Advanced Data Structures and

Algorithm Analysis

Laboratory Projects

**Project 2. Roll Your Own Mini Search Engine**

**LIU HAO XIONG**

**XIAO RUI**

**SHEN ZHE YU**

**Date: 2019-03-16**

# Chapter 1: Introduction

## Problem Description

A web search engine is a software system that is design to carry out web search. In this project, we are going to create our own search engine which can handle inquiries over “The Complete Works of William Shakespeare” (<http://shakespeare.mit.edu/>).

Our main tasks are:

1. Run a word count over the Shakespeare set and try to identify the stop words (also called the noisy words).
2. Create our inverted index over the Shakespeare set with word stemming.
3. Write a query program on top of our inverted file index, which will accept a user-specified word (or phrase) and return the IDs of the documents that contain that word.
4. Run tests to show how the thresholds on query may affect the results.

## Background of the algorithms

* Pre-processing

Since the content of the website (<http://shakespeare.mit.edu/>) remains unchanged, we get all the data in advance with a web crawler and store them in txt format. Then we employ the stemmer on these txt files. At the same time, we record the collection frequency and identify stop words. Now, the main tasks for us are indexing and searching.

* Indexing

In order to build an index file, we traverse all the files and associate every word appeared in txt files with a postlist by *std::map* container of C++, which helps us identify the word frequency and the documents including them.

* Searching

To implement the search function, we made the index file available for search queries. The index helps find information relating to the query as quickly as possible.

# Chapter 2: Algorithm Specification

## Data Structure

The core part in our program is the class InvertedFileIndex which we defined in the file InvertedFileIndex.h. We use several kinds of containers of the C++ STL in this class. They are: std::map, std::set, std::string and std::vector. The definition is as following:

class InvertedFileIndex

{

public:

InvertedFileIndex() {};

bool GetStopWord();

bool UpdateIndex();

void InsertWord(std::string word, int docID);

std::vector<std::string> QuerySearch2(std::string& query, float threshold);

~InvertedFileIndex();

private:

std::map<std::string, PostList\*> InvertedIndex;

std::set<std::string> StopWord;

std::vector<std::string> Documents;

};

The *std::map<std::string, PostList\*> InvertedIndex* is the index file we used in the search engine. The container *std::map* is a fairly well-rounded dictionary-type that provides several advantages if we need storage of keys and values. Although it’s not strictly specified, we can take it as a kind of self-balancing binary tree which has a good lookup time and insertion time.

We use the *std::set* to store the stop words we get in *bool GetStopWord().* When we are going to insert a new word into the map, we check whether the word is a stop word first. The set is an ideal container for us to record whether a word is a stop word or not.

And finally, we use the *std::vector* to store the documents names.

For the postlist, we also implement it as a class and the structure is as following:

class PostList

{

public:

friend class InvertedFileIndex;

PostList():freq(0),docID(0){};

~PostList(){};

public:

int freq;

std::vector<std::pair<int, int> > docID;

};

The integer *freq* represents the document frequency (the number of documents which contain each term). And the *std::vector<std::pair<int, int> > docID* is to record the term frequency of all the documents, where term frequency means the frequency of each term in each document.

## Descriptions of all the key algorithms

* bool InvertedFileIndex::GetStopWord();

This method first stems the documents, then helps find the stop word list.

For each document we fetched from a corresponding page of [**http://shakespeare.mit.edu/**](http://shakespeare.mit.edu/) by our spider, we employ the porter2 stemmer implemented by smasssung’s team provided on github[[1]](#footnote-1) to stem each term in the document. At the same time, we got a list of filenames of the documents. We use a map structure to record the collection frequency of each term. Once a stemmed term appears, we increment its collection frequency, except it becomes an empty string after stemming. After all documents are handled, we sort the terms according to its collection frequency, add those with higher collection frequencies than ‘henry’ to stop word list, which is our hand-filtered result for the collections of Complete Work of Shakespeare. For arbitrary documents collections, we may choose those with collection frequency higher than some times of number of documents, for example, 3.

If there is already a StopWordList.txt and FilenamesList.txt under the directory, we retrieve the stop word list and list of documents’ names from the 2 files.

