Project: Search Engine

Introductory Programming
Master of Science in Software Development
IT University of Copenhagen

November 19, 2018

Introduction

The goal of this project is to implement a large piece of software. This year, students will develop a web-based search engine. The project is divided into two parts:

1. A mandatory part where you create the core of the project.
2. Extensions to the basic core. Students who have completed all of the mandatory tasks can pick from proposed extensions, or work on their own extensions.

The project will be performed in groups, and will be implemented in Java. The project spans the last four weeks of the course. Each group is required to hand in a written report at the end of the project, which will form the basis for an oral exam.

Several software development tools & techniques will be used: version control (Git), testing (JUnit), debugging, documentation (Javadoc), benchmarking, build tools (Gradle), and code review. Becoming familiar with these tools & techniques is a central goal of this course.

The following sections describe the project in detail. Do not be discouraged by unfamiliar terms in the problem description; everything that that is needed to complete this project has already been covered in class.

Background

Search engines are basic tools in our everyday lives. Their task sounds simple: Given a query string, retrieve all websites that "match" the query string (from a database of websites), and rank them according to some measure of "importance" with respect to the query string. Achieving this goal seems straight-forward, but there are plenty of challenges to overcome, see e.g. the poster[1] of Google. The mandatory part of this project will allow you to provide basic solutions to these challenges in a guided way. More advanced features can be developed in the extension part of this course. This section provides some background on the general structure of the search engine program that you are going to develop.

Database of Websites

The database of websites is represented as a "flat" file. Consequently, the search engine program takes a filename as an argument. Each file lists a number of websites including their URL, their title, and the words that appear on a website. In particular, a website starts with a line "*PAGE:" that is followed by the URL of the website. The next line represents the title of the website in natural language. This line is followed by a list of words that occur on the page. The following example illustrates the file format:

Example File

```
*PAGE:http://www.exampleA.com/
Website A's title
Here
is
a
word
and
one
word
more
*PAGE:http://www.exampleB.com/
Website B's title
Here
is
more
and
yet
more
```

[1] `https://static.googleusercontent.com/media/www.google.com/en//insidesearch/howsearchworks/assets/searchInfographic.pdf`

Note

If a website entry contains less than two lines after the "*PAGE" line, i.e., it has either no title or no words, the entry should be **omitted**.
(We have to assume that the entry for this website is erroneous.)

We provide access to a dataset extracted from the English wikipedia (`http://en.wikipedia.org`). Each dataset has different sizes, as detailed below:

Wikipedia

- `enwiki-tiny.txt`
  `(4kb, 6 websites)`
- `enwiki-small.txt`
  `(544kb, 1033 websites)`
- `enwiki-medium.txt`
  `(11.6mb, 25011 websites)`
- `enwiki-large.txt`
  `(328mb, 25033 websites)`

