New Jersey Institute of Technology

A. V. Gerbessiotis

February 13, 2006

(100 points) Due: March 20, 2006

CIS 750

Spring 2006

Programming Assignment 2

1 Introduction

This assignment is a continuation of the previous one. The input to this assignment is the output of
`./pa1 token some-name`
where `some-name` is a generic name for either a file or directory name.

The output of the tokenizer will be first stemmed. You only need to implement the elementary (Harman) stemming algorithm described in class, even if it gives something simple and questionable.

The output of `PA1` (with or without stemming) is ordered according to `docid`. We now order the output stream based on `wordid` first (increasing order), `docid` (increasing order), `attr` (increasing order) and `wpos` (increasing order).

Using this information, you will build an inverted index where you will store information about word occurrences in the form of a combined `vocabulary` and `occurrence-list` structures as described in class (Subject 3). For testing purposes this construction will have an interesting side-effect.

At the end you will be asked to design a query system that implements simple logical operation (`AND` and `ANDNOT` operations.

This is the minimum implementation required to gain you the `100` points of this assignment. You can enrich this implementation by adding additional features. Or you can think ahead of adding even more features and components in the context of the independent work component of the course.

2 Deliverables

The relevant files will become available in `homework/pa2` as described in Handout 2. A single executable file will be the result of your compilable source code into the form of a file named `pa2`.

3 Part 1 : Stemming (10 points)

The executable file `pa2` will read the command line and behave as follows.

```
% ./pa2 stem some-name
```

The first argument in the command line (after the name of the executable file) denotes the action as before. The second argument is a file/directory name.

For action `stem` you need to take the output of `pa1` and apply to it Harman's stemming algorithm that eliminates very simple suffixes (eg. plural). The output is a stream similar to the input.

Note that it is quite possible that say wordid `25` already corresponds to word `algorithm` and there is another wordid say `35` that corresponds to word `algorithms`. If as a result of the stemming algorithm we have a construction of `algorithms` into `algorithm` you need to decide what to do with wordids 25 and 35. The design decision is yours. One way to deal with it is to convert for example `35` into `25`; an issue is what to do with wordid `25` that is left unused. If however, `25` does not exist a renaming might take place instead.

4 Part 2 : Inverted Index (70 points)

In the second part you take the output of `Part 1` or if you decide to skip this part, of `PA1` (`token` option) and generate an inverted index. You call the inverted index creation module of yours with a call described below.

```
% ./pa2 invert  some-name
```

Note that such a call in essence applies all options described in `PA1` and also, `stem`, that is it is a cumulative action.

An inverted index consists of a `vocabulary` and an `occurrence-list`.

The `vocabulary` will contain the wordids in sorted order (note that the lexicon of `PA1` contains the wordid/word combinations not necessarily in sorted order). For every entry you need to record information such as `wordid`, `ndocs`, `nhits`, and potentially (depends on your implementation and language combination) a pointer `locp` to the actual entries. Entry `ndocs` counts the number of distinct documents that contain `wordid`. 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 (type).

The (potential) pointer `locp` points to the area in which you will maintain information about the given `wordid`. Such information will be a list of lists (whether you implement it in an array, linked list or whatever else it is your decision) of occurrences of `wordid`. Every document in which `wordid` occurs will have a separate list. Therefore the number of lists for `wordid` will be `ndocs`. The elements in each such list can be be ordered by `type` and within each type by `wpos`. Note that `if you think this is not a good idea,` you can provide a written justification (eg, in the form of a comment, or a separate document-file called `Readme.txt` that will appear in the deliverables directory), and do otherwise.

4.1 Side-effects

Besides the side-effects of `PA1`, `Part 2` will generate additional side-effects.

For a start, it will generate a file named `vocabulary` that includes a dump of the vocabulary in the form of tuplets (`wordid`, `word`, `ndocs`, `nhits`). It suffices to print the four elements of the tuplet, one set of elements per line; you can ignore (i.e. not print) the surrounding parentheses.

It will also create a directory named `invindex`. Inside that directory you will create a number of files one for each word whose `wordid` is in the vocabulary. In the file whose name is after the `word` with the given `wordid` you will store the occurrence lists in the form of (`docid`,`wpos`, `type`) tuplets. The format of the file will be as follows.

The first entry in such a file will be a decimal number. If its value is `val`, it will mean that the following `val` tuples belong to the same document `docid`. Then `val` tuplets are listed (you need only print `docid, wpose, type` without the surrounding parentheses). After these `val` tuples another positive value might be listed to indicate another group of occurrences. If a `-1` appears, it will indicate the END-OF-FILE.

Note that we do not ask you to sort the groups of hits other than the obvious by `docid` order implied by their listing. You might choose to have them sorted by `type` or `wpos` or not.

The intent of all the side-effects is to be able to build an inverted index from all the generated files without requiring the re-reading of the web-pages.

5 Part 3 : Query Implementation (20 points)

Having completed Part 2 we ask you to implement a command line-based implementation of a simple two-term query language.

```
% ./pa2 AND     term1 term2 some-name
% ./pa2 ANDNOT  term1 term2 some-name
```

Each one of the operations above (implicitly) assumes that `./pa2 invert some-name` is first executed. We say implicitly because an alternative is to disregard the `some-name` directive and read the files generated by the side-effects and thus rebuild the inverted index. A more elaborate query system will be implemented as part of `Programming Assignment 3`. For the moment, `Part 3` suffices.

The outcome of `AND` 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 to the document.

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 `wpos` nor `type` to generate an answer to the query. A more elaborate processing can occur using `wpos` and `type` position that will also rank the results. Wait for this for the next programming assignment!