NCAT Search Engine Documentation

Fatlum Sadiku

October 28, 2019

v1.0

1 Introduction

The Search Engine for NCAT (Net Corpora Administration Tool) is a Search Engine specifically designed for query evaluation on linguistic corpora from the NCAT family. The current goal of this Search Engine is to provide tools for efficient search over linguistic data from non-standard varieties of English. The current version of the Search Engine is written in C++ using the GNU compiler g++ 7 on Ubuntu 18. Currently, the Search Engine consists of two parts. The **Indexer** contains modules that take a given SQL-file of data (as of version 1.0 represented by a CSV-file) and generate an Inverted Index to be used by the Searcher. The **Searcher** contains modules that, given a query, retrieves all relevant results from the Inverted Index and returns the documents containing them to the user.

In the following documentation, the *h.*files and the *cpp.*files are collapsed into one description, since the *h.*files only contain function declarations and/or class declarations. The subsections correspond to modules containing both kinds of files. The titles of the subsections are the filenames without the file endings. Every subsection therefore corresponds to two files (with the exception of files containing *main*). Within those subsections there will be a description of all the functions, methods, constructors, etc. within those modules. The functions themselves will follow a sort of tree structure according to dependencies, i.e. if *func1()* calls *func2()* and *func3()*; and *func2()* calls *func4()*, the functions will be listed as *func1(), func2(), func4(), func3()*. Where not further indicated, *vector* refers to *std::vector*, *string* refers to *std::string*, etc.

2 Indexer

The Indexer takes a given SQL-file (a CSV-version of that file in version 1.0) and constructs an Inverted Index structure to be used by the Searcher.

2.1 readfileCSV

This module reads a CSV-file and constructs an Index (not Inverted yet) by splitting up the words and indexing them. In later versions there will be a module that has the same functionality but uses an SQL-file instead.

2.1.1 RunMain

This is the quasi-main function of this module. It takes an *std::string* as its input, which is the name of the CSV-file to be indexed, and returns an *idMap* structure. The *idMap* is defined as *std::map<int, std::vector<std::string>>* and corresponds to a map containing the docIDs as keys and the words of the content, seperated as items in a vector, as values. The function reads the file by calling *readFile* and generates the *idMap* by calling *genMap*.

2.1.2 readFile

This function reads a file by taking a *std::ifstream&* and generating a *lineVec*. The *lineVec* is defined as a *std::vector<std::vector<std::string>>*, which represents the filestream as a vector containing the lines (i.e. all content and metainformation per docID) as items, which in turn are vectors of string, where each item corresponds to one column in the original table. It does so by calling *lineToVec* on each line (i.e. a row of the table).

2.1.3 lineToVec

This function takes a *string*, corresponding to one line (i.e. row of the original table) and generates a *vector*, where the elements are the columns per row. For the index itself, we only need docID, memberID and content (cleared of html-tags).

2.1.4 genMap

This function takes a *lineVec* (see 2.1.2) which corresponds to the complete input table and generates an *idMap* (see 2.1.1) from that table. It does so by calling *split* on the 8^{th} index of each vector-element (corresponsing to the cleaned up content of the document). It then associates the docID to the vector.

2.1.5 split

This function takes a *string* corresponding to the content entry for a particular docID row in the original table and returns a *vector<string>* which represents the word-separated content. This function is special, as it will be subject to revision right up until shipping. The reason for that is that it is never completely certain, what can be counted as a word or what should be included in the index. *split* goes through all the characters, using whitespaces and commas as basic separators, however, it also calls additional helper functions. As of version 1.0 there are two helper functions, both of which are predicates (*isSeparator* and *isAcronym*), of which only one (the first) has been included in *split*.

2.1.6 isSeparator

This function takes in a *char* and returns *true*, if that *char* is a separator. As of version 1.0 the characters that are recognized as separators include *".?!:;&\/()"*. This list is not exhaustive and subject to revisions. The goal is to have the cleanest output possible, where (almost) all marginal cases are taken care of. Since the Index will not be generated or expanded on a regular basis, runtime issues are of no concern.

