# Exam

April-May 2020

## 1 Introduction

In this report I describe how I implemented the missing parts of an unfinished code to make a search engine. Mainly the pre-code missed four important parts. A parser that verifies if queries are written correctly, an evaluation function which evaluate the results from the parser, an index which among other things maps document paths to a map, calculates an relevance score for each documents and performs the mentioned functions on a given query to create the final result.

### 1.1 Requirements

Here I outline the detailed requirements specified in the assignment text.

- First requirement: implement a recursive descent parser for the BNF grammar with the Boolean operators "AND, OR, ANDNOT".
- Next requirement: implement an inverted index that maps a word to a set of documents that contain this word.
- Last requirement: Implement an algorithm that arrange the documents so that the relevance of the documents is sorted higher in the result. The document that contains the most relevant word should occur higher up in the result than those who contain less relevant words.

## 2 Technical background

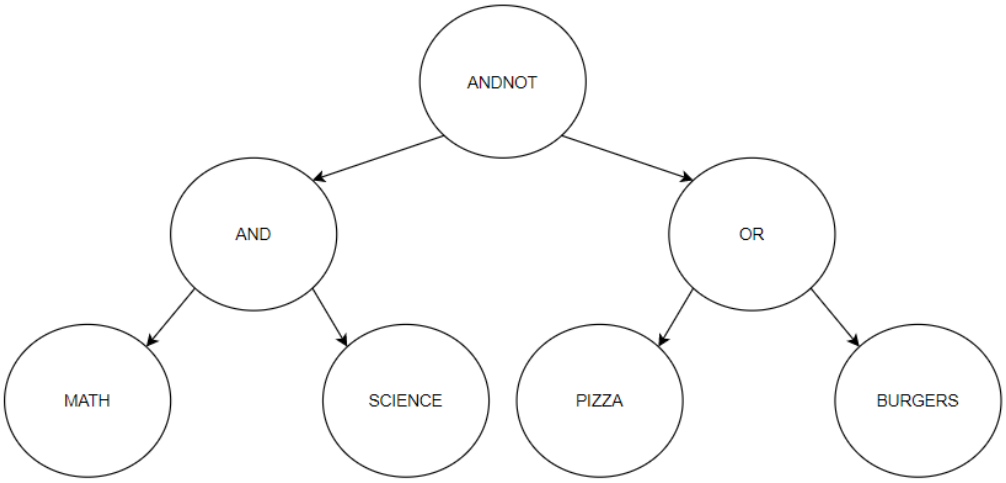
In this section I describe some of the topics covered in this exam which one should be familiar with before reading the implementation. These are hashmaps, recursive descent parsing, indexing and the tf-idf algorithm. I assume terms such as *while* and *for*-loops, *if*-statements, *linked lists*, *sets* and *arrays* are basic knowledge by now and will therefore not explain these.

A hashmap is a data structure that can link any given key to a value, which is a smart way to create fast access to an item. In this exam the values in the hashmap are the sets containing certain document paths, and the keys are words. A disadvantage of using a hashmap is that two

values could end up having the same value, which is called a collision. Collisions can be handled in many and ways but I do not think this is relevant to explain in this report.

To understand what a recursive descent parser is, it is important to know what a parser does and what recursion means. In this example a parser is an interpreter that breaks data into smaller parts to make it easier to check for given operators and verify if queries are written correctly. A parser will usually create a parse-tree which can be evaluated later.

Here is a visualization of a parse-tree:




Recursion means that something is defined within itself. In computer science this is applied when a function calls upon itself, creating a loop which helps solve the problem. So, when I talk about a *recursive descent parser* it is a parser that uses recursive functions to generate a parse-tree applying nodes from the top to the bottom. In this case the topmost nodes are operators and the leaf-nodes (at the bottom) represents specific words.

