**Boolean Information Retrieval System**

Application’s Structure

At First, Let’s Begin with the architecture of the file system relative to the source folder:

**dataset: -** All the documents that have to be searched is stored in this folder for better understanding and ideas of the working.

**venv: -** Folder containing files that helps in making workspace a virtual environment

**Code.py: -** This .py extension files explains us that it is a python file containing all codes for the Boolean retrieval system.

**InvertedIndex.txt:-** Contains the inverted index of the tokens.

**Stopwords.txt: -** Contains the words (stopwords) that have not be included in tokens, in process of making inverted index.

**BiGramInvertedIndex.txt: -** Contains the Bi-gram of all the tokens of the corpus.

**RetrievedDocument.txt:-** Contains the output documents related to the query.

Code’s Structure

Now, Coming Inside the code file(Code.py), different functions, libraries and data-structures have been used.

**Functions:-**

**LevenshteinDistance (str1,str2):-** Functions containing two string parameters as inputs and calculate the minimum edit distance between them and return the minimum edit distance value.

LevenshteinArray as 2D array has been used for finding the Levenshtein distance initialized with all values as 0 with length of str1 as number of columns and length of str2 as number of rows.

**Time Complexity:- O(n^2)**

**AND(p1,p2):-** Functions with two posting lists as parameter and performs intersection on input lists and return common documents id as output.

Used answer as list data structure for storing the intersection two posting lists document id.

**Time Complexity:- O(n)**

**OR(p1,p2):-** Functions with two posting lists as parameter and performs union on input lists and return documents id that are present in both posting list as output.

Used answer as list data structure for storing the union of two posting lists document id.

**Time Complexity:- O(n)**

**NOT(p) :-** Functions with a single parameter as posting list and returns all documents that have not been matched with this posting list.

Used answer as list data structure for storing the all the document id minus the posting lists id.

**Time Complexity:- O(n)**

**BiGramQuery(word):-** Function taking word (string) as a parameter for converting this wildcard query for bigram search.

**Time Complexity:- O(n^2)**

BiGramSearch(words):- Taking list as parameter used for searching all the word set for bigram query and append the docIds as result.

**Time Complexity:- O(n^2)**

**normalQuery(raw):-** Taking the input query as parameter and returns with spell checking using Levenshtein distance between the query word and words present in the inverted index table.Return the querywords as a list after spell correction and removing the stopwords, bit operation words

**Time Complexity:- O(n^3)**

**ParseBoolean(PreprocessedQueryString, invertedindexTable):-**

Functions for booleanQuery search and input is the pre-processed query string and inverted index table.Returns the ResultDocumentSet.

Use Stack for the Boolean parsing query.

**Time Complexity:- O(n^2)**

Data Structures:-

* We have used *python dictionaries* to store the inverted indexes. The dictionary keys are the unique words in the document space, and the values are their respective document IDs in which they occur, which are stored in a List. Looking up values in a python dictionary is very fast due to the use of HashMaps and a lookup operation can be done in O(1) time complexity.
* For the Boolean AND/OR and the Unary NOT operations, we have converted the List of document IDs into a Python Set. The Set data structure in python, which is akin to a List with only unique values, provides us an elegant and fast way to perform Boolean AND/OR operations. These operations are performed in O(n+m) time complexity, where n and m are sizes of the two sets. The NOT operation is performed in O(n) time complexity, where n is the size of the inverted index.
* We have used a stack to store the Boolean preprocesssed queries, which are popped one-by-one.

Running time for Preprocessing:

The query preprocessing function(normalQuery(raw)) performs Spell check, Stopword removal and Lemmatization. The time complexity is O(n^3), where n is the number of words in our document space.StopWord Removal and Lemmatization will take Time Complexity of O(n^2) but in Spell Checking the use of LevenshteinDistance function in a for loop causes Time Complexity to be O(n^3).Converting into BiGramQuery has also the Time Complexity of O(n^2) due to nested for loop.

Running time for Building Inverted Index:

The running time for building the Inverted Index is O(n), where n is the number of words in our document space.

Running time for Search/Retrieval:

The running time for search or retrieval is O(n), where n is the number of words in the document space but for BiGram Query Search, it will takes O(n^2) as using .union() function inside the for loop.

Instructions:-

1. Enter the Query, as asked in the terminal window.For WildCardQuery use ‘\*’ symbol inplace of the missing letters.
2. InvertedIndex, BiGram Index has been stored in the ‘InvertedIndex.txt’ and ‘BiGramInvertedIndex.txt’ respectively. You can see the indexes from here.
3. Document Names have been displayed in the terminal with docnames saved in the ‘RetrievedDocuments.txt’ related to ur queries.

Team Members:-

Shivam Shandilya 2020A7PS2094H

Sathivik