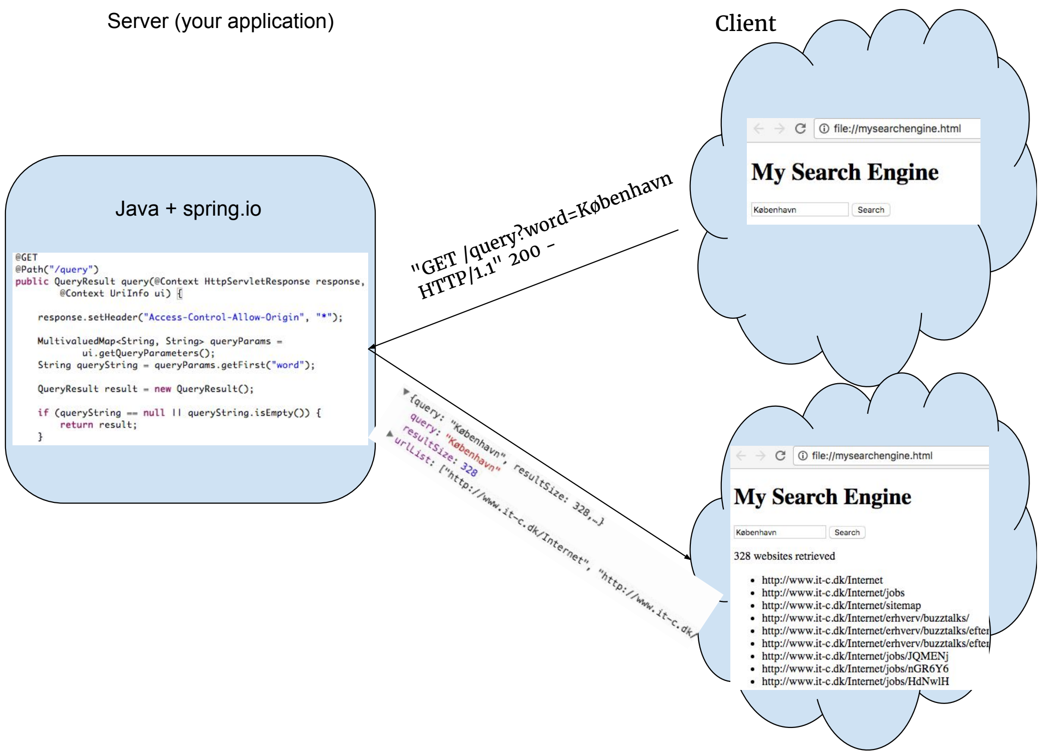
Answering a Query

Assume the user of your program wants to query the collection of websites with a query string. From an abstract point of view, the search engine has to do the following:

Recipe to process a query

1. **Check** the query. ("Does the query make sense?")
2. Retrieve the list of websites that **match** the query.
3. **Rank** these websites according to their importance with regard to the query.
4. **Return** the list of ranked websites.

Your project starts with a very basic setup: A query consists of a single word, everything else is invalid. A website matches a query when the query word is contained in the list of words found on the website. No ranking is going to take place.


Server (your application)
Java + spring.io
Client
"GET /query?word=København HTTP/1.1" 200 -
file://mysearchengine.html
My Search Engine
København
Search

```
@GET
@Path("/query")
public QueryResult query(@Context HttpServletResponse response,
        @Context UriInfo ui) {

    response.setHeader("Access-Control-Allow-Origin", "*");

    MultivaluedMap<String, String> queryParams =
            ui.getQueryParameters();
    String queryString = queryParams.getFirst("word");

    QueryResult result = new QueryResult();

    if (queryString == null || queryString.isEmpty()) {
        return result;
    }
```

```
{query: "København", resultSize: 328,…}
  query: "København"
  resultSize: 328
 urlList: ["http://www.it-c.dk/Internet", "http://www.it-c.dk/
```

```
328 websites retrieved

• http://www.it-c.dk/Internet
• http://www.it-c.dk/Internet/jobs
• http://www.it-c.dk/Internet/sitemap
• http://www.it-c.dk/Internet/erhverv/buzztalks/
• http://www.it-c.dk/Internet/erhverv/buzztalks/efte
• http://www.it-c.dk/Internet/erhverv/buzztalks/efte
• http://www.it-c.dk/Internet/jobs/JQMENj
• http://www.it-c.dk/Internet/jobs/nGR6Y6
• http://www.it-c.dk/Internet/jobs/HdNwlH
```

Figure 1: Overview of the client-server architecture. The client sends the query to the web server (using HTTP GET) and receives an answer (in JSON) that is subsequently displayed.

Prototype

You will start with the source code for a simple search engine. The program takes the name of a data file and tries to read this file according to the rules stated above. Each website is modeled as an object of the class *Website*. It will construct a list of Website objects. After constructing this list, a simple user interface launches, allowing the user to query the search engine: Given a word, it outputs the urls of all websites that contain this word.

A search engine for websites should of course be working in its native environment: inside a web-browser. We will provide a setup that makes your prototype into a web service (following a RESTful API using Spring Boot[2]), and a web-browser client for this web service. The client is written using HTML/CSS/Javascript. The client sends specific messages to your server application and expects the query result to be provided in a certain way. A schematic representation is given in Figure 1.

[2]details: `http://restfulapi.net/` and `https://spring.io`

Mandatory Tasks

The project has three mandatory tasks. These tasks should be solved consecutively, i.e., in the second task you will expand on the work you did for the first task, and so forth.

You are encouraged to use functionality that is already implemented in Java. In particular, data structures provided in `java.util`[3] and the methods for String objects in `java.String`[4] will turn out to be useful. **If you think the solution to a task involves a very complicated piece of code, consult with your TAs/teachers first!**

Note

Some tasks contain "Challenges", marked in green. Challenges are **not** mandatory tasks; they are optional extensions that you are welcome to include in your report.

Software Development Techniques

We have learned about several software development techniques in this course, in particular unit testing using JUnit and technical documentation using Javadoc. **Your final product must provide unit tests and technical documentation for all mandatory tasks.**

Getting Started

To get started, clone the following Git repository, with your group letter in place of `Z`.

```
https://github.itu.dk/wilr/ip18groupZ
```

This repository contains a copy of the search engine prototype, built with Gradle. The repository is specific to your group, so you are free to modify it as you see fit. **You must host your project in this repository**. This not only eases collaboration within your group, but also make it easier for your TAs and teachers to provide you with feedback. For details on how to use Git, Gradle, and IntelliJ, please consult the course website.

Once you have the prototype, and have successfully compiled and run the prototype, start by adding the following **necessary** basic functionality to the search engine.

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html`
[4] `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`

- Modify `FileHelper` such that it only creates a website if both the title is set and the word list contains at least one word.

What you need to know to solve this step

- familiarity with file I/O, and with lists

Task 1: Faster Queries using an Inverted Index

In this task, we explore ways to make the query algorithm more efficient, by providing a better way to search for websites containing a certain word.

The query algorithm performs lookup operations on an index of websites. The prototype index, given by `SimpleIndex`, is very inefficient: It simply iterates through the list of websites and collects those that contain the query word. An *inverted index*[5], on the other hand, allows for better performance by maintaining a mapping between words and a list of website objects that contain these words. See Figure 2 for an example.

We explore different ways to implement an index. To facilitate this exploration, we provide an interface `Index` that specifies what an index needs to do. `Index` specifies two methods: `build`, which builds an index from a list of websites provided as parameter, and `lookup`, which returns for a given query word the list of websites that contain the query.

Your first step is to provide alternative implementations of an index.

- Complete the implementation of the provided classes `InvertedIndexTreeMap` and `InvertedIndexHashMap`. These classes should implement an inverted index, using a `TreeMap` and a `HashMap` respectively.
- Use one of these new indices in `SearchEngine` in place of `SimpleIndex`.

You should now have three interchangeable indices: `SimpleIndex`, an `InvertedIndex` backed by a `TreeMap`, and an `InvertedIndex` backed by a `HashMap`.

[5]https://en.wikipedia.org/wiki/Inverted_index

$$\begin{tikzcd}
\boxed{\text{word}_1} \arrow[r] & \{\text{site}_{1,1}, \text{site}_{1,2}, \ldots, \text{site}_{1,m_1}\} \\
\boxed{\text{word}_2} \arrow[r] & \{\text{site}_{2,1}, \text{site}_{2,2}, \ldots, \text{site}_{2,m_2}\} \\
\vdots & \\
\boxed{\text{word}_n} \arrow[r] & \{\text{site}_{n,1}, \text{site}_{n,2}, \ldots, \text{site}_{n,m_n}\}
\end{tikzcd}$$

Figure 2: Abstract view of an inverted index for a database of websites containing n words in total. Each word is assigned to a list (represented as $\{\ldots\}$) of website objects that contain the word. In our notation, word "word$_i$" is contained in exactly m_i websites.

