

Instructions for MIE457F Project 1: An Information Retrieval System *Due Date: Oct. 6, 2003, midnight*

Questions to Scott Sanner

`ssanner@cs.toronto.edu`

1 Overview

The purpose of this programming project is to have you modify and implement parts of an information retrieval system while working in a structured Java programming environment.

For this project, you will be expected to complete the following three tasks:

1. Modify the current tokenizer to make it more robust.
2. Implement an inverted file data structure to efficiently store indexed document content.
3. Implement an efficient query engine that accesses the index and ranks query results based on keywords and the TF and IDF heuristics (plus others if you're feeling ambitious).

The programming portion of this project should not be too difficult but it will require working knowledge of a number of Java topics including package structure, the Make utility, the JavaDoc utility, and the Java Foundation Classes (JFC) – especially *java.lang* and *java.util*.

All of these topics will be introduced in tutorial and you will have the opportunity to explore and experiment with them during the laboratory sessions. The only knowledge prerequisite for this project is the ability to code and compile simple Java applications.

2 Background reading

In order to understand the content of this document, you should first understand the lecture slides for weeks 3 and 4 on the topics of indexing and search for information retrieval [1].

Additional helpful material can also be found in Grossman et al. [2] and Van Rijsbergen [3].

3 Getting started

All of the information to compile, run the code, and view the documentation for this project is described in the *README* file in the root directory of the project.

To get started with the project, download the archive from the following webpage (right click and *Save As* from a web browser):

```
http://www.cs.toronto.edu/~ssanner/Projects/prog.html
```

To unpack the archive into a subdirectory, use a Windows compression/decompression utility such as *WinZip*. Or, from the UNIX command line, you can type:

```
tar -xvzf P1.tar.gz
```

From this point, open a command prompt (UNIX: any shell, WINDOWS: Start->MIE Software->gams) and change into the *Project1* subdirectory. On a Windows system, use the command `type README` or on a UNIX system, use the command `cat README` to view the *README* file.

The *README* file should have all of the information required to compile, run the command-line interface, and access the JavaDoc documentation in the *javadoc* subdirectory. The L^AT_EX version of this file along with its corresponding compiled *ps* and *pdf* versions is in the *docs* subdirectory.

Note: A useful text editor for Windows is *TextPad* (Start->TextPad). For UNIX, useful text editors are *emacs*, *xemacs*, *vi*, and *pico*.

4 Existing code structure

Before starting to modify the current information retrieval system, it is first important to understand its structure.

4.1 Package structure

The package structure for this project is given in Figure 1. Details of the packages and the classes within them can best be explored through the JavaDoc documentation. However, a quick overview of each package is given below.

[root] This is the root of the *ir* and *comshell* packages and is the default package if no package is named. As in most projects, the root package simply exists to provide a top level of organization for all of the subpackages. It contains no classes and *will not be modified in this project*.

comshell This is an auxiliary package that provides support for a command line interface. Other packages such as *ir* will extend the *ComShell* class in the *comshell* package in order to implement their own command shell. *This package will not need to be modified in this project*.

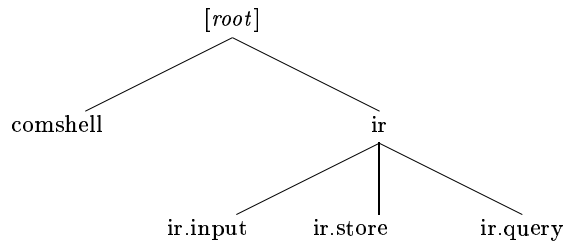


Fig. 1. Package structure for the MIE457F Project 1: Information Retrieval.

ir This is the root of the *ir* (i.e. Information Retrieval) package. This package contains one class *IRShell* that extends a *ComShell* class in order to provide a customized command line interface for this project. The commands implemented for this class make calls to the tokenizer, the indexing system, and the query retrieval system. *You will need to modify these commands to accomodate changes that you make to the code.*

ir.input This is the package that takes care of document retrieval (either from a file or the web) and tokenization. It provides a basic *Token* class as well as a general *TokenStream* interface. *You will only need to modify RawTextTokenStream to provide more robust tokenization – see below.*

ir.store This is the package that takes care of document indexing and storage of the inverted file index. There is an interface *Index* that defines the basic operations that all index implementations should support. It also includes a very simple and inefficient implementation to provide a basic example of how to support this interface. *You will need to provide an alternate index implementation that makes use of an inverted file structure. You may also add new methods to the Index interface and your index implementation to support query retrieval.*

ir.query This is the package that takes care of query retrieval. It is currently empty and is simply provided in case you want to add a specific class for performing query operations. Otherwise, you can simply implement the query interface as an additional method on the *Index* interface and its respective implementation in the *ir.store* package. *Additions to this package are optional but may help you to write code that is structurally clear.*

4.2 Compilation and Makefiles

How you compile the project will depend on what system you are using. UNIX offers a built-in utility for compiling projects called the *Make* utility. This is described below in detail. If you are using Windows, you can still compile fairly easily, but things are not *as* automated.

