COL 764 Assignment 1 - Inverted Index Construction

2021 - Version 1.1

Deadline

Submission of the complete implementation, and the report on the algorithms is due on **August 31th 2021, 11:59 PM**. All submissions are to be made on Moodle.

Weightage

This assignment is evaluated against 100 marks. The breakup of marks is given at the end of the document.

Instructions

Follow all instructions. Submissions not following these instructions will not be evaluated.

1. This programming assignment is to be done by each student individually. Do not collaborate by sharing code, algorithm, and any other pertinent details with each other. You are free to discuss and post questions to seek clarifications. Please post all discussions related to this assignment under **"Assignment 1"** channel on Teams.

2. All programs have to be written either using **Python/Java/C/C++** programming languages only. Anything else requires explicit prior permission from the instructor. The machine we use to evaluate your submissions has the following versions:

   - **C, C++**: gcc7;
   - **Java**: java 11 (we will not use OpenJDK);
   - **Python:** version 3.7.

   We recommend you to use same versions to avoid the use of deprecated/unsupported functions or take care to ensure required compatibility.

3. A single zip of the source code has to be submitted. The zip file should be structured such that

- upon deflating all submission files should be under a directory with the student's registration number. E.g., if a student's registration number is `20XXCSXX999` then the zip submission should be named 20XXCSXX999.zip and upon deflating **all contained files** should be under a directory named ./20XXCSXX999 only (names should be in uppercase). Your submission might be rejected and not be evaluated if you do not adhere to these specifications.
- apart from source files, the submission zip file should contain a build mechanism *if needed* (allowed build systems are Maven and Ant for Java, Makefile for C/C++). It is the responsibility of each student to ensure that it compiles and generates the necessary executable as specified. **Please include an empty file in case building is not required** *(e.g. if you are using Python)*.
- the list of files to be submitted, along with their naming conventions and synopsis are specified under sections 1.5 and 1.6.

4. **Note that we will use only Ubuntu Linux machines to build and run your assignments.** So take care that your file names, paths, argument handling etc. are compatible.
5. You *should not* submit data files. If you are planning to use any other "special" library, please talk to the instructor first (or post on Teams).
6. Note that your submission will be evaluated using larger collection. Only assumption you are allowed to make is that all documents will be in English, have sentences terminated by a period, and all documents are in one single directory. The overall format of documents will be same as given in your training. The collection directory will be given as an input (see below).
7. **Note that there will be no deadline extensions. Apart from the usual "please start early" advise, I must warn you that this assignment requires significant amount of implementation effort, as well as some 'manual' tuning of parameters to get good speed up and performance. Do not wait till the end.**

---

1 Assignment Description

In this assignment, your goal is to build an efficient Boolean retrieval system for English corpora. Much of the effort will be on learning ways to develop compact, efficient-to-build, and efficient-to-query inverted index structures.

1.1 Input

You are given a document collection – extracted from 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 (see the colored box below). Only content enclosed in specific XML tags need to be indexed, and, unless otherwise mentioned, that is what we refer to as the indexable portion of the document (or 'document content') in the rest of this document.

The document collection itself is specified as a directory consisting of files, with each file containing one or more documents. Note that not all files contain same number of documents, but no document spans multiple files.

We also supply a (possibly empty) list of ”stop-words” that must not be indexed.

An Example of XML Formatted Document.

An example of XML formatted document.

