CS 276 / LING 286 Spring 2016

Programming Assignment 1

Due: Thursday, April 14, 2016 at 11:59pm

Overview

In this programming assignment, you will be applying knowledge that you have learned from lecture to build a simple indexing and retrieval system for a search engine. This assignment should be done in teams of two or individually. More specifically, it involves the following tasks:

1. Build an uncompressed index over a corpus of webpages from the stanford.edu domain (c.f. Sec. Corpus), and implement retrieval for Boolean conjunctive queries.
2. Build a compressed index over the same corpus, using variable length encoding and implement Boolean conjunctive queries.
3. Write up a report describing your program and answer a set of questions.
4. For extra credit, you are encouraged to implement other compression algorithms (e.g., gamma-encoding)

You are required to submit your code on corn (which we will run with an auto-grader) and a write-up via Gradescope. See later sections for more details.

Starter Code

Download the Java starter code from: http://web.stanford.edu/class/cs276/pa/pa1-skeleton.zip. You can use `ant` or Eclipse to build the code. For a quick introduction to Java, links to helpful tutorials, and instructions to build the code, check out this document.

Corpus

The corpus you will be working with for this assignment contains webpages from the stanford.edu domain. The data for this assignment is available as a `.zip` file at: http://web.stanford.edu/class/cs276/pa/pa1-data.zip. The total size of the corpus is about 170MBs. There are 10 sub-directories (named 0-9) under the data directory. Each file under the sub-directory is the content of an individual webpage. You should assume that the individual file names are unique within each sub-directory, but not necessarily the case across sub-directories (i.e., the full path of the files are unique). All the HTML info has been stripped off from those webpages, so they only contain space-delimited words. You should not re-tokenize the words. Each consecutive span of non-space characters consists of a word token in the corpus.

Task 1: Building an uncompressed index and retrieval (50%)

The first part of this assignment is to build an inverted index of this corpus, and implement Boolean conjunctive queries. In particular, you need to implement the blocked sort-based indexing (BSBI) algorithm [1]. You should treat each sub-directory as a block, and only load one block in memory at a time when you build your index (note: you will be penalized if you construct the index by loading blocks of larger than one directory in memory at once). Note that we are abstracting away from the operating systems meaning of blocks. You can assume that each block is small enough to be stored in memory.

You will use Java for this programming assignment. You can find some sample skeleton code under the `pa1-skeleton` directory inside the assignment package.

All of your submitted code for this PA should reside in a directory `pa1-skeleton`. Under the `pa1-skeleton/task1` directory, you must supply two shell scripts, named `index.sh` and `query.sh`. Each script should invoke the necessary Java program to perform the indexing and retrieval tasks. The `index.sh` and `query.sh` scripts that invoke the `cs276.assignments.Index` and `cs276.assignments.Query` Java programs can be found under `pa1-skeleton/task1`. Prior to running the scripts, be sure to compile using `ant` or Eclipse. If you change the scripts, make sure that they can be invoked from the parent directory (however, you may not change the input arguments for the sake of our auto-grader). For example,

```
cd pa1-skeleton
ant
sh task1/index.sh [input arguments - see below]
sh task1/query.sh [input arguments - see below]
```

For convenience, we have marked where to add your code with `TODO: Your code here` comments in the Java files. Please feel free to add any helper methods you find helpful.

[1] http://nlp.stanford.edu/IR-book/pdf/04const.pdf

Indexing

The index.sh script should take two command-line arguments:

- input_data_dir: a string-valued argument. This describes the directory location of the input corpus data.
- output_index_dir: a string-valued argument. This is the location of the output directory containing the generated index. You should assume the output directory does not exist yet.

In addition, the index.sh script should **only** print the total number of files in the corpus to stdout. At grading time, we will be invoking the index.sh script as follows.

```
sh task1/index.sh input_data_dir output_data_dir
```

We will then collect timing statistics on how long the indexing takes, the size of your generated index, and the amount of memory used when building the index. We will also verify that the total number of documents you output matches the expected count.

