DD2476 Search Engines and Information Retrieval Systems

Assignment 1: Boolean Retrieval

Johan Boye, Carl Eriksson, Jussi Karlgren, Hedvig Kjellström

The purpose of Assignment 1 is to learn how to build an inverted index. You will learn 1) how build a basic inverted index; 2) how to handle multiword queries; 3) how to handle phrase queries; 4) how to evaluate a search system; and 5) techniques for handling large indexes.

The recommended reading for Assignment 1 is that of Lectures 1-3.

Assignment 1 is graded, with the requirements for different grades listed below. In the beginning of the oral review session, the teacher will ask you what grade you aim for, and ask questions related to that grade. All the tasks have to be presented at the same review session – you can not complete the assignment with additional tasks after it has been examined and given a grade. ***Come prepared to the review session!*** *The review will take 15 minutes or less, so have all papers in order.*

E: *Completed Task 1.1-1.4 with some mistakes that could be corrected at the review session, completed Task 1.5.*
D: *E + Completed Task 1.1-1.4 without mistakes.*
C: *E + Completed Task 1.6.*
B: *C + Completed Task 1.7 with some minor mistakes, showed understanding of possible reasons for these errors.*
A: *B + Completed Task 1.7 without mistakes.*

These grades are valid for review February 7, 2016. See the web pages www.kth.se/social/course/DD2476, VT 2016 ir17 - Computer assignments in the menu, for grading of delayed assignments.

Assignment 1 is intended to take around 50h to complete.

Dataset

In the three assignments in this course, you will implement a rudimentary search engine. The engine will be evaluated on a corpus of linked documents, the wiki for the US town Davis. The wiki can be found on `https://daviswiki.org`, and we also provide a ”washed” version without images or formatting, which is found in the course directory, see below.

NOTE: The wiki is constantly changing. There are therefore discrepancies between the washed copy (which is from 2014) and the online wiki.

Computing Framework

The code needed for the assignment can be found in the course directory `/info/DD2476/ir17/lab` on the CSC UNIX system. Copy the the following files to your own home directory:

- the sub-directory **`ir`**,
- the three shell scripts **`run_tokenizer.sh`**, **`run_search_engine.sh`** and **`compile_all.sh`**.
- the image file **`ir17.gif`**
- the files **`testfile.txt`**, **`testfile_tokenized_ok.txt`, `patterns.txt`**

The **`ir`** directory contains the source code skeleton which you will use as the basis for the assignments 1-3. The code of each assignment builds on that of the previous assignment.

If you are using your own computer, you will need to copy the sub-directories **`pdfbox`** and **`davisWiki`** as well.

When you have copied the **`ir`** directory to your home directory, compile the lab skeleton as follows:

```
sh compile_all.sh
```

If you are working on a Windows platform, you can instead use the **`bat`** files:

```
compile_all.bat
```

You might see some warnings; these may be ignored.

Task 1.1: Tokenization

The **`Tokenizer`** java class (found in the **`ir`** directory) performs tokenization of text. In its default setting, the program considers whitespace and punctuation as delimiters. This means, for example, that **`24/7`** will be considered as the tokens **`24`** and **`7`** (assuming that we remove punctuation). However, one can configure the tokenizer to deal with non-standard tokens (such as dates, phone numbers, mail addresses, etc.) by adding regular expressions in the file **`patterns.txt`**: If the next string of non-whitespace characters to be read from input matches one of the regular expressions in this file, the string is considered to be a single token. For instance, adding the pattern **`\d+/\d+`** to **`patterns.txt`** causes the program to emit **`24/7`** as a single token. **Your task is now to add regular expressions to `patterns.txt` so that the program handles non-standard tokens correctly.** Try to make your regular expressions general, so that each expression covers a class of non-standard tokens (e.g. dates). Try out your expressions on **`testfile.txt`** using the script **`run_tokenizer.sh`**. Your task is done when the output of the program is identical to the contents of the file **`testfile_tokenized_ok.txt`** (use the Unix **`diff`** utility to verify this). Less than 20 regular expressions should be required.

At the review

To pass Task 1.1, you should be able to run the tokenizer using your regular expressions, and let the teacher verify that the output is correct. If asked, you should also be able to explain the purpose of a particular regular expression.

Task 1.2: Basic Inverted Index

When we have sorted out the tokenization issues, we can start working on index construction. To start the search engine, write

```
sh run_search_engine.sh
```

or, on a Windows platform:

```
run_search_engine.bat
```

Note that if you are using your own computer, you might need to edit the script and change the path to the **`davisWiki`** data set.

