CS 345: Homework 6 is the Mini-Project and worth more points (Due: Dec 13, 2016)

Rule 1. Teams of no more than three students as explained in syllabus (Handout 1). Work, i.e. source code only with no classes or other executable/binary files, to be emailed in electronic form as a single-file tar or zip archive (and in no other format).

Rule 2. Submitted archive will be processed on an AFS machine (afsconnect1.njit.edu or afsconnect2.njit.edu or osl11.njit.edu), where testing would take place. (At a minimum, the archive should contain a text file `HW6_ABC.txt` as explained below.) The code should be compilable/interpretable and executable on one or the other similar AFS machines in any language available there (emphasis in C, C++, Java, Python, Perl).

A file-based desktop search engine: HiFiDSE

You will implement the components of the **Higlander File-based Desktop Search Engine**. I will be mispronouncing it `high feed sea` rather than `high phi DSE`.

0.1 Project deliverables.

After compilation the executable unit produced should be named `hifidse`. If your code is interpreted the wrapper function should be `hifidse` possibly with an appropriate suffix as needed. We shall refer to that file as `hifidse` for the remainder. Every source file you submit must include in the form of comments in its 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_ABC.txt` 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. The `ABC` are the initials of the first names of the three members of a team. (Naturally if a team has fewer members this can be truncated.)

0.2 Objectives of the Mini-Project

A crawler has already downloaded Web-based information into a single directory. Within that directory there might be subdirectories of files or individual file(s).

You will first examine the files in the directory and its subdirectories for files that are text searchable based on filename attributes. Those files will then be tokenized, i.e. a file will be split into "words" that can be English words, numbers, dates etc, flattened (case removal), stemmed and then the resulting index-terms will be indexed. Implicit in all considerations would be the requirement that the system will be main-memory based: you don't need to address issues related to the question 'What if i run out of RAM, or CPU cycles?'. The index itself will be file-based, one file per index term.

A small query engine will also be built that will facilitate very simple queries (conjunction, disjunction of two or three arguments, negation of one argument).

0.3 Overview

Step1: **Searchable Documents**. The locally-stored web-pages available in a single directory will be processed. Subdirectories will be recursively searched. Files that are text-searchable will be identified. Some assistance is provided in the protected area of the course Web-page in the form of program `recdir.c` available as link L5 there. It's up to you whether you want to use it or not (it has only been tested on a Unix system). Text-searchable files are to be uncompressed text files (.txt,.html,.htm,.c,.cc,.java,.cpp,.h). This step will generate as a by-product file `step1.txt` that will store URL (Uniform Resource Locators) or in fact URI (Uniform Resource Identifiers) and map them to numbers (**docID**s). You are then about to start the process of parsing these and only these text-searchable files and tokenizing their contents and performing linguistic analysis in the form of stopword elimination and stemming.

Step 2: **Tokenization**. This step involves the tokenization of every text-searchable file (aka document). IMPLICIT in the tokenization is case-folding: everything becomes lower-case! You identify and extract tokens of interest (words, numbers, dates, etc) along with other useful context information (such as word position in the text, font-size and other context for an html text document).

Step 3: **Stopword Elimination.** Steps 2-4 are part of the parsing phase and the generation of the Forward Index: first identify tokens and then convert tokens into an index-term by eliminating some (stopwords) or obtaining from several slightly different tokens one index-term (stemming). For the sake of this assignment we will concentrate only on elementary `stopword` removal. The output of this phase will be the tokens identified in Step 2 minus the stopwords; this stream of keywords will then be further processed in Step 4.

Step 4: **Stemming.** The fourth step uses the output of Step 3 to further eliminate and reduce the number of potential index terms by performing stemming. The surviving tokens (after stopword elimination) as transformed by stemming become the index-terms. You now map the `word` of an index-term to numbers (`wordID`).

