**Question1: Use the web crawler you built in Project 1 that crawled a limited space, looking for text and html files. You will probably need to modify how you saved the words from the pages that you traversed to support this query engine. Describe in detail what you changed to support the saving of words for this project.**

Project1:

Text from the pages are tokenized and stored in to the documents, tokenized words are formed by removing the space, special characters, return, tab.

words = lower\_text.split("[\\s\\t\\r.,():?\\//@~-]+");

// Pulling text only if length is > 0

**if**(text.length() > 0)

{

**if**(url.contains("txt") || url.contains("htm") || url.contains("html")){

//Logic to pull text into **new** .log text document and save it into project folder (Workspaces)

**try** {

System.setOut(**new**PrintStream(**new**FileOutputStream("Doc"+docid+".log")));

}

**catch** (FileNotFoundException e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Project2:

In this project the words that starts with a character are only stored in to the document. The words that are numbers and symbols cannot be seen in this project. We redefined the definition of the word with removal of word starting with the digit or special character.

**For**(**int** i=0;i<words.length;i++){

**if**(words[i].substring(0, 1).matches("[a-zA-Z]+")){

dict.add(words[i].toLowerCase());

System.out.println(words[i]);

}

**Question2: You will need a dictionary of words. What is your definition of “word”?**

Word:

Word is defined as a group of characters starting with an alphabet without a space, return character, tab, and special characters. To tokenize words ***split*** function is used with a regular expression. (Expression: [\\s\\t\\r.,():?\\//@~-]+")

**Will you generate the dictionary while navigating the pages or as a separate step? Explain your approach**.

Dictionary is generated while navigating through the pages according to my programming. Each page is given a separate docid while it is crawled. Here to store the dictionary a dynamic collection known as TreeSet is used. Treeset is not a fixed set of collection. Each time a document is crawled, the text in that page is retrieved to a document. The words that placed in to the Treeset are only words that starts with a character and without space or special characters. When once words are stored into Treeset, then the words are sorted and duplicates are eliminated. At each page the words are collected and stored in the dictionary without duplicates. This process repeats for every page until it crawls all pages.

**Static TreeSet<String>dict = new TreeSet<String>()**

**for**(**int** i=0;i <words.length;i++){

**if**(words[i].substring(0, 1).matches("[a-zA-Z]+")){

dict.add(words[i]);

System.out.println(words[i]);

}

}

**If needed, you can define a fixed size for the dictionary based upon the results from Project1, and allowing for a 10% increase.**

Since there is no fixed size for the dictionary, it is not necessary to increase the size by 10%.

**Question3:** **For purposes of this project, you may assume a maximum of 20 documents. You will need to create a term frequency matrix for all documents.**

Term frequency matrix is built after creation of dictionary, after completion of crawling based on last document id we are creating 2D array. In BasicCrawler.java the dictionary is created, after completion of complete crawling the dictionary is passed from BasicCrawler.java to BasicCrawlController.java. at that instant last document id is stored in variable “docid”. So based on docid and dictionary list size 2D array is created which is called by “dict\_term\_matrix”.

Initially dict\_term\_matrix is set to 0. As shown below.

//First initializing complete matrix to 0s

for(int a=0; a < dict\_list.size();a++){

for(int b = 0;b < basicCrawler.docid+1;b++){

dict\_term\_matrix[a][b] = 0;

}

}

Then first word from the dictionary is retrieved. This word is checked in the every document and the frequency of that word in each document is displayed in the term frequency matrix. For example: In our dictionary first word is ‘a’. ‘A’ is moved to term frequency matrix and then it will search how many times ‘a’ has occurred in the first document. It will count the frequency of the occurrence of ‘a’ and then it is assigned in term frequency matrix. Similarly, the frequency of occurrence of word ‘a’ in all other document is assigned in term frequency matrix. Same method is being applied to all other words.

