CS 242: Information Retrieval And Web Search Project Report

- ***Multithreaded Web Crawler***
- ***Lucene Indexing and Searching***
- ***Hadoop Indexing and Searching***
- ***User Interface***

By,
Tanaya Vadgave (SID: 861242264)
Shravani Madhavaram (SID:861306757)

----------**INDEX**----------

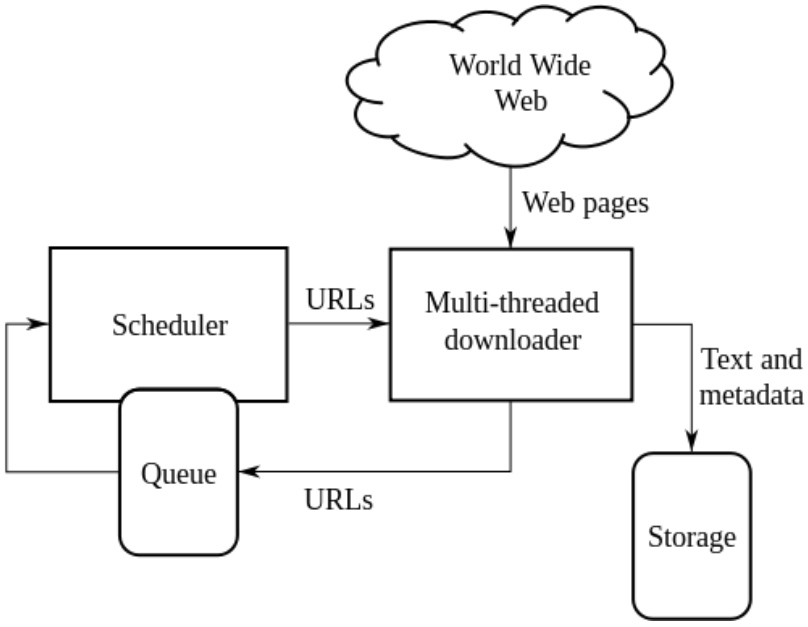
Number	Title	Page Number
1	Introduction	3
2	Crawling Architecture	3
3	Crawling Strategy	4
4	Lucene Indexing	5
5	Lucene Searching	7
6	Hadoop Indexing	8
7	Hadoop Searching	9
8	Run Time of Lucene Indexing vs Run Time of Hadoop indexing	9
9	User Interface and Snippet Generation	10
10	Results	11
11	Instructions to deploy the project	14
12	Limitations of System	15
13	Obstacles and Solution	15
14	Collaboration	16
15	References	16

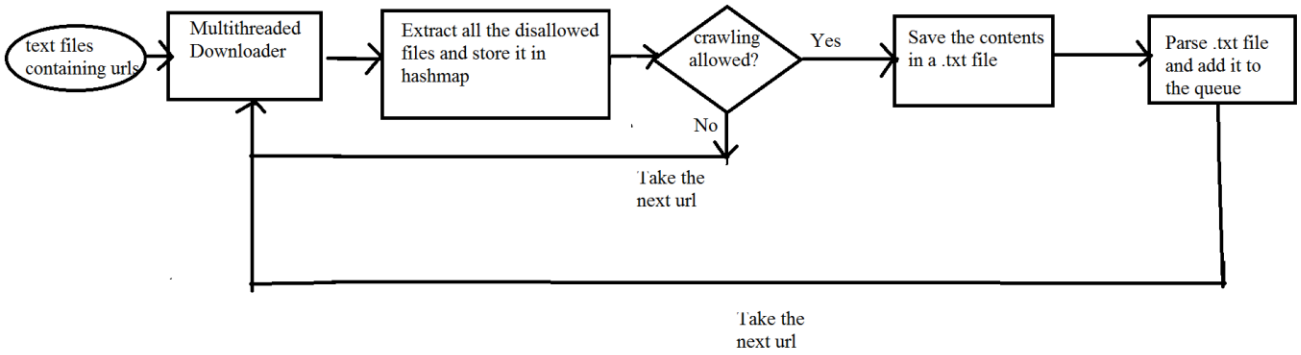
1. Introduction

Search Engine is Software system that is designed to search information on the World Wide Web. In the CS242: Information Retrieval and Web Search project, we are building such Search Engine to search '.edu' content from World Wide Web. We are using JSOUP to parse and download the .edu data from web. After that we are indexing the content using Lucene and Hadoop. We have implemented ranking algorithms to get the best possible results from indexed content and finally displayed the content using UI.

2. Crawling Architecture

Web crawler is used to find and download web pages automatically which provides the collection for searching. Beginning with root URL, it fetches each page, parse it for links to pages it has not seen, and adds the new link to queue. When it fetches a page with no unseen links and when the queue is empty, it stops.





To get the web pages we will be using 'jsoup' which provides an API to extract and manipulate data from URL or HTML file. We will be restricting the pages to '.edu' category. Then,

a. We will parse robot.txt file of the webpages
b. Check whether the particular site is allowed to crawl or not
c. If the site is not allowed to crawl, extract the pages/directories which the site has not allowed for crawling
d. Recursively crawl remaining pages to get >5GB data and store that in the disk.
e. We have downloaded about 5.5GB(Approximately 78k files) of data.

3. Crawling Strategy

a. Downloading:

We first take a text file as an input which contains the seed pages for our web crawler. In our Start.java class (which contains the main function) we implement a **Queue** which takes the seed pages' link.

For each seed in the text file a new thread is being created until 50 threads are created. In the multiThreading.java class threads are being started, and crawling method of Parse.java class is being called, which starts the crawling of data. First we check whether the url can be crawled or not by parsing robots.txt for that url's domain to check "/" content in disallow section and then, maintaining a **Concurrent Hashmap** of robots.txt url's domains as key and list of urls(disallowed URLs/Directories) which can't be crawled. Then we will check whether the url is there in the concurrent hashmap, if it's not available we will just parse the robots.txt and store it in the concurrent hashmap and check whether crawling is allowed. If the url can be crawled, we will add the url to the **hashmap** which we create to add already visited urls to avoid duplicate URLs and then download the file in .txt format. After that parse the file and add all the urls which has .edu hostnames to the initial Queue. Before adding the url to the queue we will check for the url in the concurrent hashmap of already visited nodes and then add it to the queue if it's not there in the hashmap.

b. Multithreading:

Web crawling speed is governed not only by the speed of ones own internet connection, but also by the speed of the sites that are to be crawled. Especially if one is crawling sites from multiple servers, the total crawling time can be significantly reduced, if many downloads are done in parallel. Multi-threading is solution for this.
In our project, we are using 'ThreadPoolExecutor-High level API' for multi-threading purpose. The main advantage of using it is it can be customized according to need.

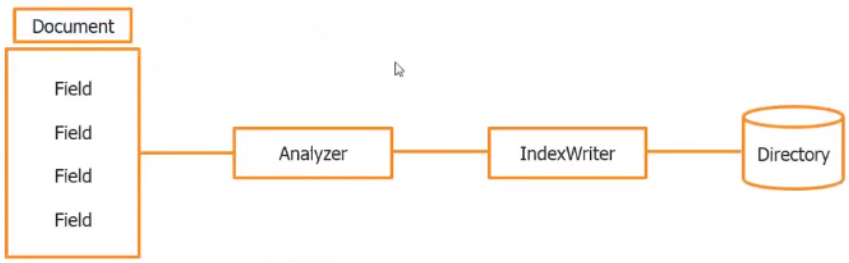
Multithreading is used while crawling (Parsing and downloading) stage. Threads from the thread pool work synchronously on the queue (which consists of URLs to be parsed), poll the URL from the queue, parse it and download the webpage.

c. Duplicate File Check:

On web, there can be two different URLs but having the same content. Those duplicate contents are not supposed to be downloaded back. We are performing this duplicate check by calculating check sum of URLs and stored in the checksum map. If any new URL has same checksum then it is a duplicate page and thus will not be stored again. Code snippet:

```
public static String getCheckSum( String str ) throws Exception
{
MessageDigest md = MessageDigest.getInstance( "MD5" );
byte[] bytesOfStr = str.getBytes( "UTF-8" );
byte[] mdBytes = md.digest( bytesOfStr );
//convert the byte to hex format
StringBuffer str = new StringBuffer("");
for (int i = 0; i < mdBytes.length; i++)
{
str.append(Integer.toString((mdBytes[i] & 0xff) + 0x100, 16).substring(1));}
return str.toString(); }
```

4. Lucene Indexing



Classes used when indexing documents with Lucene

a. Indexing process is one of the core functionality provided by Lucene. Following diagram illustrates the indexing process and use of classes. IndexWriter is the most important and core component of the indexing process.

   We add *Document*(s) containing *Field*(s) to *IndexWriter* which analyzes the *Document*(s) using the *Analyzer* and then creates/open/edit indexes as required and store/update them in a *Directory*. IndexWriter is used to update or create indexes. It is not used to read indexes.

b. We applied similar classes showed in above diagram for indexing document
   - We used Lucene inverted index data structure
   - As soon as 'IndexCreation' function is called, analyzers are initialized to different fields of documents (Detailed description of 'Analyzer choice' is in next section).
   - Then, index writer is initialized with specified 'IndexDirectory' where indices are to stored.
   - Files in the 'dataDirectory' (Which have already been downloaded by the crawler) are iterated one after the other.
   - Analyzer are applied to different fields and indices are created and stored in Lucene index document. Files with extension .fdt, .fdx, write_lock, .cfe, .cfs are observed inside IndexDirectory.
   - Indexing operations handled by the code:

     Add Document: to create the indexes on the newly available contents. We are using this operation for indexing file contents, file name and file path.
     Update Document: to update indexes to reflect the changes in the updated contents. It is similar to recreating the index
     Delete Document: update indexes to exclude the documents which are not required to be indexed/searched

c. Text Analyzer Survey and choice

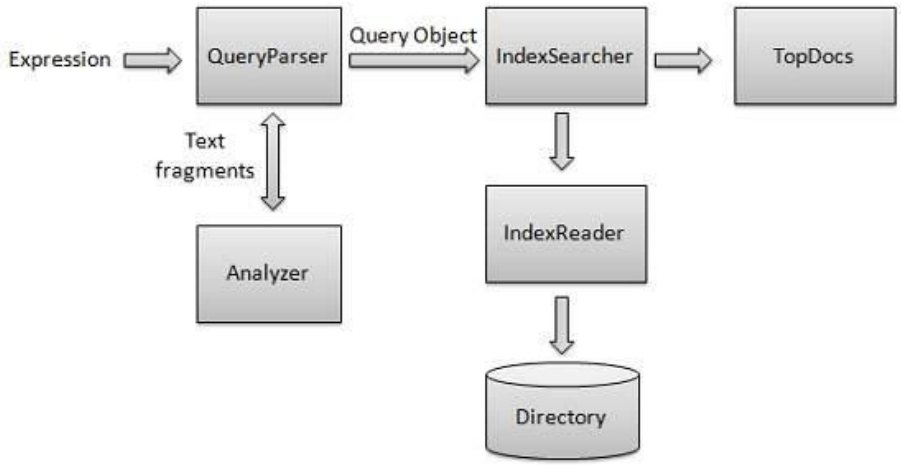
   In our project, we needed to use field operation which specifies a way or controls the way in which contents of a field are made searchable. We compared following text analyzers to mad a choice of which analyzer to be used for which field:
   - WhitespaceAnalyzer: This analyzer spilts the text in a document based on whitespace.
   - Simple Analyzer: spilts the text in a document based on non-letter characters and then lowercase them.
   - Stop Analyzer: removes the common words like 'a','an','the' + standard Analyzer functions
   - Standard Analyzer: Handles names, email address etc. It lowercases each token and removes common words and punctuation if any.
   - Keyword analyzer: Keyword Analyzer you mentioned *doesn't split the text at all* and takes all the field as a single token. It is used for things like ID or phone numbers, but not for usual text.
   - PerFieldAnalyzer: Used for specifying different analyzer on per field basis.

d. Our choice of Analyzer:

We have used Per-field analyzer. We kept standard analyzer as the default analyzer. Fields analyzed:

Field	**Analyzer**
Title	Keyword
Anchor	Keyword
Metadata	Keyword
Body	Standard
Filename	Not Analyzed

5. Lucene Searching



Searching Process

Searching process is one of the core functionality provided by Lucene. Following diagram illustrates the searching process and use of classes. IndexSearcher is the most important and core component of the searching process.

We first create *Directory*(s) containing *indexes* and then pass it to *IndexSearcher* which opens the *Directory* using *IndexReader*. Then we create a *Query* with a *Term* and make a search using *IndexSearcher* by passing the *Query* to the searcher. *IndexSearcher* returns a *TopDocs* object which contains the search details along with document ID(s) of the *Document* which is the result of the search operation.

a. Query Parser Analyzer choice: we have used 'Standard Analyzer' at query parser

For Searching Lucene has default TF.IDF method. But we have also implemented Page Rank algorithm as well to get best possible results. Through UI we gave 2 options as: Lucene-TF and Lucene:Page Rank.
We also compared the top score results of these two ranking methods. We got different results for these two methods, as Page rank calculation is query independent.

b. Page Rank Algorithm:

Each web page has number of outgoing links and number of incoming links. For all unique URLs (which we found out during the crawling process), we created a HashMap for each URL as key and value as set of outgoing URLs. Similarly, we created another HashMap to save incoming URLs. We maintained the graph using these two HashMaps.
After that, we implemented Page Rank algorithm by calculating page rank for each page in the graph using formula:
PR(A)=(1-d)/N + d*(PR(T1)/C(T1)+…+ PR(Tn)/C(Tn))
We also checked till the convergence is reached. We stored the page rank in one text file 'pageRankValues.txt' once for the data. And the results saved in the file are used to get top 10 results for query search. After searching the query, we collected top 10 results and displayed those on UI highest page rank page to lowest page rank page. [Note: for output please check 'results' section of this report.]

6. Hadoop Indexing

a. Sequential Files:

As Hadoop stores all the HDFS files metadata in name node's main memory (which is a limited value) for fast metadata retrieval, so Hadoop is suitable to storing small number of large files instead of large number of small files. One more disadvantage of maintaining small files, from map reducing perspective is that, processing more number of small files will require a lot of map tasks, as each file will be processed as one separate file split in map reduce job. So if we merge small files into large files, map reduce job can done quickly.
These small files are merged into a sequential file which is a data structure in Hadoop in order to make our Map Reduce jobs more efficient. A sequential file is like a container collectively stores all small documents in a compressed Key value pair where key is document name and value is the content of the document. Hadoop indexing is done using a single Mapreduce job , which makes our solution very fast and efficient.

b. Map Reduce:

Sequential file is given as input to the map which takes each document and tokenizes the content and removes all the stop words and calculates the frequency of each word in each document. Output of map function will be in the form of key value pair where key would be each word and the value would be document id and frequency of the word in that document (word, {document id, frequency}).
Partitioner technique where the key value pairs are routed to different reducers is implemented. The Output of the reducer is using the technique of splitting the input files based on the first character. We get the output in multiple files based on the first character of the keyword. So reducing is done using 26 reducers. The advantage of using the partitioner is the execution time of mapreduce job is reduced significantly.
Reducer which takes the output of the mapper as the input will read all the values of a key and concatenates all the values. Output of the reducer will be in the form of key value pair where key would be each word and value will be the list of all the documents and word frequency in each document (word, [{document id1, frequency},{document id2, frequency}..]).

c. Handling Stop words:

Hash map of google stop words is created in the map function. While reading data from each file, we check each word whether the stop words hash map contains the word, if it does we don't add the word to the output. Hash map is used to create google stop words because searching is quick using hash map.