You should now see the GUI where search queries are entered and search results presented. The **`-d`** flag on the line above tells the program which directory to index. You might specify any number of directories by

```
-d directory1 -d directory2 -d directory3
```

etc. The program indexes all text files and readable PDF files in the specified directories.

Enter some words in the search text box and press Enter. You will now see in the result text area:

```
Found 0 matching document(s)
```

In order for the system to actually find some matching documents, you will now need to add code to some of the classes in the **`ir`** directory. The **`processFiles`** method in the **`Indexer`** class reads files and produces a stream of tokens to be indexed. However, currently the program is incomplete, and no index is built. **Your task is to complete the classes `HashedIndex`, `PostingsList` and `PostingsEntry` (and possibly add classes of your own), so that the program builds an inverted index.**

When you have finished adding to the program, compile and run it, indexing the **`davisWiki`** data set. Try the search queries

zombie

which should result in **36** retrieved documents, and

attack

which should result in **228** retrieved documents.

At the review

There will not be any examination of Task 1.2, it is merely a preparation for Task 1.3.

Task 1.3: Multiword Queries

The search engine implemented in Task 1.2 can only handle single word queries; it will now be extended to queries consisting of several words.

Extend the `search` method in the `HashedIndex` class **to implement the intersection algorithm** (page 11 in the textbook), so that you can do multiword searches.

(You can choose between intersection queries and phrase queries in the "Search options" menu. The "Intersection query" option should be selected by default, so you do not need to do anything. The option ”Ranked retrieval” and the ”Ranking score” menu relates to Assignment 2, and the button bar at the bottom of the GUI relates to Assignment 3.)

When you have finished adding to the program, compile and run it, indexing the **`davisWiki`** data set. Try the search queries

zombie attack

which should result in **15** retrieved documents,

money transfer

which should result in **105** retrieved documents.

Furthermore, design your own query (or queries) with 3 words or more, and try it on the dataset.

At the review

To pass Task 1.3, you should be able to start the search engine and perform a search in intersection query mode with a query specified by the teacher, and show that your seach engine returns the correct list of documents. You should also be able to explain all parts of the code that you edited, draw the data structure on paper, and explain from that figure how an intersection query is executed.

Task 1.4: Phrase Queries

Modify your program so that it is possible to search for contiguous phrases like "money transfer" using the techniques described in Section 2.4.2 in the textbook. Only documents that contain that exact phrase should be returned; documents that include the words at separate places should **not** be returned. Note that the algorithm on page 39 is **not** exactly what you are looking for.

You will need to add code to the **`search`** method in the **`HashedIndex`** class, so that when this method is called with the **`queryType`** parameter set to

`Index.PHRASE_QUERY`, the search query should be treated as a phrase query. When the method is called with the parameter set to **`Index.INTERSECTION_QUERY`**, the program should behave as in Task 2 above.

Compile the program and run it, indexing the **`davisWiki`** data set. You can choose between intersection queries and phrase queries in the "Search options" menu. Choose the "Phrase query" option and try the same search queries as before,

zombie attack

which should result in **14** retrieved documents,

money transfer

which should result in **2** retrieved document.

Run your own query/queries as phrase query.

At the review

To pass Task 1.4, you should be able to start the search engine with and perform a search in phrase query mode with a query specified by the teacher, and show that your search engine returns the correct list of documents. You should also be able to explain all parts of the code that you edited, draw the data structure on paper, and explain from that figure how a phrase query is executed.

You should also prepare a comprehensive answer to the question *Why are fewer documents generally returned in phrase query mode than in intersection query mode?*

Task 1.5: What is a good search result?

The objective of this task is to **reflect on what constitutes a good answer to a query**. Run the program from Task 1.4, indexing the data set `davisWiki`. Choose the "Intersection query" option in the "Search options" menu.

Search the indexed data sets with the following query:

graduate program mathematics

Go through the list of 22 returned documents and look at them on the web. A document called `Doc_Name.f` have the web address `https://daviswiki.org/Doc_Name`. (There are discrepancies, see above.) Assess the relevance of each document for the query. Use the following four-point scale:

(0) Irrelevant document. The document does not contain any information about the topic.
(1) Marginally relevant document. The document only points to the topic. It does not contain more or other information than the topic description.
(2) Fairly relevant document. The document contains more information than the topic description but the presentation is not exhaustive.

(3) Highly relevant document. The document discusses the themes of the topic exhaustively.