2.1.7 isAcronym

This function is currently (as of version 1.0) under revision. The goal is to design a function that checks separators as to whether they are part of acronyms. If we did not do that, *U.S.A* would be separated as the words *U, S* and *A*.

2.2 indexer

This module generates the Inverted Index from an *idMap* (see 2.1.1) and returns an *InvertedIndex* structure. The *InvertedIndex* structure is defined as *std::map<std::string, std::map<int, std::vector<int>>>*. This corresponds to a regular Inverted Index with positional inverted lists, called *postingsList* in the program and in the documentation.

2.2.1 generateIndex

As of version 1.0 , this is the only function in this module. The only addition to 2.2 is that the positions start at 0.

2.3 indexerMain

This contains the main function of the Indexer. It calls *RunMain* (see 2.1.1) to generate the *idMap* and from that generates the Inverted Index by calling *generateIndex* (see 2.2.1). In addition to that, as of version 1.0 , it serializes the Inverted Index using the *Boost* library by generating a binary archive. In later versions of the Search Engine, this will be replaced by keeping the Index keys in memory while saving the individual *postingsLists* on disk. This will be achieved by using the *STXXL* library.

3 Searcher

The Searcher is the main part of the Search Engine. Its goal is to take a given query and run it thorugh a preprocessor to clean it up and prepare it for the search functions. The cleaned up query then gets parsed into a *SearchTree* structure (for more details see 3.3), which then gets evaluated by a recursive algorithm (described in 3.3) and at each inner node calls the appropriate function from *Searcher* (section 3.2). The result of any query is thus given as a positional postings list, which retrieves the appropriate data from the actual corpus (this part has not been implemented as of version 1.0).

3.1 RetrievePostings

As of version 1.0 , this module only contains a trivial function to retrieve positional postings lists from the Inverted Index. In later versions, and in particular after STXXL has been implemented (see section 2.3), this will be expanded to make retrieval of lists as efficient as possible considering that the entire Index cannot be kept in memory.

3.1.1 retrieve

This function takes a reference to a *string*, the key (and search term), and to a *InvertedIndex* and retrieves the value for that key.

3.2 Searcher

This module contains functions used to evaluate different *postingsList* structures. One function, *mergePos*, is used as an auxiliary for most other functions (described in 3.2.1) and will therefore be explained first. Except for that function, all other functions return *postingsList* and all functions except for *mergePos* and *exact_phrase* take two *postingLists* as input, sometimes with additional flags. A *PostingsList* refers to what in standard IR parlance is called a positional inverted list and is defined as *std::map<int, std::vector<int>>*.

3.2.1 mergePos

This function does a basic merge on the *vectors* containing the positions for each docID. It takes in two *vectors*, called *left* and *right* and merges their elements in linear time using set semantics. It returns a vector<int> which contains each position once (which is trivial since positions cannot be the same on the same docID).

3.2.2 intersect

This function realizes the AND operator over two *postingsLists*, called *left* and *right*, in linear time. It returns a *postingsList*. It calls *mergePos*.

3.2.3 unionize

This function realizes the OR operator over two *postingsLists*, called *left* and *right*, in linear time. It returns a *postingsList*. It calls *mergePos*. The reason it is called *unionize*, instead of *union* is that the latter is a reserved keyword in C++.

3.2.4 complement

This function realizes the NOT operator over two *postingsLists*, called *left* and *right*, in linear time. It evaluates the set equivalent $L \setminus R$ and returns a *postingsList*. It calls *mergePos*.