Step 5: **Sorting aka Inverting the Forward Index and building the (inverted) index.** The output stream is then ordered (aka sorted) based on `word` first (increasing order), then `docID` (increasing order) and finally `offset` (increasing order). Using this information, a file-based inverted index is to be built, one file per index-term; file name will be the word itself. Inside each file, 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 are maintained. For testing purposes this construction will have an interesting side-effect.

Step 6: **Query Engine.** You will be asked to design a query system that implements simple logical (`AND` and `OR, NOT`) operations.

This is the minimum implementation required to gain you the full points of this assignment. You can enrich this implementation by adding additional features such as searching more file types, do a more thorough parsing or more elaborate linguistic analysis.

1 Step1: Text-Searchable Document identification

The program `hifidse` will read the command line and behave as follows. (If you use Java or Python this invocation will be different of course.)

```
% ./hifidse searchable name
```

The first argument in the command line (after the name of the executable) denotes the action. The next argument is an arbitrary name that corresponds to a directory or a file name. For action `searchable` if name is a file-name then you need to determine whether it is text-searchable. If it is a directory you need to determine what files in that directory or its subdirectories are text-searchable. Thus names and filenames have three attributes: `DIR` for a directory, `TXTS` for a text-searchable file and `TXTN` for a non text-searhable file. A `TXTS` file is one with a suffix in the list below.

`.html , .htm , .txt , .cc , .cpp , .c , .h , .java`

The execution results in an output stored in file `step1.txt`. The number of lines is the number of names identified in the directory and its subdirectories. Each line is a triplet containing the corresponding path called `docURL` relative to `name`, its attribute `docATTR` and `docID` that has been assigned to `docURL` by you. In the example below `docID` are consecutive starting from 1. You do not need to do so: you may start from 0 or use some other assignment.

```
% ./hifidse searchable  alexg
% cat step1.txt
alexg DIR  1
alexg/courses DIR  2
alexg/courses/cs345 DIR  3
alexg/courses/cs345/index.html  TXTS 4
alexg/courses/cs345/my.txt      TXTS 5
alexg/courses/cs345/my.java     TXTS 6
alexg/courses/cs345/handouts.html  TXTS 7
alexg/courses/cs345/handouts DIR 8
alexg/courses/cs345/handouts/syllabus.pdf TXTN 9
```

2 Step2: Tokenization

If action is `token` then every `TXTS` file will be parsed and tokenized. YOU DECIDE what constitutes a token: the simplest rule is everything between consecutive whitespace. A `tokendebug` provides a more informative (file-based) output. The side-effect at the end of this execution will be file `step2.txt` or `step2d.txt` depending on whether a `token` or `tokendebug` was issued.

```
% ./hifidse token directoryname
% ./hifidse tokendebug directoryname
```

The tokenization phase is probably the most difficult part of this assignment and the most time consuming. You need to decide how to parse a document and what constitutes a token. You might make the job at hand easier if you use a parser program such as lex or yacc for this part (and the time involved to familiarize yourselves with them, if you have never used them before). An hour or two reading a manual page or the extensive documentation for the GNU equivalents names flex and bison might save you time building a tokenizer from scratch. Tokenizing a `TXTS` file is easier. Interesting tokens that will become index terms are going to be words (e.g. alphanumeric strings starting with a character) or non-trivial numbers. Collectively we will call all these interesting tokens `words` even if some of them are numbers. During this phase streams of tuples such as `(docID,word,offset,attrCDE)` or `(docURL,word,offset,attrVAL)` will be generated silently into `step2.txt` or `step2d.txt`.

I. `Text-searchable` documents will be represented by the corresponding `docID` or `docURL` for a `token` or `tokendebug` invocation.

II. Folded tokens will be represented by a `word`.

III. An `offset` is a word offset (first word, second word, etc). It can start from 0 or 1.

IV. Names `attrCDE` (for attribute code) and `attrVAL` (for attribute value) provide information about the `word` and its context. For non HTML (HTM) files there is only one `attrCDE` and `attrVAL`, 0 and PLN (for plain) respectively. For a word inside the title of an HTML/HTM document i.e. inside `<TITLE>, </TITLE>`, 1 and TTL are to be used. For a word inside an HTML anchor i.e. words surrounded by a pair of `<A>, </A>` anchor tags, use 2 and ANC. If surrounded by a `<H1> to <H3>` use 3 and H13, if by `<H4> to <H6>` use 4 and H46. Other attribute codes and values are possible.