- *If you are using UNIX Makefiles (optional)*: In every package directory there is a corresponding Makefile (of the same name) that describes how to compile or process all of the files in that directory. For the most part, you should only need to modify this file to add additional classes or subpackages to be compiled. These are defined by the *PACKAGES* and *CLASSES* variables in the Makefile and *you simply need to add the appropriate entry whenever you add a subpackage or class to a package.*

For compilation on either platform, refer to the *README* file in the root directory (*Project1*) for more information.

4.3 Using JavaDoc documentation

There is a reasonable amount of code already in this project and the JavaDoc documentation will be your most effective tool for getting a high-level overview of the package and class structure. JavaDoc documentation uses a web browser interface and allows you to click on any package, class, or method to explore it further.

For information on building JavaDoc documentation, refer to the *README* file in the root directory (*Project1*). This explains how to build and view the code documentation for this project.

4.4 Running the command-line interface

The command-line interface as implemented in the class *IRShell* in the *ir* package provides a powerful way to interact with the information retrieval system. The user simply issues commands for indexing documents or URLs and viewing/querying the current index. Furthermore, the user can save commands to a script file and execute this automatically from the command-line or even the UNIX/DOS prompt.

An example script follows:

```
// A sample IRShell script
echo on
help
newindex MyIndex
adddoc "f1.txt"
adddoc "http://www.cs.toronto.edu/~ssanner"
listpostings "Scott"
listpostings "the"
timer start
query "Scott"
timer stop
query "the"
timer stop
echo off
return
```

The output of this script on sample data is given below:

```
> [help]

listenv                - list env vars and bindings
setenv <var> <val>    - set env variables
echo {on|off}         - turns com line echoing on/off
exec <filename>      - execute a script
return                - returns from script
help                  - this menu
timer {start|stop}   - start or stop timer
quit                  - quit application
//                    - comment

newindex <name>      - make a new index
adddoc <doc>         - add a doc (file or web page) to index
listpostings <string> - list postings in index
query <string>       - query index and rank docs

> [newindex MyIndex]

New index named 'MyIndex' created...

> [adddoc "f1.txt"]

Adding file: 'f1.txt'

<'This', Source:'f1.txt', Pos:0, Line:1, LinePos:0>
<'is', Source:'f1.txt', Pos:5, Line:1, LinePos:5>
<'the', Source:'f1.txt', Pos:8, Line:1, LinePos:8>
<'first', Source:'f1.txt', Pos:12, Line:1, LinePos:12>
    ---REMOVED TO SAVE SPACE---

> [adddoc "http://www.cs.toronto.edu/~ssanner"]

Adding file: 'http://www.cs.toronto.edu/~ssanner'

<'Scott', Source:'http://www.cs.toronto.edu/~ssanner', Pos:91, Line:4, LinePos:11>
<'Sanner's', Source:'http://www.cs.toronto.edu/~ssanner', Pos:97, Line:4, LinePos:17>
<'Academic', Source:'http://www.cs.toronto.edu/~ssanner', Pos:106, Line:4, LinePos:26>
<'Home', Source:'http://www.cs.toronto.edu/~ssanner', Pos:115, Line:4, LinePos:35>
    ---REMOVED TO SAVE SPACE---

> [listpostings "Scott"]

Postings for 'Scott':
```

```
<'Scott', Source:'http://www.cs.toronto.edu/~ssanner', Pos:259, Line:14, LinePos:6>
<'Scott', Source:'http://www.cs.toronto.edu/~ssanner', Pos:91, Line:4, LinePos:11>

> [listpostings "the"]

Postings for 'the':

<'the', Source:'http://www.cs.toronto.edu/~ssanner', Pos:7494, Line:235, LinePos:52>
<'the', Source:'http://www.cs.toronto.edu/~ssanner', Pos:4742, Line:157, LinePos:60>
<'the', Source:'f1.txt', Pos:58, Line:3, LinePos:8>
<'the', Source:'http://www.cs.toronto.edu/~ssanner', Pos:6349, Line:201, LinePos:46>
    ---REMOVED TO SAVE SPACE---

> [timer start]

> [query "Scott"]

Query result for 'Scott':

< 2: 'http://www.cs.toronto.edu/~ssanner' >

> [timer stop]

Elapsed time: 6 ms

> [query "the"]

Query result for 'the':

< 15: 'http://www.cs.toronto.edu/~ssanner' >
< 3: 'f1.txt' >

> [timer stop]

Elapsed time: 7 ms

> [echo off]

Top level stream returned, exiting...
```

For full directions on running the command shell, see the *README* file in the root directory of the project.