3.2.5 proximity

This function realizes the various proximity operators. As of version 1.0 two different proximity operators have been incorporated into the search engine, namely *NEARn* and *WITHINn* where $n \in (0, 100)$ is an integer. *NEARn* denotes the second word being at most n positions to the left or the right of the first word. *WITHINn* denotes the second word being at most n positions strictly to the right of the first word. It returns a *postingsList*. In addition to the input lists, the argument list contains *dist*, *leftbound* and *op*. The first and third of those are straightforward. They are of *int* type and contain the distance (n in *NEARn*, for example) and the operator (*NEAR*, *WITHIN*, etc.),

respectively. However, *leftbound* needs an explanation. It is of *bool* type and its value depends on whether the parent operator (the operator node that is a parent of the current proximity operator node) has the current node as a left or right child (more information in section 3.3). The reason for this is that proximity operators are not associative (at least not in regards to positional information). In a query such as `foo NEAR7 bar NEAR6 baz`, if the parentheses are set as `(foo NEAR7 bar) NEAR6 baz`, the left operator needs to return the rightmost position for each pair, whereas in `foo NEAR7 (bar NEAR6 baz)`, the right operator needs to return the leftmost position for each pair.
The function works by first calling *intersect* (see section 3.2.2) and then checking for each position per docID whether there are matches for distance.

3.2.6 exact_phrase

This function does not realize an operator. Instead it is called when a search term is not a single word but rather an exact phrase, which is demarcated by quotiation marks. It takes a phrase, defined as a *std::vector<postingsList>*, by reference, as well as a *bool*, called *leftbound* (see 3.2.5 for an explanation), and returns a *postingsList*.
This function works by first iteratively calling *intersect* over the *postingsLists* in the phrase, for example `(((...(foo AND bar) AND baz) AND ...)`. The resulting *postingsList* contains the docIDs for all documents that contain all the words in the phrase. That *postingsList*, however, also contains the positions of all words within a particular document. Since no position can occur twice per docID, the function then only checks whether we can find a window of positions that fulfills the following criterium. For an index i in a given positional list, there must be an index $j = i + m$, such that the value at i, denoted by p_i is exactly m smaller than the value p_j. More precisely, if $j - i = m$ then $p_j - p_i = m$. This condition, however, can also hold if we have two consecutive partial matches, such that their summed length is at least m. This means that once the function finds a match, it still checks whether the window we looked at actually corresponds to the phrase in question.

3.3 SearchTree

This module contains the classes *SearchTree* and *TreeElement*, as well as their attributes, constructors, methods and some additional auxiliary functions which are called by some methods. It generates a Search Tree from a legal query and evaluates that Search Tree by calling the appropriate functions from *Searcher* (see 3.2) in a recursive manner.
Before going on, however, it is in order to explain what the Search Tree actually represents. In previous versions of this Search Engine (written in Python), it was called a Parse Tree, which better reflects its use. For any given query, the order of operations is denoted by parentheses. The **Preprocessor** uses a particular binding hierarchy to introduce parentheses if none are given by the user (more information in section 4). The Search Tree module then takes that parenthesized query and generates a tree structure, in which inner nodes represent the operators (AND, OR, NEARn, etc.) and leafs represent either a single word or an exact phrase. The calculated *postingsLists* move up the tree until the operator with the weakest binding in the query is encountered.

3.3.1 class TreeElement

This class represents a node of the Search Tree. It is a friend of *SearchTree* and contains the private member variables *key* of type *string*; *type* of type *int*; as well as *parent*, *left* and *right*, all of type *TreeElement**. The *key* can be any word, exact phrase, operator or opening and closing parenthesis, *type* then defines whether it is a single term (type=1), an exact phrase (type=2), an operator (type=3), an opening parenthesis (type=4) or a closing parenthesis (type=5). The rest simply refer to a parent node or a left or right child, respectively. The constructors and methods will be explained in sections 3.3.3 through 3.3.5.

3.3.2 class SearchTree

This class represents the tree itself, i.e. the collection of *TreeElements* and their connection. It has two members: *root* of type *TreeElement* and *current* of type *TreeElement**. *current* is used by almost all member functions to keep track of where in the tree we are. The construcors and methods will be explained in sections 3.3.9 through 3.3.13.

