COL 764 Assignment 1 - Inverted Index-based Search System

2023 - Version 1.0 (dated 13th August 2023)

Deadline

Submission of the complete implementation, and the report on the algorithms is due on **August 28, 2023, 11:59 PM**. All submissions are to be made on Moodle.

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 call signatures as given in the instructions below.
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 ranked retrieval system for an English corpora. Much of the effort will be on learning ways to develop compact, and efficient-to-query inverted index structures, and evaluate their performance.

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 indexing is *full text* – i.e., you should not be doing any stop-word elimination or stemming on the document collection.

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.

An Example of XML Formatted Document.

An example of XML formatted document.
Each document is enclosed in `<DOC>..</DOC>` tags. Within each document, following XML tags are present:

1. `<DOCID>...</DOCID>`: Encloses the document identifier. Note that the document identifier here is not necessarily an integer. It is upto you to remap these ids to integers for indexing purposes (if required).
2. `<TITLE>...</TITLE>`: Encloses the title of the document. It could be empty.
3. `<CONTENT>...</CONTENT>`: This encloses the main content of the document. Note that this content in turn could contain some SGML/HTML/XML tags, and it could be empty.

Contents under `TITLE` and `CONTENT` tags are to be indexed. 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. 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>
<DOCID>FT911-1</DOCID>
<TITLE>
FT  14 MAY 91 / (CORRECTED) Jubilee of a jet that did what it was designed
to do
</TITLE>
<CONTENT>
Correction (published 16th May 1991) appended to this article.
'FRANK, it flies]' shouted someone at Sir Frank Whittle during the maiden
flight of a British jet. 'Of course it does,' replied Sir Frank, who
patented the first aircraft gas turbine. 'That's what it was bloody well
designed to do, wasn't it?'
```

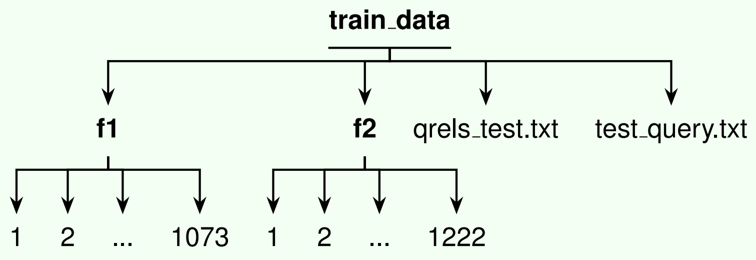
```
...
</CONTENT>

</DOC>
<DOCID> FR940104-0-00002 </DOCID>
<TITLE> <\TITLE>
<CONTENT>
<!-- PJG STAG 4700 -->
<!-- PJG ITAG l=94 g=1 f=1 -->
<!-- PJG /ITAG -->
<!-- PJG ITAG l=69 g=1 f=1 -->
<!-- PJG /ITAG -->
<!-- PJG ITAG l=50 g=1 f=1 -->
 DEPARTMENT OF AGRICULTURE
 ...
 </CONTENT>
 </DOC>
```

1.2 Corpus Details

The training corpus is available in HPC for download at the following path:
$/home/cse/phd/csz208507/scratch/COL764\_IR\_2023/train\_data$

Directory Tree



Here *train_data* is a folder, f1 and f2 are subfolders. Subfolder f1 contains files named *1, 2, ..., 1072, 1073* and subfolder *f2* contain document files named *1, 2, ..., 1021, 1022*. Each file contains one or more documents (see the document format description above).
You are required to index documents contained in files in **both** directories. *test_query.txt* file contain queries. *qrels_test.txt* contain results for all the queries in *test_query.txt*.

qrels_test.txt format

Lines of qrels_test.txt are of the form

```
qid   iteration      docid         relevancy
Q         0       SZF08-175-870        1
Q         0       SZF08-175-871        0
```

where **qid** is the Query ID,
iteration is the feedback iteration (almost always zero and not used),
docid is the official document id that corresponds to the ”DOCID” field in the documents, and
relevancy is a binary code of 0 for not relevant and 1 for relevant.

The order of documents in a qrels_test file is not indicative of relevance or degree of relevance. Only a binary indication of relevant (1) or non-relevant (0) is given. Documents not occurring in the qrels file were not judged by the human assessor and are assumed to be irrelevant in the evaluations .

test_query.txt format

There are 100 queries in the test_query.txt file.

An example of Query in test_query.txt file

