COL762 - Information Retrieval and Web Search
Assignment 1

Shravan Nawandar (2019MT10764)

Contents

1. ABSTRACT

The assignment involves building an efficient Boolean Retrieval System for the English Corpora. The algorithm broadly involves reading the documents to build posting lists for the tokens followed by compressing the posting lists using a variety of compression strategies. These compressed lists are used to implement single and multi keyword retrieval. We have also compared the performance of the 3 compression strategies implemented on some specified metrics.

2. INTRODUCTION

In this assignment, the goal is to build an efficient Boolean Retrieval System for the English corpora. The input document collection is a benchmark collection in TREC, consisting of English documents. Each document has a unique document id, and the overall document is represented as a XML fragment.

The overall task in the assignment is divided into 3 subtasks:

- Invert the document collection and build an on-disk inverted index, consisting of a dictionary file and a single file of all postings lists.
- Implement three different postings list compression models: c1, c2 and c3.
- Support single keyword and multi- keyword retrieval.

3. IMPLEMENTATION DETAILS

Implementation involves reading the documents, tokenizing them, building a posting list for all the tokens. Each file contained multiple documents, the file was split into the documents and each document was handled separately for the process mentioned henceforth. The documents were handled using BeautifulSoup for the same. These documents were cleaned, ridden of stopwords and finally tokenized. The dictionary was maintained to keep a log of these tokens, and each processed document was appended to the posting list of the tokens identified in that document.

This is followed by compressing the posting list using a compression strategy of choice. These compressions are crucial as the posting lists are practically too large to fit into memory. These compressed postings lists are written in a binary file whereas the starting address and the length of the posting list for each token is stored in a dictionary. The start memory address will help in locating a particular posting list with ease in the binary file. The length will help us identify where that particular posting list ends and make sure that we don't start reading positng list for next token.

During the retrieval phase, queries are read line by line and the same steps are performed on the tokens obtained after which we retrieve the postings list for each token from the binary file. These queries are also cleaned, stemmed and tokenized the same way as the documents were. Then, their corresponding posting lists are searched in the compressed indexed binary file, decompressed and gap-decoded. We store the intersection of all these gap-decoded posting lists in a separate list which eventually stores the set of all retrieved documents for the particular query.

3.1 Tokenizing

Each file in the data set consists of multiple documents. Each document is separated by the **”DOC”** tag. We first split the file into documents and then for each document we read the data within the list of tags specified in the input as [**xml-tags-info**]. The document identifier for each document is specified in the **”DOCNO”** tag.

The data is then split into tokens based on a list of delimiters which are **white-space and ,.:;”’** using the re library with the help of the following command.

```
re.split(” ,|␣|:|\\.|’|‘|\n|\”|;|\[|\]|\{|\}|\(|\)”,info)
```

We convert all the tokens to lower case characters for uniformity.

The next step involves removal of stop words from the list of tokens. Stop words are the words which occur very frequently in the documents and hence do not add any significant meaning to the query. Words like a,the,and etc. are usually considered to be stop words. We have been provided with a list of stop words in the input and before processing any token we check whether it is in the list of stop words or not.

Finally all the tokens are stemmed. Stemming involves removal of endings likeing from words. Here, I have used the porter stemmer to stem the tokens.

3.2 Building the Postings List

After tokenizing the data as specified above we use a python dictionary to store the posting list for each token. The posting list for each token is a simple python list.

We index the Document Identifiers linearly starting from 1 in the order in which they are retrieved. Hence, we append the document index in the posting list for a token if the token is present in the corresponding document. To handle the case of duplicates we check if the document is already present at the end of the list before appending it.

3.3 Compressing the posting list

Given the posting list now consists of a list of integers and each integer requires 4 bytes of memory to be stored irrespective of the magnitude of the number, we use compression strategies to reduce the number of bytes requires to store the list.

We first gap encode the posting list which helps to reduce the magnitude of the document ids. That is, instead of encoding document identifier numbers in

$$[n1, n2, n3, ...],$$

one could encode

$$[n1, n2 - n1, n3 - n2, ...]$$

as these gaps would then be smaller than encoding the actual integers.
This assignment involves using 5 different compression strategies which are described below along with the implementation strategies used for them:

- **C1 compression:**
  Encoding: Split the bin(x) in chunks of 7 bits, and each byte that is to be stored now contains these 7 bits in lsb with MSB set to 1 in all bytes except last one(least significant byte). The data in the first byte is padded with zeros to left if needed.

  Decoding: Simply read until the byte with 0 MSB and process all 7 lsb bits of read bytes accordingly.