5 Your tasks

Following is an outline of the specific tasks you need to perform for this project.

5.1 Tokenizer modification

The tokenizer is not currently very robust. For example, if the line “This is text.” is encountered, you probably would not want “text.” to be a token. Rather, you would like to remove the “.” and just return the token “text”. Such a capability is currently not implemented but will need to be implemented for this project.

Specifically, for the purposes of this project, we expect a token to consist of any *consecutive* group of alphanumeric (A-Za-z0-9) characters. Any symbols (e.g., '!@:) or whitespace should be discarded and considered to be a delimiter that separates tokens.

For example, here is a sample input line:

Here's a *random* postal-code:M6G 2L9!

Here are the set of tokens that should be extracted:

1. here
2. s
3. a
4. random
5. postal
6. code
7. m6g
8. 2l9

Note that all tokens are lowercase – although it makes the search engine case insensitive, it typically yields better search results.

It is your job to modify `ir.input.RawTextTokenStream` *to follow the above outlined scheme for tokenization and to convert all tokens to lowercase.*

5.2 Inverted file index implementation

You will need to build a *new* class that implements the *ir.store.Index* interface. This implementation should be based on the inverted file data structure given in the lecture slides [1]. Additionally the data structure should make it possible to calculate the TF metric for any keyword and document combination and the IDF metric for any keyword... these metrics are covered in the lecture slides but are also briefly reviewed in the next section. The TF and IDF values *do not* need to be explicitly stored (see example below) but they must be able to be calculated efficiently (i.e., constant time).

Here are two sample documents that can be stored in an inverted file:¹

Doc1.txt:

¹ Note that the second line in each document is just to provide a reference for the position of characters on the first line - it is not part of the document!

```
1: This is line 1.
012345678901234567      <- Position count (not a line)
```

Doc2.txt:

```
Here is another line.
012345678901234567890  <- Position count (not a line)
```

Following are two text diagrams roughly indicating how the inverted file structure should be organized:

Document ID and maximum term frequency (per document) storage:

```
‘Doc1.txt’ -> [DOC-INFO ID:0 MAX-FREQ:2]
‘Doc2.txt’ -> [DOC-INFO ID:1 MAX-FREQ:1]
```

Inverted file data structure:

```
‘1’          -> { [ ID:0 -> POS-LIST:{0,16}] }
‘another’    -> { [ ID:1 -> POS-LIST:{8}] }
‘here’       -> { [ ID:1 -> POS-LIST:{0}] }
‘is’         -> { [ ID:0 -> POS-LIST:{8}], [ ID:1 -> POS-LIST:{5}] }
‘line’       -> { [ ID:0 -> POS-LIST:{11}], [ ID:1 -> POS-LIST:{16}] }
‘this’       -> { [ ID:0 -> POS-LIST:{3}] }
```

There are a few things to note about the above data structure:

- -> indicates a map.
- {} indicates a collection (either a list or set based on requirements).
- [] indicates a structure with fields - the first word is the struct name, the words preceding colons are field names.
- Although TF is not explicitly stored, it can be easily calculated from the number of term entries in a document (found by finding the size of the POS-LIST field for a term and document) divided by the max term frequency for that document (found by looking at the MAX-FREQ for a document).
- Although IDF is not explicitly stored, it can be easily calculated from the number of documents in which a term appears (found by finding the number of document postings for a keyword) and the total number of documents (assuming document IDs are allocated consecutively, just look at the max document ID).

It is your job to design and implement this inverted file data structure. You may modify this structure to suit your needs but it is imperative that it be efficient for query retrieval on large indices. I.e., the linear search method used in the current indexing class `ir.store.SimpleListIndex` should not be used – it is inefficient and only provided as an example implementation of the `ir.store.Index` interface.

When you implement this interface, make sure to implement `getPostings(String s)` so that we can view the contents of your inverted file structure. You can print

the contents in any reasonable manner so long as it indicates for any keyword, the set of documents and positions within those documents that the keyword occupies. See the implementation in `ir.store.SimpleListIndex` for an example.

You will benefit substantially by using the built-in Java Collection and Map classes for these data structures (especially `ArrayLists`, `TreeSets`, and `HashMaps`). The Java Collection and Map classes will be covered in tutorial.

