CS 345: Homework 7 (Due: Dec 9, 2014 by Noon)

Rules. Teams of no more than two students as explained in syllabus (Handout 1). This is to be handed back no later than noon time the day it is due in electronic form as a single tar or zip file decompressible on an AFS machine (afsconnect1.njit.edu or afsconnect2.njit.edu), where testing would take place. (At a minimum, the zipfile should contain a text file `HW1_ABCD_EFGH.txt` as explained below.) The code should be compilable/interpretable and executable on one or the other identical AFS machines in any language available there (emphasis in C, C++, Java, Python, Perl).

1 The index and query engine of FDSEE

This assignment is a continuation of HW6. The input to this assignment is the output of Step 4 (stemming) of HW6 or Step 2 or 3 (token or stopword) depending on the level of implementation of HW6.

1.1 Objectives of the assignment

The output before or after stemming of previous phases is already ordered (in fact grouped) by `docID` and `wpos` as documents are tokenized in some sequence and from the first word or character to the last. All tokens of a given document are extracted before moving to the next document.

In Step 1, you order the output stream based on `wordID` first (increasing order), then `docID` (increasing order) and finally `wpos` (increasing order). Using this information, you will build an inverted index where you will store information about word occurrences (i.e. inverted lists) in the form of a combined `vocabulary` and `occurrence list (inverted list)` structures as described in class and summarized in this document for completeness. For testing purposes this construction will have an interesting side-effect.

In Step 2, you will be asked to design a query system that implements simple logical (`AND` and `OR, NOT`) operations.

This is the minimum implementation required of this assignment. You can enrich this implementation by adding additional features.

1.2 Project deliverables

After compilation the executable unit produced should be named `fdsee`. If your code is interpreted the wrapper function should be `fdsee` possibly with an appropriate suffix as needed. We shall refer to that file as `fdsee` in the remainder. Every source file you submit must include in the form of comments in the first 5 lines the names of the members of the group including the last four digits of their NJIT IDs. In addition a file named `HW6_ABCD_EFGH.txt` (or `HW6_ABCD.txt` if the team has only one member) needs to be included (that also conforms to the first-5-line convention) that includes instructions for compilation/interpretation, bugs, and anything else of interest (eg extensions). The `ABCD, EFGH` are the last 4 digits of the NJIT IDs of the students of the team.

1.3 Step 1: Inverted Index

```
% ./fdsee invert  filename
% ./fdsee invert  dirname
```

For this step, the output of Step 2-4 of Homework 6 is being processed. Thus `invert` can take the output of `stem` or `stopword` or `token` or you can just make it behave as if it combines in itself the steps of `step, stopword, token` that you have implemented and then it build an index by reordering this stream. The reordering involves sorting of the stream by `wordID`, `docID`, `wpos` and in that order. Then an index is built consisting of a vocabulary and occurrence lists.

1. Vocabulary and occurrence lists. An inverted index consists of a `Vocabulary` and `occurrence` or `inverted lists`.

2. Vocabulary: technical details. The `Vocabulary` will contain the `wordID`s that survived stemming sorted by `word` not `wordID`. (Thus algorithm might be in the Vocabulary but algorithms will not if it got stemmed into algorithm.) For every entry in the Vocabulary you need to record information such as `wordID`, `Ndocs`, `Nhits`, and potentially a pointer `LocP` to the actual entries.

Entry `Ndocs` counts the number of distinct documents that contain the word (original or stemmed). Entry `Nhits` counts the number of occurrences of `wordID` in all documents (a given word might occur more than once in a document) and in any possible form (attr or stem).

Although `LocP` traditionally points to `Ndocs` linked lists of total length `Nhits`, for out case `LocP` will point to a file named `word` or `word.wordID`. This file would include all the tuples re `wordID`, sorted first to last as explained earlier.

1.3.1 Side Effects

Besides the side-effects of `fdsee` of HW6 additional side-effects will be generated.

Side-Effect 1: Vocabulary file. Command `invert` will generate a file named `fdsee_vocabulary` that includes a dump of the vocabulary in the form of tuples (`wordID`, `word`, `Ndocs`, `Nhits`). It suffices to print the four elements of the tuplet, one set of elements per line without the surrounding parentheses or the commas but include one space inbetween. Note that this file is ordered by `word`! This file will be located in the following directory along with the rest of the index.

Side-Effect 2: Directory `fdsee_invindex`. The vocabulary file will coexist in this directory created by command `invert`. Inside that directory you will create a number of files one for each word in the vocabulary. In file `word` or `word.wordID` you will store the occurrence lists of `word` in the form of triplets (`docID`,`wpos`, `attr`). Note that obviously, one does not need to store in it `wordID` or `word`. The format of the file will be as follows: one tuple per line, with the three elements as specified earlier. Parentheses or commas are optional, but use a space to separate the elements. The number of lines of that file should be `Nhits`.

1.4 Query engine implementation

Having completed the index we ask you to implement a command line-based implementation of a simple up to three-term query language.

```
% ./fdsee and2    term1 term2                        ; term1  AND term2
% ./fdsee and3    term1 term2  term3                 ; term1  AND term2 AND term3
% ./fdsee or2     term1 term2                        ; term1  OR  term2
% ./fdsee or3     term1 term2  term3                 ; term1  OR  term2 OR  term3
% ./fdsee andnot  term1 term2                        ; term1  AND -term2
% ./fdsee and2not term1 term2  term3                 ; term1  AND term2 AND -term3
% ./fdsee next5   term1 term2
```

Each one of the operations above (implicitly) assumes that `./fdsee invert dfname` is first executed. (That is, `fdsee_invindex` already exists and is well formed.)

The outcome of `and2` is to read the occurrence lists of the wordIDs of `term1` and `term2` and find their common intersection docID-wise, i.e. find those documents that contain both terms. The output is a list of the documents containing both words, one per line. Each line prints not only the `docID`s but also the qualified URLs of each document. The `and3` allows for a 3-term conjunction.

The outcome of `or` is that of a disjunction and the two variants behave analogously to `and2, and3`.

The outcome of `andnot` is to read the occurrence lists of the wordIDs of `term1` and `term2` and find those documents in which `term1` appears but not `term2`. The printout of the result is as before. For `and2not` the documents that contain `term1, term2` but not `term3`,

Note that for this part we only need to use `docid` information and neither `wpos` nor `attr` to generate an answer to the query. A more elaborate processing can occur using `wpos` and `attr` position that will also rank the results.

Alternatively, you may implement the following

```
% ./fdsee boolean "mterm1  op1 mterm2 op2 mterm3 op3 mterm4"
% ./fdsee next5   term1 term2
```

where `mterm` is either a `term` or `-term` and the − indicates a NOT. An `op` then can be either an `AND` or an `OR`. Conjunctions have higher precedence than a disjunction! The operator `next5` checks whether the terms `term1` or `term2` appear next to each other in a document (i.e. their `wpos` differ by 5 or less. ■

Date Posted: 8/27/2014