Introductory Programming 2022 Final Project: Search Engine

Group 16
Simon Olafsson German Alexander Garcia Angus Mads Berg Andreas Elsberg
{simol,gega,madbe,elsb}@itu.dk

December 17, 2022

Introduction

Statement of Contribution

Andreas Elsberg and Simon Olafsson were responsible for implementing task 4-5. German Alexander Garcia Angus and Mads Berg was responsible for implementing tasks 1-3. Everyone has contributed equally to every part of the project across all tasks, when it comes to coding and researching to find information on task solutions.

1 Task 1: Refactoring

When we started refactoring this project we quickly realised different issues in the source-code that we initially got handed. We noticed that, throughout the program, there generally was bad coding practices, very high coupling and low cohesion, method overriding, none type-specific type declarations,

Anyone familiar with object oriented programming languages will know that the aim is in fact the opposite. Low coupling and high cohesion.

General bad coding practices

We noticed that throughout the code the type declaration of each type consisted of the type `var`, which is a non type-specific type declaration. This is considered bad coding practice in the world of programmers coding in Java in specific. The reason for this is, that the code becomes harder to figure out for others, and oneself after a certain amount of time.

Solution We simply exchanged all of the `var`-types with their specific types. Example:

```
1 var result = new ArrayList<List<String>>();
2
3 // after refactoring it, it became:
4
5 List<List<String>> result = new ArrayList<>();
```

Method overriding

One of problematic parts of the code that we noticed, was that there was method overriding (meaning two codes with the same name) of the method search(). One of the methods looked as the following:

```
 public void search(HttpExchange io) {
    String searchTerm = io.getRequestURI().getRawQuery().split("=")[1];
    var response = new ArrayList<String>();
    for (var page : search(searchTerm)) {
      response.add(String.format("{\"url\": \"%s\", \"title\": \"%s\"}",
          page.get(0).substring(6), page.get(1)));
    }
    var bytes = response.toString().getBytes(CHARSET);
    respond(io, 200, "application/json", bytes);
  }
```

And the other search() method looked as such:

```
public List<List<String>> search(String searchTerm) {
    var result = new ArrayList<List<String>>();
    for (var page : pages) {
      if (page.contains(searchTerm)) {
        result.add(page);
      }
    }
    return result;
  }
```

General Refactoring our code

When approaching the task for refactoring the code handed to us, we first tried to see if there were some of the methods doing more tasks and if we could divide these tasks into more methods or even new classes.

```
  public OldWebServer(int port, String filename) throws IOException {
    try {
      List<String> lines = Files.readAllLines(Paths.get(filename));
      int lastIndex = lines.size();
      for (int i = 0; i < lines.size() - 1; i++) {
        if (lines.get(i).startsWith("*PAGE")) {
          pages.add(lines.subList(i, lastIndex));
          lastIndex = i;
        }
      }
```

```
11      } catch (FileNotFoundException e) {
12        e.printStackTrace();
13      }
14      Collections.reverse(pages);
```

The above has been refactored into three classes called FileReader, Page and WebPages

our three classes called FileReader, Page and WebPages:

FileReader Class

- Responsible for reading the file content of the path "config.txt"
- The class has one field with an arraylist and two methods:

  AddFileContent Method:

  Reads the strings from the file into the arraylist

  getFile Method:

  Returns content of the arraylist of strings

Page Class

Each page object has a responsibility of containing websites in the form of an arraylist. **It has a file with an arraylist and three methods:**

- od: reads a string and puts it into the array
- getWords Method: returns the content of the arraylist of strings.
- getURL Method: gets the URL from the arraylist of strings **NOTE:** This method is only used in the PageComparator class to sort the webresults in ascending order

WebPages class

The overall responsibility of this class is to contain all the page objects and put them into a set of webpages.

It has 4 fields:

- List<List<String» - List containing lists of strings.

- HashSet<Page> – Will contain all the page objects in a set
- FileReader object – to read the file.
- List<String > - Will contain the filecontent (list of strings) that's read from the FileReader class

2 Task 2: Modifying the Basic Search Engine

Modify the program task: For task 2 we started by identifying where the output was generated and found it inside the code.js. in the getElementByID method.