We will specify the tag that contains the document identifier, (`<DOCNO>` in this example), and the tags that contain the indexable portion. For example, we could ask you to index `<HEAD>` and `<TEXT>` in this example collection.
You may safely assume that **all** documents in the collection have their document identifiers within the specified tag, and their indexable portion is strictly within the specified tags – i.e., if a document does not have any of the specified tags, you need not index anything from that document (the document itself can be dropped).
Finally, you can also assume that all documents are well-formed (i.e., no unbalanced XML tags, no arbitrary Unicode characters – only English ASCII terms etc.)

```
<DOC>
<DOCNO> AP880212-0001 </DOCNO>
<FILEID>AP-NR-02-12-88 2344EST</FILEID>
<FIRST>u i AM-Vietnam-Amnesty     02-12 0398</FIRST>
<SECOND>AM-Vietnam-Amnesty,0411</SECOND>
<HEAD>Reports Former Saigon Officials Released from Re-education
 Camp</HEAD>
<DATELINE>BANGKOK, Thailand (AP) </DATELINE>
<TEXT>
   More than 150 former officers of the
overthrown South Vietnamese government have been released from a
re-education camp after 13 years of detention, the official
Vietnam News Agency reported Saturday.
   The report from Hanoi, monitored in Bangkok, did not give
specific figures, but said those freed Friday included an
ex-Cabinet minister, a deputy minister, 10 generals, 115
field-grade officers and 25 chaplains.
   It quoted Col. Luu Van Ham, director of the Nam Ha camp south
of Hanoi, as saying all 700 former South Vietnamese officials who
had been held at the camp now have been released.
   They were among 1,014 South Vietnamese who were to be released
from re-education camps under an amnesty announced by the
Communist government to mark Tet, the lunar new year that begins
Feb. 17.
   The Vietnam News Agency report said many foreign journalists
and a delegation from the Australia-Vietnam Friendship
Association attended the Nam Ha release ceremony.
</TEXT>
</DOC>
```

If you are using Python, have a look at `BeautifulSoup` library for parsing XML. If

you are using other programming languages, talk to the instructor (or ask in Teams) and get approval for any other library(ies) you want to use.

1.2 Task

The overall task in this assignment consists of the following three subtasks:

1. Invert the document collection and build an on-disk inverted index, consisting of a dictionary file and a single file of all postings lists. You should use only the stop-word list that is supplied to you, and if it is empty then you must not eliminate any stop-words. You should use only the Porter stemmer (from `https://tartarus.org/martin/PorterStemmer`) to stem terms before indexing. Use `<white-space>` and `,.:;"'` as the set of punctuation, as delimiter, for tokenization.

2. Implement three different postings list compression models: $c1$, $c2$ and $c3$, whose details are given below in Section 1.4, and compute various performance metrics over them.

3. 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.

   Your program will have to take as input a large list of single and multi-word keyword queries, and write the results to a separate file. The specific format of the output file is described later on in this document. Note that the queries are to be evaluated in exactly the same order as given.

Note that you are not **required** to implement an efficient/compact dictionary, but you can choose to do so if it helps you in improving the performance metrics specified below.

NOTE:

The postings list compressions are considered for evaluation only if Boolean retrieval works correctly before and after applying these compressions on the index.

1.2.1 Bonus Task

In addition to the above, there is a bonus task – to implement one of the following two postings-list compression strategies: $c4$ or $c5$, which are considered the state of the art.

Qualifying for Bonus Task

You will qualify for bonus marks only if (a) Boolean retrieval works with $c1$, $c2$ and $c3$ compressions; and (b) Boolean retrieval works with either $c4$ or $c5$ chosen for the bonus task.

1.3 Metrics of Interest

Apart from functional implementation of the overall task, you should report the following metrics on the given document collection:

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 Speed: The total time taken to compress all the postings lists. It has to be reported for each compression strategy. This should be measured in milli-seconds.

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:

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

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

During evaluation, we also plan to use these metrics to compute the top performing submissions (who will earn additional points).

1.4 List Compression Methods

In this section, we will describe various compression techniques that you are expected to implement as part of the assignment. For simplicity, we call these compressions as $c1$, $c2$, $c3$, $c4$ and $c5$.

Gap encoding: Considering that the postings lists are organized with strictly increasing document ids, compressions $c1$ — $c4$ can be optimised to encode only gaps. That is, instead of encoding document identifier numbers in

$$[n_1, n_2, n_3, ...],$$

one could encode

$$[n_1, n_2 - n_1, n_3 - n_2, ...]$$

as these gaps would then be smaller than encoding the actual integers (refer to class notes for concrete examples of gap encoding).

Compression c1: Works with chunks of binary data as the data in computer memory itself follows this representation.

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 siginficant byte). The data in the first byte is padded with zeros to left if needed.

example: consider $x = 111119$ in base-10, whose binary representation is

$$bin(x) = 1\ 1011\ 0010\ 0000\ 1111$$

then $encoding(111119)$ generates three bytes with

$$10000110\ \ 11100100\ \ 00001111$$

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

Note that we use Gap encoding with $c1$.