```
<top>
<num> Number: 301
<title> International Organized Crime
<desc> Description:
Identify organizations that participate in international criminal
activity, the activity, and, if possible, collaborating organizations
and the countries involved.
<narr> Narrative:
A relevant document must as a minimum identify the organization and the
type of illegal activity (e.g., Columbian cartel exporting cocaine).
Vague references to international drug trade without identification of
the organization(s) involved would not be relevant.
</top>
```

where a query is enclosed in `<top>...<top>` tags. With in each query, the following XML tags are present:

1. `<num>` : Encloses the query `( also known as topic )` identifier. Note that the query identifier here is not necessarily an integer.
2. `<desc>` : Encloses the query/topic.
3. `<narr>` : Encloses the Narrative or say query need.

You are expected to make use of content under `<desc>` and `<title>` tags. The `<narr>` content is optional – you are free to experiment with its use for queries.

1.3 Task

This assignment consists of the following three subtasks:

1. Invert the document collection and build an on-disk inverted index, consisting of (at least) (i) a dictionary file and (ii) a single file consisting of all postings lists. You should not perform any stop-word elimination and stemming. For tokenization you have two options
   (a) **Simple tokenizer**: Use `<white-space>` and `,.:;"'` as the set of punctuation, as delimiter, for tokenization.
   (b) **BPE tokenizer**: While there are many implementations and pre-trained BPE tokenizers, you are expected to implement BPE tokenizer yourself. You may assume that tokens don't span word boundaries detected in the simple tokenizer. You can use a random 25% sample of the given corpus as the training corpus for the tokenizer.
2. Implement postings list with two different storage models:
   (a) **No compression**: In this you will have no compression of document ids, and/or TF scores etc. However, you may use the document ids mapped to integer space, and delta encoding as required.

(b) **Variable byte compression**: For document id and TF, you must use variable byte encoding.

3. You must provide a ranked retrieval system using TF-IDF scoring using the index you have constructed above.

   (a) **Term Frequency (TF)**: We will use
   $$tf_{ij} = 1 + \log_2(f_{ij})$$
   as the term-frequency component, where f_{ij} is the number of times the index-term i appears in document d_j.

   (b) **Inverse Document Frequency (IDF)**: We will use
   $$idf_i = \log_2\left(1 + \frac{N}{df_i}\right)$$
   where df_i is the number of documents that the index-term i appears in, and N is the total number of documents in the collection.

   Your program will have to take as input a large list of keyword queries, and output 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:

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

1.4 Metrics of Interest

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

Performance of Ranked Retrieval: You will required to generate up to 100 best results for each query. We will use $F_1@\{100, 50, 20, 10\}$ as the metric to evaluate how well your retrieval performance is. We expect the performance of all submissions for the case of normal tokenizer to similar.

Retrieval Efficiency: Average time taken per query (including decompression of list(s), dictionaries, tokenization of queries as required) from the time query is passed to the system to the time all (top-100) results are retrieved. It is computed as follows:
$$Efficiency = \frac{|T_Q|}{Q}$$
where T_Q is the time taken for answering (and writing the results to the file) for ***all*** given queries, and Q is the total number of queries. ¡¡¡¡¡¡¡ HEAD

(For Bonus Marks) Index Construction Efficiency: You can optionally claim to have your index construction speed be evaluated (in uncompressed setting). If your construction takes more than threshold time (to be announced soon), then you will be awarded -10 marks. Otherwise, top-10% fastest implementations will get 10 bonus marks, next-25% will get 7.5, and remaining will get 5 bonus marks. If you would like to take this up, mention it in the submitted report, along with the construction time you have measured.

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]` {**`0|1`**} {**`0|1`**}
   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.
`{0|1}`	specify the compressions that will be applied – `0` specifies no compression, and `1` specifies variable byte compression. **Please print "`not implemented`" on console if you have not implemented the compression**.
`{0|1}`	specify the name of the tokenizer used by the program – `0` specifies the simple tokenizer and `1` specifies the BPE tokenizer.

   `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 `tf_idf_search.sh` for performing TF-IDF retrieval using the index you have just built above. It will be called via a shell script as follows:

   `tf_idf_search.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 adopt 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   iteration      docid          relevancy
Q        0       SZF08-175-870          1.0
```

giving document id (a string, as given in the collection) retrieved by query qid ('Q') with relevancy (i.e., the TF-IDF score as a floating point number). It is expected that for each qid, the docno's are sorted in decreasing order of their relevancy values. The result file may not contain NULL characters.

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 PDF report should also contain details of how the experiments were conducted, what the results are –speed of construction, performance on the released queries, query execution time etc.
- Only `BeautifulSoup`, `lxml`, `nltk`, and libraries distributed with python (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.
- You are also required to share the details of directories on HPC, that contain Index and Dictionaries that you built.

1.7 Evaluation Plan

- 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 tf_idf_search.sh <arguments>
```

- Note that all evaluations will be done on not only the queries that are released, but on a set of "hidden" queries. We will use a different query file than test_query.txt given in the shared folder, although the format of the query file will remain the same. Thus, your implementation should be able to handle new terms in the query.

It is important to note that instructor / TA will not make any changes to your code. If your code does not compile / execute as expected, you will be called for a meeting with the TA where you can make minor changes to the code. There will be a penalty of 10% of the overall marks for such a case – however minor the change is going to be. It is your responsibility to guard against this.

1.8 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)	30
BPE tokenizer	15
simple tokenizer	5
Variable byte compression	10
Ranked retrieval performance	15
Retrieval efficiency	10
Report	10
Shell scripts	5
Total	100
Bonus	10

Note that in ranked retrieval performance, and retrieval efficiency, the evaluation is relative. That is, top-10% will get full marks, next 10% will receive 9/10 of the marks, next 10% will receive 8/10, ... and so on.