RMIT

Computer Science & IT, School of Science

ISYS 1078 / 1079 — Information Retrieval

Assignment 1: Indexing

Due: Midday, 29/3/2017 (week 5)

This assignment is worth 20% of your overall mark.

Introduction

In this assignment, you will implement an inverted index and use it to store term occurrence information. You will need to design and implement appropriate data structures, write your data to disk, and be able to read it back.

This assignment is intended to give you practical programming experience for implementing core information retrieval functionality, and to deepen your understanding of the inverted index data structure.

The "Information Retrieval" blackboard contains further announcements and a discussion board for this assignment. Please be sure to check these on a regular basis – it is your responsibility to stay informed with regards to any announcements or changes. Login through `https://learninghub.rmit.edu.au`.

Assignment Teams

This assignment should be carried out in groups of *two*. It is up to you to form a team. Please choose your team carefully, as you will need to work with the same person on Assignment 2, which builds on Assignment 1.

Once you have decided on your team, you should register using the form at:
`https://goo.gl/forms/CRqcyuMtRXXTGlkq2`

(Note: If you are a postgraduate student and have strong reasons for needing to complete the assignment individually, you may apply to do so by sending an email to the lecturer, explaining your reasons. However, bear in mind that the requirements and available marks will be the same as for pairs, so you are strongly advised to work in a team.)

Teaching Servers

Three CSIT teaching servers are available for your use: `(titan|saturn|jupiter).csit.rmit.edu.au`. You can log in to these machines using ssh, for example:

```
% ssh s1234567@jupiter.csit.rmit.edu.au
```

where `s1234567` is your student number. These servers host the document collection and other data for the assignments in this course. You are encouraged to develop your code on these machines. If you choose to develop your code elsewhere, it is your responsibility to ensure that your assignment submission can be compiled and executed on `(titan|saturn|jupiter).csit.rmit.edu.au`, as this is where your code will be run for marking purposes.

Important: You are required to make regular backups of all of your work. This is good practice, no matter where you are developing your assignment solutions.

Plagiarism

RMIT University takes plagiarism very seriously. All assignments will be checked with plagiarism-detection software; any student found to have plagiarised will be subject to disciplinary action as described in the course guide. Plagiarism includes submitting code that is not your own or submitting text that is not your own. Allowing others to copy your work is also plagiarism. All plagiarism will be penalised; there are no exceptions and no excuses. For further information, please see the *Academic Integrity* information at `http://www1.rmit.edu.au/academicintegrity`.

General Requirements

This section contains information about the general requirements that your assignment must meet. *Please read all requirements carefully before you start.*

- You must implement your programs in Java, C, C++, PHP, or Python. Your programs should be well written, using good coding style and including appropriate use of comments. Your markers will look at your source code, and coding style may form part of the assessment of this assignment.
- The submitted programs must be your own code. You should not use existing external libraries that implement advanced data structures. Where libraries (or in the case of scripting languages, built-in features beyond simple low-level data types) are used for data structures such as hash tables, they **must** be clearly attributed, and it is up to you to demonstrate a clear understanding of how the library is implemented in the discussion in your assignment report.
- Your programs may be developed on any machine, but **must** compile *and* run on the course machines, `(titan|saturn|jupiter).csit.rmit.edu.au`. If your submission does not compile and run on these machines, it will **not** be marked.

- You *must* include a plain text file called "README.txt" with your submission. This file should include the name and student ID of all team members at the top. It needs to clearly explain how to compile and run your programs on `(titan|saturn|jupiter).csit.rmit.edu.au`. The clarity of the instructions and useability of your programs may form part of the assessment of this assignment.
- Paths should not be hard-coded.
- Where your programs need to create auxiliary files, these should be stored in the current working directory.
- Parts of this assignment will include a written report, this may be in *plain text* or *PDF* format.
- Please ensure that your submission follows the file naming rules specified in the tasks below. File names are case sensitive, i.e. if it is specified that the file name is `gryphon`, then that is exactly the file name you should submit; `Gryphon`, `GRYPHON`, `griffin`, and anything else but `gryphon` will be rejected.

Programming Tasks

Have a look at the file `/KDrive/SEH/SCSIT/Students/Courses/ISYS1078/2017/a1/latimes` on `(titan|saturn|jupiter).csit.rmit.edu.au`. It is part of the TREC TIPSTER test collection, and consists of various articles from the *LA Times* newspaper.
Here is an example of the markup format:

```
<DOC>
  <DOCNO> LA010189-0001 </DOCNO>
  <DOCID> ... </DOCID>
  <DATE> ... </DATE>
  <SECTION> ... </SECTION>
  <LENGTH> ...  </LENGTH>
  <HEADLINE>
     The article headline.
  </HEADLINE>
  ...
  <TEXT>
     The text content of the document.
  </TEXT>
</DOC>
```

Individual documents are wrapped in `<DOC> ... </DOC>` tags. These indicate the beginning and end of each individual document in the collection file.

