Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Scalable algorithms for data analysis (CS6713)
Assignment 4, Submission deadline : 7th July 2020, Marks : 30

In this assignment, we will implement a Hierarchical Bloom Filter tree for document similarity search. Hierarchical Bloom Filter (HBF) approach is used in digital forensics, bioinformatics etc. The goal is to create an index of a set of documents which supports similarity (nearest neighbor) search given a query document. In bioinformatics, genome sequence collections are indexed in place of documents. The set of articles could be diverse. In such cases, data driven approaches what embed the documents (such as doc2vec) in some compact space could suffer from the issue of highly diverse and limited collection of documents. In this assignment, we will use a simple Bloom filter based approach to efficiently index such a document collection.

Corpus Indexing

We build a Hierarchical Bloom filter over an input corpus (document collection). A hierarchical bloom filter tree is a tree whose nodes are Bloom filters. The Bloom filter parameters m, k and the set of hash functions are identical for all the Bloom filters in the tree. Each leaf node corresponds to a Bloom filter for one of the documents in the corpus. You can assume that the number of documents in the collection, denoted as N, is a power of 2. The Bloom filter for a document is created by treating the document as a bag of words (that is, a set of its words) where a words are simply white space separated terms in the article without any stemming. Now, a Bloom filter for this word set can be easily constructed. We denote the resulting Bloom filter as a document Bloom filter. The basic idea is that Bloom filters of two very similar documents will have lots of 1s in common. In other words, their Bloom filters will have a high Hamming similarity. Searching for a document similar to the query document now amounts to identifying a target document in the corpus whose Bloom filter has high Hamming similarity to the query document Bloom filter.

We now create a tree over the document Bloom filters for fast querying as follows. Consider a complete binary tree with N leaf nodes (that is $2N-1$ nodes in total) where each node in the tree holds a Bloom filter. Each leaf node holds a Bloom filter for one of the documents in the collection. Bloom filter for an internal node is defined recursively as follows: Bloom filter for an internal node is obtained by performing union of the Bloom filters corresponding to its two child nodes. Recall that, union of two Bloom filters is obtained by doing bitwise OR operation on the two Bloom filters. In other words, Bloom filter of an internal node corresponds to the union of all its leaf document Bloom filters. In your implementation, you may include the document leaf nodes in the final tree in any arbitrary order.

Querying

Given a query document q, first construct its corresponding Bloom filter with the same set of parameters and hash functions as in indexing. The search starts from the root node. The search uses a real number t as a threshold parameter where $0 \leq t \leq 1$. The search at a tree node u proceeds recursively as follows. Let $BF(q)$ and $BF(u)$ denote Bloom filters of query q and node u respectively. Perform bitwise AND of $BF(u)$ and $BF(q)$ to obtain a bit vector say s. If the number of 1s in s is at least t times the number of 1s in q, then the search proceeds recursively to both children of u. If the recursive search ends at one or more leaf (document) nodes, select the document among them whose Bloom filter has the least Hamming distance to the query Bloom filter as the final output document. If the search does not reach any of the leaf documents then declare the search result as empty (i.e, there are no similar document in the collection).

Such a hierarchical structure on the document Bloom filters result in an efficient and fast index. In the ideal case, the similar article is searched after inspecting only $O(\log(N))$ Bloom filters (nodes). This in practice could be a significant improvement over the naive approach of N comparisons for each query document.

Implementation Details

- Your implementation is a single executable named `hbf` (standing for Hierarchical Bloom Filter).
  Usage: `hbf corpus_folder query_folder m k t`

  Here, `corpus_folder` contains the text document files that are to be indexed and `query_folder` contains a set of query documents. The parameters `m` and `k` are the standard Bloom filter parameters (the number of bits and the number of independent hash functions), and t is a real number which is the threshold parameter as discussed earlier. For this assignment, it is sufficient to create an in-memory tree.

  You can re-use your Bloom filter implementation from your first assignment. For each file in `query_folder`, a separate query is performed on the tree and the file name in the `corpus_folder` corresponding to the matched document is printed. If the search returns empty then print "EMPTY". Along with the matched document, you also need to print the number of nodes in the tree that were visited while performing the query. In addition, after completing all document searches, you need to print the total number of articles in `corpus_folder` and the average number of tree nodes visited for a query.

- **Programming language:** You could use either C++ or Java for your implementation.

- **Additional Marks:** Additional 10 marks for those who creates the index tree based on a hierarchical clustering where similar documents in the corpus belong to the same subtree. In the original implementation, the leaf ordering in the tree is arbitrary. Having

similar documents in a subtree ensures that in its corresponding Bloom filter (that is, the Bloom filter corresponding to the subtree root), most 1s are due to a significant fraction of the documents in it. In other words, the Bloom filter does not become over dense. As a consequence, false hits at internal nodes are reduced and search performance is improved.