Boolean conjunctive retrieval

The script query.sh will take one command-line argument, which is the directory location of the index you built. Your retrieval program should read from stdin a sequence of word tokens separated by space, which forms the Boolean conjunctive query. There could be any non-zero number of query terms. Your program should print to stdout the list of documents containing the query terms, one document file name on each line, **sorted in lexicographical order**. The document file name should only include the subdirectory name and the file name. For example,

```
0/crypto.stanford.edu_
0/crypto.stanford.edu_DRM2002_
1/crypto.stanford.edu_cs142_
1/crypto.stanford.edu_cs155_hw_and_proj_
```

If the conjunctive query has no results (could also be caused by any of the terms in the query is not being found in the corpus), your program should output

```
no results found
```

to stdout.

It is important to make sure that you are **not loading the whole index into memory**. The whole point of indexing is that you can seek to the corresponding index position of a query term on the disk, and just read into memory the posting list of that term, without having to load the full index.

You are required to **implement efficient intersection of postings lists**. You will be penalized for using the programming language's set intersection operation. You are required to **order the terms by postings list length** to optimize query performance.

You will be penalized if you do not implement this optimization. We will be invoking your script with the following command.

```
sh task1/query.sh index < query.txt
```

where `index` is the directory where your index resides and `query.txt` contains a list of query terms, all in one line.

Toy dataset

It is usually a good idea to verify your programs output on a small toy dataset first during development. We have provided you with a small toy dataset and a set of sample queries and outputs, under the `toy_example` directory.

There is a script called `grader.sh` under the same toy example directory. It is similar to the auto-grader script we use for final evaluation. You should be able run it to verify your program meets our input/output requirements. Also, a set of development queries and their outputs are given at the `dev_queries` and `dev_output` directories. The script `run_dev.sh` is provided to help you run these development queries.

Grading

To ensure your index is built correctly, you will be tested on 20 Boolean conjunctive queries of one or multiple terms. For those queries, you will get 1.5% of the final grade for each query you answer correctly, for a total of 30% of your grade. You will earn another 20% if none of the following penalties apply to your program. You will be penalized by

1. 5% if your program does not load one block at a time in indexing (see our notes at the beginning of Task 1 description).
2. 5% if you do not order the query terms by postings list length in retrieval
3. 4% if your retrieval program loads the whole index into memory rather than simply loading the postings lists of just the query terms.
4. 3% if the total number of files your indexer outputs does not match the correct count.
5. 3% if the timing statistics of your retrieval algorithm are way out of the norm.

The maximum combined grades for this task is 50%.

Task 2: Building a compressed index and retrieval (30%)

In the second task, you should build a compressed index using gap encoding with variable byte encoding for each gap. The input/output format and code structure is the same as Task 1, but you should provide scripts under directory `pa1-skeleton/task2`. We will also use an auto-grader to build a compressed index with your program, and run a set of queries. Similar to Task 1, we will be invoking your scripts in a similar fashion:

```
cd pa1-skeleton
ant
sh task2/index.sh [input arguments - discussed above]
sh task2/query.sh [input arguments - discussed above]
```

Grading

Similar to Task 1, you will be tested on 20 conjunctive queries (1% for each query you answer correctly, for a total of 20%). And you will earn another 10% if none of the following penalties apply:

1. 5% if your compression algorithm does not achieve a reduction in index size.
2. 5% if your query response time with the compressed index is way out of norm.

The combined total grade of this task is 30%. Note that if your program does not implement the variable length encoding compression algorithm, you will receive **no credit** for this task.