The pseudo code is as following:

bool InvertedFileIndex::GetStopWord()

{

if(there is already a stop word and a document list)

restore them to StopWord and Documents;

else{

OriDir <- "ShakespeareComplete";

DstDir <- "StemmedShakespeare";

create DstDir

open the OriDir

map<string, int> term\_freq;

while((entry <- readdir(dir)) != NULL)

{

filename = entry->d\_name;

if(filename == "." || filename == "..") continue;

out.open(DstDir + "\\" + filename);

in.open(OriDir + "\\" + filename);

while (in >> to\_stem)

{

stem the word

if(to\_stem == "") continue;

term\_freq[to\_stem]++;

write to the corresponding destination file in DstDir

}

store the filename in FilenamesFile

}

copy the map to a vector and sort

for(auto it = vec\_tf.begin(); it != vec\_tf.end(); ++it)

{

if(it->first != "henri") // for Shakespeare Complete Works

add it to StopWord

else break;

}

return true;

}

* bool InvertedFileIndex::UpdateIndex();

The function UpdataIndex() is used to scan all the files in the folder and then handle every word. We use a FOR statement to traverse all the string in the *std::vector<std::string> Documents.* And then if a file is opened correctly, we handle all the words in this file with the function *InsertWord(string word, int docID)*.

The pseudo code is as following:

bool InvertedFileIndex::UpdateIndex()

{

string word;

string dirname <- "StemmedShakespeare";

int docID <- 0;

for all the document names:

open file "StemmedShakespeare\\docname.txt"

if (the file docname.txt is opened)

for all the words in docname.txt

if the word is not a stop word

InsertWord(word, docID);

endif

end

else

print ("Open File Failed")

endif

docID <- docID + 1;

end

return true;

}

* void InvertedFileIndex::InsertWord(string word, int docID);

The function InserWord() is used to insert a word into the map *InvertedIndex* correctly. If the word is not included in the map before, then we insert a new key. If the word is already in the map, we then update the PostList. The pseudo code is as following:

void InvertedFileIndex::InsertWord(string word, int docID)

{

auto map\_it <- InvertedIndex.find(word);

if the word is not included in the map

//Insert a new key

PostList \*p <- new PostList();

(\*p).freq <- 1;

(\*p).docID.push\_back(pair<int, int>(docID, 1));

InvertedIndex.insert(pair<std::string, PostList\*>(word, p));

else if

//update the PostList

if there is already the same word in this document

(\*(map\_it->second)).docID.back().second ++;

else if the word has never appeared in this doc yet

(\*(map\_it->second)).freq ++;

(\*(map\_it->second)).docID.push\_back(pair<int, int>(docID, 1));

endif

endif

return;

}

* vector<string> InvertedFileIndex::QuerySearch2(std::string& query, float threshold)

This method implements the search procedure, returning the list of documents containing any word in the given query in RANKED order.

First we find the stemmed terms of the given query in inverted index to get their document frequency df. Then we sort the terms by their df in ascending order and abandon part of words in light of the given threshold (cell(threshold \* number of terms) terms are held) .

Then, we get the poslist of these words and add all documents that contain any one of the words to the doc\_list. Here we take the union instead of intersection since we will rank the retrieved documents by an index called tf-idf, which equals to term frequency in a particular document multiplied by log (N/df) (called idf), where N is the total number of documents. By calculating the score, sum of tf-idf of every term, for every document, we succeed to rank every document by the score.

The pseudo code is as following:

vector<string> InvertedFileIndex::QuerySearch2(std::string& query, float threshold = 1.0)

{

terms << query;

while (terms >> term)

{

Stem the term.

If(the term is in StopWord) continue;

If(term in InvertedIndex) insert it to termlist

}

if (termlist.size() == 0)

{

cout << "Can't find query in the documents, please change the query and try again." << endl;

Return an empty vector.