What you need to know to solve this step

- understanding interfaces and abstract classes
- using `java.util.Map`
- ability to build an inverted index (graphically) from a given set of websites, e.g., for three websites containing the words "copenhagen university", "university", and "copenhagen of" you should be able to draw and explain the mapping:

$$\begin{tikzcd}
\boxed{\text{copenhagen}} \arrow[r] & \{\text{website}_1, \text{website}_3\} \\
\boxed{\text{university}} \arrow[r] & \{\text{website}_1, \text{website}_2\} \\
\boxed{\text{of}} \arrow[r] & \{\text{website}_3\}
\end{tikzcd}$$

Your next step is to design a small experiment that evaluates these indices with respect to the running time of the `lookup method`. This will gauge how the choice of underlying datastructure affects the performance of an (inverted) index.

For this, we have provided `IndexBenchmark`, which is a skeleton of a JMH benchmark that you can build with Gradle. Your task is to modify and use it as follows:

- Modify `IndexBenchmark` such that it uses a list of query words of 10–20 words that are used for calling `lookup` on the specific index.
- For each of the three indices: Modify `SearchEngine` to use that index, run the benchmark, and save the output in your repository (include and discuss these in the report).
- Modify `SearchEngine` to use the fastest index from now on.

Note

The relevant part of the output has the following form:

```
Benchmark                          Mode  Cnt       Score    Error  Units
IndexBenchmark.measureAvgTime      avgt   xx     xxx.xxx ± xx.xxx  ns/op
```

For debugging, Gradle also outputs (if you scroll up) minimum and maximum time, standard deviation, etc.

Task 2: Refined Queries

This task will allow your search engine to work with more complex queries, as it is common in a search engine.

Example

The query

President USA OR Queen Denmark OR Chancellor Germany

should return all websites "containing" at least one of the following word sequences:

(i) President USA

(ii) Queen Denmark

(iii) Chancellor Germany

Here, "containing" means that all of these words can be found on the website (but they do not have have to occur adjacent to each other).

More specifically, your search engine must do the following for a query string `str`:

Multiple Words If `str` has the form[6]

$$\mathrm{w}_1\mathrm{w}_2\ldots\mathrm{w}_n,$$

i.e., it contains n words split by a blank "␣", a website matches the query string if and only if it contains these n words. (However, we don't require that they are adjacent to each other or in order.)

[6]We assume that "OR" does not occur as a word in the database files.

Merging via OR Suppose that `str` can be decomposed into parts split by the keyword “OR” as follows:

$$\mathrm{w}_{1,1} \ldots \mathrm{w}_{1,m_1} \mathrm{OR} \mathrm{w}_{2,1} \ldots \mathrm{w}_{2,m_2} \mathrm{OR} \ldots \mathrm{OR} \mathrm{w}_{n,1} \ldots \mathrm{w}_{n,m_n}$$

A website matches the query if there exists an index i between 1 and n such that the text on the website contains each word $\mathrm{w}_{i,1}$, $\mathrm{w}_{i,2}$, …, w_{i,m_i}. This means that at least one of these word sequences has the “Multiple Words” property from above.

Advice: Start slowly and extend the functionality step-wise. Try to keep your code extendable. For example, you could implement these features in the following steps:

(i) Support queries “$\mathrm{w}_1\mathrm{w}_2 \ldots$”, without the “OR”.

(ii) Support a single “OR” in the query.

(iii) Support multiple “OR” words in the query.

It is recommended to put all the logic in `QueryHandler`. In the `getMatchingWebsites` method, first, decompose the query into its components. For a single word, the query still retrieves lists of websites from the inverted index. For the multiple words feature, a collection of lists must be checked for websites that appear in all lists. The “OR” feature is easiest implemented by merging the results of multiple word queries. Make sure to remove duplicates, i.e., a website is only allowed to be part of the result at most once!

Challenge

1. Add one or both of the following two features to your query engine:

   URL Filter If `str` contains a word starting with “site:”, let `url` be the characters following “site:” until the next blank “␣”. A website matches the query if and only if its URL contains `url` as a substring. All words starting with “site:” have to be removed from the actual query string.