7. Hadoop Searching

Ranking for Hadoop is done by implementing BM25 and Page Ranking together. Page rank for all the pages is calculated initially by storing all the outgoing links and then calculating the page rank for each url. BM25 model is calculated by the formula:

$$\sum_{i \in Q} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n_i-R+r_i+0.5)} \cdot \frac{(k_1+1)f_i}{K+f_i} \cdot \frac{(k_2+1)qf_i}{k_2+qf_i}$$

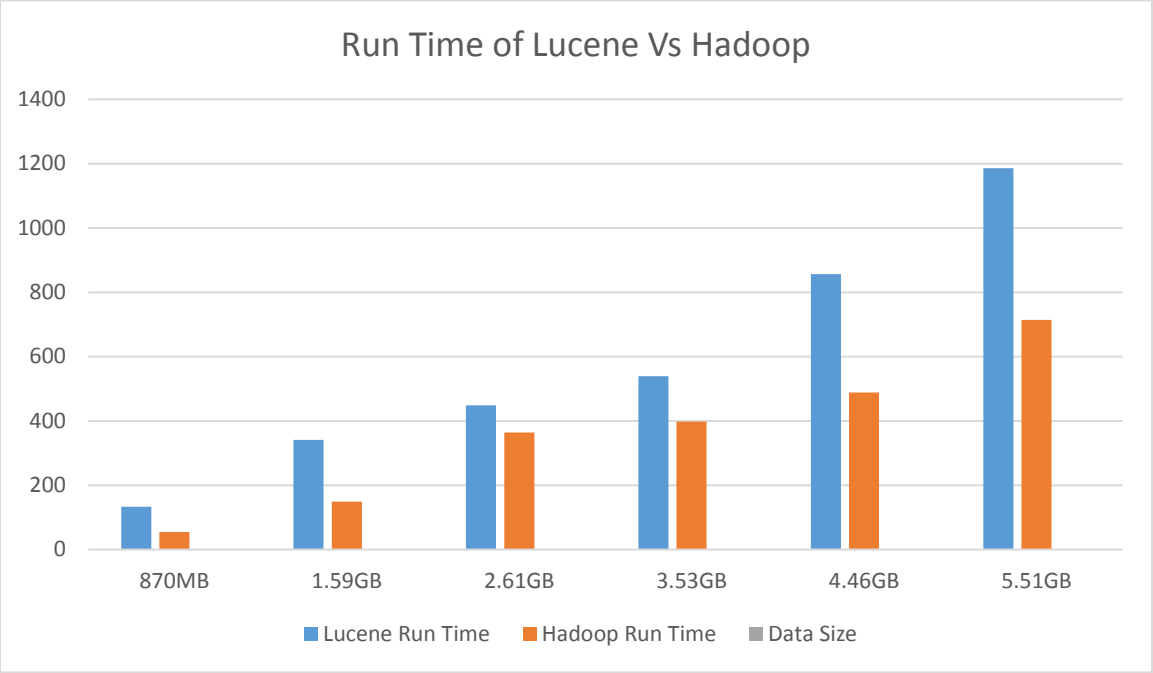
Initially the indexed file from Hadoop is taken as input and hash map is created where key is the word and value is also a hash map which has key as document name and value as the frequency of the word in that. For a given query, documents are retrieved for each word and BM25 score is calculated for each document. Then we retrieve the page rank for each document and add BM25 score to make it more accurate and then add all the pages are added to a priority queue where we can retrieve the top 10 results. We have designed **'Page Rank'** algorithm for Hadoop search as well. To get best possible results, we have obtained the combined ranked output as BM25+Page Rank.

8. Run Time of Lucene Indexing vs Hadoop Indexing

a. Result Table:

Data Size	Lucene Time(seconds)	Hadoop Time(seconds)
870MB	133	54
1.59GB	341	149
2.61GB	449	364
3.53GB	529	338
4.46GB	856	488
5.51GB	1186	714

b. Graph



c. Observation

Hadoop indexing is approximately twice as fast as Lucene Indexing.

9. User Interface and Snippet Generation

We used JSP-Servlet, HTML, CSS to design our User Interface. We have kept additional feature as focus to the search window and buttons.