Solution We changed the code by adding a simple if/else statement, so that if the data.length is == to 0 (meaning the result of the search), then the output should be "No web page contains the query word" instead of "0 websites retrieved":

```
%
document.getElementById('searchbutton').onclick = () => {
    fetch("/search?q=" + document.getElementById('searchbox').value)
        .then((response) => response.json())
        .then((data) => {
            console.log(data)
            if (data.length == 0) {
                document.getElementById("responsesize").innerHTML =
                    "<p>No web page contains the query word<p>";
            } else {
                document.getElementById("responsesize").innerHTML =
                    "<p>" + data.length + " websites retrieved<p>";
            }
```

Modify the reading of the code task

To complete this sub-task we created a new section under the "readFile" method in the WebPages class. We noticed that every URL has the same format, such that the title comes after the last occurrence of the character '/'. In order to make the title comparable with the actual title in the next line after the URL, we used the replace method to replace the underscores "_" with white spaces "\s", and then it is saved in the "wordsAfterSlash" local variable.

To make sure that the title is set, we use the if-statement to compare if wordsAfterSlash equals the nextString, which is the next index after the URL. Afterwards we check if linenumber+2 starts with "*PAGE" to ensure that every page has at least one word contained in its list.

Solution Here follows our solution for the task:

```
1    if (filecontent.get(linenumber).startsWith("*PAGE")) {
2
3        String targetString = filecontent.get(linenumber);
4
5            String nextString = filecontent.get(linenumber+1);
6
7            int lastWord = targetString.lastIndexOf('/');
8
9            String wordsAfterSlash = targetString.substring(lastWord + 1, targetString.
                 length()).replace("_", "\s");
10
11           String linefromfile = filecontent.get(linenumber);
12
13           if(linefromfile.startsWith("*PAGE") && wordsAfterSlash.equals(nextString) &&
14           !filecontent.get(linenumber+2).startsWith("*PAGE")) {
15
16             storingRawPages.add(filecontent.subList(pagenumber, linenumber));
17
18             pagenumber = linenumber;
19
20           }
```

3 Task 3: Inverted indexes

Approach For this task we had to implement an inverted index, which maps webpages to keys using the java datastructure HashMap. For future use we could have explored the possibility of using a TreeMap to compare benchmarking results. Due to time constraints we decided go with a HashMap.

Solution The first implementation was not optimized in regards to speed, where every webpage was iterated over immediately, which was very time consuming for the compilation. Instead we decided to put every webpage that matches a specific searchTerm into a HashSet to avoid duplicates. Afterwards the HashSet is put into the HashMap as a value to the specific searchTerm, where the searchTerm is the key in the map. As shown in the following piece of code:

```
1    public void makeInvertedIndex(HashSet<Page> websites) {
2
3
4        for (Page newPage : websites) {
5
```

```
            for (String word: newPage.getListOfWords()) {

                HashSet<Page> existingList = new HashSet<>();

                if(index.containsKey(word)){

                        index.get(word).add(newPage);

                } else {

                    existingList.add(newPage);

                    index.put(word, existingList);
                }

            }
        }
    }
```

Our search method called getSearchResults was then refactored by using a lookUp method from the InvertedIndex class. By inserting a searchTerm in the parameter of the lookUp method, the method will either return a HashSet containing the searchTerm or an empty Set.

```
    public HashSet<Page> lookUp(String searchterm) {

        if(index.containsKey(searchterm)){

            return index.get(searchterm);

        } else {

            HashSet<Page> empty = new HashSet<>();

            return empty;
        }
    }
```

Test To make sure that our new implementations had the right functionality, we implemented unit tests. An example of such test follows here.

```
    @Test void makeInvertedIndex_containsOneElement(){
        InvertedIndex tempindex = new InvertedIndex();
        Page temppage = new Page();
        temppage.addLine("hej");
        HashSet<Page> tempset = new HashSet<>();
```

```
        tempset.add(temppage);
        tempindex.makeInvertedIndex(tempset);

        set = tempindex.lookUp("hej");
        assertEquals(1, tempindex.getMapSize());
    }
```

The test checks if pages are added correctly to the Map inside the InvertedIndex.