3.3.3 TreeElement constructors

The *TreeElement* class contains four constructors. The first takes no arguments and sets *key* to the null string, *type* to 1 and all the neighboring nodes to *nullptr*. The second constructor takes *key* and *type* as arguments and sets all neighboring nodes to *nullptr*. The third constructor takes *key*, *type* and *parent* as arguments and sets the children to *nullptr*. The final constructor takes all members as arguments. This one is only used for testing purposes and is not necessary for the construction of the Search Tree.

3.3.4 TreeElement::printInfo

This method prints key, type and the keys of all neighboring nodes in a pretty format. It takes an optional argument *s* of type *string*, which is used by *SearchTree::printTree* (see 3.3.11).

3.3.5 TreeElement::changeChildren

This method changes left and right children of a given node and is only used for testing purposes.

3.3.6 getOperatorType

This function is used by *SearchTree::evaluate* (see 3.3.13) to determine the specific type of operator. It takes in a *string* and returns an *int* For AND, OR and NOT, the type is 1, 2 and 3 respectively. For NEAR the type is between 100 and 199, and for WITHIN, the type is between 200 and 299, where the last two digits represent the distance.

3.3.7 isOperator

This function is used by *SearchTree::generate* (see 3.3.12) to determine if a given *std::string* is an operator. It returns a *bool*.

3.3.8 splitString

This method is used by *SearchTree::evaluate* (see 3.3.13), in particular if the input is an exact phrase. It splits the input phrase into a vector of words.

3.3.9 SearchTree constructors

The *SearchTree* class contains only one constructor, namely the one that constructs an empty Search Tree, where *current* is a pointer to *root* and *root* is initialized by the empty TreeElement constructor (see first constructor in 3.3.3)

3.3.10 SearchTree::insertElement

This method takes two arguments, *query_ i* of type *std::string* and *type* of type *int* and returns *void*. These correspond to the *key* and *type* members of the *TreeElement* class (see 3.3.1). Depending on the type, this method inserts a given Tree Element into the Search Tree or generates an edge to a left or right empty child. If *query_ i* is an opening parenthesis, depending on whether the left child exists or not, it generates a left or right empty child, respectively. If it is a closing parenthesis, *current* moves up a node. If *query_ i* is an operator, it generates a node with two empty children and finally, if it is a single word or an exact phrase, the method generates a leaf.

3.3.11 SearchTree::printTree

This method moves down the Search Tree recursively and calls *TreeElement::printInfo* on each node with an offset depending on the depth of the recursion, i.e. the depth of the tree. It returns *void*.

3.3.12 SearchTree::generate

This method takes a *query* of type *std::vector<std::string>* as an argument and returns *void*. It does so by assigning *type* information to each element of the input and calling *SearchTree::insertElement* with the correct type. In order to determine whether a given input is an operator, it calls *isOperator* (see 3.3.7).

3.3.13 SearchTree::evaluate

This method is the central method for searching. It takes an *InvertedIndex* as an argument and returns a *postingsList* that matches all search terms and all operators. It does so by moving down the tree recursively. If it encounters a single word, it calls *retrieve* (see 3.1.1). If it encounters an exact phrase, it calls *splitString* (see 3.3.8) to split the phrase into a vector, then calls *retrieve* on all words, pushes the *postingsLists* onto a vector and then calls *exact_ phrase* on that vector. Finally, if it encounters an operator, it calls *getOperatorType* (see 3.3.6) to determine the exact operator and then, depending on the operator, calls one of the binary functions described in section 3.2. As of version 1.0 the *leftbound* flags for the *proximity* operators are set to default settings (see 3.2.5 for more information).

3.4 SearcherMain

This module contains the *main* function of the Searcher. As of version 1.0 there are two important short-term fixes which will be changed in later versions. For one, instead of calling the **Preprocessor** to parse the query, there is an auxiliary function called *breakLine*, which transforms a legal query into a *vector* ready for Search Tree generation. In addition to that, the *main* function loads the Inverted Index as a binary archive, using the *boost* library, before it evaluates any queries. The latter will be changed once *STXXL* is incorporated into the project.

4 Preprocessor