**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.**

Solution

Project 1:

Splitting text into words using regular expression, which will split based on space, tab, return, special characters. After splitting we considered all of them as words.

//Below is the Regular expression used to split text into words

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();

}

Project 2:

In this project, we are more specific about words. We have removed all the numbers and special characters from the word listing, since these are not defined as words. Our crawler will search if the first letter of the word starts with a letter then only it will be added otherwise it will be ignored. Ignored words include numbers and other special characters. The major difference between project1 word definitions to project2 is, now words should start with alphabet, word shouldn’t start with any special character or integer.

//For each word which are splited using regular expression, using for loop to compare word if it starts with alphabet

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

{

//difference from project1 to project2, making sure if words which are going to added to dictionary has to start with letter

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

{

dict.add(words[i]);

}

**Question2: You will need a dictionary of words. What is your definition of “word”? Will you generate the dictionary while navigating the pages or as a separate step? Explain your approach. If needed, you can define a fixed size for the dictionary based upon the results from Project1, and allowing for a 10% increase.**

Definition of a word:

Definition of word according to our project is:

1. Split text into words according to special characters, spaces, tab, return for that the regular expression we used is “("[\\s\\t\\r.,():?\\//@~-]+")”
2. Now, we pulled words into dictionary only if word starts with letter.

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

We are generating the dictionary while navigating the pages itself. When crawler starts crawling a page, it will assign a doc id to that page. For that doc id the words will be scraped by using Treeset list. The advantages of using TreeSet are: It is dynamic, you don’t have to assign its length. It will automatically sort the words alphabetically. So, whenever the word is retrieved, it will be placed in the dictionary according to its character position. When the crawler completes its crawling for one page it will move to another and start appending the words of later page. Then the same process will continue. This is how dictionary of words is created.

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

**/\*for each word below logic checks if it starts with letter\*/**

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

{

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

dict.add(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, we are dealing with dynamic Treeset, there is no need to provide size to dictionary. Here in our project dictionary doesn’t need fixed size.

**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**

Solution:

As we are building term frequency matrix after complete creation of dictionary, after completion of crawling based on last document id we are ceration 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 time we will have last document id stored in variable “docid”. So based on docid and dictionary list size we are creating 2D array called “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. This way the term frequency matrix is created.

Note; As we are retrieving words into documents according to docid. I request you to place those documents one folder and provide it in below code while testing

controller.addSeed("http://lyle.smu.edu/~fmoore/");

// controller.addSeed("http://lyle.smu.edu/~fmoore/");

//controller.addSeed("http://lyle.smu.edu/~fmoore/");

/\*

\* Start the crawl. This is a blocking operation, meaning that your code

\* will reach the line after this only when crawling is finished.

\*/

controller.start(BasicCrawler.class, numberOfCrawlers);

//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>();

//Provide file name where exactly files are present in your folder structure

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? [10 points]**

Solution:

After creating dictionary, term frequency matrix. We have provided an option to user can give query also able enter multiple queries separated by space. Using while loop if used enter “STOP” then searching will be done, program terminates. Below is the code to describe above questions.

List<Integer> doc\_array = **new** ArrayList<Integer>();

List<Integer> query\_array = **new** ArrayList<Integer>();

repeat:

//while loop to make sure user can enter multiple queries

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

**int** validcount=0;

**double** temp;

**double** qtemp;

**double** dtemp;

**double** idf;

**double** tf\_idf;

**double** qidf;

**double** qsquaresum=0;

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

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

**double** cosine=0;

**double** cotemp=0;

List<Double> cosinesim=**new** ArrayList<Double>();

List<Double> dlist = **new** ArrayList<Double>();

List<Double> qsquare = **new** ArrayList<Double>();

List<Double> crosstemp2=**new** ArrayList<Double>();

List<Double> qdlist=**new** ArrayList<Double>();

List<Double> qtf\_idf\_list=**new** ArrayList<Double>();

List<Double> qidf\_list=**new** ArrayList<Double>();

List <Double> idf\_list=**new** ArrayList<Double>();

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

//BufferReader to read user input

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

String query=qr.readLine();

//Once user enter “stop” then while loop stops and program terminates

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

{

System.***out***.println("Done Searching");

**break**;

}

**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? GRADUATE STUDENTS: Also include in the display, the first 20 words of the document. If you stemmed the words, this can be the stemmed version. [40 points]**

Solution:

Cosine similarity is calculated once user enters query against all documents. Document Url is printed for top N results. We considered N = 2 in our code, also it displays TOP 20 words from that URL. Below is the code and screenshots.

List<Integer> doc\_array = **new** ArrayList<Integer>();

List<Integer> query\_array = **new** ArrayList<Integer>();

repeat:

//while loop to make sure user can enter multiple queries

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

//different variables created to calculate cosine similarty

**int** validcount=0;

**double** temp;

**double** qtemp;

**double** dtemp;

**double** idf;

**double** tf\_idf;

**double** qidf;

**double** qsquaresum=0;

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

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

**double** cosine=0;

**double** cotemp=0;

List<Double> cosinesim=**new** ArrayList<Double>();

List<Double> dlist = **new** ArrayList<Double>();

List<Double> qsquare = **new** ArrayList<Double>();

List<Double> crosstemp2=**new** ArrayList<Double>();

List<Double> qdlist=**new** ArrayList<Double>();

List<Double> qtf\_idf\_list=**new** ArrayList<Double>();

List<Double> qidf\_list=**new** ArrayList<Double>();

List <Double> idf\_list=**new** ArrayList<Double>();

//user enters query

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

//logic to read input

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

String query=qr.readLine();

//if user enters “stop” program terminates

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

{

System.***out***.println("Done Searching");

**break**;

}

query\_list.clear();

//Query is splited based on “ “ and store in list

String qwords[] = query.split(" ");

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

{

query\_list.add(qwords[i]);

}

//logic to check if words are there in dictionary else it will reject query and aging ask for user input

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

**int** query\_count = Collections.*frequency*(query\_list,

query\_list.get(i));

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

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

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

qidf\_list.add(qidf);

query\_array.add(query\_count);

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);

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

**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);

}

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

**for** (**int** k = 0; k < BasicCrawler.*docid* + 1; k++) {

tfidf\_list[j][k] = 0;

}

}

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

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

**for** (**int** k = 0; k < BasicCrawler.*docid* + 1; k++) {

match\_tf\_list[j][k] = 0;

}

}

**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]);

}

}

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

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

**double** dc = 0;

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

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

;

{

dtemp = Math.*pow*(tfidf\_list[j][k], 2);

dc = dc + dtemp;

**if** (j == dict\_list.size() - 1) {

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

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

}

}

}

}

}

**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);

}

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

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

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

qdlist.add(cotemp);

}

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

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

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

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

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

System.***out***.println("Cosine similarity: " + cosine);

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));

}

}

}

}

}

}

}

Screenshots:





