Introductory Programming 2021 Final Project: Search Engine

Group 13

Emiliano Gustavo Giusto
Fabian August Clement Harlang
Fernando Sabater Miara
Klaus Engell Lundstrøm
`{emgi,faha,fers,kllu}@itu.dk`

February 20, 2025

Introduction

This report describes Group13's Search Engine project for the Course Introductory Programming on ITU Copenhagen. The document will address different parts of our projects in the following order:

- **Refactoring:** A short description of our refactoring process and the new classes implemented.
- **Search Engine:** Modifying search engine ("no query result", stop words and stemming processing).
- **Inverted Index:** Implementation of inverted index for enhanced search performance using HashMap and Trie data structure.
- **Refined queries:** Working with logical operators (AND/OR) and Regex in search queries.
- **Ranking and score structure:** Ranking Algorithms and search relevance.
- **Extensions**: Extra functionalities we have added to the project (front-end and auto-completion).
- **Conclusion:** General thoughts about the project's solution, including known subjects of improvement (shortcomings, further implementation etc.) .

Project's code base is downloaded from our repository and compressed as a single zip file called `group13.zip`.

The code can furthermore be viewed and forked on ITU's Github: `https://github.itu.dk/emgi/IP-project/`. Here it's also possible to get an overview of commits and other project insight.

Distribution of task

We have utilized project management tool Trello to keep track of various tasks assigned to each team member. Our code base and version control is being managed in a GitHub repository using remote subbranches that are later merged to the master branch. We roughly split up the project responsibility in categories as follows:

Code base contribution by team members:
Creating alternative Indexer: Fernando
Inverted index and ranking: Emiliano
Score and refined queries : Klaus
Front-end and auto-completion: Fabian

However, many tasks have overlapped and everyone has done their part to equally contribute to the code base and/or documentation of the project. Testing and documentation were distributed according to relevance for team members main area responsibility i.e "describe what you've coded as a javadoc comment and test with JUnit". A more detailed and elaborate description of these matters are reserved as objective for this report.

Coding collaboration:
During this project we collaborated in a common repository by creating branches and pull requests. Our master branch was configured as protected and only accepted changes through a pull request with at least one approval from another collaborator (group member).

1 Task 1: Refactoring (creation of new classes)

Description
As the first activity of the project, we went through the code and it's different classes together to comprehend it's structure and logic. We realized that the Web server class had redundant and irrelevant code for the functionality of merely initiating a web server. It was therefore decided to allocate these lines of codes to a Search Engine class and refactor the code with principles of low coupling and high cohesion. We decided moreover to dedicate an entire sub folder (dba) for classes relevant to "the database" i.e classes manipulating with the indices, words and queries fetched from the /data/enwiki-(...) text files. Classes were also added to our custom Java package 'searchengine' for the sake of code resuse and inheritance.

Approach
After installing the Gradle framework and had the web server running on localhost:8080 we tested the generic project code. Line by line code was moved to the Search Engine class ensuring no exceptions were thrown and the code would still compile after refactoring it. Relevant comments were added to the class followed by testing of the web server with the already existing web server test class, we were provided with, to make sure the HTTP service was still running as expected.

Result

After completion of above steps we had a cleaner and more semantic structured code (each class does what the naming implies) that was easier to read and would serve as a better starting point for proceeding with the project's other tasks than initially. The refactoring process furthermore gave us an opportunity to think about abstraction and implementation of the other classes yet to be made and how they would interact with one another.

2 Task 2: Modifying search engine

Description

After allocating the Search engine class to it's own file we looked into to possible improvements. First of all the prototype version didn't return an error message if no results were found for a specific search term. That was the first thing we decided to implement. Next, we wanted to get rid of the "stop words" for more efficient searching i.e removing articles and proposition words. Finally we wanted to utilize "stemming processing" to reduce words to their roots to achieve better search result with less search term sensitivity.

Approach

No results found
Search queries are being fetched through a GET HTTP request to the "/search" endpoint and the response is returned in JSON format. While the search results are loading we implemented a loader animation. Then the length of the data is being reviewed by our code. If it equals 1 then innerHTML is changed to "1 website is found", otherwise "x websites are found". If no results were found for the query and exception is thrown and the #responsesize element displays the message "Sorry, no results found".

Stop words
The search engine is set to divide the search input received from the back-end first based on OR operators and second by the spaces between words. Once this list is conformed, it is filtered removing all "stop words"; words that have no real meaning in a sentence in English language (i.e.: for, at, in, on, by, among others). All words were moreover transformed into lowercase and surrounding symbols i.e. $(:, ?/;)$ etc. were removed.

Word stemming
Given an arbitrary search term like “cat”, the search engine should respond with relevant web pages containing that specific words, however sometimes similar variations of the words like “cats” or “catty” instead of “cat” might be as relevant as the exact search term

but won't retrieve the desired web pages in the search result. A solution to this is to enhance the search engine with the process of "stemming".

Stemming is the notion derived from natural language processing where a word is reduced to it's root form i.e no prefixes and suffixes. For this task, a library provided by Carmen Alvarez (2016) was used featuring a version of a Porter's Stemmer. A few minor corrections were made on Stemmer Java class to improve the functioning to our case.

Result
In general these improvements made the search engine more dynamic by making it susceptible to similar (but different) search terms as well as filtering out redundant "stop words". Most importantly it would now also return an error message if no result were found. These initial steps made the code ready for further implementation namely inverted index and more advanced queries.

3 Task 3: Inverted index (HashMap and Trie data structure)

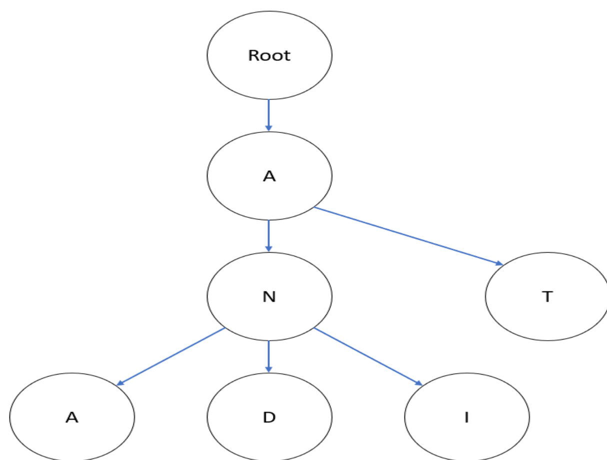
Approach
On our approach, inverted indices can either be build based through a `Trie` data structure or a `HashMap` data structure. We decided on implementing and benchmark both to see which alternative would be more efficient for our project.

Description
Prototype version simply iterated through a list of web pages to find all entries matching a given query. This method is not very efficient. An inverted index however is the best practice data structure for document retrieval systems like a search engine. The idea is to map search terms to web pages in which they appear using a paired HashMap structure like:
$hello(1, 1)$
$world(1, 2)$
The objective of this task is to reduce the loading time for the search engine to respond to a desired query as computers process number comparison faster than string comparison. Enabling such functionally will thereby make a faster assessment of which web pages are relevant for the search term. Our index has a word-level structure implemented with a HashMap which is automatically handled by the Trie data structure

Trie data structure
After thorough research, we found out that a trie is, theoretically, a more efficient information retrieval data structure, and hence it might have a better searching performance in our project. Before its implementation, we thought it could also be useful to have an efficient

functionality to retrieve web pages while the user was typing with a dynamically typed search while we were showing the results as we were going down the nodes, but unfortunately we did not manage to complete this additional task on time. The main advantage of implementing this data structure is its search complexity, which has time O(m), where m is the length of a given string, whereas a Hashmap can vary from O(N), where N is the number of key-value pairs present and it would be the worst time complexity, up to O(1). On the other hand, the greatest constraint of a trie is related to its memory usage, where every character has to be allocated to memory, instead of utilizing a chunk of it for the whole entry as in most hash tables (quote: https://en.wikipedia.org/wiki/Trie).

With regards to the coding process followed, we created a new branch called trie-search where the implementation was done, and we ended having merge conflicts with the main branch we could not resolve due to changes in multiple files across the project. In the end, we did not merge that branch, but instead we created a new one called trie-merge-master. The trie data structure is made up of nodes and leafs, where the first character of a given string is inserted into the first node and then, the second character will be assigned to its children node. In case no such path exists, it will be created by the putIfAbsent method. When the last character is reached, hence the last assignable node to this word is reached as well, the status of it will be changed to "leaf", by setting the isLeaf field as true. If it is the first time we reach such stage, then a new word will be created for the leaf, if not we will just add an occurrence to the word. In terms of the searching algorithm, we are iterating over the given passed in key, and we are getting the child node which contains the following character, until we reach the leaf. Null is returned in case such key was not founded.



The trie data structure: Nodes and leafs

Results

We conducted a manual benchmark by getting the starting and finishing times of the search method for every search, and these were the results:

	HashMap		Trie	
Search term	Results	Time	Results	Time (s)
danish or argentina or tesla or calculator or somalia or lexicon	5004	0	5365	0
united states	16362	11	16362	16
technology or analog	4255	0	4513	0
a	0	0	12	0
russia or japan or elon musk or camera or security	8671	1	8855	0
john or william or juan	7439	1	7739	1
english or wikipedia or chinese or europe	12676	3	12774	3

HashMap / trie structure

There were some discrepancies on the number of web pages retrieved in case a complex search was requested, which affected our performance in case more or less results were returned. Despite these, we could not find significant time variations in terms of searching for a given word, and a better all around performing data structure through our manual benchmark in the time given, and although theoretically the trie should have performed better, our results were inconclusive. Namely, for the search term "united states", the HashMap data structure took 5 seconds less than its counterpart, but on the search "russia or japan or elon musk or camera or security" the trie took 0 seconds whereas the HashMap took 1 second. In terms of future optimizations, we would focus on the performance of the sorting algorithms and reducing the discrepancies on the number of web pages retrieved. As a conclusion for this task, we saw an opportunity to enhance our learning experience by undertaking a challenging approach through this new algorithm, which exceeded the contents of the course but we thought it might be interesting to explore.

4 Task 4: Refined Queries

Description
At first keyword combinations were not possible for the search engine to handle. To implement more complex queries, that would display web pages that contain at least one of the words e.g
$(Word1 \quad Word2 \quad Word3)$ / $(Word1 \quad OR \quad Word2 \quad OR \quad Word3)$

additional coding had to be done. To enable this multiple words property as a function, we went from allowing words that were split by blanks and with no requirement of being adjacent to then allowing two words separated by OR and finally multiple words separated by OR.

Approach

Multiple words and merging via OR

In our approach to this task, we ended up creating a class (Search Engine) that covers both task 4 and 5. In this part we will cover how we processed the input. We started by breaking down the input string according to two factors. The first one was "OR", and the second was " "(space between input words).

At first we split the input string according to the OR operator. Further along this will help us when we calculate the score on both sides of the OR, and then process the element with the highest score. We stored the disjointed input into multiple Lists. Furthermore we then iterated through the elements of this list, to split the strings with the " " variable. To deal with spaces in an input string, we had to use the variable "

In order for our code to accept inputs with more than one word, we dealt with spaces between words as an "and". When the inputs were separated, there were furthermore stored in multiple Lists.

Result
These Lists became useful for calculating the score of an input, and matching the score to the input.

After the splitting and storing of the original input, we ended up merging the lists into a final hashMap that matches the web pages to the score.

5 Task 5: Ranking Algorithms

Description

Nowadays, search relevance is alpha omega as beside holding the property of conveniently showing users the most relevant search result for their search term it's also a whole field of its own (Search Engine Optimization) that has an increasingly commercial value as more businesses are gaining customers online.

The question is meanwhile: How can the web pages be displayed in an adequate order of relevance? Ranking Algorithms can take care of this. The idea is to assign a value to each page based on the number of occurrences that the combination of words input by the user has.