TF-IDF is an algorithm that tells us how important a word is to a document, in a collection of documents. In other words, the TF-IDF will give a document a score that tells the user how relevant the document is. The TF stands for *Term Frequency* and is calculated by counting how many times a word occurs in a document and dividing this number by the total number of words in that document. The IDF stands for *Inverse Document Frequency* and is calculated by taking the logarithm of the total number of documents in a corpus divided by how many documents in that corpus that contains a specific word.

$$\text{TF-IDF} = \frac{\text{Occurrences of a word in a document}}{\text{Total number of words in a document}} * \text{Log}\left(\frac{\text{Total number of documents}}{\text{Number of documents containing a word}}\right)$$

# 3 Design/implementation

When I started on the completion of the search engine, I first analyzed the pre-code and got a rough understanding on how the application should work. The pre-code provided me with a file called indexer, which needs one command line argument when it is run, namely the directory containing all the document files that will be indexed. Before the program is run the indexer file will tokenize the text from a given document into single words, and add these words to a list. The mentioned list along with a document path and an index structure were the things I was provided when I implemented the *addpath* function, the first missing part of the given code. I implemented the *addpath* function in a way that it will iterate over the words in the given list and check whether they are contained in a specific map which I can access via the index structure. If a word is not contained in the map the function will create a new set, add the given path to the set and lastly add this set to the map. If a word were to exist in the map the provided *map_get* function will fetch the map and the document paths along with a key that matches the current word. The function will then check if the given path is already contained in the set of paths. If the path is contained in the set, nothing will be done, but if it is not contained it will be added.

The TF-IDF algorithm is implement as part of the *addpath* function. The main reason is that I have to iterate over the given list of words to get the necessary variables to calculate the TF-IDF score and this is something I already do in the *addpath* function. When I iterate over the list of words, I use created a function that counts how many times a word occurs in the given document. This function will return an integer which is then divided by the total number of words in the given list. The result represents the TF-part of the algorithm. The TF-value will then be multiplied with the logarithm of the total number of documents which is divided by how many of the total documents containing a specific word. The total number of documents is the size of the given directory and how many documents that contains a word I found by using the provided *set_size* function. The TF-IDF score is then assigned to a query result type which shares a relation to a document.

When the indexing of the documents is done the indexer application will create a web-interface where a user can enter queries. The entered query will then be sent to the indexer application in the form of a string, divided into single strings and returned as a list. An example of an entered query could be “math OR theory” and the results will be the list “math” → “OR” → “theory”. At this point I implemented the *index_query* function which processes this list of query words and returns a list of documents that contains words from the query. The *index_query* function creates this list by creating a parse-tree of the given query words and then proceed to evaluate this parse-tree. After the query is evaluated and have returned a set of documents, this set is then iterated over and each element is placed inside a list.

To evaluate a query, I implemented a recursive descent parser. This parser takes a list as an argument and sends it through a variety of recursive functions which among other things checks whether the operators AND, OR and ANDNOT occurs in the query. Depending on which operator or word occurred in the list the parser will remove this word and return a node

of a specific type with two child-nodes. These two child-nodes could either contain more operators or words from the list depending on how far in the query the parser have gotten. The result of the recursive descent parser is described as a parse-tree.

The evaluation of the parse-tree is solved by traversing the tree and evaluate each node-type using a built-in feature in the C-programming language. In the case that the node contains a word that is not an operator, I fetch the map and the values linked to this word. The returned value is then the set of document paths that this value contains. If an operator is found it will at some point contain two words as it's child-nodes. If the operator is an AND-node the provided *intersection* function will return the documents containing both search words. In the case an OR operator is found the provided *union* function will return a set of documents containing all the search words, and lastly if the ANDNOT operator is found the *difference* function will return all documents containing only the first search word. If there should be a case where a node-type is not recognized I simply create a new, empty set and return it to avoid any further problems.

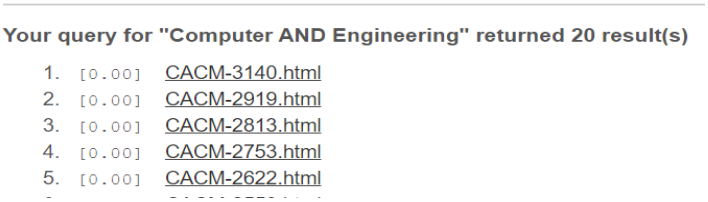
# 4 Discussion and evaluation

First of all, I have tested my search engine and it works quite correctly. I tested it by writing queries and checking whether the returned documents contains the words I searched for. However, I met some problems when it comes to the TF-IDF score. I tried to calculate the score using this formula:

$$\frac{\text{Occurrences of a word in a document}}{\text{Total number of words in a document}} * \text{Log}\left(\frac{\text{Total number of documents}}{\text{Number of documents containing a word}}\right)$$

But because many of the results ended up being a number between 0,01-0,07 the score did not appear in the webserver, which was quite surprising to me. For this reason, I made a decision and removed the value that caused this problem in my implementation, namely the "total number of words in a document".

