INF-1101-2: Data structures and algorithms

Exam: Search engine

April 24th, 2023

1 Introduction

For this exam, the assignment is to implement a set of functions for searching and sorting files based on a score relevant to the query – input given by the user of the search engine. The functions should be able to create a data structure, an index, and a parser following the given grammar. The main assignments are:

- Create the required functions to create and handle an index

- Create a parser to handle the input from the search engine

2 Technical Background

The C programming language will be the only language used for the implementation of the index structure, and Makefile [1] will be used as the compiler for the program. The Index ADT is built up by using several other ADT's, in this instance we use hash map, lists and sets. The parser implemented will be a top-down recursive descent parser. Let's go deeper into what a hash map and a recursive descent parser really is.

2.1 Hash map

A hash map [3], or hash table, is an abstract data structure that allows for efficient storage and retrieval of key-value pairs. A hash map utilizes a technique called hashing to map keys to specific locations in the memory. The key has a hash function applied which will calculate a unique index where the corresponding value will be stored. This index acts like an address and enables fast access to the value stored behind the given key. Hash maps allow for a low complexity for insertion, deletion, and retrieval operations, making it a great data structure for larger amounts of data.

2.2 Recursive descent parser

For the code to really understand the query it's given by the user, a parser needs to be implemented to translate the query to understandable syntax for the code. Following the tip from the assignment, a recursive descent parser [4] is used as the parser in this implementation. A recursive descent parser is a top-down parsing technique that breaks down a given input to smaller parts, following the rules given to it from a given grammar. For this parser, a BNF (Backus-Naur form) grammar is applied. The parser should differentiate between terminal and non-terminal symbols – the non-terminal symbols will correspond to parsing functions in the parser. The non-terminal symbols in this parser are AND, ANDNOT and OR. The parser will parse through the input until it's either successfully parsed or an error occurs.

2.3 BNF grammar

BNF (Backus-Naur Form) grammar [5] is a type of notation used to describe the syntax of a programming language, in this instance the C programming language. The grammar consists of rules which will differentiate valid and invalid expressions in the program. For this program, the BNF grammar should look for Boolean operators (AND, ANDNOT and OR) for giving the right results of a query input.

2.4 TF-IDF, scoring system

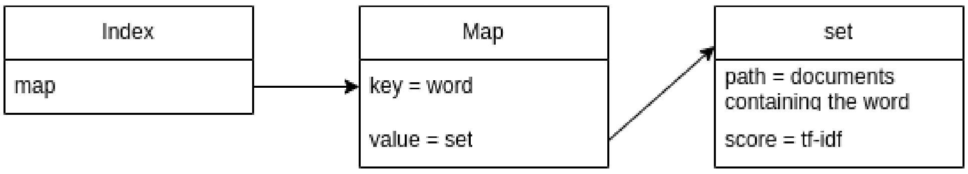
When returning documents to the user searching in the search engine, it's great to know which documents that's the most relevant to what's being searched for. The way to differentiate the different files is by scoring them using the tf-idf algorithm [6]. Tf-idf (term frequency – inverse document frequency) is an algorithm that finds the frequency of a word in a document relevant to the frequency of the word in all the documents in a set of files. The algorithm will return a value which corresponds to the relevancy and frequency of the given word – the higher the score, the higher the frequency.

2.5 Reverse indexing

The index will do reverse indexing (inverted indexing) [7], an efficient way to map documents containing certain terms making it easier to search and retrieve them later. Reverse indexing creates indexes that map to a set of documents that all contain a term, instead of storing all documents with their respective contents. This way of indexing allows for fast retrieval of documents relevant to the search query, while also providing an easy way to identify the relevant documents to the given query.

3 Design and Implementation

The pre-code comes with a lot of different files for all components of the code, so following the structure of the code the parser and the index will have their own file and header file. The index abstract data structure should be able to create an index for storing and retrieving documents. The documents should be able to be retrieved when a word in the document appears in the input query. The index contains a map with all different words in every document given to it, the map has a word as a key and a set as the value, every set of every key contains every document containing the key.



3.1 Add path to word

Every word in the map structure needs a path – the path from which document it originated from – and a score – for the relative frequency of the word in the document. The function for adding a path to a term is used when building the index structure. The code will go through every document given to it, tokenize it, and then add a path to all tokenized words. The words will be added to the index structure's map as a key, and then a set will be created as the value for that key. The set will contain elements of their own structure, query results with a score and a path. The score will be the frequency of the term in that exact document, meaning only the term frequency of the tf-idf algorithm.

If a term is already in the index's map, its path and score will be added to the set of the key that's already in the map. If the term is not in the index's map, a set will be created, and the term is added to the map. Now there will be no duplicates of keys in the map, and every key will contain every possible path for the key (term).