5.3 Query engine implementation

Once you have completed the Tokenizer modifications and have built an inverted file data structure for indexing documents, you will need to implement a query routine that takes as input a set of space-separated keywords (keyword order, proximity, and boolean combinations do not need to be considered for this project) and ranks all documents having at least one matching keyword according to the following scheme:

The term frequency of keyword i in document j is given by tf_{ij} :²

$$tf_{ij} = \frac{f_{ij}}{\max_i f_{ij}} \quad (1)$$

The inverse document frequency of a keyword i is given by idf_i :^{3,4}

$$idf_i = \log \left(\frac{n}{d_i} \right) + 1.0 \quad (2)$$

For determining the rank of a document, the weight contribution of a keyword i to a document j containing that keyword is given by w_{ij} :

$$w_{ij} = \frac{(\log(tf_{ij}) + 1.0) \cdot idf_i}{\sum_{k \neq j} ((\log(tf_{ik}) + 1.0) \cdot idf_i)^2} \quad (3)$$

Finally, the rank of document j is given by r_j :

$$r_j = \sum_i w_{ij} \quad (4)$$

Obviously, the higher the value of r_j the higher the rank of document j . Documents with higher rank should be listed before documents of lower rank when the query algorithm displays its results.

To perform this ranking, you must implement the `getQueryResults(Set query_keywords)` method in the `ir.store.Index` interface. You may find it useful to build a separate class for handling queries and you may do this in the `ir.query` package. (This way, the query algorithm and data structure do not need to be implemented in

² In equation 1, f_{ij} is the frequency of keyword i in document j .

³ In equation 2, n is the total number of documents and d_i is the document frequency of keyword i , i.e. the number of documents in which keyword i appears.

⁴ For this project, assume all logarithms are base 10.

the same class. This is useful if you want to change the query algorithm without changing the data structure!)

The design of the query retrieval algorithm is your choice, however it must do a few simple things:

1. Retrieve *any* documents containing at least one of the given set of keywords.
2. Assign a weight r_j to each of these documents as calculated by the above formulae.
3. Sort the document set by this weight.
4. Display the matching documents for the user in rank order.

It is your job to design and implement `getQueryResults(Set query_keywords)` *for the inverted file data structure class that you created for the previous tasks. As with other elements of the inverted file index, you will benefit substantially from using the Java Collection and Map classes. Additionally, you will benefit from understanding how the current query retrieval algorithm in* `ir.store.SimpleListIndex` *works since it provides a simple but efficient method for ranking and sorting documents.*

You must implement the query algorithm described above. However, if you are feeling ambitious and want to provide additional query algorithms for extra credit, see the Extra credit section for some ideas.

6 Submission and marking

6.1 Turning in your project

The due date for this project is **Oct. 6, 2003 at midnight**. To turn in the project, use either a *tar* or *zip* utility such as *WinZip* to compress the ***Project1*** directory along with its subdirectories and email this *single compressed file* to Scott Sanner at *ssanner@cs.toronto.edu*. You will be sent notification within 24 hours that the project has been received and that it can be successfully decompressed.

Use of the ECF compression utilities will be covered in lab.

In the event that there are any complications with the submission process (e.g., a corrupted file), we will request to view the directories you are submitting to verify that no file has been modified past the due date/time. Thus, **you must ensure that you have a copy of the code on your local *H* drive on the ECF machines (so we know that the timestamp is valid) and you must *not* modify it beyond the due date (otherwise the timestamp will indicate this and we will not be able to accept your project).**

6.2 Evaluation

Your program will be evaluated using some command-line scripts that we develop. These will simply be more complex versions of the example script *test.irsh* given in the root directory of the project.

Evaluation will be based on the following criteria:

1. *Completeness* – Have you completed all of the requirements?
2. *Clarity* – Is your code well-structured and understandable?
3. *Commenting* – Do you have general as well as JavaDoc-specific comments?
4. *Correctness* – Are there any errors which lead to incorrect program behavior?
5. *Efficiency* – How efficient is your implementation for queries on very large indices?

6.3 Extra credit

For extra credit, you may consider the following project enhancements:

1. Add word stemming/linguistic morphology capabilities to the tokenizer.
2. Use a regular expression lexical analysis package such as the PERL5 utilities or JLEX for better tokenization.
3. Evaluate the tradeoffs between a hashed inverted file format and a sorted inverted file format (or other formats).
4. Implement alternate retrieval ranking schemes (using different metrics or taking into account additional information such as keyword proximity).
5. Implement boolean combinations (AND,OR,NOT) for the query system.

References

1. Fox, M.S.: Lecture Slides, Weeks 3 and 4, Information Retrieval: Indexing and Search. On-line: <http://www.mie.utoronto.ca/courses/mie457f/> (2003)
2. D. A. Grossman, O. Frieder: Information Retrieval: Algorithms and Heuristics. Kluwer Academic Publishers, Boston (1998)
3. Rijsbergen, C.J.V.: Automatic Text Analysis. Butterworth, London (1979)