Text Retrieval with Apache Lucene
First Programming Task of Multimedia Retrieval
SoSe2014

The task:

Write a program using Apache Lucene 1 (at least version 3.6, currently the newest version is 4.7), which fulfills the following expectations:

For a given folder the parser should be able to create an index of all files contained in this folder and all its subfolders.
The file analysis is based on english language using the Porter Stemmer.
The parser should be able to index file types like Plain Text and HTML as well.
Depending on the file type the HTMLParser should be used (you can find it in the demo package of Lucene).
The search should be implemented by using standard functionality of IndexSearcher.
When showing the results, also the path and the title (if it is a HTML file) should be given to the user.

Our solution:

According to its context within the Multimedia Retrieval subject, the presented task corresponds to experimenting with aspects of a modern search engine. As we discussed in class, a search engine could be modeled to consist of a crawler, an indexer and a search interface. In this task we focused specifically on adapting and using the indexer, as provided by the Lucene open source library for information retrieval.

To utilize the index we provided support for 2 phases: the Build index phase, when the index is created. And the Search phase, when (only after creating the index) searches can be performed.

Our solution was implemented in a java class called THIndexer (Text and HTML indexer). Apart from GUI and IO related variables, this class is essentially composed of the following elements:

Class elements:

1) private static Analyzer analyzer: Lucene variable allowing text-analysis via tokenizing. It also removes stop words, and uses the porter stemmer algorithm for improving the indexing.
The first part of analyzing the retrieved files is tokenizing. This corresponds to the extraction useful of text units from the files, called tokens. The tokenizing occurs before the creation of the index. By discarding stop works, the index is reduced.

The concept underlying the porter stemmer is extracting stems. By using them the index might provide better performance than using the words themselves. A query that uses a stemmed word will actually match a larger number of words than a query using only the original token. However not all results might be entirely accurate.
For our implementation we used specifically the EnglishAnalyzer.

2) private static FSDirectory index: Lucene variable for managing the directory where the index will be stored. By referring to this variable, we can access the index. By default the created index will be stored on the index subfolder of the current directory.

3) private static IndexWriter writer: Lucene variable allowing to add files to the index.

4) private static IndexWriterConfig config: Lucene variable for configuring the index. One of the properties that can be configured is the analyzer itself, so that when a document is added the analysis is performed.

Class funtions:
This class is composed of the following functions:

Basic Functions:
1) public THIndexer(): Constructor. This only initializes the GUI.
2) private void initComponents(): This function is called by the constructor to initialize the GUI.

Functions used in the Build index phase:
3) private void browseBtnActionPerformed (ActionEvent): Allows to use the browse button to select the folder to be indexed.
4) private void buildIndexBtnActionPerformed (ActionEvent): This function calls the addFileToIndex function over the folder passed as parameter.
5) private boolean addFileToIndex (File, IndexWriter): This function scans iteratively each element in the folder passed as parameter. For each element: if it is a folder, it calls itself recursively over it. If it is a txt or html file, it adds it to the index. If the file is a txt file, then its body is added to the index and the english analyzer from the index does the parsing. If it is a html file, then it is parsed by the jsoup html parser before passing it to the english analyzer. The function returns true if the folder was successfully indexed, and false otherwise.
Note that this function assumes the functionality that would be implemented by the crawler (namely, the exploration of subfolders), and also by the indexer.

Functions used in the Search phase:

6) private void searchBtnActionPerformed (ActionEvent): This function implements the searching over the index. It assumes that the index was successfully created (which is enforced by the GUI). It first parses the query string. Then, using Lucene variables, searches the index and prints the results.
Note that this function, next to the GUI, assumes the functionality corresponding to the search interface. Naturally much of the proposed interface could be improved, but it would exceed the requirements for this task.
As an additional resource we provided within the results, an explanation of why each file is returned, and specifically why it is sorted in that position, according to the Lucene scorer. This helps to understand the inverse frequency indexing carried out by the Lucene index.

7) void main (String []): main.

User Guide

Next to this documentation you will find 2 folders, one named THIndexer and one named Test. The former includes the code itself, and the later a test folder for testing the indexer.

The THIndexer folder includes 2 subfolders: lib and src. The lib folder includes all the .jars used by our implementation. The src folder includes the code itself, in the file: THIndexer.java.

In the test folder we include several versions of books by Lewis Carrol, downloaded from guternberg.org.

The easiest way to use this program is to load the THIndexer folder as an existing project into Eclipse. Then, surfing into the THIndexer, src, default package subfolder, and running the THIndexer.java from there.

The GUI is fairly explicit, and hopefully easy to use.
First the user can touch the Browse button, and select the folder she wants to index. The user must press the Build Index button to activate the search functionality. The search functionality will only be possible once the index is created. Be careful to pass an adequate folder for the indexing.
When the search button is activated, then the user can input text and perform searches.

We suggest performing the following searches over the test folder:

- Alice
- Snark
- Jabberwocky
- The
- a

(For the last 2 queries, no result should be produced, which proves that stop words were removed)

- actualizations
- actually
- actuals

(The last 3 queries should produce the same result, proving that porter stemming is being used. Specially given that the words actuals or actualizations appear in no document.)