INF-1101 - fei003 - a2

This report details the implementation of an indexer that supports indexing of text documents and evaluation of queries to filter words within the indexed documents. The filtering mechanism is based on a context-free grammar (CFG) that describes the unambiguous rules for determing the legal expressions of the query language.

1. Theoretical background

The assessment can be into two main problems. Each problem can then be further split into sub-sub problems. For instance:

- **Parsing:** ambiguity, sanitizing input, error handling etc.
- **Indexing:** containers, algorithms, caching etc.

The biggest problem of this assesment is the filtering mechanism described in the introduction. To be able to filter the words in the documents, the program requires a parser that can take the input (query) of a user and produce something meaningful out of it.

The data structures and algorithms used in this assessment was already done in the pre-code of the assessment. This report will not go in detail on the given data structures and algorithms.

1.1 Parsing

[1] describes ambiguity in a sentence as a sentence that can be understood in two or more ways, and it's often referred to as *structural ambiguity*. The nature of CFGs has it origins in human languages, and is therefore prone to being ambigous. The CFG used in this asessment is called *Backus Naur Form (BNF)*.

The BNF describes our CFG, which is a set of rules used to describe or query language. The syntax is:

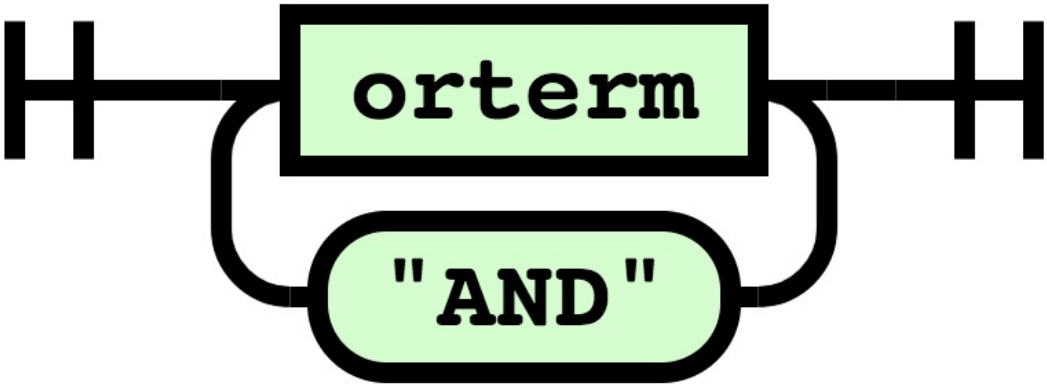
- `query ::= andterm | andterm "ANDNOT" query`
- `andterm ::= orterm | orterm "AND" andterm`
- `orterm ::= term | term "OR" orterm`
- `term ::= "(" query ")" | <word>`

The syntax diagrams below helps visualize the syntax.

Query:

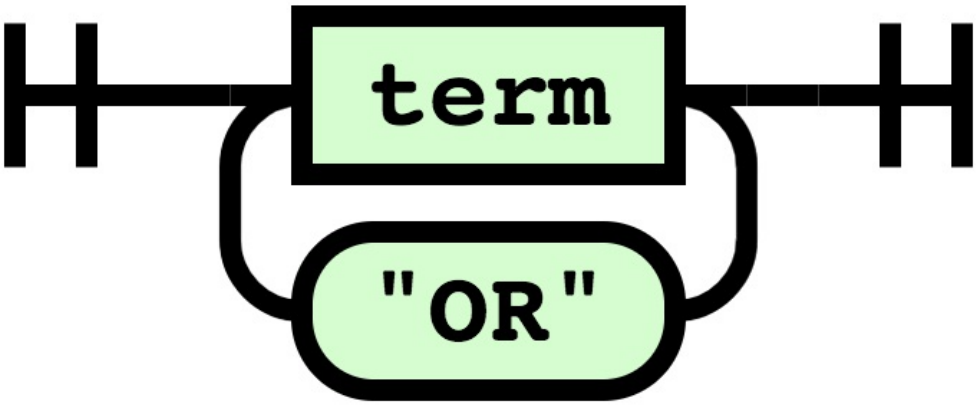
A query is an andterm or an andterm combined with "ANDNOT" and a query.

Andterm:



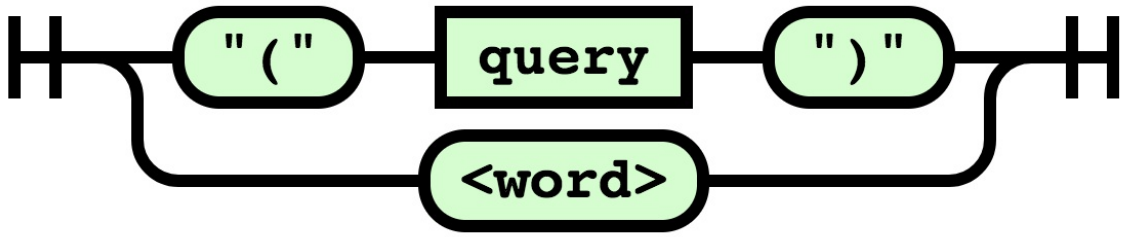
An andterm is an orterm or an orterm and "AND" combined with an andterm.

