Code Search

Ajit Jadhav, Arvind Deshraj, Junaid N Z, Rohan S
Indian Institute of Information Technology, Sri City, Chittoor

Abstract—**The goal of this information retrieval system is to find all relevant code samples for a natural language search query. In this paper, we present the design of our system that affords efficient retrieval of code samples for a required purpose as specified by the user query. Our system uses implementation of positional inverted index and k-gram index. Based on these indexes the code samples are retrieved for the given query as well as queries similar to the given query.**

Keywords—code search, positional inverted index, k-gram index

I. Introduction

In software development activities, source code examples are critical for understanding concepts, applying fixes, improving performance, and extending software functionalities. Previous studies have even revealed that more than 60% of developers search for source code every day. With the existence of super-repositories such as GitHub hosting millions of open source projects, there are opportunities to satisfy the search need of developers for resolving a large variety of programming issues. We propose to build a search engine for the same cause.

II. System Architecture

A. *Positional Inverted Index*

Inverted index is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. It is the most popular data structure used in document retrieval systems, used on a large scale in search engines.

We are using a positional inverted index where for each term we store the postings along with their positions in the document(as shown in Fig. 1). A positional inverted index is created using the code as documents.

B. *K-gram Index*

A k-gram is a contiguous sequence of k items from a given sample. In a k-gram index, the dictionary contains all k-grams that occur in any term in the vocabulary. Each postings list points from a k-gram to all vocabulary terms containing that k-gram(as shown in Fig. 2).

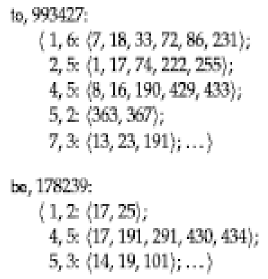
We are creating 3-gram, 4-gram and 5-gram indexes on the given set of documents. These indexes will help in wild card query processing for the given query.

C. *Query Processing*

For a given natural language query, we perform basic analysis of the query like stop word removal. In computer search engines, a stop word is a commonly used word (such as "the") that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query. When building the index, most engines are programmed to remove certain words from any index entry. The list of words that are not to be added is called a stop list. Stop words are deemed irrelevant for searching purposes because they occur frequently in the language for which the indexing engine has been tuned. In order to save both space and time, these words are dropped at indexing time and then ignored at search time.

For further processing of the query we can use stemming and lemmatization. For grammatical reasons, documents are going to use different forms of a word, such as *organize*,*organizes*, and *organizing*. Additionally, there are families of derivationally related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set. The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. *Stemming* usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma. For our search purposes, we use stemming.*

The query can be a normal query as well as a wild card query. Thus, for a given query, if we have a standard query, we directly retrieve the documents using the inverted positional index. But if we have a wild card query, we first find a list of words from the 3-gram, 4-gram and 5-gram indexes, then we merge the results and using this result, we run document retrieval on the positional inverted index. This gives us results in a wide range thus allowing us to get more number of relevant results.



```
to, 993427:
   ⟨ 1, 6: ⟨7, 18, 33, 72, 86, 231⟩;
     2, 5: ⟨1, 17, 74, 222, 255⟩;
     4, 5: ⟨8, 16, 190, 429, 433⟩;
     5, 2: ⟨363, 367⟩;
     7, 3: ⟨13, 23, 191⟩; . . . ⟩

be, 178239:
   ⟨ 1, 2: ⟨17, 25⟩;
     4, 5: ⟨17, 191, 291, 430, 434⟩;
     5, 3: ⟨14, 19, 101⟩; . . . ⟩
```