V. When `token` is issued for a text-searchable file containing the token `Algorithms` then the following quadruplet would be generated. Note that case folding will immediately turn the A of the token Algorithms into the (case-folded) word algorithms. The 1 means word-offset is 1 as Algorithms is the one and only word of the file. Moreover for a .txt file an attribute code of 0 and value of `PLN` only make sense.

`(5,algorithms,1,0)`

or

`(alexg/courses/cs345/my.txt,algorithms,1,PLN)`

Note. At this phase we still deal with `word` but no `wordID`. However we deal with `docURL` and `docID` available after Step1.

Note. In files `step2.txt` or `step2d.txt` you may not include parentheses and commas that we used in the example above; you may replace them with space (see sample `step1.txt` contents). Thus the latter quadruplet (aka output of step2d.txt) can appear as

`alexg/courses/cs345/my.txt algorithms 1 PLN`

3 Step3: Stopword Elimination

The output of Step2 is used to eliminate stopwords from that streams of tuples. Thus `stopword` behaves similarly to `token` (rather than `tokendebug`). It generates an output stream with fewer tuples; yet offset and other information does NOT change. A file `step3.txt` will be generated which is `step2.txt` minus the offending tuples.

```
% ./hifidse stopword name
```

The list of stopwords is as follows.

```
  i a about an are as at be by com en for from how in is it  of on or that the
this to was what when where who will with www
```

4 Step4: Stemming

The stream input for this step is the output of Step 3 (if implemented) or Step 2 otherwise.

```
% ./hifidse stem name
% ./hifidse stemdebug name
```

For action `stem` or `stemdebug` you need to apply Harman's stemming algorithm that eliminates very simple suffixes (eg. plural). File `step4.txt` will be generated or `step4d.txt` as needed. The output for `stem` is a stream similar to that of `token` or `stopword` BUT WITH ONE MAJOR EXCEPTION: instead of `word` the corresponding `wordID` appears in the tuple. Note that I have written `wordID` rather than say "stemmed word". For option `stemdebug` the corresponding `indexterm` appears in the tuple obtained from the corresponding `word`.

The original token `Algorithms` for the past 2 steps has still been around as the word `algorithms` in the output of `token, tokendebug` or `stopword`. With `stemdebug` it will appear as `indexterm` named `algorithm` in `ste4d.txt` whereas for `stem` the word `algorithms` will be replaced by a number the `wordID` assigned to the newly-derived indexterm `algorithm`. We delayed assigning `wordID`s for a good reason: the original text could have included tokens `Algorithm, algorithms, Algorithms` generating two words after case-folding: `algorithm` and `algorithms`. Delaying `wordID` assignment for Step4 means that there is one index-term is `algorithm` that DOES NOT EVEN SHOW-UP in the original text.

```
 HarmanStemmingAlgorithm(word) [ Steps are in order and not swappable...]
1. If word ends in -ies but not -eies or -aies
    then -ies --> -y;
2. If word ends in -es  but not -aes, -ees or -oes
    then -es  --> -e;
3. If word ends in -s but not -us or -ss
    then -s   --> -;
```

Note. If you decide not to do stemming, you still need to create `step4.txt` or `step4d.txt` i.e. convert `word` into `wordID`s but `indexterm` would be the same as `word`.

4.1 Side Effects

The `stem` or `stemdebug` invocation will also create a file `step4LEX.txt` This is the list of index-terms with their associated `wordID`s computed in this step. The list is ordered by `wordID` and is printable, one tuple per line. `wordID`'s do not need to be consecutive nor start from zero or one for that matter.