  Implementation: For each element in the gap encoded list we keep extracting bits in sets of 7 and add 128 to all but the last byte as defined in the encoding strategy. This list of 8 bit numbers is appended to the resultant compressed posting list.
  While decoding we continue reading bytes until a number less than 128 is encountered which signifies that the next byte would mark the start of a new document id.

- **C2 compression:**
  Encoding: Encodes each integer with O(log x) bits and thus gets close to optimal number of bits for each integer in its compression. Let, lsb(a,b) denote b least significant bits (lsb) of the binary representation of the number a, l(x) = length(bin(x)) and U(l) denote the unary representation of a number n. Unary representation of a number n is simply (n-1) 1's followed by a zero. We encode the integer x as
  U(l(l(x))) .* lsb(l(x),l(l(x))1) .* lsb(x,l(x)-1) where .* implies simply concatenating the terms.

  Decoding: Read the unary code first to find l(l(x)) and then the next l(l(x))-1 bits to find l(x), finally read next l(x)-1 bits to find x. We append 1 at the most significant bit (msb) to the read bits while finding the actual integers l(x) and x.

  Implementation: We first aim to make a binary string representation for the encoding defined above collectively for the entire posting list of a token to minimize the number of bits wasted in padding. For each term in the posting list we convert it to its binary representation which gives us l(x) and following the same process for l(x) we get l(l(x)) and hence eventually the encoding for the term. This is repeated for all the terms and a combined binary string is made for the entire posting list which is padded with appropriate number of zeros to make it possible to convert the binary string into list of bytes. We also store the number of zeros required for padding. The binary string is then read in groups of 8 and converted to a byte number and written in the binary file.

  Similar process is used while decoding where we convert the bytes read into a binary string and then ignore the zeros padded initially and use the decoding strategy defined above to decode the string.

- **C3 Compression:**
  Encoding: Compresses each postings list entirely using Google's fast general-purpose compression library called snappy. Snappy simply takes a sequence of bytes as input and generates another sequence of bytes (compressed).

  Decoding: Each time postings-list is required, we bring its compressed version to memory, decompress it, and use the resulting decompressed version.

  Implementation: We store the gap encoded list as a byte array in a binary file where each element in the list has been allotted 4 bytes. We also store the starting memory address and length of the byte array for each posting list for further use.
  While decoding the output of snappy decompression is read into a binary file and using the starting memory address and length of each posting list the gap encoded list is obtained again.

3.4 Boolean Retrieval:

The query file is read line by line and each line is split into tokens which undergo stemming and stop words are removed to obtain the resulting list of queries to be searched for. Each query is treated the same way as document content and tokenized the same way. We keep a result list which stores intersection of the posting list of all the tokens. Whenever, a new token is found, we update the result list by the intersection of the result list and the posting list of the new token.

We support the following Boolean retrievals using the dictionary and index -

- **Single keyword retrieval** :- Return all document-ids whose document content contains the specified keyword.
- **Multi-keyword retrieval** :- Return all document-ids whose document content contains all the specified keywords. The implementation of multi-keyword retrieval requires the use of single keyword retrieval, as we find the intersection of all posting lists.

The intersection operation takes O(l1+l2) time where l1 and l2 are the sizes of the two lists because the lists are in sorted order and a linear pass over both the lists is sufficient to find their intersection.

4. EVALUATIONS

4.1 Metrics of interest:

- **Index Size Ratio (ISR):** Compute this metric as:

$$ISR = \frac{|D| + |P|}{|C|}$$

  where $|D|$ is the size of the dictionary file, $|P|$ is the size of the postings file, and $|C|$ is the size of the entire collection. All sizes are measured in number of bytes they occupy on disk. ISR has to be reported for inverted index without any of the compressions, and after applying each compression to the postings list.

Compression Strategy	ISR	Compression Speed	Query Speed
c0	0.3356	11325	37230
c1	0.124	14673	41300
c2	0.1052	102289	88300
c3	0.1728	39059	78000

Table 1. Metric Values for compression strategies

- **Compression Speed:** The total time taken to compress all the postings lists, which is the time difference between indexing with a compression strategy and indexing without any compression. It has to be reported for each compression strategy in units of milliseconds.
- **Query Speed:** Average time taken per query (including decompression of list(s) if required). This should be reported in µ-seconds per query, and computed as follows:

$$QuerySpeed = \frac{|TQ|}{Q}$$

where TQ is the time taken for answering (and writing the results to the file) for all given queries, and Q is the number of queries.