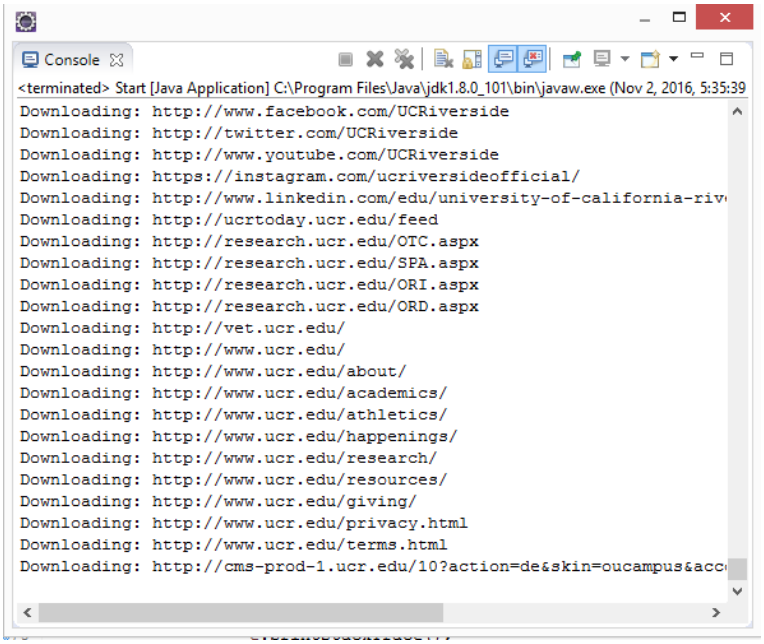
When obtained top 10 search results for the query, we are printing following things:

a. Filenames in which the results are obtained
b. Ranking Scores of top 10 results in the descending order
c. Title, clicking on which one can navigate to respective page
d. Snippet of about 3-4 lines. Firstly, character count and term position is calculated. Then, following conditions are checked to get best possible snippet

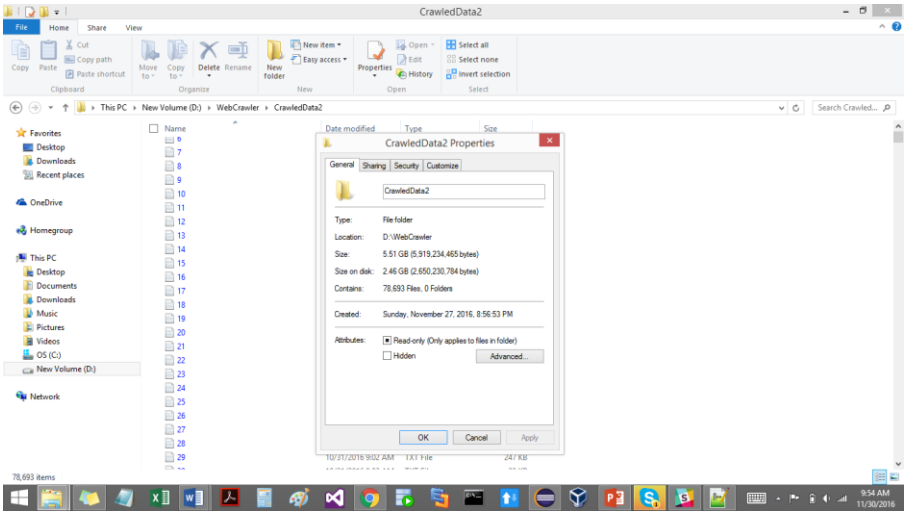
   i. If entire query is not present, use the first term of the query
   ii. If search term is appearing at the beginning of the document
   iii. If search term is appearing at the end of the document
   iv. If search term is appearing at the middle of the document
   v. Error/document length is not sufficient, print top 4 lines of the page

10. Results

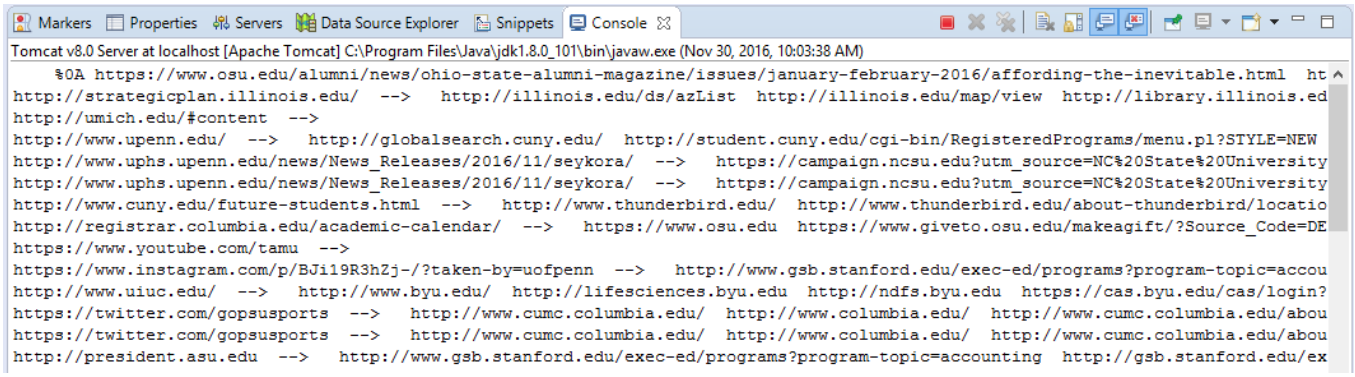
a. Downloading webpages corresponding to following URLs:



b. Size of Crawled Data



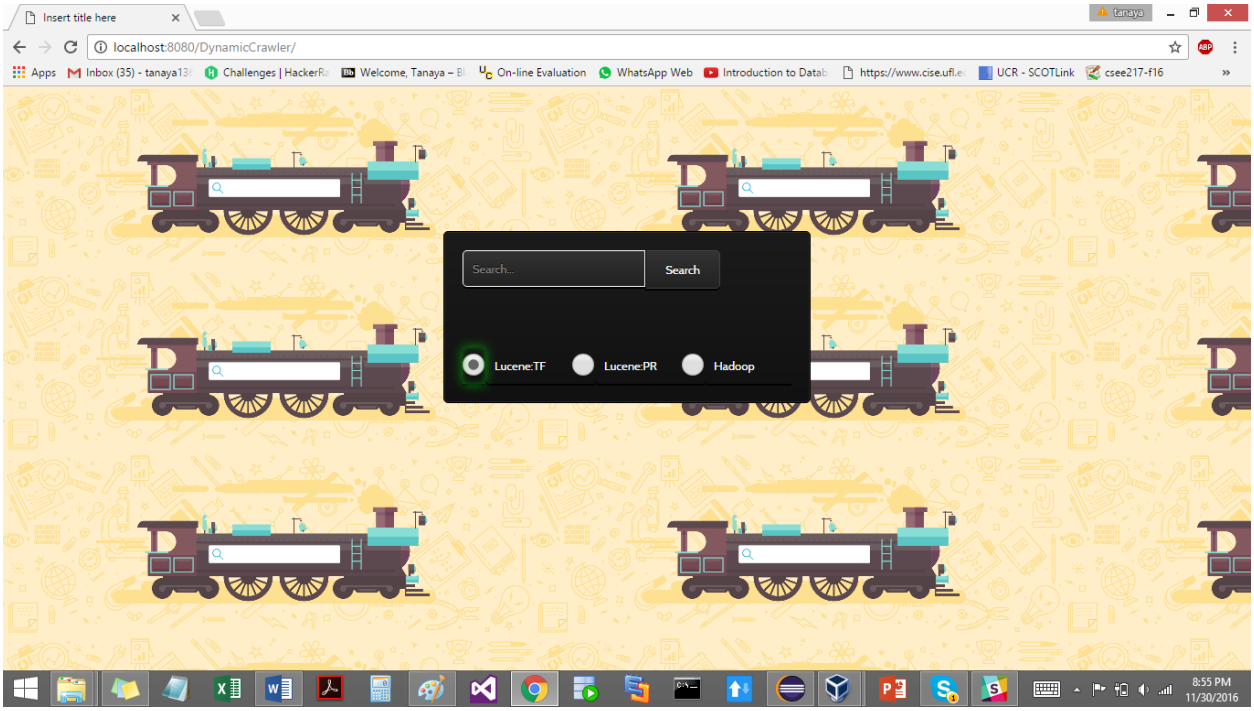
c. Web Graph Display

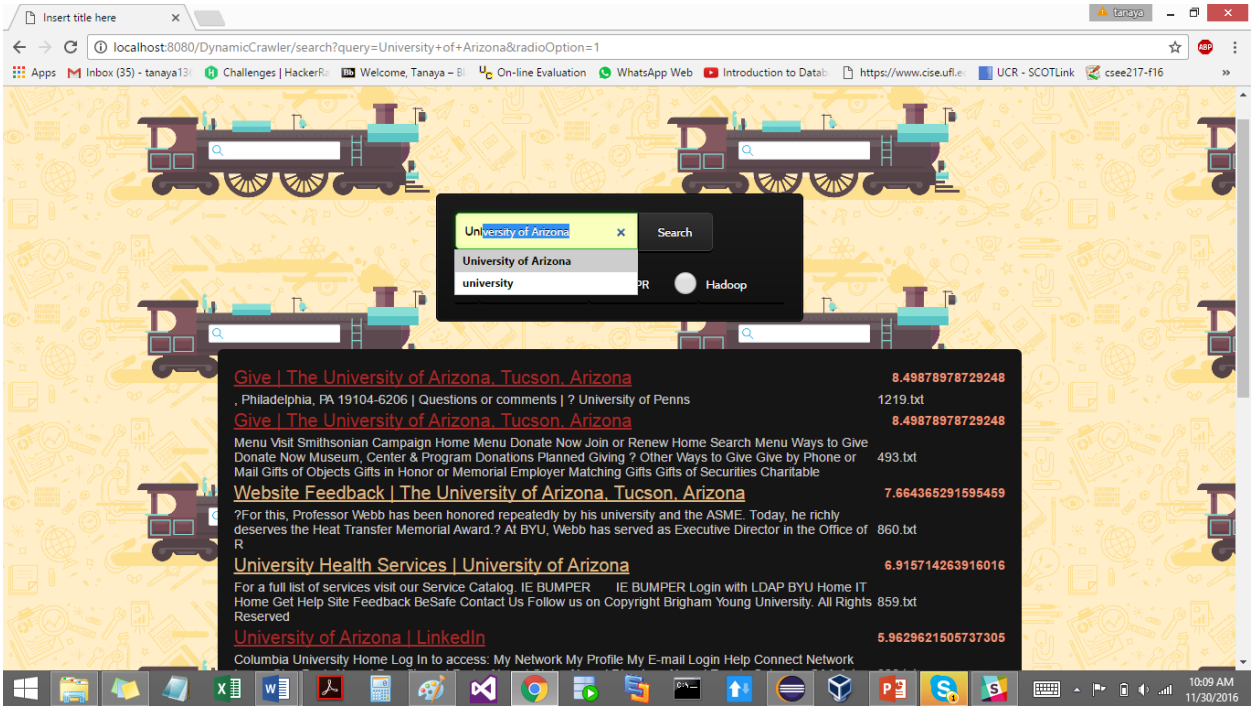


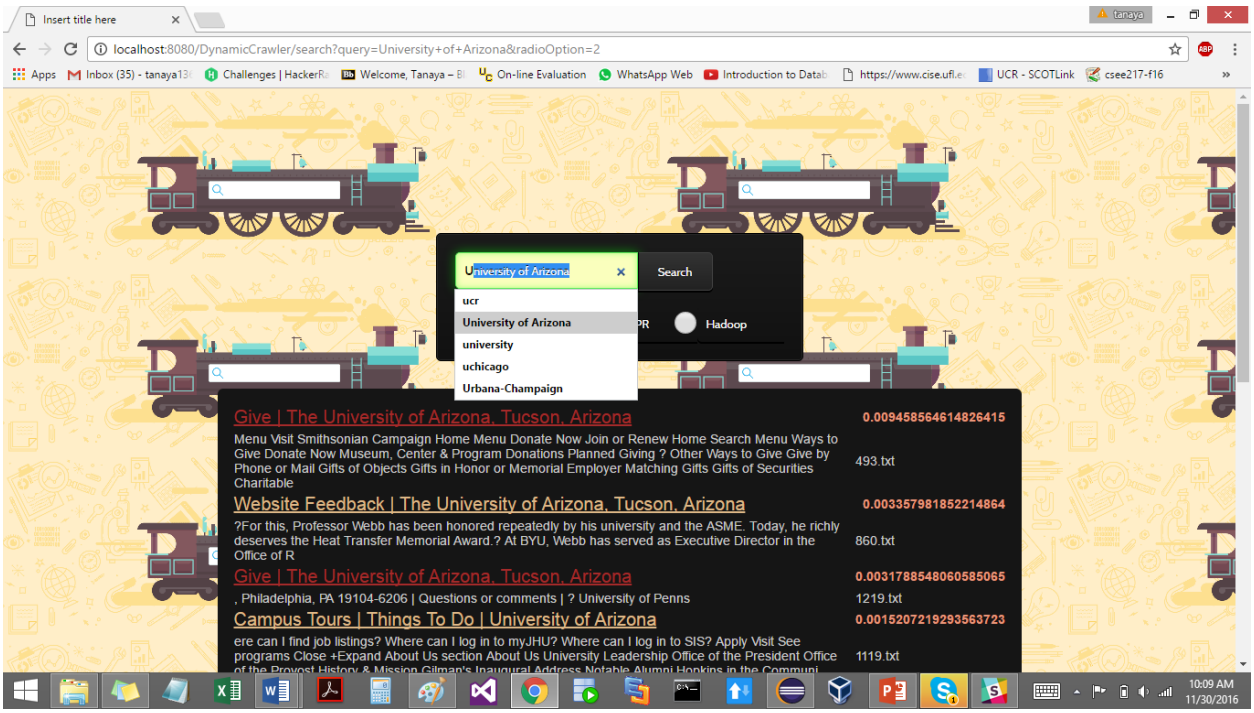
d. Data of the file 'pageRankValues.txt ' which has Page Ranking score for each page

```
http://news.columbia.edu/tomomori - 2.3603463E-4
http://www2.cuny.edu/libraries/about/ - 0.12404009
http://jobs.byu.edu/ - 0.022488289
http://mednews.stanford.edu/media/index.html#policy - 2.3603463E-4
http://dellmedschool.utexas.edu/prospective-students - 0.032864757
http://www.princeton.edu/artsandtransit - 2.3603463E-4
https://finserve.byu.edu/ - 0.003569368
http://www.azte.com - 2.3603463E-4
https://twitter.com/intent/retweet?tweet_id=801110293753593856 - 2.3603463E-4
http://si.convio.net/site/PageNavigator/enews?s_src=siod_sig_sihome - 2.3603463E-4
http://www.gradschool.psu.edu/admissions/ - 0.015791591
http://www.colorado.edu/alumni - 2.3603463E-4
http://www.asu.edu/admin/ - 0.007928343
http://bgs.byu.edu/ - 0.030492445
http://www.lib.utexas.edu/ - 0.04993793
https://t.co/gvOPjZPh4v - 2.3603463E-4
http://www.bmcc.cuny.edu - 0.15325604
https://www.insidehighered.com/audio/2016/11/16/all-cool-kids-aren%E2%80%99t-doing-it#.WCxQqulFEy4.mailto - 2.3603463E-4
http://parents.stanford.edu/vital-information/ - 0.032738477
http://besmart.com/ - 2.3603463E-4
https://cas.byu.edu/cas/login?service=https://my.byu.edu/uPortal/Login - 0.03297188
http://www.thesundevils.com/ViewArticle.dbml?ATCLID=208254939&DB_OEM_ID=30300 - 2.3603463E-4
http://fightingillini.com/ - 2.3603463E-4
https://giving.columbia.edu/giftguide/schoolprograms - 0.061629154
http://studentaid.psu.edu/ - 0.015791591
http://home.byu.edu/home/copyright - 0.6115266
http://global.umich.edu/newsroom/study-abroad/ - 2.3603463E-4
http://admissions.utexas.edu/ - 0.032864757
http://stanford.edu/search/ - 0.37044376
http://visit.illinois.edu - 0.051646736
```