Here is an example of what the TF-IDF score was:

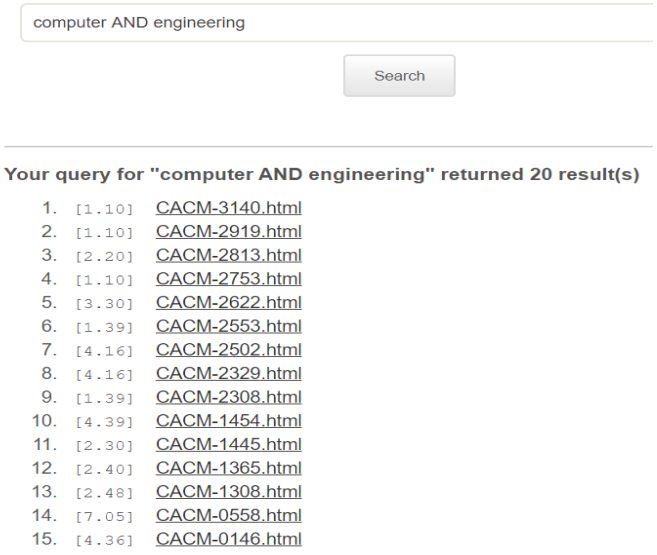


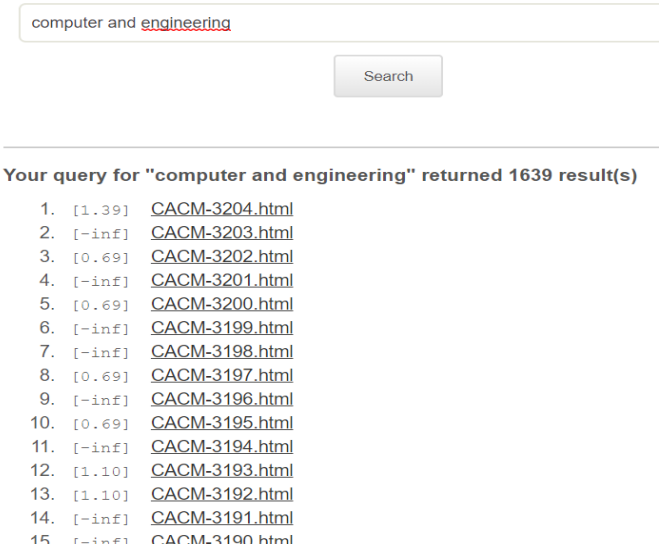

After I removed this value different values appeared in the web-server and they give a correct relevance score to the documents. I validated each document score by calculating the percentage of a word in a document. This method gave a word a higher percentage based on how many times it occurs in the document compared to the total number of words contained in the document. Whether or not the scoring is correct, this check tells me that a document with a higher score is more relevant. Another minor flaw is that I did not manage to sort the

documents so that the ones with the highest scores appeared higher in the list. Lastly, I must admit that the TF-IDF algorithm is not very flexible in this case. That's because I did not implement a function that counts how many documents that exists in the given directory, I simply wrote this number since I know how many files we were given. This will surely affect the scoring system if another directory is given.

Another thing worth mentioning is that the search engine does not support writing query operators with small letters. I'm not quite certain on why it won't support small letters because when I'm parsing a query, I'm comparing the query word to an operator with both big and small letters. Trying to use small letters also messes up the score for each document, rating some of them to have the value: " [-inf] ".

Here are some examples:




# 5 Conclusion

In this report I described how I implemented the missing parts of a code to create a functional search engine. This involved implementing an index structure, a recursive decent parser, an evaluation function and other functions to help solve minor problems. The mentioned functions mainly connected document paths to specific words, ranked documents and evaluated queries. I have learned a lot about writing code in C by writing this exam. Some examples are how to use tools to debug the code and how to create a better overview, which I will try be better at in the next assignment.

# 6 References

[1] https://stackoverflow.com/questions/2592043/what-is-a-hash-map-in-programming-and-where-can-it-be-used

https://codefying.com/2015/12/31/inverted-index-in-c/

https://www.onely.com/blog/what-is-tf-idf/