Compression c2: 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, i.e., $U(1) = 0, U(2) = 10, U(3) = 110, ...$

Given these, we define the encoding and decoding in $c2$ as follows:

encoding:

$$U\left(l\left(l\left(x\right)\right)\right) \odot lsb\left(l\left(l\left(x\right)\right), l\left(l\left(x\right)\right) - 1\right) \odot lsb\left(x, l\left(x\right) - 1\right)$$

The symbol $\odot$ just denotes concatenation of bits.

example: For x = 119, $bin(119) = 111\ 0111$, $l(119) = 7$, $l(l(119)) = 3$, $U(3) = 110$, $lsb(3, 2) = 11$, $lsb(119, 6) = 11\ 0111$

$$\textit{encoding(119)} = 110\ 1111\ 0111$$

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. Don't forget to append 1 at the *most significant bit* (msb) to the read bits while finding the actual integers $l(x)$ and x.

Compression c3: This compression is different from the previous two because it compresses each postings list entirely using Google's fast general-purpose compression library called *snappy* (`https://github.com/google/snappy`). Python bindings are available from python-snappy (`https://github.com/andrix/python-snappy`). Note that snappy being a general-purpose compression library, it simply takes a sequence of bytes as input and generates another sequence of bytes (compressed). It is up to you to suitably represent the sequence of numbers in the (gap-encoded) postings list as a string and feed it to snappy compress/decompress functions.

encoding: Given a postings list p consisting of gap-encoded document identifiers, compress it using snappy library.

decoding: Each time postings-list is required, bring its compressed version to memory, decompress it, and use the resulting decompressed version. You can assume that after decompression the result fits in memory.

Compression c4: considers how flexibility in parameters can help in compression. For integer x and parameter $b = 2^k$ with $k > 0$, consider $q = \lfloor (x-1)/b \rfloor$ and $r = x - q * b - 1$, $C(r) = bin(r)$ is to represent integer r with k bits in binary, $U(l)$ is unary representation of l. The optimal value for k is based on trade off between bits required for unary representation and $C(r)$ for each x

encoding: $U(q+1).C(r)$
Either some fixed number of bits, over the entire index, can be used before each encoded integer to encode the k or chose previous integer compression methods c1 or c2.

example: consider $x = 119$, with say $k = 6, b = 64$, then $q = 1, U(1) = 0, r = 54, C(54) = 11\ 0110$.

$$\textit{encoding(119)} = 011\ 0110$$

decoding: read the unary prefix to find q, then read next k bits to find r to finally get back x

Compression c5: considers inverted list as a whole in the hope that it can compress better with the observed sequence patterns. Firstly, find the k such that most(say 80 to 90%) of the integers in the list can be represented using k bits per integer and then find suitable b such that these values fit in range $[b, b + 2^k - 2]$.

encoding: each number is shifted with respect to b to make the numbers fall in the range $[0, 2^k - 2]$. The numbers which fall out (i.e., $n_i > 2^k - 2$) can be represented with separate simple list or try using any previous encoding scheme on each number in this separate list.

example: Consider an inverted list, $l = [8, 11, 12, 19, 24, 34, 42, 62, 214, 422]$, now considering b = 8 and k = 6 the encoded list will then be $[0, 3, 4, 11, 16, 26, 34, 54, *, *]$ with a separate list of $[214, 422]$. Either some fixed number of bits, over the entire index,

can be used before each list to encode the b and k or chose previous integer compression methods c1 or c2. Mark with a symbol(say bit sequence corresponding to $2^k - 1$) in the encoded list at corresponding positions so as to place integers from the separate list in correct positions during decoding.

1.5 Program Structure

In order to achieve these, you are required to write the following programs:

1. **Inverted indexing of the collection:** Program should be named as `invidx_cons.{py|c|cpp|C|java}`.
   It will be called via a shell script as follows:
   `invidx.sh [coll-path] [indexfile] [stopwordfile]` {**`0|1|2|3|4|5`**} **`[xml-tags-info]`**
   where,

`coll-path`	specifies the directory containing the files containing documents of the collection, and
`indexfile`	is the name of the index files that will be generated by the program.
`stopwordfile`	is a file that contains the stopwords that should be eliminated. The file could potentially be empty. If not empty, it contains one stopword in each line.
{`0|1|2|3|4|5`}	these specify the compressions that will be applied – `0` specifies no compression, and `1`—`5` correspond to $c1$ to $c5$ as listed above. **Please print "`not implemented`" on console if you have not implemented a particular technique**.
`xml-tags-info`	is a file that contains the tag for document identifier in the first line, followed by tags that contain the indexable portion in each line, starting from the second line. For the example in section 1.1, this file would have `HEAD` and `TEXT` in each line. You can assume that document ids are ***always under <DOCNO> tags***.