5 Step5: Inverted Index

```
% ./hifidse invert  name
```

At this point the stream is the one appearing in `step4.txt`. We have tuples `(docID,wordID,offset,attrCDE)`. This is the forward index. We need to invert it to generate the inverted index, i.e. the index. Inversion involves sorting. We need to bring close all tuples that share the same indexterm i.e. have the same `wordID`. We sort thus the tuples first by `wordID` then by `docID` and then by `offset` and followed by `attrCDE`.

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 `indexterm` and its `wordID` generated by stemming in a file named `step5VOC.txt`. It is similar to `step4LEX.txt` except that the Vocabulary is sorted by `indexterm` not `wordID` and contains more information per `indexterm` i.e. line in the file. Every line of the `Vocabulary` file is related to an entry `indexterm` that records information such as the corresponding `wordID` but also `Ndocs` and `Nhits`. A pointer `LocP` to the actual entries (occurrence or inverted lists) need not be recorded. For this implementation the location `LocP` is implied: it is a filename obtained from `indexterm` and/or `wordID`.

Entry `Ndocs` records the number of distinct documents that contain the `wordID` after stemming (e.g. all occurrences of algorithms and algorithm). Entry `Nhits` counts the number of occurrences of `wordID` in all documents (a given word might occur more than once in a document).

3. Inverted lists: technical details. The inverted list part of the index are files under (inside) a directory named `step5.dir`. Although `LocP` traditionally points to `Ndocs` linked lists of total length `Nhits`, for our case `LocP` for a specific `wordID` is implied: a file named `indextermwordID.txt` the associated filename for `indexterm` with `wordID`, `indexterm.txt`, the choice is yours. An inverted list (file contents) would include all the tuples associated with `wordID` in sorted order by `docID`, `offset` and `attrCDE`. A tuple is triplets (`docID`,`offset`, `attrCDE`). Note that obviously, one does not need to store in the inverted list `wordID` or the `indexterm`.

The format of the file is 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 of the tuple. The number of lines of that file should be `Nhits`.

5.1 Side Effects

File `step5VOC.txt` contains the Vocabulary. A `wordID` generates a one-line entry. Entries are sorted by `indexterm` and contain the `indexterm` name, `wordID`, `Ndocs` and `Nhits`. Directory `step5.dir` contains the multiple files of the index i.e. the inverted lists. Directory location is relative to the invocation of `invert`.

6 Step6: Query Engine

Having completed the index we ask you to implement a command line-based implementation of a simple query language that deals with up to three terms. The assumed input for this step is the index built in `step5.dir`, the vocabulary `step5VOC.txt` plus other files (eg step1.txt) of previous steps, as needed.

```
    ; Assuming docID(term1) == docID(term2) ...            do
% ./hifidse and2    term1 term2                          ; term1  AND term2
% ./hifidse and3    term1 term2  term3                   ; term1  AND term2 AND term3
% ./hifidse or2     term1 term2                          ; term1  OR  term2
% ./hifidse or3     term1 term2  term3                   ; term1  OR  term2 OR  term3
% ./hifidse andnot  term1 term2                          ; term1  AND -term2
% ./hifidse next5   term1 term2                 ; |offset(term1)-offset(term2) | <= 5
```

The outcome of `and2` is to read the inverted/occurrence lists of the wordIDs of `term1` and `term2` and find their common intersection docID-wise, i.e. find those documents that contain both index-terms. The output is a list of the documents containing both index-terms, one per line. Each line prints not only the `docID`s but also the `docURL` of each document. Thus the output is similar to that of Step1 minus the DIR,TXTS,TXTN noise. The `and3` allows for a 3-term conjunction.

The outcome of `or` is that of a disjunction and its 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.

Note that for this part we only need to use `docID` information and neither `offset` nor `attr` to generate an answer to a query. A more elaborate processing can occur using `offset` and `attrCDE` information that will also rank the results. But ranking could have been a step 7 that we decided to not include it!

Finally, operator `next5` checks whether the terms `term1` or `term2` appear next to each other in a document (i.e. their `offset` differ by 5 or less.

■

Date Posted: Aug/31/2016