//Getting Dict list from basic crawler

List<String> dict\_list = new ArrayList<String>(basicCrawler.dict);

List<String> doc\_list = new ArrayList<String>();

//Term Frequency Matrix

int[][] dict\_term\_matrix = new int[dict\_list.size()][basicCrawler.docid+1];

//First initializing complete matrix to 0s

for(int a=0; a < dict\_list.size();a++){

for(int b = 0;b < basicCrawler.docid+1;b++){

dict\_term\_matrix[a][b] = 0;

}

}

//For loop to get files according to docid

for(int i=1;i < basicCrawler.docid;i++){

//list which will have words from file

List<String> words\_list = new ArrayList<String>();

File f = new File("C:/Users/Madhuri/Desktop/Info\_Retrival/Info\_Project/webcrawler/Doc"+i+".log");

//to check if file exists or not

if(f.exists()){

try (BufferedReader reader = new BufferedReader(new FileReader(f))) {

String line = null;

//logic to load words into list

while ((line = reader.readLine()) != null) {

words\_list.add(line);

doc\_list.add(line);

}

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

//for loop for dict size

for(int j=0; j < dict\_list.size();j++ ){

//for loop for document words list

for(int k=0;k < words\_list.size();k++){

//comparison

if(dict\_list.get(j).equals(words\_list.get(k))){

//this counts term frequency according to word in dictionary by using how many terms are there in document

int Count = Collections.frequency(words\_list,dict\_list.get(j));

//put term frequency in matrix

dict\_term\_matrix[j][i] = Count;

}

}

// System.out.println(dict\_term\_matrix[j][i]);

}

}

else{

continue;

}

}

**Question4:** **The user should be able to enter multiple queries, consisting of one or more query words separated by space. What happens if a user enters a word that is not in the dictionary?**

Below is the code used to enter the multiple queries; this code runs until the users gives “STOP” as the input.

repeat:

**while**(**true**){

System.***out***.println("Enter our query here. When you want to stop searching please press STOP");

BufferedReader qr=**new** BufferedReader(**new** InputStreamReader(System.***in***));

String query=qr.readLine();

**if**(query.equalsIgnoreCase("STOP"))

{

System.***out***.println("Program Terminated");

**break**;

}

}

If the program reads the input as stop word then it breaks the execution and exits from the console.

When a query is entered and if the query words are present in the dictionary, then the program tells the user that the query is not in the dictionary. Along with that it also allows the user to give the input once again.

repeat:

**while**(**true**){

**for**(**int** i=0; i<query\_list.size();i++){

validcount=Collections.*frequency*(dict\_list, query\_list.get(i));

**if**(validcount==0){

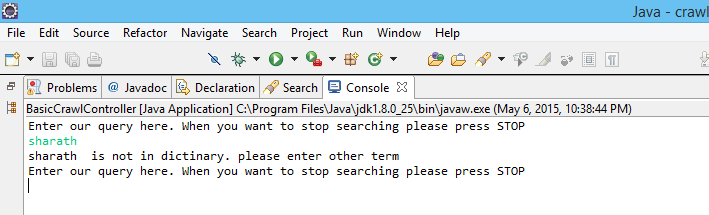
System.***out***.println(query\_list.get(i)+" is not in dictinary. please enter other term");

**continue** repeat;

}

}

Output:



**Question5:** **Compute the cosine similarity of the query against all documents. Display the measure and document URL in descending numerical order for the top N results. What value of N would you pick? Also include in the display, the first 20 words of the document. If you stemmed the words, this can be the stemmed version.**

To calculate Cosine Similarity, these are the following steps:

Calculate Document frequency (df) and idf:

**for**(**int** i=0;i<dict\_list.size();i++){

**int** doc\_count=Collections.*frequency*(doc\_list, dict\_list.get(i));

temp=BasicCrawler.*docid*/doc\_count;

idf=1+(Math.*log10*(temp));

//System.out.println("idf "+idf);

idf\_list.add(idf);

doc\_array.add(doc\_count);

}

Calculate term frequency of the query and idf:

**for**(**int** i=0;i<query\_list.size();i++){

**int** query\_count=Collections.*frequency*(query\_list, query\_list.get(i));

query\_array.add(query\_count);

qtemp=BasicCrawler.*docid*/query\_count;

// System.out.println("queryCount "+query\_count);

qidf=1+(Math.*log10*(qtemp));

qidf\_list.add(qidf);

}

Calculate tf-idf for document and query:

Document:

**for**(**int** k=0, j=0;k<idf\_list.size() && j< dict\_list.size();j++){

**for**(**int** i= 1;i<=BasicCrawler.*docid*;i++){

tf\_idf=dict\_term\_matrix[j][i]\*idf\_list.get(k);

tfidf\_list[j][i]=tf\_idf;

// System.out.println("tfidf "+tfidf\_list[j][i]);

}

}

Query:

**for**(**int** i=0;i<query\_list.size();i++)

{

qtf\_idf\_list.add(query\_array.get(i)\*qidf\_list.get(i));

qsquare.add(Math.*pow*(qtf\_idf\_list.get(i), 2));

qsquaresum=qsquaresum+qsquare.get(i);

}

Calculate dot product for all documents and query:

**for**(**int** i=0;i<query\_list.size();i++){

**for**(**int** j=0;j<dict\_list.size();j++){

**for**(**int** k=1;k<=BasicCrawler.*docid*;k++){

**if**(query\_list.get(i).equals(dict\_list.get(j))){

match\_tf\_list[i][k]=tfidf\_list[j][k];

}

}

}

}

**double**[][] crosstemp= **new** **double**[query\_list.size()][BasicCrawler.*docid*+1];

**double** crosstemp1=0;

**for**(**int** i = 0,j=0;i<query\_list.size() && j<dict\_list.size();i++,j++){

**for**(**int** k=1;k<=BasicCrawler.*docid*;k++){

crosstemp[j][k]=qtf\_idf\_list.get(i)\*match\_tf\_list[j][k];

}

}

**for**(**int** j=1;j<=BasicCrawler.*docid*;j++){

**for**(**int** k=0;k<query\_list.size();k++){

crosstemp1=crosstemp1+crosstemp[k][j];

}

crosstemp2.add(crosstemp1);

}

||Query|| && ||Document||:

||Document||:

**for**(**int** k=1; k<=BasicCrawler.*docid*;k++){

**double** dc=0;

**for**(**int** i=0;i<query\_list.size();i++){

dtemp=Math.*pow*(match\_tf\_list[i][k],2);

dc=dc+dtemp;

**if**(i==query\_list.size()-1){

// System.out.println("DC "+dc);

dlist.add(Math.*sqrt*(dc));

}

}

}

||Query||:

**for**(**int** i=0;i<query\_list.size();i++)

{

qtf\_idf\_list.add(query\_array.get(i)\*qidf\_list.get(i));

qsquare.add(Math.*pow*(qtf\_idf\_list.get(i), 2));

qsquaresum=qsquaresum+qsquare.get(i);

}

qsquaresum=Math.*sqrt*(qsquaresum);

||Query||\*||Document||:

**for**(**int** i=0;i<dlist.size();i++){

cotemp=qsquaresum\*dlist.get(i);

// System.out.println("COTEMP "+cotemp);

qdlist.add(cotemp);

}

Cosine Similarity:

Cosine Similarity(Query,Document1) = Dot product(Query, Document1) / ||Query|| \* ||Document1||

**for**(**int** i=0,j=0;i<qdlist.size() && j<crosstemp2.size();i++,j++)

{

cosine=crosstemp2.get(j)/qdlist.get(i);

List<String> top20\_list = **new** ArrayList<String>();

**if**(cosine>0 && cosine!=Double.***POSITIVE\_INFINITY***){

**int** later\_i = i+1;

System.***out***.println("Cosine Similarity: "+cosine+ " & value of i: " +later\_i);

System.***out***.println("URL: "

