely contain some information related to this type. Another approach could be to assign the score they had as it came from the query of the examination type.

Other methods for this problem that did not work. Generally, given that we want to assign better scores to the documents that are related to a specific examination type could be to implement a weighting method of the terms, so that that the documents that contain the word “type = {test, diagnosis, ...}” could get a better score. This method did not give proper results.

Query Processor.

The query processor is a class that parses a query, removes the stop points (eg. “.”, “,” “()” ...) and returns a map that consists of the query terms and their TF. Also, it can load files of stopwords just like the index creation, but this is optional: The BioMedicRetriever removes the terms that are not present in the vocabulary, thus, adding this files or not results in the same result.

2.3 Checking

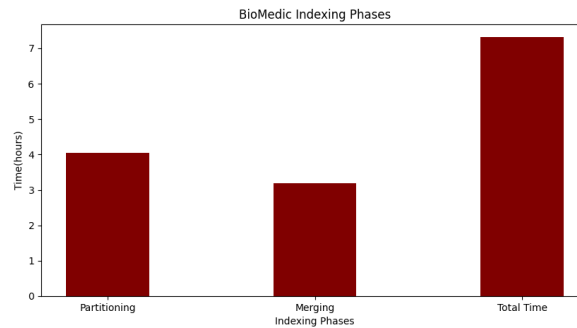
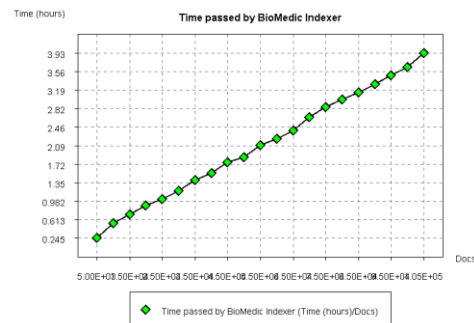
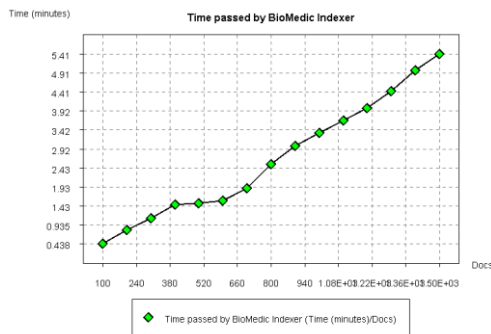
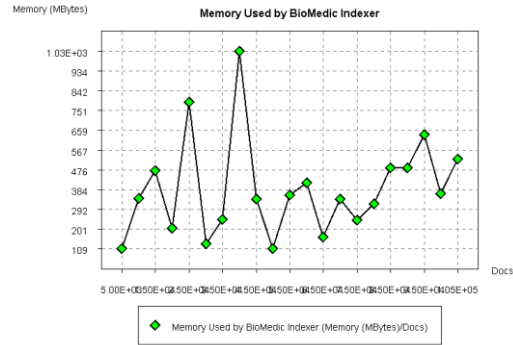
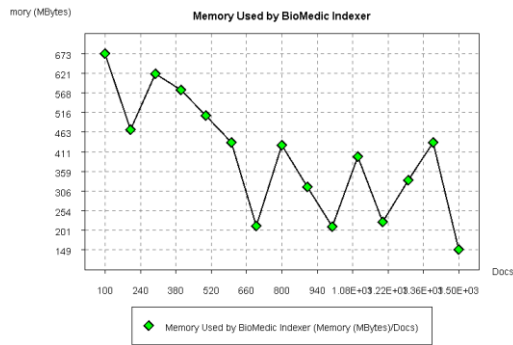
To check if the merging and qa are working fine, we did the following checks. First of all, after the indexing of the big collection, we checked if a norm is calculated for every document ID. In addition, we printed the biggest parts of the RandomAccessFiles and we checked that the contents are stored properly. Regarding query answering, we made some simple queries and we checked if the terms appear in the resulting files. We also made plenty of queries (randomly) to check that the system isn't crashing. Note that an extended evaluation of the results of this (and any) system is the topic of the next phase of this project.

3 Experimental Evaluation

The experiments contacted on a machine of 8GB Memory, 256GB SSD NVMe Disc, and a 4-Core (8 Hyperthreads) i5 8th-gen CPU, running Windows 11.

3.1 Index Creation Evaluation

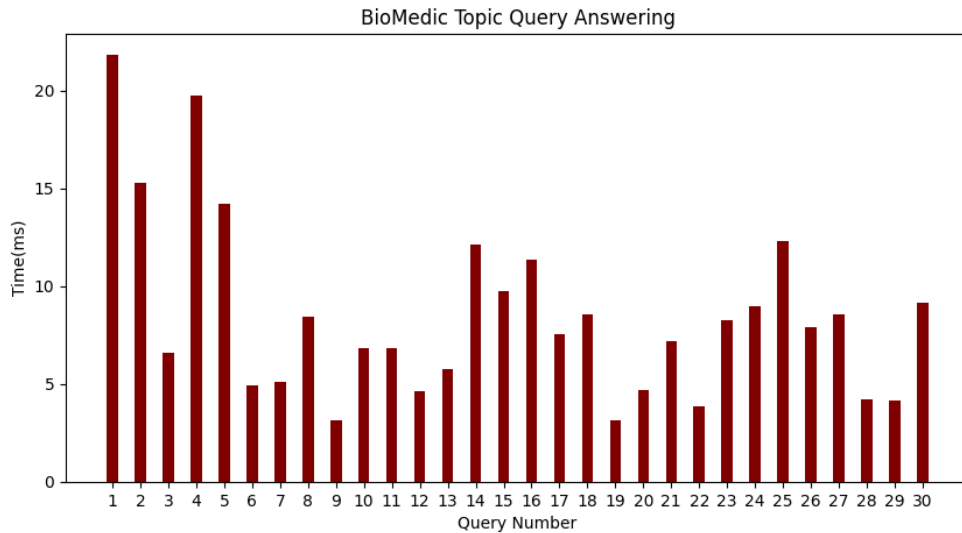
The evaluation of the indexing phase is presented in two graphs: A graph showing the memory usage correlated with the document count, and a graph showing the total time passed correlated with the document count. For both of them we show two versions: One for the whole indexing process and one for the indexing of a subset of the collection.



Note that the total time needed to index a directory is also based on the threshold we choose. Running the tool on a better system with greater threshold (~30.000) and 16GB memory with an 8-core CPU (16 Hyperthreads) the total indexing time was around 5 hours. However, choosing a smaller threshold to maintain the total memory used in small levels (see corresponding figures) could be beneficial as the program can run in any standard machine.

3.2 Query Answering Evaluation

For query answering, we show the total response time needed for the queries created from the files of “topics.txt”, using the summaries of these queries. The Results can be seen in the graph below.



Generally, an average response time of the BioMedicEngine is a value in the range of MS.

Note: For this phase we don't care/evaluate for how relevant are the returned results, we only care about the implementation of the vector model and the performance. The query quality evaluation is a part of the next phase.

This evaluation is performed automatically from the system, using statistic gathering and analysis, and can be reproduced easily.

4 Conclusion

This report is a presentation of a vector-model-based Search Engine. It contained a basic tutorial related to "how to run", a basic explanation of the architecture, a presentation of the methods that were used to implement the engine and the experimental evaluation. The next part of this project is about creating a general search engine evaluation tool, and examine the accuracy of the system presented above.