The <DOCNO> ... </DOCNO> tags contain a unique identifier for the document. You will need to keep track of this to refer to documents in the collection.

The <HEADLINE> ... </HEADLINE> and <TEXT> ... </TEXT> tags contains the text content of the document. This is the actual content of the document that you will need to index.

Your task is to write two programs. The first will index the collection, gather appropriate term distribution statistics, and write the index data to disk. The second program will load the index data, accept text terms as input, and return the appropriate term distribution statistics for each term.

1 Indexing

Your indexing program must be called `index` and accept a number of optional command-line arguments. The invocation specification should be:

```
% ./index [-p] <sourcefile>
```

(or equivalent invocation in another programming language) where the *optional* flag `-p` will print the terms being indexed, and the mandatory argument `<sourcefile>` is the collection to be indexed. These are described in more detail below.

Note that your implementations must be *efficient*, making use of appropriate algorithms and data structures. This will be part of the assessment criteria.

1.1 Parsing Module (10%)

Your first task is to parse the data, tokenising and extracting content terms, and removing any punctuation and excess markup tags. Punctuation consists of any symbols that are not letters or numbers. You will need to carefully consider issues of token normalisation (for example, how to deal with acronyms and hyphenated words).

All content terms should be *case-folded* to *lower-case* as they are indexed.

Your program must be called `index`, and should accept an *optional* command-line argument `-p`. This flag indicates that your program should print all content terms, in the same order as in the original document, to the standard output stream, stdout. If the flag is not specified, your program should produce *no* output to standard out. An example of how your program might be run is as follows:

```
% ./index -p /KDrive/SEH/SCSIT/Students/Courses/ISYS1078/2017/a1/latimes
```

(or equivalent invocation). As your parser encounters each new document, you will need to assign an ordinal number as a document identifier. These can be assigned sequentially

(i.e. the first document is 0, the second is 1, and so on). This is how the documents will be referred to in the inverted list information (see below).

You also need to track the unique document identifier specified in the `<DOCNO>` tags, and how these map to the integer identifiers that you assign. Your program should therefore always write an output file *to disk*, called `map`, which contains this mapping information. Auxiliary files such as this must be written to the *current working directory*.

1.2 Stopping Module (10%)

Stopping is the process of excluding words that have grammatical functions, but contain no meaning themselves, from consideration during indexing. Examples of such *stopwords* are `the`, `of`, and `and`. A stoplist is available on `(titan|saturn|jupiter).csit.rmit.edu.au` at /KDrive/SEH/SCSIT/Students/Courses/ISYS1078/2017/a1/stoplist.

Extend your program `index` with a module to stop the input terms. Your program should accept an *optional* command-line argument, `-s <stoplist>`, where `stoplist` is a file of stopwords that are to be excluded from indexing. An example of how your program should be invoked is:

```
% ./index [-s <stoplist>] [-p] <sourcefile>
```

Note that the `-p` option must still be available; when it is specified, your program should print all content terms that are *not* stopwords to the standard output stream.

An efficient lookup structure to store the content of the `<stoplist>` file is a hash table. It is up to you to choose a suitable hash function for text strings. You will be asked to explain your implementation in the report (see below).

1.3 Indexing Module (20%)

Extend your program `index` to build an inverted index. The inverted index needs to store term occurrence information for all content terms that occur in the file that is being indexed. For each term t, you need to store:

- The document frequency, f_t, a count of the number of documents in the collection in which term t occurs.
- An inverted list for term t, which consists of postings of the form

  $$< d, f_{d,t} >$$

  where:

  - d is the document in which t occurs

– $f_{d,t}$ is the within-document frequency, a count of how often t occurs in document d

For example, if the term `insomnia` occurs in two documents in the collection, twice in document 10, and three times in document 23, then the inverted list would be:

```
insomnia:  2 <10, 2> <23, 3>
```

Note: the punctuation symbols are only used to make the list human-readable. The actual internal representation of the inverted list would just store a sequence of integers, represented appropriately, for example `2 10 2 23 3`. Your stored lists must **not** include the extra punctuation between items. See the lecture notes for further details on the inverted index and inverted lists.

You can assume that you will be working with collections that are small enough to fit in main memory.

When your index program has finished constructing the inverted lists for each unique term that occurs in the collection, the data should be written to disk. Your program should write *three* data files:

1. `lexicon`, containing each unique term that occurs in the collection and a "pointer" to the inverted list for that term
2. `invlists`, which contains the inverted list information, consisting only of numerical data
3. `map`, which contains the mapping information from document id numbers (as used in the inverted lists) to the actual document names

These files must be written to the current working directory.

Since the lexicon and inverted lists are stored separately, your `lexicon` file will need to include information about the file offset position in `invlists` where the inverted list for the corresponding term occurs. Your program should **not** simply read the `invlists` file sequentially from the start each time it is accessed.

It is up to you to design the particulars of the implementation. However, please read the next section on *Querying* carefully first, as this is likely to have implications for how you store your data. You will be asked to explain your implementation in a report (see below).