+ BasicCrawler.*url\_list*.get(later\_i));

File f = **new** File(

"C:/Users/Madhuri/Desktop/Info\_Retrival/Info\_Project/webcrawler/Doc"

+ later\_i + ".log");

// to check if file exists or not

**if** (f.exists()) {

**try** (BufferedReader reader = **new** BufferedReader(

**new** FileReader(f))) {

String line = **null**;

// logic to load words into list

**while** ((line = reader.readLine()) != **null**) {

top20\_list.add(line);

}

} **catch** (IOException x) {

System.***err***.format("IOException: %s%n", x);

}

System.***out***.println(top20\_list.size());

**if** (top20\_list.size() > 20) {

**for** (**int** d = 0; d < 21; d++) {

**if** (!top20\_list.get(d).isEmpty())

System.***out***.println(top20\_list.get(d));

}

} **else** {

**for** (**int** e = 0; e < top20\_list.size(); e++) {

**if** (!top20\_list.get(e).isEmpty())

System.***out***.println(top20\_list.get(e));

}

}

}

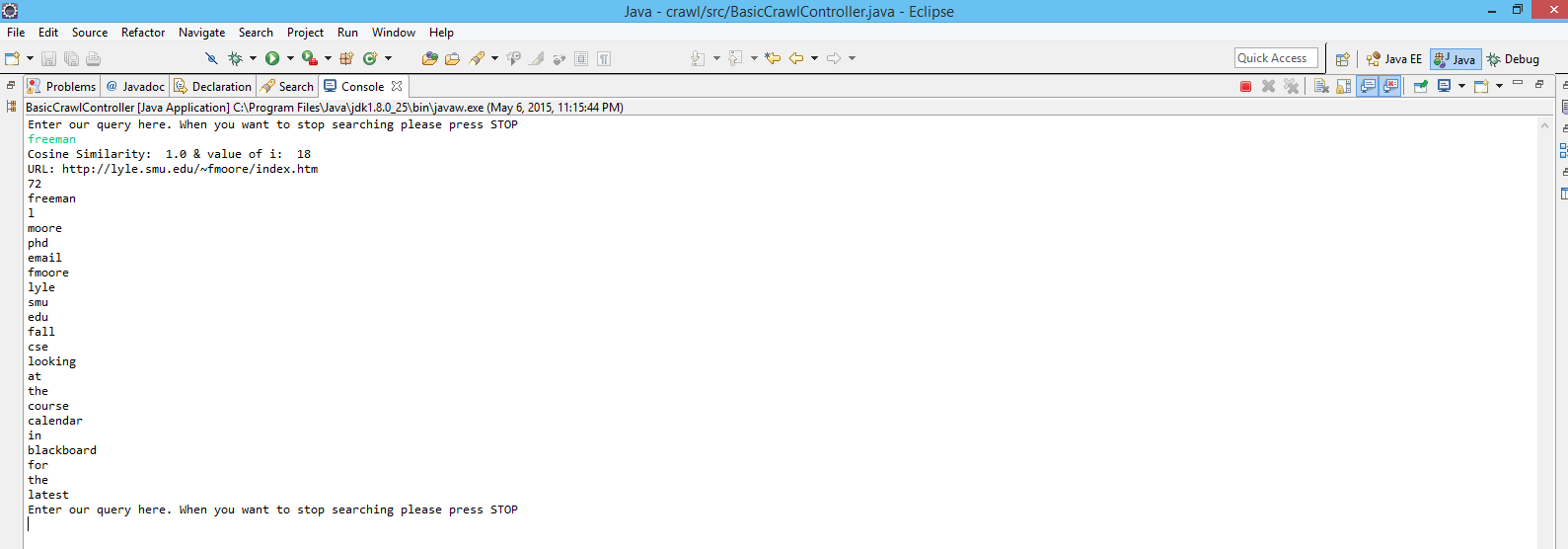
}

}

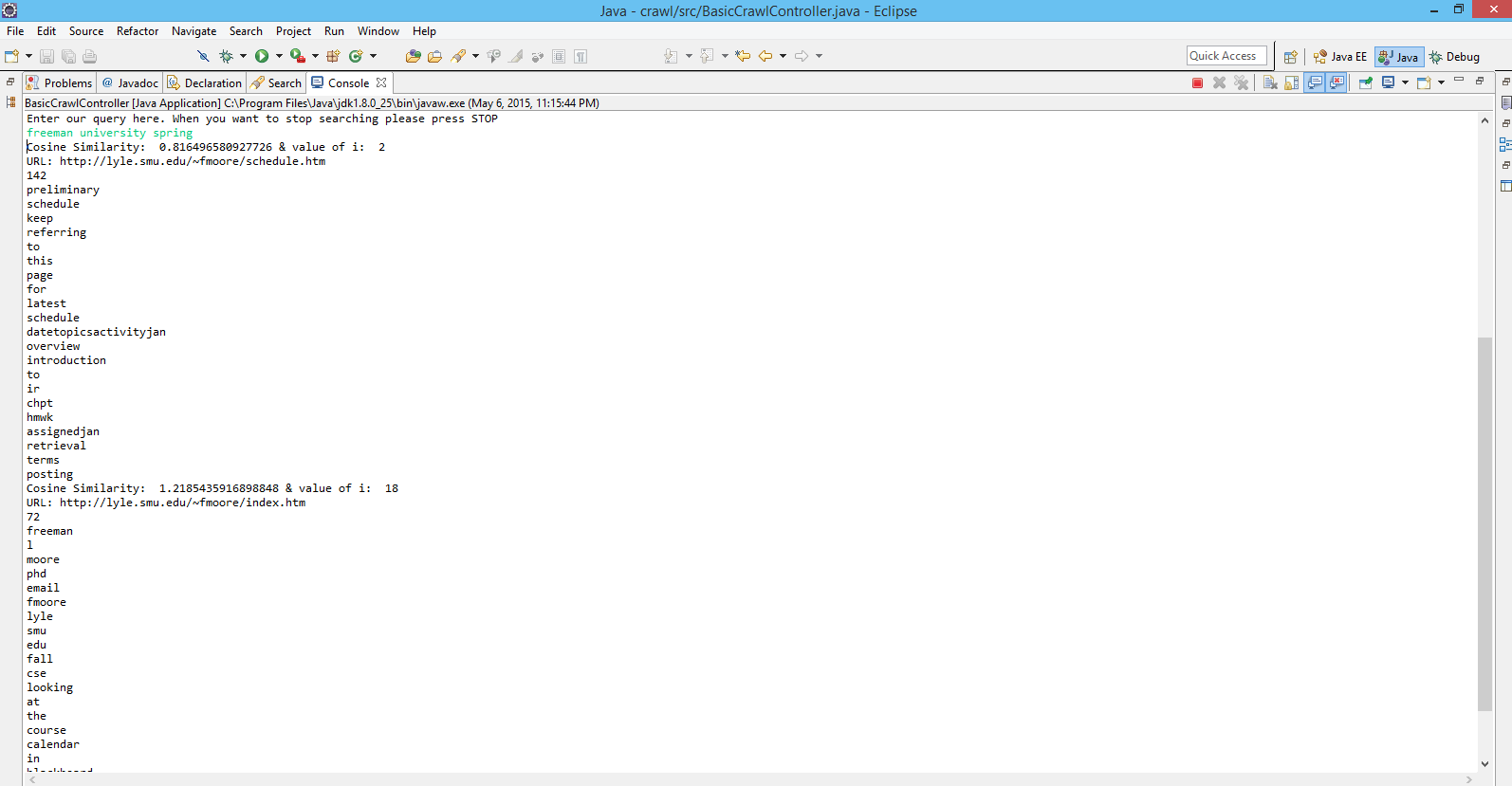
}

OUTPUT:

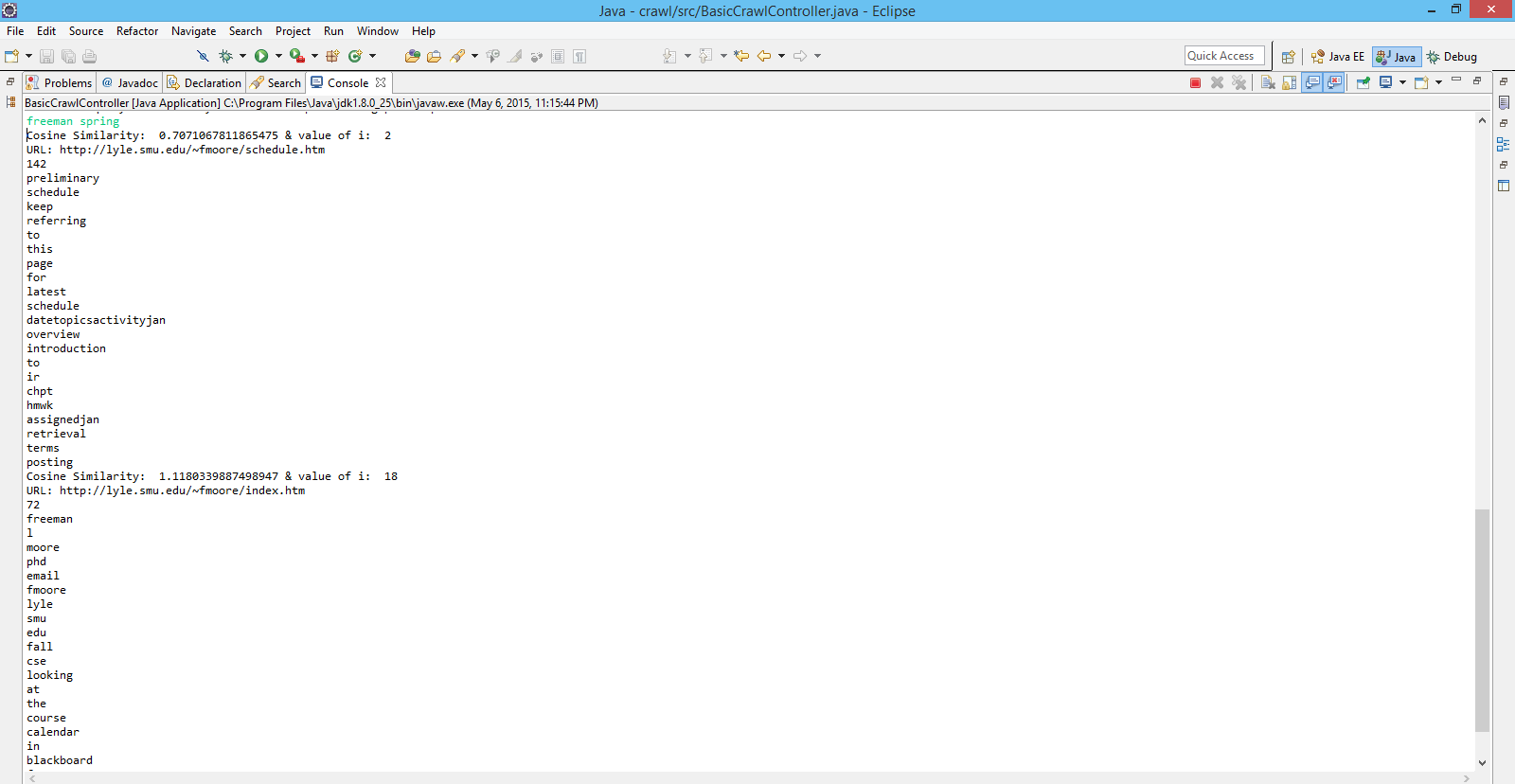
For Input: Freeman



For Input: Freeman University Spring



For Input: Freeman Spring



For input : Freeman College