[E. Sormunen. Liberal relevance criteria of TREC—Counting on negligible documents? *ACM SIGIR*, 2002]

Edit the results into a **plain text file** using the following space-separated format, one line per assessed document:

`QUERY_ID DOC_ID RELEVANCE_SCORE`

where `QUERY_ID` = 1, `DOC_ID` = the name of the document, `RELEVANCE_SCORE` = [0, 1, 2, 3]. Name the text file **`FirstnameLastname.txt`** and send it to **`jboye@kth.se`.**

It should be noted that there is no objectively correct relevance label for a certain query-document combination! It is a matter of judgement. For difficult cases, write a short note (not in the labeling file, but a separate note) on why you chose the label you did. At the review, you will present three difficult cases.

Compute the **precision** (relevant documents = documents with relevance > 0) and **recall** (up to an unknown scale factor) of the returned list. To get a recall estimate up to an unknown scale factor, assume the total number of relevant documents in the corpus to be 100.

At the review

To pass Task 1.5, you should show the text file with labeled documents, in the correct format. You should have emailed it before presenting. You should be able to explain the concepts precision and recall, and give account for the precision and recall (up to a scale factor) of the returned document list.

You should also, for at least 3 documents in the list, expand on why you chose the labels you did.

NOTE: You will not be failed in the review for selecting the ”wrong” labels. However, you might be asked to redo this task if you are unable to motivate your labels in a satisfactory manner.

Task 1.6: What is a good query? (C)

In this task, you will again use the same search settings as in Task 1.5. The objective is here to **reflect on how a user formulates a query based on an information need**, and how the query is merely an incomplete representation of the latent information need.

Consider the following information need:

Info about the education in Mathematics on a graduate level at UC Davis

Design a query which you think will return documents relevant to the information need. Do an intersection search using this query, and

1) look at the number of documents – if there are more than 25, the query is too general, if there are fewer than 2, the query is too specific.

2) when there is a reasonable number (<25), label the returned documents as ”relevant” or ”non-relevant” in exactly the same manner as in Task 1.4. **Remember to save query and returned list for the review**.

Now, go back to the information need description and ponder if you can improve the precision and recall of the returned list by removing, adding or exchanging words in the query. Do a search using the new query, and see if a higher share of relevant documents were returned. Repeat this until you have found an ”optimal” query for the information need description.

At the review

To pass Task 1.6, you should show the optimized query, as well as the full list of query variants leading up to the final query. You should also expand on why you think that the final query gave better precision and/or recall than the earlier variants.

Furthermore, prepare a comprehensive answer to the question *Why can we not simply set the query to be the entire information need description?*

Task 1.7: Large Inverted Indexes (B or higher)

In realistic applications, the complete index is too large to fit in working memory. In this task, you will implement a methodology for storing the index on disk, reading to working memory during query execution.

> Tip: If you are working on the CSC UNIX system, you might not have enough disc space on your account. You can then use the `tmp/` directory on the local hard drive of the Linux machine you are using.

You are free to use any kind of solution, subject to the following requirements:

1) You can use any class in the Java standard edition library, but don't use third-party libraries or systems (specifically: Don't use a database system).
2) The **computational complexity of evaluating a query on this index should be sub-linear in the number of unique terms in the index**. (This excludes, for instance, solutions in which all postings lists are saved in one standard text file.)
3) The **method should NOT hold the entire index in working memory during execution**. For the method to be scalable to indices larger than what fits in working memory, most or all of the index should be kept on disc, only to be read into working memory when needed. (This excludes, for instance, solutions in which the entire index is simply kept in memory as in Tasks 1.2-1.4 and written to one file when finishing the GUI.)

When you have a working solution, try to index and search the Guardian corpus (`/info/DD2476/ir17/lab/guardian`). This corpus is not enormous, but still too big to store in main memory. It will thus give you a nice illustration that your code can handle big data sets.

At the review

To pass Task 1.7, you should be able to explain the storing solution you are using, execute the indexing step on the `davisWiki` corpus (which is manageable since the data set is quite small) and allow the teacher to search the index.

Specifically, you should be able to 1) start the GUI, indexing Davis wiki articles and storing the index on disc, 2) perform the search queries in Tasks 1.2-1.4 with the same answers as above, 3) quit the GUI, saving the index, 4) restart the GUI WITHOUT indexing again, and 5) again perform the search queries in Tasks 1.2-1.4 with the same answers as above.

You should also be able to explain how the computational complexity, i.e., how the computation time for a certain query varies with the size of the searched dataset, can be evaluated experimentally. (Note that you do not have to do this evaluation since the existing dataset is much too small.)

Moreover, you should be able to explain all parts of your code.