2 Querying (20%)

Write a program called `search` that loads your index data, and searches it to retrieve the inverted lists for particular terms. An example of how your program should be invoked is:

```
% ./search <lexicon> <invlists> <map> <queryterm_1> [...  <queryterm_N>]
```

(or equivalent invocation) where `<lexicon>`, `<invlists>` and `<map>` are the inverted index files that your `index` program created. The program will also receive a variable number of query terms (but at least one query term) as command-line arguments. Each of these terms should be looked up in the lexicon, and the appropriate inverted list data fetched from the inverted list file. For each `queryterm`, you need to print the following information to standard out:

- The current query term;
- The document frequency f_t;
- A list of each document in which the term occurs (using the identifier from the `<DOCNO>` field, which you can look up in your `map` file), and the within-document frequency $f_{d,t}$.

An example of how your program might be run is as follows:

```
% ./search lexicon invlists map insomnia
```

(or equivalent invocation). If the term `insomnia` occurs in two documents, twice in document FT911-22 and three times in document FT911-2032 then the output should be:

```
insomnia
2
FT911-22 2
FT911-2032 3
```

Your implementation should follow these specifications:

- The `lexicon` and the `map` should be loaded into memory when your program is invoked, and stored using an efficient look-up data structure such as a hash table.
- The `invlists` data should *not* be pre-fetched into memory. Instead, the inverted list data should be read from disk as each query term is processed. That is, your program should read *only* that section of the `invlists` file that corresponds to the list data for a particular term.
- The output of your program *must* follow the format given in the example above, and must *not* produce any additional output beyond what is specified in the assignment requirements.

3 Report (40%)

Create a file called `report.txt` or `report.pdf` and answer the questions below. Remember to clearly cite any sources (including books, research papers, course notes, etc.) that you referred to while designing aspects of your programs.

The answers in your report should *not* include pasted extracts from your source code (the source code will be submitted separately, as part of the assignment). However, you may wish to make use of pseudo-code to explain particular concepts, algorithms, or implementation choices, if needed.

3.1 Index Construction (15%)

Explain your inverted index implementation. As part of your explanation you need to clearly describe how you:

- gather term information while your `index` program is parsing the collection
- tokenise terms, including how you deal with punctuation and markup tags, and handle acronyms and hyphenated words
- merge the information together for the final `lexicon` and `invlists` files

This explanation should be around a page in length, but no longer than one and a half pages.

3.2 Stoplist (5%)

Briefly explain how you implemented your stoplist module, including the hash function that you used, and why it is appropriate. This explanation should be around half a page in length.

3.3 Index Search (15%)

Explain your index search implementation. As part of your explanation you need to clearly describe:

- the data structures you used for your `lexicon` when it is held in memory
- how you look up corresponding term occurrence information in your `invlists` file

This explanation should be around a page in length, but no longer than one and a half pages.

3.4 Index Size (5%)

How large is your inverted index? How does this compare with the size of the original collection? Why is it larger/smaller than the original collection?

3.5 Limitations

This assignment requires you to design and implement a large indexing system for text documents. In this section, you should briefly outline any known limitations or bugs in your approach or implementation. For example, perhaps you noticed that your code will crash in certain conditions, and despite your best efforts you just couldn't figure out why. Or, perhaps due to time constraints you were forced to make a less than ideal design decision.

Please be up-front in listing known issues, and demonstrate your understanding of IR system requirements by explaining what you think an "ideal" system might do in addition (or instead).

3.6 Contributions

In this section, you should provide details about what each team member contributed to the submitted assignment materials. Note that both team members will receive the same mark. However, you *must* include this section.

4 Optional Extension: Compression (Up to 10% bonus marks)

ONLY attempt this section if you have completed all previous sections of the assignment.

If you submit a solution to this extension exercise, you will need to submit two versions of your code, one without compression (i.e. all previous sections of the assignment), and one with compression (i.e. incorporating the following requirements). You MUST clearly explain in your README.txt file which files correspond to which part of the assignment, as well as how each should be compiled and run.

1. Extend your `index` program so that the inverted lists are compressed using *variable-byte* coding before they are written to disk, and decompressed after being read from disk.

2. Extend your report to include a subsection called *Compression*, and explain your implementation of the chosen compression scheme. What is the size of your inverted index using compression? How does this compare to the uncompressed inverted index? Write no more than half a page.

What to Submit, When, and How

The assignment is due at Midday, 29/3/2017 (week 5). Assignments submitted after this time will be subject to standard late submission penalties.

- All assignment files (including source code, README file, report) must be submitted as ONE single zip file, named as your student number (for example, 1234567.zip if your student ID is s1234567). Please do NOT submit the assignment document collection file, or any index or map files that your programs create as output when run.
- You only need to submit one copy of the assignment per team. Please remember to indicate all team members at the top of your README file, and remember that you also need to register your team through the form at: `https://goo.gl/forms/CRqcyuMtRXXTGlkq2`
- Assignments should be submitted via the IR course BlackBoard, which you can access through the LearningHub. Submission instructions will be posted there before the due date.