Orterm:



An orterm is a term or a term and "OR" combined with an orterm.

Term:



A term is an opening parenthesis "(" followed by a query followed by a closing parenthesis ")", or a word.

The syntax diagrams are a graphical alternative to the BNF, they describe the possible paths between two points by going through other nonterminal and terminals (terminals being round, nonterminals squared).

What we can see from the figures above is that the syntax is recursive. We are actually looking at a *recursive descent parser* with a *top-down* design

1.2 Indexing

The contract of the index was given in the pre-code
[2] defines an abstract data type (ADT) as a data type whos operations are accessible through an interface, and has its implementation hidden from the client (a program that uses an ADT). This means that the same set of operations can have different implementations, and clients can switch betweeen implementations without breaking existing code (as long as the contract of the interface is held).

Indexing in itself is the process of *collecting*, *storing* and *parsing* data for facilitating information retrieval. The purpose of this is to optimize speed and performance when trying to find relevant data for a given search query [3].

The indexer used in this assessement was a so-called *inverted indexer*, meaning that unique words (keys) are stored and mapped to the locations (values) of the content (in our case doucments on the computer). Basically we are looking to find every document where term `t` occurs.

1.2.1 Query and result ranking

When the user enters a query in the web interface (located at localhost:8080), the query is then sent to the indexer which tokenizes the query string and stores it in a list. The values returned from the index is a list of documents that match the give query.

The list of documents ordered by relevance, first document being the most relevant and so on and so fort. For determing the relevance of the documents two algorithms called *term frequency (TF)* and *inverse document frequency (IDF)* is used. Where TF denotes the number of times the term `t` occurs in document `d` and `idf` denotes the logarithmically inverse faction of the documents that contains the word `t` divided by the number of documents `D` containing the word `t`.

The problem with raw TF is that all words weighted equally important when it comes to relevance. Therefore, it's combined with the `IDF` which yields an algorithm called `tf-idf`.

3. Design

3.1 Index

For implementing the index ADT, an interface that described the operations with its return types was pre-defined in the precode. All there was to do was to decide which data structures to utilize inside the index. The chosen data structure was a hashmap that contained a set of documents. The documents contains the path to the file and its ranking score.

The index works in this way:

For each document and for each word in the document:

- Does hashmap contain word?
    - Create document and set, put in map
- else, set contains document?
    - add the score
- else, create new document

The index is then ready to be used by the indexer application.

3.2 Parsing

The indexer application starts a web server on localhost that serves the web interface. This is where the user can enter the query.

The parser interprets the given query, and as previously stated, works in recursive way. The parser splits the query into sub problems and for each word in the query it evaluates which operation to do on the set of documents.

For the Terminals ("OR", "AND", "ANDNOT") three different set operations are used; difference, intersection and union. This was also implemented in the pre-code.

If the query contains no terminals the indexer application implicitly adds the "OR" terminal between each word in the query.

4. Implementation

The application is implemented using the C programming language with the `Apple LLVM version 9.1.0 (clang-902.0.39.1)` compiler on a `x86_64-apple-darwin17.5.0` architecture with a `posix` thread model. Apart from the obvious standard libraries, most of the re-used code is already credited to people it involves inside the source code (thanks S. V. Valvaag, M. Stenhaug and E. H. Graff).

5. Discussion

When implementing the index the chosen container inside was a hashmap, this was because the hashmap has a time complexity of O(1) because of the hashing. The container inside the hasmap was the `aatreeset` given in the pre-code, this set ensures that the set operations inside the

recursive parsing fucntion has a time complexity of 0(N).

The caveat of the hashmap is that a hashmap with a bad hashing function can yield the same index when hasing different words, this is problematic in some cases. Another disadvantage is that it takes up a lot of memory when the size grows.

There is unfortunately some bugs in the parser still that were not found during the development, and the error handling inside the parser is also very generic.

In hindsight, the way that the index is implemented is could also be done differently. The excessive use of table lookups and set lookups is what´s causing the indexing to take > 3 seconds when starting the program.

6. Conclusion

In this report the details of how a index ADT that supports indexing of documents and evaluation of queries to find occurences of words in the documents is described. The resulting index was an inverted index containing the words and all the locations of the documents with the words.

The intention of the index was to use it for a basic search engine that had a simple web interface for evaluating queries on a set of documents located on the host tbat returned a link to the matching documents.

The Users of this web interface could search on either single words or a combination of words using the language described in this report.

References

[1] Cecilia Quiroga-Clare, *Language Ambiguity: A Curse and a Blessing* [Online]. Available: http://www.seasite.niu.edu/trans/articles/Language%20Ambiguity.htm

[2] Robert Sedgewick, 1997. _Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching: Fundamentals, Data Structures, Sorting, Searching. 3 Edition.