4 Task 4: Refined Queries

Task 4 involves expanding the search engine to work with more complex queries, more specifically having the search engine taking more than 1 search term and having the search terms being split by the word "OR". We split the task into steps as advised, and started by implementing the support of multiple search words. For the implementation of supporting multiple search words, we created the **getSearchResults** method in the **SearchEngine class.** we identified that the string searchTerm came in the format of **%20 between each word. Eg. "Queen%20Denmark".** We used the split method to split the string at "%20" and saved it in a String array.

Example of the code below:

```
    public List<Page> getSearchResults(String searchTerm) {

      Map<Page, Double> siteRank = new HashMap<>();

      String[] subSearches = searchTerm.toLowerCase().split("%20or%20");

        for (String subsearch : subSearches) {

            String[] searchWords = subsearch.split("%20");

            setOfPages = query.buildPages(index, searchWords);

            computeTFIDF(siteRank, searchWords,setOfPages, index, query);

        }

      result = getDescendingListFromMap(siteRank,result);

      return result;


    }
```

This method uses helper methods from the Refined Query class such as buildPages, getOverlap (finding the intersection of websites in the HashSets of pages, where pages has to contain the same word/words) and the method will result in a List of pages that is going to be sent to the webserver.

5 Task 5: Ranking Algorithms

For this tasks we integrated both the TF(Term frequency) and the TF-IFD(Term Frequency-Inverse document frequency) ranking in the getSearchResults method in the SearchEngine class through the helper method computeTForTFIDF in the same class. Furthermore we have a final boolean field called TrueForTF that is set to false by default, as we are interested to show the TF-IDF score for the pages, but keeps the possibility of changing the boolean to true, if we are interested in the TF-ranking at some point.

the following method for computeTForTFIDF is shown below:

```
1   public double getTForTFIDF(boolean trueForTF, String searchTerm, Page page,
        InvertedIndex idx) {
2
3        if(trueForTF == true){
4           return tfscore.TFScore(searchTerm, page);
5         } else {
6           return tfidfscore.TFIDFscore(searchTerm, page, idx);
7         }
8     }
```

The TF class is used to calculate the frequency of a specific word or the impact of a word inside a page. This calculation is done using the searchTerm and the set of pages found by using the lookUp method in the InvertedIndex class. The TF score, which is the frequency of a word divided by the total amount of words inside a page, will calculated using the TFScore method.

Furthermore we made an IDF class to calculate the Inverse-document frequency. This class has a method called IDFScore which calculates the IDF by using the Math.log method on the calculation of the total amount of pages in the database divided with the amount of pages the searchTerm exist in.

These scores will be allocated to the TFIDF class, that will multiply the TF with the IDF to get the TF-IDF score of a page. Each page will get an TF-IDF score by storing the score in a page object.

We implemented this solution inside the getSearchResults method in the SearchEngine class. Furthermore, if a searchTerm consists of "OR", then it has to split by that occurence

and if a searchTerm contains for example "java OR asia" or "java asia OR borneo", then it has to pick the highest score or sum of the two sides split by "OR".

Further description of a given TF-IDF score to a page depending on the searchterm, using the results from our searchengine implementation, which is shown in the Appendix section in figure 1.

To solve the task of ranking the pages in descending order, we inserted each page as keys and a given TF-IDF score to the page as values in a HashMap. Afterwards we used streams to convert it into a list of pages in the correct descending order. To show how the pages are ordered, figure 2 in the appendix section shows results and their respective TF-IDF score ranked in descending order.

To compare the TF score to the TF-IDF ranking of a page, we provided the search results of "Queen Denmark" in figure 3 in the appendix section. The first 7 pages are the same when using the searchterm "Queen Denmark" but then as shown in figure 3, after the page "Alexandra of Yugoslavia" the difference on the ranking shows. As seen on the left side of the figure, we have the TF scores for the search result of "Queen Denmark". These pages are ranked according to the word frequency of both "Queen" and "Denmark", but not the amount of pages the word occurs in. So the TF frequency is very primitive ranking compared to the TF-IDF score.