}

Sort the term list according to frequency of every term;

term\_num = int(termlist.size() \* threshold);

if (term\_num == 0) term\_num++;

pair<map<int, float>::iterator, bool> ret;

for (i <- 0; i < term\_num; ++i)

{

idf <- log(float(N) / termlist[i].second->freq);

for (auto it : termlist[i].second->docID)

{

Compute tf-idf;

ret.first->second += tf-idf;

}

}

for (auto iter : doclist)

{

DocList.push\_back(iter);

}

Sort the DocList according to score;

for (auto it\_doc : DocList)

Filenames.push\_back(Documents[it\_doc.first]);

return Filenames;

}

* void Test(std::vector<std::string> res, std::string query);

This method tests the correctness of our Inverted index, by check if the document retrieved really contains words or phrases in a query, and make sure every related document is retrieved. By ‘related’, we mean the document contains at least a word in a query.

We set a bool related\_doc to tag every document, with a default value false. We visit all the documents and read them term by term just as what we did in the word count step. For each term, we check if it is in our query. If it is, we set related\_doc to true.

Then we compare the list of documents tagged by true with the actual documents we retrieved. If and only if the two are no different we get our inverted index right.

A special case we handle here is that when all the words inquired become stop word after stemming, we should check if the retrieved list is empty.

The pseudo code is as following:

void InvertedFileIndex::Test(vector<string> res, string query)

{

for (auto it : res) FileList.insert(it);

terms << query;

while (terms >> term) //Get the terms in the query.

{

stem the term;

termlist.insert(term);

}

Test if all terms are in the StopList;

Delete all stop words from termlist;

If(all terms are in the StopList)

{

If(there are no documents retrived)

{

cout << "Correct: Query is in StopList, and there are no documents retrived." << endl;

return;

}

else

{

cout << "Error: Query is in StopList, but there are documents retrived." << endl;

return;

}

}

bool related\_doc;

for (every document)

{

related\_doc = false;

while (in >> term)

{

it\_set = termlist.find(term);

If(term is in termlist)

{

related\_doc = true;

it\_set = FileList.find(filename);

If (filename isn't in FileList)

{

cout << "Error: " << filename << " contain the term " << term

<< ", but is not retrived." << endl;

return;

}

break;

}

}

If(no terms in termlist are in this document)

{

If(filename is in FileList)

{

cout << "Error: Document " << filename << " is retrived, but in fact it is irrelevance." << endl;

return;

}

}

}

cout << "Correct: The documents retrived are all relevance, and other documents are all irrelevance." << endl;

return;

}

## Sketch of the main program

In our main function, we first create a InvertedFileIndex object named SearchEngine and then use the method GetStopWord() to get stop words and store them in the set StopWord. After that, we call the method UpdateIndex() of the SearchEngine to scan all the documents and build our index file. Then we start searching words with the help of the index file, as well as test the correctness of the Inverted Index we built. We call the function QuerySearch2() and Test() until the user inputs “q” to exit. The pseudo code is as following:

int main()

{

InvertedFileIndex SearchEngine;

SearchEngine.GetStopWord();

SearchEngine.UpdateIndex();

std::cout << "Please enter the query, and if you want to quit,

just enter 'quit!':" << std::endl;

std::string str;

std::getline(std::cin, str);

while (str != "q")

using namespace std::chrono;

high\_resolution\_clock::time\_point t1 <-

high\_resolution\_clock::now();

std::vector<std::string> res;

int times <- 10000;

for i <- 0 to times-1

res <- SearchEngine.QuerySearch2(str, 1.0);

end

high\_resolution\_clock::time\_point t2 <-high\_resolution\_clock::now();

duration<double> time\_span <-

duration\_cast<duration<double>>(t2 - t1);

SearchEngine.Test(res, str);

std::cout << "Search Engine spends " << time\_span.count() <<”s”

<< times << " times and retrives " << res.size()

<< " documents." << std::endl;

int i <- 0;

for all the files we get

i++;

std::cout << "No." << i << ": " << it << std::endl;

end

std::cout << std::endl << "Please enter the query, and if you

want to quit, enter 'q':" << std::endl;

std::getline(std::cin, str);

end

return 0;

