Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Search Engines and Information Retrieval Systems

« Assignment 1 - Boolean retrieval »

January 20th - February 10th

Author
Rémi Domingues

Teachers
Johan Boye
Hedvig Kjellström

Scholar year 2014-2015

Instructions

This report describes the implementation of a search engine demonstrator.

To execute it, first set the right working directory in *src/SearchGUI.java* :

public static final String homeDir = "/MY_PATH/search-engine";

In the following, we will consider that the working directory '.' of your shell is the content of the variable homeDir.

Compilation : *javac -encoding ISO-8859-1 -Xlint :none -cp . :./pdfbox src/*.java*

Execution : *java -Xmx1024m -cp . :./pdfbox src.SearchGUI -d ./davisWiki*

davisWiki is a folder containing files to be indexed. Any number of directories can be specified (*-d directory1 -d directory2*). Every text and readable PDF files in those directories will be indexed by the search engine.

1.1 Basic inverted index

Our words are storing using the following data structure :

— **HashTable** containing for each **word** (key) a list of documents (word, PostingsList)

— **PostingsList** is a **Sorted linked list** containing *PostingEntries*, i.e. pairs *<documentID, PostingsEntry*

— **PostingsEntry** is a **sorted list** providing better performances than standard lists thanks to the use of **skip pointers**

Therefore, if we iteratively parse the documents, following an ascending order per document ID and offset per document, the method *HashedIndex.insert()* has a best-case average complexity of O(1).

The search engine can now achieve queries composed of a single word :

— zombie => 36 documents

— attack => 228 documents

1.2 Multiword queries

We have adapted the **intersection algorithm** in page 11 of the book *Introduction to Information Retrieval* in order to process multiword queries. Every document containing both words will be returned (we do not check the word offset at the moment, hence this is not a phrase query).

Below is described the intersection algorithm :

```
documents = []
for i in tokens.size() do
    docIterators[i] = getPostings(tokens[i]).iterator()
end for
while Every iterator has a successor do
    maxDocID = getMax(docIterators)
    n = 0
    for all docIterators i do
        if i.docID == maxDocID then
            ++n
        end if
    end for
    if n == 1 then                ▷ We have found a document containing every tokens
        documents.add(maxDocID)
    else                          ▷ At least one iterator is different than maxDocID
        for all docIterators i do
            while i != null AND i.docID < maxDocID do
                i = i.next()
            end while
        end for
    end if
end while
return documents
```

This algorithm has been slightly improved by sorting the list containing the document lists in ascending order. Therefore, since the algrithm will start with the smallest list, we should iterate less times every list when this is not relevant (when processing more than two words).

The search engine can now achieve queries composed of a single word :
— zombie attack ⇒ 15 documents
— money transfer ⇒ 106 documents
— sleeping on campus ⇒ 82 documents

1.3 Phrase queries

To process phrase queries, we have replaced *documents.add(maxDocID)* by the following code :

```
if(queryType == intersectionQuery)
    documents.add(maxDocID)
else if (queryType == PHRASE_QUERY)
    offsetIterators = []
    for each docIterator di
        offsetIterators.add( di.offsetIterator())
    End for
    // Ready for ranking by number of sentences per document
    positions = getSentencePositions(offsetIterators)
    if(!positions.isEmpty())
        documents.add(maxDocID, positions.getFirst())
```

The *getSentencePositions()* function is described below for 2 words. It returns the starting index of every sentence in the given document.

```
List<Integer> getSentencePositions(pp1 = positions(p1), pp2 = positions(p2))
    r = []
    while(pp1.hasNext())
        tmp1 = pp1.next()
        while(pp2.next() < tmp1);
        tmp2 = pp2.previous()
        if(tmp2 == tmp1 + 1)
            r.add(tmp1)
        pp2.next()
    return r
```

Below is the algorithm for the phrase queries and N variables. It also returns the starting position for every sentence matching the given query and document.

```
List<Integer> getSentencePositions(List<ListIterator<Integer>> offsetIterators)
    r = []
    // Iterator on the lists of offsets
    offsetIterIterator = offsetIterators.iterator()
    pp0 = offsetIterIterator.next()

    // For each offset of the first list
    while(pp0.hasNext())
        tmp0 = pp0.next()
        tmp1 = tmp0
        sentence = true

        // For every other list of offsets, we check if if contains an offset
            following the previous one
        while(offsetIterIterator.hasNext())
```

```
            pp2 = offsetIterIterator.next()

            // While the current offset is lower than the previous one
            while(pp2.hasNext() AND pp2.next() < tmp1);
            tmp2 = pp2.previous()
            // If the second word is not exactly after the previous one
            if(tmp2 != tmp1 + 1)
                // The previous word position cannot lead to a sentence, we jump
                    to the next one
                sentence = false
                break;
            // The second word follows the previous one, we try the others
            pp2.next()
            tmp1 = tmp2

        // The document contains the right sentence
        if(sentence)
            r.add(tmp0)

        // Position the list iterator on the first list of offsets
        offsetIterIterator = offsetIterators.iterator()
        // The first iterator is already known
        offsetIterIterator.next()

    return r
```

The search engine can now achieve queries composed of sentences :
- zombie attack ⇒ 14 documents
- money transfer ⇒ 2 documents
- sleeping on campus ⇒ 19 documents

Why are fewer documents generally returned in phrase query mode than in intersection query mode ?

As observed in the previous algorithm, the a phrase query is an intersection query on which we add a constraint. This constraint requires that a document contains the given word in the specified worder and without any other word between them. Therefore, we only return the documents in the intersection query that also satisfy this new constraint.

1.4 What is a good search result ?

1.4.1 Query evaluation

The following intersection queries return the given results, detailed in doc/-query_evaluation.txt

- skiing trip ⇒ 6 documents
  - `https://daviswiki.org/Northern_California` ⇒ 2 or 3 ? (2, destination advice)

  - `https://daviswiki.org/Pre-2010_Reviews` ⇒ 0 or 1 ? (1, snowboard, equipment, destination)
  - `https://daviswiki.org/Outdoor_Adventures` ⇒ 2 or 3 ? (3, trip offers, ski)
- university rowing team ⇒ 5 documents
  - `https://daviswiki.org/Davis_Life_Magazine` ⇒ 1 or 2 ? (1, only mention mention the team)
  - `https://daviswiki.org/UC_Davis_Budget_Cuts` ⇒ 1 or 2 ? (2, not much information, but important information about the team)
  - `https://daviswiki.org/Protests` ⇒ 0 or 1 ? (1, it's about the team)
- tourist attractions ⇒ 4 documents
  - `https://daviswiki.org/Central_Coast` ⇒ 2 or 3 ? (3, wide document but gives hints)
  - `https://daviswiki.org/Northern_California` ⇒ 1 or 2 ? (1, area description but not for tourists, give links)
  - `https://daviswiki.org/UC_Davis_Budget_Cuts` ⇒ 2 or 3 ? (2, only a minor part of the document is interesting)

1.4.2 Precision and recall

Using the following formulas, we calculate the precision and recall for each query :

$$precision = \frac{|\{relevantdocuments\} \cap \{retrieveddocuments\}|}{|\{retrieveddocuments\}|} \tag{1}$$

Precision describes the relevance of the retrieved documents, i.e. the amount of junk retrieved. It is the number of correct results divided by the number of results returned by the query.

$$recall = \frac{|\{relevantdocuments\} \cap \{retrieveddocuments\}|}{|\{relevantdocuments\}|} \tag{2}$$

Recall gives an information about the completeness of our search engine, i.e. how many relevant documents have been retrieved divided by the number of relevant documents.