On the right side of the figure, we have the TF-IDF scores of "Queen" and "Denmark", the middle part of the results list differs to compare to the TF ranking of the pages. That is because the TF has been multiplied with the IDF, which gives a more complex but precise result, as the number of pages found with the words are included. That is why we decided to use the TF-IDF score as default in our searchengine, as it gives a more precise calculation in the matter of ranking the pages.

Test To test if we assigns the TF and TF-IDF score correctly to a searchTerm consisting of either a single words or multiple words, that may be split by "OR", we implemented multiple test to show the validation of the implementation. Below is shown examples of two of our tests:

A test of the TF ranking to show that the first page contains a higher ranking than the last:

```
1
2 @Test void computeTForTFIDF_TFscoreTopscoreOverBottomscore(){
3         boolean TrueForTF = true;
4         Map<Page, Double> siteRank = new HashMap<>();
5         String[] searchword = {"java"};
6
7         setOfPages = query.buildPages(index, searchword);
8         se.computeTForTFIDF(TrueForTF, siteRank, searchword,setOfPages,index,query);
```

```
        result = se.getDescendingListFromMap(siteRank, result);

        Page bottomscore = result.get(result.size()-1);
        Page topscore = result.get(0);

        assertTrue(bottomscore.getCurrentScore() < topscore.getCurrentScore());
    }
```

A test of the TF-IDF ranking to show that the first page contains a higher ranking than the last:

```

@Test void computeTForTFIDF_TFIDFscoreTopscoreOverBottomscore(){
        boolean TrueForTF = false;
        Map<Page, Double> siteRank = new HashMap<>();
        String[] searchword = {"java"};

        setOfPages = query.buildPages(index, searchword);
        se.computeTForTFIDF(TrueForTF, siteRank, searchword,setOfPages,index,query);
        result = se.getDescendingListFromMap(siteRank, result);

        Page bottomscore = result.get(result.size()-1);
        Page topscore = result.get(0);

        assertTrue(bottomscore.getCurrentScore() < topscore.getCurrentScore());
    }
```

A test to show that a searchTerm split by OR results in the correct pages and the correct number of pages.

```
@Test void getSearchResults_with3MultiWordsSplitByOr() throws IOException{

        result = se.getSearchResults("President%20USA%20OR%20Queen%20Denmark%20OR%20
            Chancellor%20Germany");

        Page presidentUSA = new Page();
        Page queenDenmark = new Page();
        Page chancellorGermany = new Page();

        for (Page page : result) {
            if(page.hasWord("President") && page.hasWord("USA")){
                presidentUSA.addLine("President");
                presidentUSA.addLine("USA");
            }
            if (page.hasWord("Queen") && page.hasWord("Denmark")){
                queenDenmark.addLine("Queen");
                queenDenmark.addLine("Denmark");
            }
```

```
18              if (page.hasWord("Chancellor") && page.hasWord("Germany")){
19                  chancellorGermany.addLine("Chancellor");
20                  chancellorGermany.addLine("Germany");
21              }
22          }
23
24          assertTrue(presidentUSA.hasWord("President"));
25          assertTrue(presidentUSA.hasWord("USA"));
26          assertTrue(queenDenmark.hasWord("Queen"));
27          assertTrue(queenDenmark.hasWord("Denmark"));
28          assertTrue(chancellorGermany.hasWord("Chancellor"));
29          assertTrue(chancellorGermany.hasWord("Germany"));
30          assertEquals(91, result.size());
31      }
```

6 Conclusion

When coming into the project, none of the team members had any prior experience with webserver implementations or searchengines for that matter. We are overall satisfied with every members contribution to the project and the implementations of our InvertedIndex class and SearchEngine class, which are able to handle complex queries with a higher speed using a hashMap compared to the first untouched version, that we were given. Furthermore, every search results in a list of pages that are ranked in a descending order using the TF-iDF algorithm to give each found page a ranking score.

Every class and methods contains javadocs to explain the logic and use of each respective class and method. Furthermore, we also documented each test with a description.

The test coverage shows 77 percent (according to jacoco test reports) due to the fact that our WebServerTest did not work after new implementations. This is the case because the order of the Pages are loaded differently as compared to the original implementation. As none of us had written the Webservers methods initially we left it as was. We have done tests to all the none-trivial methods that wasn't originally there. (getters, setters and main). Additionally, we were told by our supervisor that we were not required to test the methods that wasn't done by us.

