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

## Background of the algorithms

# 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::UpdateIndex();

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 = 1.0);

QuerySearch2() is to get the index of the documents which contain terms in the query.

First we turn the query into seperate terms which are stemmed and not in StopWord terms and their frequency will be stored in vector termlist. Then, sort the term list according to frequency of every term.

* 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

# Chapter 4: Analysis and Comments

1. **Analysis of each function**

**Function: Create**

Description: Create a linked list using the given array, with a head node, which stores the length of the list.

Time Complexity: O(N)

Space Complexity: O(N)

**Function: Insert**

Description: Insert node to the given position

Time Complexity: O(N)

Space Complexity: O(1)

**Function: Sort**

# 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.***

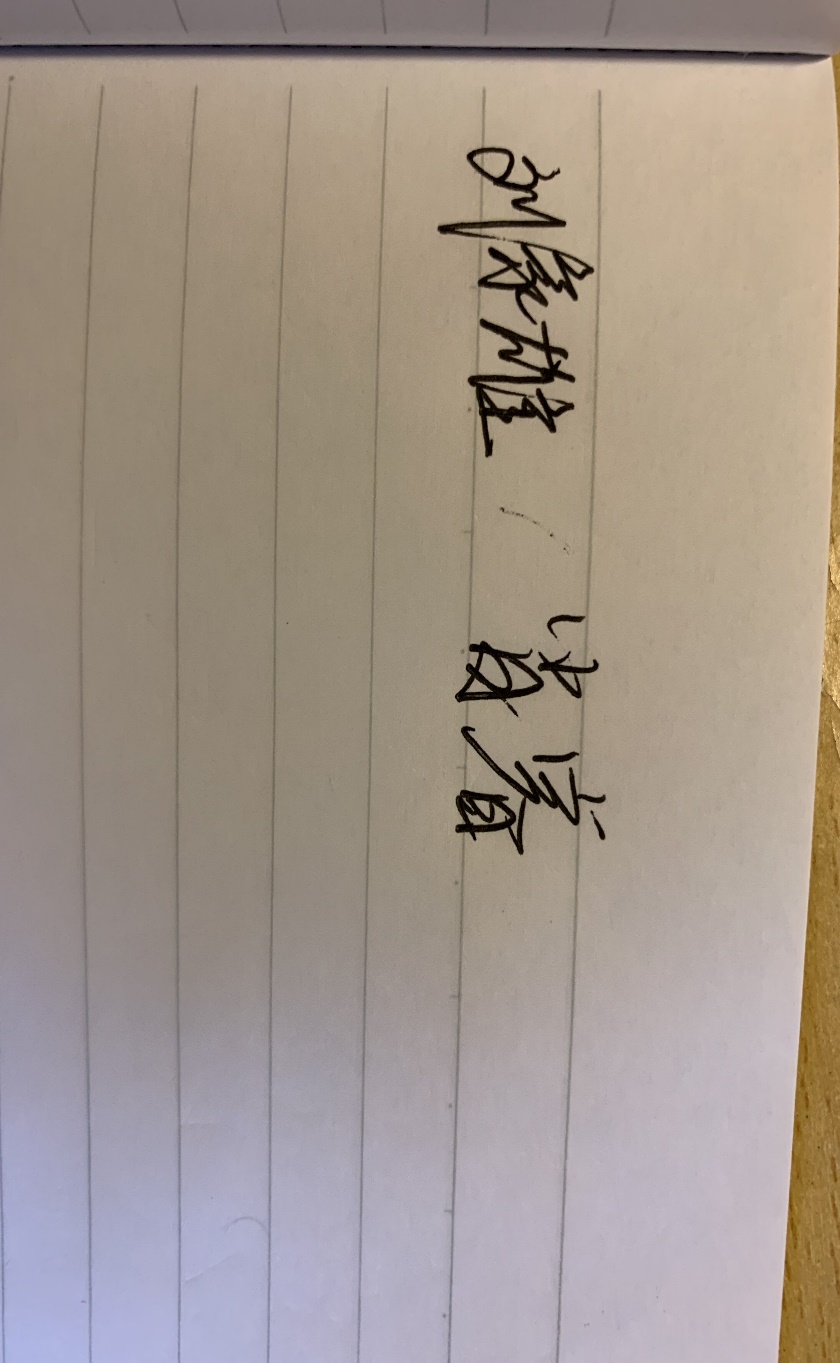
# Author List

**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)