   `invidx.sh` is a BASH shell-script that will be a wrapper for your program. Do not use this script for building/compiling your assignment (which should be done using `build.sh`).

   The program should generate **two** files:

   (a) **`indexfile.dict`** contains the dictionary and

   (b) **`indexfile.idx`** contains the inverted index postings. Note that it is expected to be a *binary file* (not printable text) containing the sequence of document identifiers for each postings list. Further, it can also contain –if needed– a mapping between document identifier given in the collection, and the document number used in the

index. There is no restriction on how these (and any other) info will be stored in the postings list file.

Construct an appropriate inverted index consisting of postings-list and dictionary structure which are stored on disk in the specified filenames.

2. **Search and rank:** Your submission should also consist of another program called `boolsearch.sh` for performing Boolean retrieval using the index you have just built above. It will be called via a shell script as follows:

```
boolsearch.sh [queryfile] [resultfile] [indexfile] [dictfile]
```

`queryfile`	a file containing keyword queries, with each line corresponding to a query
`resultfile`	the output file named `resultfile` which is generated by your program, following format (see below)
`indexfile`	the index file generated by `invidx_cons` program above which should be used for evaluating the queries
`dictfile`	the dictionary file generated by the `invidx_cons` program above which should be used for evaluating the queries

Any other additional information (e.g., compression technique used) must be stored as part of index file.

Result Output Format: We will adapt the format used by the **trec_eval** (`https://github.com/usnistgov/trec_eval`) tool for verifying the correctness of the output using a program. Therefore, it is **very** important that you follow the format exactly as specified below. The result file format instructions are given below:

Output Format

Lines of results_file are of the form

```
qid     docno          sim
Q0      ZF08-175-870   1.0
```

giving document numbers (a string, as given in the collection) retrieved by query qid ('Q' followed by the line number of the query in the query-file starting from 0) with similarity sim (a float, always the same value for all results in Boolean model). The result file may not contain NULL characters.

Note that the qid and docno are the main strings for evaluation in each line and sim could just be 1 for all documents in these boolean retrieval tasks.

1.6 Submission Plan

All your submissions should strictly adhere to the formatting requirements given above. You might choose to split your code by creating additional files/directories but it is your responsibility to integrate them and make them work correctly. You can also generate temporary files at runtime, if required, **only within your directory**.

- You should also submit a README for running your code, and a PDF document containing the implementation details as well as any tuning you may have done. **Name them** README.{txt|md} **and** 20XXCSXX999.pdf **respectively**.
- The set of commands for evaluation of your submission roughly follows -

  Tentative Evaluation Steps (which will be used by us)

  ```
  $ unzip 20XXCSXX999.zip
  $ cd 20XXCSXX999
  $ bash build.sh
  $ bash invidx.sh <arguments>
  $ bash boolsearch.sh <arguments>
  ```

- Only `BeautifulSoup`, `PorterStemmer`, `snappy` and native libraries (e.g. `re`) will be available in the Python evaluation environment. **You should not import and use dependencies of these libraries directly in your code.** Include the source code of any imports in your submission for other languages (after getting prior permission from the instructor). No external downloads will be allowed for any language environment at runtime.

1.7 Tentative breakup of marks assignment

In general, a submission qualifies for evaluation if and only if it adheres to the specifications given above (arguments, structure, use of external libraries, correct output format, input format adherence, etc.). Given this requirement, the marks assignment for correct implementation of:

basic inverted index (postings list + dictionary)	20
c1, c2, and c3	10 marks each (30 marks)
Boolean retrieval (single term)	5
Boolean retrieval (multi-term)	10
Report	10
Shell scripts	5
Marks for being in the top-10 of the at least two of three metrics given above (ISR, Compression Speed, Query Speed)	20
Total	100
Bonus	15