3.2 Index query

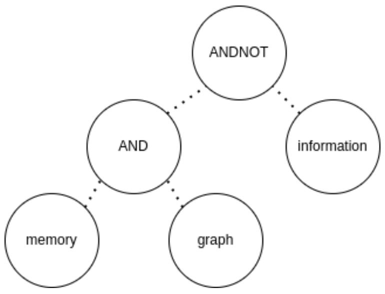
The index query function is supposed to request data from the index data structure and return the data corresponding to the query input. The functions search for the input query in the index's map and return the value of the key – the value being a set of documents containing the term searched for. The set will be returned as a list with the corresponding score connected to it. Before returning, the inverse document frequency will be calculated. Meaning the frequency of the term frequency times the number of documents containing the term divided by the total number of documents in the file. The list is then sorted based on the score of the document, with the highest score being the first to be returned.

3.3 Recursive descent parser

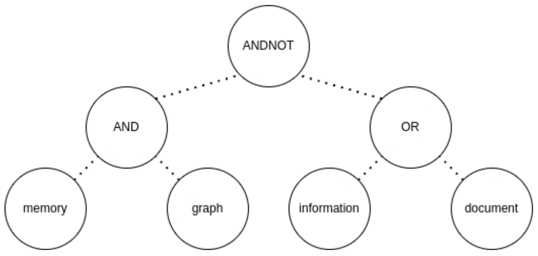
As already said, the code needs a parser to be functional with the index data structure and the grammar given in the assignment. Following the tips from the assignment, a recursive descent parser is implemented as the chosen parser. A recursive descent parser is a type of top-down parsing technique used to analyze a given input sequence and from that build a parse tree based on the given grammar – the BNF grammar. The parser checks the input sequence for terminal and non-terminal symbols. Terminal symbols are symbols that cannot be broken down to smaller symbols. Non-terminal symbols cannot be broken down to smaller symbols and will specify the structure of the language that's been described in the assignment.

The parser provided in the code will parse the query input and return a parse tree representing what is being searched for. The parser will through functions check for the presence of the given operators ANDNOT, AND, and OR. If any of these operators are present in the input query, they will be put as the root of the tree if it's only one operator. But if there are more operators use in the input query, the tree will put the first operator as the left child and the second operator as the root and if there are three operator the first will be the left child, the second is the root and the last one is the right child. The left and right child of the operators will be the words being searched for.

For example, if the input query is "(memory AND graph) ANDNOT information)", the parse tree created will look like this:



And if the input query is "(memory AND graph) ANDNOT (information OR document)", the parse tree will look like this:



4 Experiments

To see how good or bad a code is, we usually go through different tests and experiments to see how good the code really is. In this instance the experiments done on the code will be; time required to build an index of different sizes, the time required to search through the index depending on how many words are input and searching in different sized indexes.

In the pre code there is a function included for testing time that will return the time at the exact moment it is called. By creating two different variables - one for the time before the index is created, and a new one for the time the index is created – the difference between those variables will be the time used to create and build the index. This is the logic behind the experiment and is the same as when checking for time required to search through different indexes.

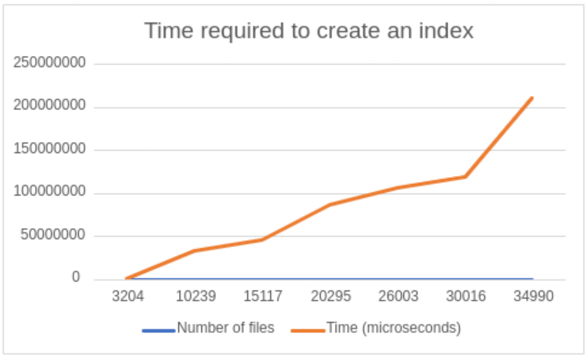
Will also test the code through the Valgrind [2] program which will create a summary of all memory leaks in the program.

5 Results

The results will just be tables and graphs showing what the results of the different tests are, the discussion about the results will come afterwards.

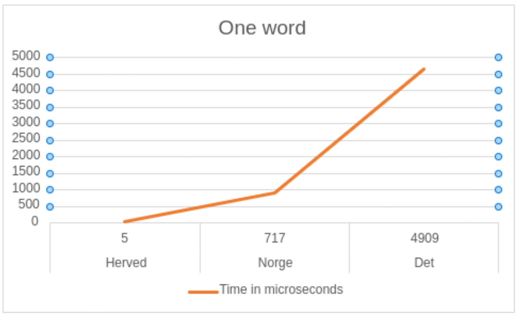
5.1 Time required to create an index

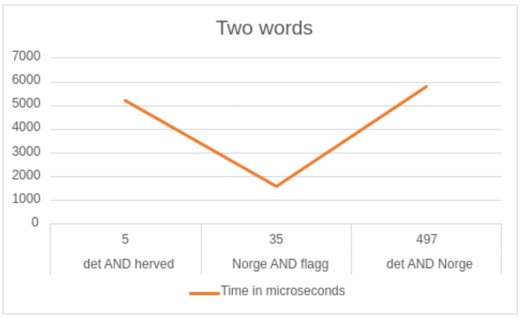
Files origin	Number of files	Time in microseconds	Time in seconds
cacm	3 204	535 831	0,54
Wikipedia	10 239	32 774 982	32,77
Wikipedia	15 117	45 560 105	45,56
Wikipedia	20 295	87 604 757	87,60
Wikipedia	26 003	106 313 709	106,31
Wikipedia	30 016	119 801 134	119,80
Wikipedia	34 990	211 405 437	211,40