e. Some User Interface results for a query







11. Instructions to deploy

'IRCode' folder, there will be 2 sub-folders: WebCrawler, DynamicCrawler.
WebCrawler sub folder has another sun folder: WebCrawlerRun which has the bat file to run.

a. To run Web Crawler, do following:

- Go to the folder WebCrawlerRun through command prompt.
- Run the WebCralwer.bat file along with 4 arguments:
  Argument1: seed URL file
  Argument2: maximum number of pages
  Argument3: hop limit
  Argument4: destination directory where crawled data is stored
- Example Command:

```
https://www.asu.edu/sites/default/files/css/css_hbVdKIcjrsyzM_RfN1N23EQ3BvAc1QiT
3NaRgfz3is0.css
https://www.asu.edu/sites/default/files/css/css_hbVdKIcjrsyzM_RfN1N23EQ3BvAc1QiT
3NaRgfz3is0.css
http://continuing-ed.phoenix.edu/coursebasket/publicCourseBasket.do?method=load
https://interdisciplinary.stanford.edu/
http://www.thunderbird.edu/
Terminate batch job (Y/N)? Y

D:\WebCrawler>WebCrawler.bat test.txt 80000 6 "D:\WebCrawler\CrawledDataTest"
```

b. Dynamic Crawler (Contains: indexing, searching, UI, Page rank)
There are two ways to deploy crawler on Tomcat server.
   i. Run the DynamicCrawler.war file on tomcat server8 using command line
   ii. Open Eclipse
   Import the project(It has all required Jars)
   Open JavaEE prospective
   Open src/main/resources
   Open config.properties
   Make changes in downloadedFiles(path where you have saved all downloaded files), index(path where you have saved all indexed files), hadoopindex(path where you have saved hadoop indexed file)
   Run the project on tomcat server. (We had run our code on Apache Tomcat8)
   Open browser. Search: http://localhost:8080/DynamicCrawler/
   The GUI will open. Select any of 3buttons- Lucene:TF, Lucene:PR, Lucene:Hadoop to make a choice for searching.

   [Note: The source code folder contains sampleHadoopIndexFile which you can use for testing the code]

12. Limitations of System

a. Dynamic web pages are not handled
b. HTTP protocol(jsoup.connect) is handled but FTP/HTTPS protocols are not handled
c. PDF, Images links are discarded from crawling
d. Lucene limitation: Lucene indexing is slower for very large data. It can be made faster using multithreading.

13. Obstacles and Solution

a. Obstacle: The first obstacle that we faced was when we encountered same pages being downloaded multiple times.

   Solution: Now we kept a hashmap which stores already downloaded files. Before adding the url to the queue we check whether the url is already there in the hashmap, If the hashmap contains the url we won't add the url to the queue otherwise we will add the url to the queue. After incorporating this, we removed this obstacle and now we only crawl different web pages.

b. Obstacle: First, we had thought about sequential processing of queue to parse URLs. It was way too longer and slower process for collecting 5GB+ data.

   Solution: We used 'Thread Pool Executor' to manage multithreading.

14. Collaboration

a. URL Extraction- Shravani
b. Downloading Page Contents- Shravani
c. Robot.txt parsing and duplicate pages check- Tanaya and Shravani
d. Multithreading: Tanaya
e. Lucene Indexing and searching- Tanaya
f. Hadoop indexing and searching- Shravani
g. Page Rank- Tanaya
h. User Interface- Tanaya and Shravani
i. Project Integration and Testing- Tanaya
j. BAT file creation and report- Shravani and Tanaya
k. Snippet Generation- Tanaya

15. References

a. Jsoup Tutorial

http://o7planning.org/en/10399/jsoup-java-html-parser-tutorial#a864736
https://jsoup.org/cookbook/extracting-data/example-list-links

b. Apache lucene file format

http://lucene.apache.org/core/3_5_0/fileformats.html

c. Lucene Tutorial

https://www.tutorialspoint.com/lucene/lucene_indexing_process.htm

d. Hadoop

Class material