Due to time constraints, we would have liked to develop the implementation even further, as some pages are contains the same amount of words with the same occurence of searchTerm words like "queen denmark", which then does not fully rank and sort them in a complete manner.

Furthermore, we would also have liked to benchmark the speed of the program by comparing use of a hashMap versus a TreeMap in our inverted index.

We also had problems with committing to the github main repository during the project, but even though the number of commits is different for each member, every member has contributed equally. Every member has been present at every meeting, supervision and the writing of the code.

7 Appendix with figures

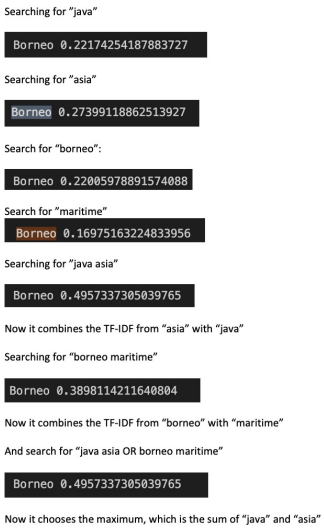

Figure 1: TF-IDF ranking examples

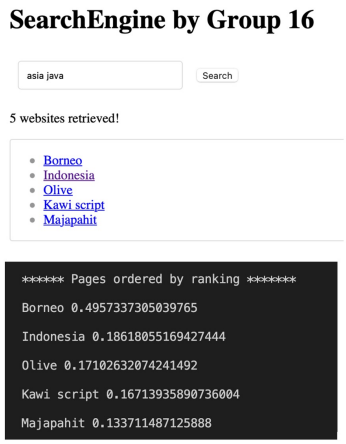

Figure 2: Example of search results being ranked in descending order with the highest sum chosen for page "Borneo"

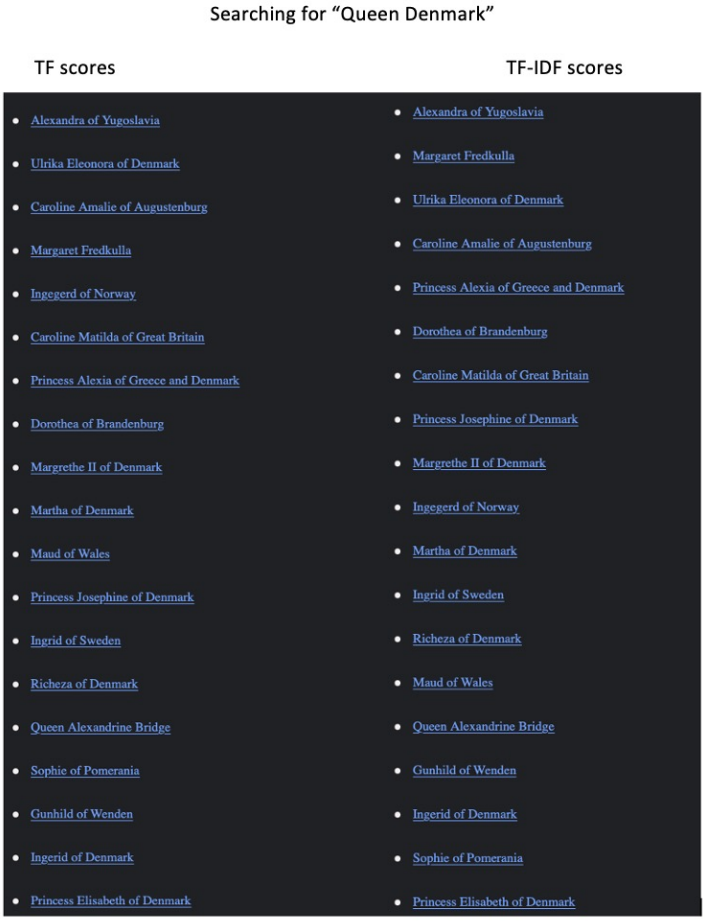

Figure 3: Example of search results being ranked in descending order by using TF and TF-iDF respectively