Approach

In our project the model used for ranking calculation was tf–idf. The acronym stands for "term frequency–inverse document frequency", which states that the score of a word in relation to an specific page where the word is present is equal to the number of occurrences of that word in the web page divided by the number of total occurrences of that word in all pages in the data set (in our case, our local database). This score has a close relation with relevancy according to this model.

After all tf–idf scores has been determined for all pages related to an specific term, then we must combine it with all the calculated scores for the set of web pages retrieved from other terms (in case the input has multiple words). In that particular case, the calculation is based on which logical operator is linking those words: In words joined by AND operator (an space in the input) the resulting score of an specific web page is the addition of the scores of that web page for both words; in contrast, in words joined by OR operator, the resulting score is the maximum of them.

Result

Finally, once all calculations by the tf-idf model are made, each web page that can be found in the result set will have a positive number mapped to it and that's the criteria for order them by descending score (or relevancy).

6 Extensions

Auto-completion using JQuery

As an extra feature we decided to implement auto-completion using Jquery library. The Initial version of this extension simply worked by hard-coding a bunch of keywords into a JavaScript array and then use these as source of suggestions to any input values of the #searchbox element. Later on, the extension advanced to using a copy of the local data txt-files as source instead. This decision, however created two issues. One of wrong formatting (Jquery accepts only object format) and another one of repetitions (the data source has many repeated words). To solve the formatting issue a "split-string by new line method" was implemented. All data is now stored as array objects. To avoid repeated keywords being suggested, and thereby solving the second issue, the array is being filtered using a filter function that returns only unique keywords to a final variable. For performance purposes the script was moreover adjusted to trigger only if the length of the search term is ≥ 3. Less characters than that would typically be too few characters to correctly "guessing" the search term and would beside that also return too many misleading suggestions that would only deteriorate the response time of the search engine.

Key event

The initial search engine would only process the queries as an 'onclick-event'. This was extended with an 'onkey-event' allowing the input value to be submitted on enter. This functionalities is assimilating other common search engines.

Improved Client GUI

Changes have been made to the overall appearance of the search engine. The search box has been centered and increased in height and width to achieve a more modern and responsive design. The styling was made using Bootstrap 5 framework. The background color has a linear gradient from blue to green and the font style was changed to Roboto. Various other CSS improvements were made to other elements including buttons, headlines and links.

7 Conclusion

This project has given us insightful knowledge on how a search engine works and very useful and essential tools on how to improve it. We've become acquaintance with relevant document retrievals terminology like: stemming, inverted index, regular expressions and ranking algorithm and even more importantly we've explored more advanced functionalities of GitHub by collaborating with remote branched with pull requests and mutual reviewing processes. From the prototype version we have managed implementing a much more sophisticated search engine that now is extended with following enhancements:

a) Utilizes an inverted index based on the hashMap trie data structure for better search performance and document retrieval

b) Allowance of more advanced input queries and multi-word properties to be processed using regular expressions.

c) Ranking of the different matching web pages in descending order based on the score combination of a given query.

d) Better overall GUI with improved design, auto-completion and key events.

8 Shortcomings and subjects of further improvement

Although we made many improvements to the code there are still certain areas that could be subject for further improvements. Some functionalities we would consider for an extended version includes:

Exact search term option: Majority of document retrieval systems allow input value in quotation which initiate a search query that is responded with documents only in which the entire string is contained.

Page description below headline: Popular search engines like Google displays an abstract of the page below the headline. This is efficient when searching for information as it helps the user assess relevancy.

Pagination: Some of our search results are way above thousands. The list seems infinite if you don't have a maximum of results per page. This feature would therefore be worth implementing.

Real database: This might be beyond the scope of the course since we haven't officially been introduced to databases design yet. However, it is obvious that "flat databases" in text-files format aren't best practice. Implementing the system with real relational databases like **MySql** or **PostgreSQL** would be an instructive and challenging task worth considering further down the line.