}

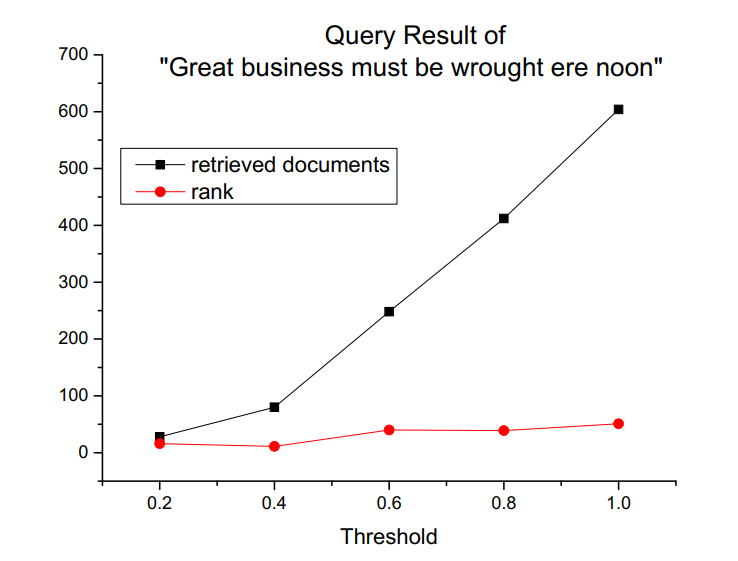
# Chapter 3: Testing Results

Test six different cases for correctness of our program, including an empty query, a stop word, a word in the documents, a word not in the documents, a phrase in the documents, a phrase not in the documents, as is shown in the Graph1. It should be noted that it’s impossible to predict what documents will be retrieved and how they will be ranked, so we design a test program which is introduced above for testing the correctness of out InvertedIndex and SearchEngine. The results here is based on the test program, and we won’t present the results of SearchEngine.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case | Testing Purpose | Expected Results | Actual Behavior | The possible cause of a bug | Status |
| Please enter the query, and if you want to quit, just enter 'q':  Enter threshold: **1** | Query = “” | Correct: Query is in StopList or empty, and there are no documents retrieved. | Correct: Query is in StopList or empty, and there are no documents retrieved. | Empty term may be stored in InvertedIndex wrongly | Pass |
| Please enter the query, and if you want to quit, enter 'q':  **and the**  Enter threshold: **1** | Query is a stop word | Correct: Query is in StopList or empty, and there are no documents retrieved. | Correct: Query is in StopList or empty, and there are no documents retrieved. | Stop word may be stored in InvertedIndex wrongly | Pass |
| Please enter the query, and if you want to quit, enter 'q':  **highness**  Enter threshold: **1** | Query is a word in documents | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | InvertedIndex may omit some documents containing term, or add irrelevance documents wrongly | Pass |
| Please enter the query, and if you want to quit, enter 'q':  **googlegoogle**  Enter threshold: **1** | Query is a word not in documents | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Terms not in documents may be stored in InvertedIndex wrongly | Pass |
| Please enter the query, and if you want to quit, enter 'q':  **To live in prayer and contemplation**  Enter threshold: **1** | Query is a phrase in the documents | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | InvertedIndex may omit some documents containing term, or add irrelevance documents wrongly | Pass |
| Please enter the query, and if you want to quit, enter 'q':  **hhhhh ggggg wwwww**  Enter threshold: **1** | Query is a phrase not in the documents | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Correct: The documents retrieved are all relevance, and other documents are all irrelevance. | Terms not in documents may be stored in InvertedIndex wrongly | Pass |