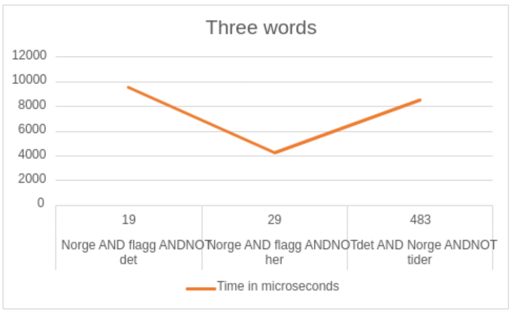


5.2 Time required to search through an index of a given size

Charts for searching for one, two and three words.







5.3 Valgrind results

```
==62005== HEAP SUMMARY:
==62005==     in use at exit: 7,282,324 bytes in 244,864 blocks
==62005==   total heap usage: 809,821 allocs, 564,957 frees, 21,318,972 bytes allocated
==62005==
==62005== LEAK SUMMARY:
==62005==    definitely lost: 2,256,870 bytes in 70,540 blocks
==62005==    indirectly lost: 4,993,204 bytes in 174,323 blocks
==62005==      possibly lost: 0 bytes in 0 blocks
==62005==    still reachable: 32,250 bytes in 1 blocks
==62005==         suppressed: 0 bytes in 0 blocks
==62005== Rerun with --leak-check=full to see details of leaked memory
==62005==
==62005== For lists of detected and suppressed errors, rerun with: -s
==62005== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
zsh: profile signal  valgrind ./assert_index
```

6 Discussion

Discussion of the different results from all tests done on the program.

6.1 Results of tests

The first test, checking how much time is required to create an index, shows how the program can handle a certain amount of data. From the numbers, we can see a steady rise in time required for the bigger and bigger amounts of files. Every increase is approximately 5k (k for thousands) files every step. The 3204 files that came with the pre-code use much shorter time than 10k files from Wikipedia. This is probably because the Wikipedia files are larger files with much more information, meaning every file takes longer to handle. The jump in time required from 30k to 35k files is much higher than the jump from 20k to 25k files. This shows that at a certain point, the number of files becomes so big that the whole process of creating an index is slowed down. I tried running for 40k and 50k files, but the program got killed before it could finish building the index. I suspect the reason for it is because of too many memory leaks. The suspicion comes from the results of the Valgrind test that shows the program has a lot of memory leaks.

6.2 Time required to search for words

From the charts we can see that longer lines of query input do not correspond to the higher time used to search. The main thing making a search use more time is the number of files corresponding to a term. The term "det" is in a lot of documents, and every time the term is used in a searching query the time used is higher than the other ones. From this it's the number of files in the set of results that determines how long it takes to search for terms. The larger the set, the longer the time it takes to return all the documents to the user. This corresponds to the amount of set operations done to return the result. More results require more set operations done; therefore, more time required to finish.

6.3 Searching in different sized indexes

With the results from the testing for different sized query inputs, it's easy to conclude that an index with more files takes longer time to be searched through because it will be more files containing the terms searched for. The searching through the hash map of terms is the same no matter the size of the index. Meaning there will not contribute to extra time used for searching. There is no test done for searching in different sized index, since we got the conclusion from searching on different sized query inputs.

6.4 Valgrind results

The Valgrind tests show that a total of 2,256,870n files were definitely lost while running the assert index file. This is a lot of bytes of memory, a whole 1.9 mega bites of memory. By using the command "valgrind --leak-check=full *file name"*, you get a summary of which functions are allocated to the memory but not freed afterwards. Most of the memory leaks are from creating sets, creating maps, and creating new nodes in the parser. Tried to resolve this by freeing the memory where I thought it should be freed but with no success. The only thing it did was to make the program crash when searching because there were no sets and maps there since they were freed from memory. I concluded that it was better to use my time on something else since I couldn't see a way out of the big amount of memory leaks.

7 Conclusion

The conclusion of the assignment is that it's a success. The program works as expected. The search works for several words, works with the Boolean operators, and gives a stable score from the tf-idf algorithm. Tests for the parser showed it could correctly identify the different Boolean operators and do what the operators should. I wished I would have more time to fix the memory leaks so the program performance was higher, but not everything can be perfect.

References

[1] Makefile, GNU Make:

https://www.gnu.org/software/make/manual/make.html

[2] Valgrind, Valgrind org:

https://valgrind.org/

[3] Hash map, Wikipedia:

https://en.wikipedia.org/wiki/Hash_table

[4] Recursive Descent Parser, Wikipedia and Geeks for Geeks:

https://en.wikipedia.org/wiki/Recursive_descent_parser

https://www.geeksforgeeks.org/recursive-descent-parser/

[5] BNF grammar, Wikipedia:

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[6] Tf-idf, Wikipedia:

https://en.wikipedia.org/wiki/Tf%E2%80%93idf

[7] Reverse indexing, Wikipedia:

https://en.wikipedia.org/wiki/Reverse_index