- skiing trip
  - $precision = \frac{4}{6} = 0.666$
  - $recall = \frac{4}{1000}$
- university rowing team
  - $precision = \frac{4}{5} = 0.8$
  - $recall = \frac{4}{1000}$
- tourist attractions
  - $precision = \frac{4}{4} = 1$
  - $recall = \frac{4}{1000}$

1.5 What is a good query ?

Are there any historical monuments in Davis ? is a query expressed to another human. If we are trying to adapt computers so that humans can interact with them like another human, our search engine is not complex enough to handle such a query. Stop words must be removed, we obtain the following queries :

Various queries have been tried :
- history monument (Davis) : $p = \frac{2}{5}, r = \frac{2}{1000}$
- historic monument (Davis) : $p = \frac{1}{3}, r = \frac{1}{1000}$
- historical monument : $p = \frac{0}{1}, r = \frac{0}{1000}$
- monument : $p = \frac{4}{15}, r = \frac{4}{1000}$
- monument Davis : $p = \frac{3}{13}, r = \frac{3}{1000}$
- historic : 181 documents
- historcal : 296 documents
- history : 1418 documents

Considering that only 15 documents contain the word document, it is very likely that the 3 last requests have a very very bad precision, bad enough so that those cannot be compensated by a higher recall.

'monument' has the best recall, since it contains very few constraints, but 'history monument' has the best precision. However, 'monument' has a F1 score of 0,0078 whereas 'history monument' barely reaches 0,0039. Therefore, 'monument' is the optimal query.

History monument

```
Bomb_Shelter 1
Monticello 0
Northern_California 0
Train_Station 3
UC_Davis_Geology_Department 0
```

Monument

```
AliPezeshkpour 0
Bomb_Shelter 1
Cell_Phone_Towers 0
County_Road_95 0
Downtown_Phone_1 0
DylanBeaudette 0
East_Area_Water_Tank 0
Griffin_Lounge 2
MichaelNielsen 0
Monticello 0
MONUMENT 0
Northern_California 0
The_Foundation 3
Train_Station 3
UC_Davis_Geology_Department 0
```

Why can we not simply set the query to be the entire information need description ?

The query contains many useless words, called stop words, such as 'are', 'there' and 'any' which do not bring any valuable information for the search. 'in' could be valuable since it describes an area linked to a location (Davis). However, since our search engine does not support yet such a functionality, we won't consider it.

The keywords 'historical', 'monuments' and 'Davis' should be transformed to lemmas in order to match more words in the indexed documents.

'Davis' is not really important here since the Wiki is mainly about Davis.

1.6 Large inverted index

1.6.1 File structure

A 'cache' directory. This one contains N+2 files, with N the number of unique words.

Each PostingsList (one PostingsList per word) is stored in a binary file.

The HashMap containing the document IDs is also stored.

The same goes for a TreeSet containing for each token, its popularity, i.e. the number of times it has been part of a query.

1.6.2 Data structures

=> Map<string, postingslist> : document IDs, allow the access to documents path from their ID in O(1) => Map<string, postingslist> : index, allow the access to postings in updating of the popularity in O(1) => TreeSet<PostingsList> : popularity set, to keep the postingslist sorted by popularity in order to remove the words below the 10% threshold. Insert / delete when updating the popularity in 2*log(N)

1.6.3 Indexing

The postings list of the 10% having the highest frequency (the word popularity is its frequency during indexing) are also always in memory (this figure should be adapted depending on the available memory and index size).

Each time Y documents have been indexed, we read from disk, merge in memory and override the existing files for the words which do not belong to the 10% threshold (around 15 000 words) and remove the postings entries from memory. When the indexing is over, the remaining words in memory are updated on the disk, and the words below the 10% threshold are removed from memory.

The popularity of every word is set to 0, the document IDs and popularity set are also written on disk.

1.6.4 Querying

Each query increases the popularity of the requested words by one. If the word is not in memory, we parse the corresponding file to load it. When every word are in memory, we apply the search query. Then, the result is returned.

Every K requests, the words below the 10% threshold are removed from memory.

1.6.5 Exiting

When choosing "Save and exit" in the menu, the popularity set is updated on disk, in order to load in priority the popular word when restarting the search engine.

1.6.6 Starting

When launching the search engine, the files containing the popularity set and the document IDs are loaded in memory, the 10% most popular words are loaded.