Next we will test how the thresholds on query may affect the results. According to our method to retrieve the documents, the amount of documents retrieved and time used in search should decrease as threshold gets smaller. We will choose a phrase “Great business must be wrought ere noon” from document “macbethmacbeth35.txt”, and mark the rank of this document with different threshold.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case | Testing Purpose | Actual Behavior | Rank | Time |
| Please enter the query, and if you want to quit, just enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **1** | Threshold = 1 | Search Engine spends 168.342s in 10000 times and retrieves 604 documents. | No.51/604 | 16.8ms/search |
| Please enter the query, and if you want to quit, enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **0.8** | Threshold = 0.8 | Search Engine spends 104.852s in 10000 times and retrieves 412 documents. | No.39/412 | 10.5ms/search |
| Please enter the query, and if you want to quit, enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **0.6** | Threshold = 0.6 | Search Engine spends 63.8881s in 10000 times and retrieves 248 documents. | No.40/248 | 6.4ms/search |
| Please enter the query, and if you want to quit, enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **0.4** | Threshold = 0.4 | Search Engine spends 26.1671s in 10000 times and retrieves 80 documents. | No.11/80 | 2.6ms/search |
| Please enter the query, and if you want to quit, enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **0.2** | Threshold = 0.2 | Search Engine spends 15.077s in 10000 times and retrieves 28 documents. | No.16/28 | 1.5ms/search |
| Please enter the query, and if you want to quit, enter 'q':  **Great business must be wrought ere noon**  Enter threshold: **0** | Threshold = 0 | Search Engine spends 14.6351s in 10000 times and retrieves 28 documents. | No.16/28 | 1.5ms/search |



An appropriate threshold (in this case, 0.4) shows better recall and precision.

# Chapter 4: Analysis and Comments

1. **Complexity analysis of each function**

**Function: InvertedFileIndex::GetStopWord()**

Description: Do word count and provide a stop word list, as well as stem the documents.

Time Complexity: O(NlogM), where N is the total number of words in all the documents, M is the number of different words in the collection.

Space Complexity: O(M)

**Function: InvertedFileIndex::** **InsertWord(std::string word, int docID);**

Description: Insert a word into inverted index

Time Complexity: O(logM)

Space Complexity: O(1)

**Function: InvertedFileIndex::UpdateIndex();**

Description: Build the inverted index according to the stemmed documents.

Time Complexity: O(NlogM)

Space Complexity: O(min(MK, N)), where K is the total number of documents. This is an upper bound estimation, in most cases it can be lower. In fact, the space occupied by the inverted index is O(2\*(sum of number of documents containing it for each term) + M).

**Function: InvertedFileIndex::QuerySearch2(std::string& query, float threshold);**

Description: response to the query, search terms in inverted index.

Time Complexity: O(Q), where T is the number of terms in the query

Space Complexity: O(Q)

1. **Complexity analysis of the whole program**

To build a inverted index, we call the method GetStopWord() once and UpdateIndex() once. The process has a O(NlogM) time complexity and needs an upper bound of O(min(MK, N)) to store the inverted index (as stated above, the space occupied by the inverted index is actually O(2\*(sum of number of documents containing it for each term) + M), in most cases significantly lower than the given bound).

The process of query just calls the method QuerySearch2, so its complexity is just as described in the analysis of the function.

1. **Comments**

The program we present here is our ver2.0 search engine. In the ver1.0 , we didn’t rank the retrieved documents, just did the intersection. Correspondingly we didn’t record term frequency in the postlist.

In ver2.0 we retrieved the union of documents related to query terms and rank them since we assume that documents with highest ranks tend to be those contain all the terms. Also, we can avoid abandoning documents

In real-world situation, we may just truncate the retrieved list to control the number of documents provided to the user, so as to increase precision and ensure recall.

1. **Question:** **What if we have 500 000 files and 400 000 000 distinct words? Will your program still work?**

That means K = 500000 and M = 400000000. It will become impossible to store the index (safe for word count considering a typical memory volume of 8GB on contemporary PCs).

A natural solution is to make use of the storage space. As long as N is not too large, it is always possible to store the index in hard disk, let alone employing distributed system.

We may simply first store the index in some blocks. When the temporary index size reach a set blocksize, write it into a block and free the memory. Finally we merge the blocks.

How to implement the merge operation becomes the key point here. We provide a method below:

When building the inverted index, we label each term by id (call it termID). Before write the block into the hard disk, we first sort the index by termID. In the merge step, we open all the blocks in hard disk at the same time and choose the record with lowest termID among all the blocks (notice that here we only need to choose among one record of each block). Once the newly read record has a different termID with the last one, which means there is no more record related to the last one left in blocks, we write it to the file storing the whole index.