Fig. 1. Example of positional inverted index

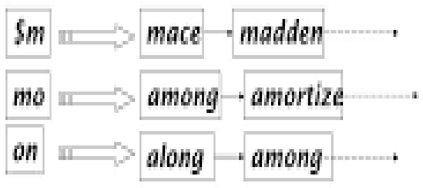


Fig. 2. Example of bigram inverted index

III. Experiments

A. Dataset

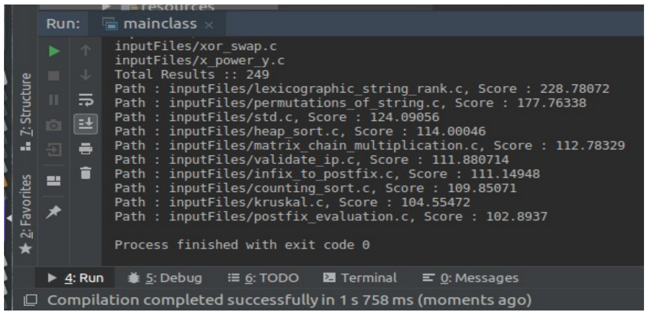
The dataset consists of C code samples. It was collected from OpenGenus Foundation’s Cosmos repository which contains a collection of a variety of data structure and algorithm codes. All the .c extension files were collected from the collection giving us a dataset of 260 C codes (approximately 400 KB). These files were used as documents and indexes were created accordingly.

IV. Conclusion

We ran multiple queries on different indexes and we got certain results. We could see that on an average K-gram index was giving bad performance. As in lot of documents were retrieved because of matching grams.

For queries like **“arrange all elements in increasing order in an array”** we could see that close to 200 files were retrieved which severely effects the precision.

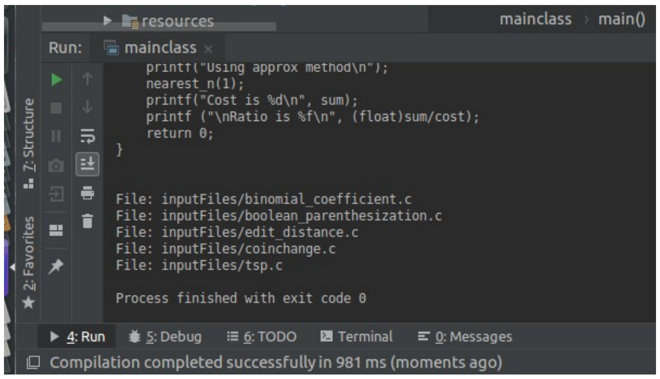
We could see that wildcard queries were returning better results. For queries like **“dyn*c”** results like coin changing, edit distance came up which are very relevant.



```
inputFiles/xor_swap.c
inputFiles/x_power_y.c
Total Results :: 249
Path : inputFiles/lexicographic_string_rank.c, Score : 228.78072
Path : inputFiles/permutations_of_string.c, Score : 177.76338
Path : inputFiles/std.c, Score : 124.09056
Path : inputFiles/heap_sort.c, Score : 114.00046
Path : inputFiles/matrix_chain_multiplication.c, Score : 112.78329
Path : inputFiles/validate_ip.c, Score : 111.880714
Path : inputFiles/infix_to_postfix.c, Score : 111.14948
Path : inputFiles/counting_sort.c, Score : 109.85071
Path : inputFiles/kruskal.c, Score : 104.55472
Path : inputFiles/postfix_evaluation.c, Score : 102.8937

Process finished with exit code 0
```

Compilation completed successfully in 1 s 758 ms (moments ago)



```
    printf("Using approx method\n");
    nearest_n(1);
    printf("Cost is %d\n", sum);
    printf ("\nRatio is %f\n", (float)sum/cost);
    return 0;
}


File: inputFiles/binomial_coefficient.c
File: inputFiles/boolean_parenthesization.c
File: inputFiles/edit_distance.c
File: inputFiles/coinchange.c
File: inputFiles/tsp.c

Process finished with exit code 0
```

Compilation completed successfully in 981 ms (moments ago)

The code can be improved a lot by implementing the concepts of multiple query generations by utilizing word/sentence embedding. We can also break down the code documents and make it go through graphical analysis or AST similarities can be used to detect semantic clones. The codes can be subjected to more customized tokenizers which will be able to map indexes better.