   Prefix Search If a word ends with a “*” character, this word matches all words that start with the string before the “*” character. For example, the word “ho*” matches “hole”, “home”, “house”, ….
2. Given that an n-word multiple-word query means to intersect lists of sizes $S_1, \ldots, S_n$, in which order would it be most efficient to intersect them?
3. Implement the query engine using an implementation of `java.util.Set`. Report on performance differences you observe via a small experiment.
4. How long does it take to answer a multiple-words query with n words, assuming that word w_i yields a result list of size S_i? Assuming that the input contains $m-1$ “OR” keywords (and thus m multiple-word queries), how long does it take to merge the m result lists of size $M_1, \ldots, M_m$?

What you need to know to solve the task

- good understanding of the methods provided by `java.util.List` and `java.util.String`, or perhaps the `java.util.regex` package.
- this assignment usually requires more "thinking about how to do it" than "actual coding". Before starting to solve the assignment, make up some examples and discuss how the process of answering the examples should look like.

Task 3: Ranking Algorithms

In this task you will improve the results of your search engine by providing ways to rank a list of websites with regard to their importance for a specific query.

The idea is simple: If a website S contains a word w, we compute a number $s(w, S)$ (the "score") that shows how important this word is on the website. (The higher the number, the more important the word). You can have a look at Wikipedia[7] to see the descriptions of some well-known approaches.

Example

Assume the query is

President USA OR Queen Denmark OR Chancellor Germany

Further assume we come up with the following scores of the words on some website (e.g. a news article) (these scores are all made up here):

President	USA	Queen	Denmark	Chancellor	Germany
12	12	1	26	2	3

Computation of score:

- each part (split by **OR**): add scores together

President USA	Queen Denmark	Chancellor Germany
24	27	5

- final score for website with respect to query: take the maximum!
- score for this website for the query is **27**

[7] `https://en.wikipedia.org/wiki/Tf-idf`

1. Add an interface `Score` to your program. It should have a method `getScore` that takes a string, a Website object, and an Index object, and returns a floating point number, for example `double` or `float`.

2. Implement the term frequency score in a class `TFScore`.

3. Implement the term frequency-inverse document score in a class `TFIDFScore`.

4. Modify your program such that each website that matches the query is assigned a score. For a multiple-word query, the score of the website is the sum of its score values for all words in the query. For an OR between two multiple-word queries, the score of the website is the **maximum** of the scores in these two queries.

5. Modify your program such that the list of websites that match a query is sorted by their score to the query in descending order. (Use methods provided in java.util!)

6. Provide queries and their results that show differences in the result quality, obtained by using these two score functions. Give a short description of how they differ.

Challenge

Implement the Okapi BM25 ranking algorithm.
(`https://en.wikipedia.org/wiki/Okapi_BM25`)

Hint

One way of implementing ranking algorithms is to store (tentative) scores in maps that assign each website to its score. To convert this map into a list with elements being sorted in descending order by their score, please use:

```
//Convert Map<Website, Float> map to List<Website> sorted by value
map.entrySet().stream().sorted((x,y) -> y.getValue().compareTo(x.getValue())).map(
    Map.Entry::getKey).collect(Collectors.toList());
```

What you need to know to solve the task

- purpose of an interface and how to create and implement one
- math operations such as taking the logarithm
- think about a clean way to integrate the scoring into the query engine

Extensions

This section introduces ideas for extensions you can work on after finishing all mandatory tasks. You are encouraged to work on your own ideas, but are required to **discuss them with a TA/teacher** first.

Solve Challenges in Mandatory Tasks. You may pick any challenge from the mandatory task part and solve it.

Improve the Client GUI. Change the client code such that the result of searches are displayed in a nicer way. (for instance, provide a title of the page is provided, a short summary of the text displaying an occurrence of the search string, add pagination to show only a certain number of results on the front page.)

Add an Autocompletion Feature. Add an autocompletion feature that proposes search words to the user. This should be solved by a client/server interaction as in the normal query procedure. You should make yourself familiar with JQuery UI to solve this task with only very few lines of Javascript.