# Appendix: Source Code (in C++)

# Declaration

***We hereby declare that all the work done in this project titled "*** ***Project 1. Binary Search Trees" is of our independent effort as a group.***

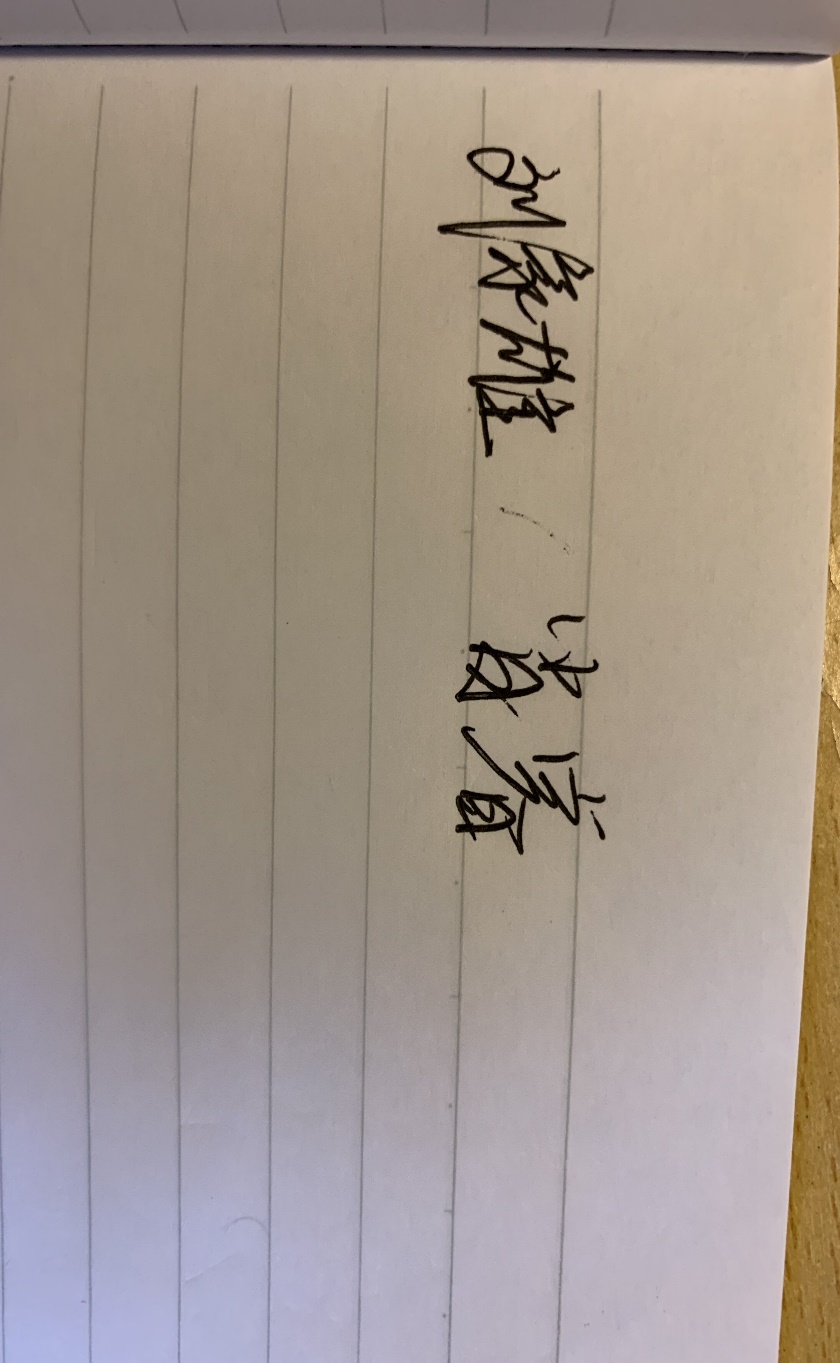
# Author List

**Programmer: LIU HAO XIONG 3160103863**

**Tester: XIAO RUI 3160105533**

**Document: LIU HAO XIONG & XIAO RUI**

# Signature



1. https://github.com/smassung/porter2\_stemmer [↑](#footnote-ref-1)