Task 3: Report (20%)

Please write up a 1-2 page report and submit it alongside with your code. It should contain the following sections:

1. A brief description of how your program is structured and the key steps in your indexing and retrieval algorithms. Make sure you report statistics on the size of the index (compressed and uncompressed), statistics of retrieval time for the development queries. We provide helper scripts which output relevant timing and memory statistics. (5%)
2. Answers to each of the following questions: (5% each)
   a) In this PA we asked you to use each sub-directory as a block and build index for one block at a time. Can you discuss the trade-off of different sizes of blocks? Is there a general strategy when we are working with limited memory but want to minimize indexing time?
   b) Is there a part of your indexing program that limits its scalability to larger datasets? Describe all the other parts of the indexing process that you can optimize for indexing time/scalability and retrieval time.
   c) Any more ideas of how to improve indexing or retrieval performance?

Please note if you go drastically beyond the 2-page limit, we may penalize you for an overly long report.

Grading

Your write-up will contribute to 20% of your final grade.

Extra credit: experiment with additional compression methods (15%)

Implement one more index compression method (e.g., gamma-encoding), and supply your code in the same structure as Task 1 and 2 under directory `extra_credit`. Section 5.4 in the textbook points to a few techniques and related publications. You can also look into ”Delta Encoding”, a compression technique very similar to Gamma Encoding, which can achieve an even better compression rate. Also, Jeff Dean's slides point to a few interesting encoding techniques that are used in Google. Similar to Tasks 1 and 2, we will be invoking your scripts with

```
cd pa1-skeleton
ant
sh extra_credit/index.sh [input arguments - discussed above]
sh extra_credit/query.sh [input arguments - discussed above]
```

Also include in the report a description of your index compression method as well as a discussion of the space/speed trade-off of this additional compression method, in comparison to Task 2 and Task 1.

Grading

You will be awarded 10% for answering 20 queries correctly (0.5% each). Note that since the objective of this task is to achieve better compression rate, you'll receive 0% out of 10% if the size of your compressed index is larger than those of Tasks 1 and 2. 2% will be given for the space/speed discussion in your report. An extra 3% will be given to the top 10 teams who answer all 20 queries correctly and achieve the best compression rates. You will be penalized if the retrieval times are abnormally long. We will inspect your code for implementation correctness if necessary.

Submission

We will use corn and Gradescope for grading and collecting the assignment. Make sure that all `.sh` scripts are executable. There are two steps for submission:

1. Submitting your code on corn using our `submit` script
2. Uploading your report as a pdf file to Gradescope

Submitting Code

Before, submitting **please make sure that** your code follows the following directory structure to ensure auto-grading goes smoothly.

```
pa1-skeleton
├── build.xml
├── src
│   └── <any source files of your program>
├── task1
│   ├── index.sh
│   ├── query.sh
│   └── <any additional files your program needs>
├── task2
│   ├── index.sh
│   ├── query.sh
│   └── <any additional files your program needs>
└── extra_credit (optional)
    ├── index.sh
    ├── query.sh
    └── <any additional files your program needs>
```

In addition, it will be a good idea to run the `grader.sh` script on the toy data set, and the `run_dev.sh` script on the development queries prior to submission. Note that the indexes for Task 1, Task 2, and Extra Credit have to be constructed from scratch (that is, they are not built upon indexes from other tasks). Also, remember that indexes in Task 2 and Extra Credit will need to be stored in binary format.

Note that we will compile and run your program on Farmshare using `ant` and our standard `build.xml`. If you did not complete the assignment on Farmshare, please verify your code compiles and runs on it before submission.

You will submit your program code using a Unix script that we've prepared. To submit your program, first put your files in a directory on Farmshare (e.g. corn.stanford.edu). Then, from your parent directory (this would be `pa1-skeleton` in the above directory structure), submit, i.e.:

```
cd pa1-skeleton
/afs/ir/class/cs276/bin/submit
```

If you are working in a team, only one team member needs to submit, but remember to indicate your partner's SUNetID when prompted by the submit script.

Uploading Report to Gradescope

Please upload your report to Gradescope under the Assignments section. Again, if you are working in a team, only one team member needs to submit, but remember to add your partner as a team member for the assignment on Gradescope.