Personalized Queries. Add a feature to the server such that it stores search queries in a file and uses previous searches in the autocompletion feature.

Make Your Own Web Crawler. Provide a program that starts with a given URL and builds a collection of websites by following hyperlinks on the website. Parse this into a flat file format, but modify the format such that it also allows you to store the hyperlinks that are present on a website.

> **Note**
>
> To avoid problems with your provider and/or being blacklisted from webservers, make sure to connect to only few (e.g., 30) websites per minute.

Implement PageRank. After writing your own web crawler, you can implement the PageRank algorithm to rank websites. (That is the algorithm which is the base of Google's web search.) See `https://en.wikipedia.org/wiki/PageRank` to get an overview of the algorithm. Compare the pagerank ranking to other rankings obtained by methods you implemented in Task 3.

Provide a "Fuzzy-Search" feature. If a search produces no result, it might be because the user performed an erroneous keystroke, so that, e.g., an "a" became an "s". In such cases you can choose to return pages which contain words almost matching the word searched for. A general technique to measure the similarity of two words is the Levenshtein distance[8]. To build an an efficient index, you may consider to use a "q-Gram index", see, e.g., the lecture notes of Prof. Hannah Bast[9].

Finding Similar Websites. Design a tool that, given a website A, will search for web sites similar to A. Here, a clustering algorithm such as K-Means[10] using the vector space model[11] can be the algorithm of choice.

Checklist for Mandatory Tasks

Use the following checklist to make sure your project fulfills all mandatory tasks.

- ☐ Tasks 1–3 have correspondent Java code that solve them, in particular,
  - ☐ you have implemented the inverted indexes, and benchmarked them,
  - ☐ your searchengine understands complex queries,
  - ☐ your searchengine is able to rank results to a query, using different scores.
- ☐ each class and each method is documented using Javadoc
- ☐ each class is accompanied with a unit test, in which you test the public methods exposed by the class (no tests need to be provided for main and "getter" methods)

Report

Your group should hand in a report covering at least the mandatory tasks. You can find out how to hand in written work in the study guide[12]. The report should start with the standard front cover[13], followed by a table of contents and a foreword. The foreword should shortly

[8] https://en.wikipedia.org/wiki/Levenshtein_distance
[9] https://daphne.informatik.uni-freiburg.de/ws1516/InformationRetrieval/svn-public/public/slides/lecture-05.pdf
[10] https://en.wikipedia.org/wiki/K-means_clustering
[11] https://en.wikipedia.org/wiki/Vector_space_model
[12] http://studyguide.itu.dk/SDT/Your-Programme/courses-and-projects/submitting-written-work
[13] http://studyguide.itu.dk/SDT/Your-Programme/Forms

(in 1 page or less) describe what is covered in the report, i.e., which tasks and extensions you managed to solve and implement, and which group member contributed to the solutions.

The following structure of the report is advised:

- Cover page
- Introduction

1. Walk-through for Mandatory Task 1
2. Walk-through for Mandatory Task 2
3. Walk-through for Mandatory Task 3

4.–... Extensions

For each of the mandatory tasks you should:

1. Describe how the task was solved, which data structures were used, and how they were used (when applicable). This description must be in natural language and also explain the architectural choices that were made in order to solve the task. Students are advised to include UML diagrams. Java code may only be used to support the description. Students are encouraged to provide figures and examples.
2. Reflect of the choice of data structure or algorithm used in solving the task. This can, e.g., be achieved by providing benchmark results for running times. Give a precise statement of the experimental setup and reflect on parameter choices used in the benchmarks.
3. Describe how you tested your code for correctness. Include a pointer to unit tests that you wrote.

The report should be no longer than 40 A4 pages (excluding the appendix).

In addition to the project report you should hand in the source code for **each relevant version** of your search engine as a **single zip file**. It should be clear what code belongs to which task (for example by using a separate folder for each version) and clear instructions on how to run the code should be included. Make sure that your code compiles, is well-documented, and contains unit tests for all non-trivial methods.