Shakole Search Engine

Author Names

(Group 12)

Guan Jiarui

Qi Yue

Zheng Ming

Date: 2022-03-19

Chapter 1: Introduction

1.1 Problem Description

The task of this project is creating a mini search engine based on "The Complete Works of William Shakespeare". For input word or phrase, we need to return the document ID that contains them.

We use a posting list to index and search. We process word, mark stop words, and do the query threshold. We maintain an AVL tree in it to fasten term-access for the bonus part.

We also add the category-search function for users to search documents in a certain category.

1.2 Background

1.2.1 Inverted File Index

Inverted file index is a word-oriented indexing mechanism where each file can be represented by a series of keywords. A typical inverted index consists of a glossary (also known as an index item) and an event list (also known as a file linked list).

1.2.2 AVL Tree

AVL tree is also a binary search tree in essence, which is characterized by:

1. It is first and foremost a binary search tree.
2. With equilibrium condition: the absolute value of the difference between the height of the left and right subtrees of each node (equilibrium factor) is at most 1.

In other words, AVL trees are essentially binary search trees (binary sort trees, binary search trees) with balanced functions.

Chapter 2: Algorithm Specification

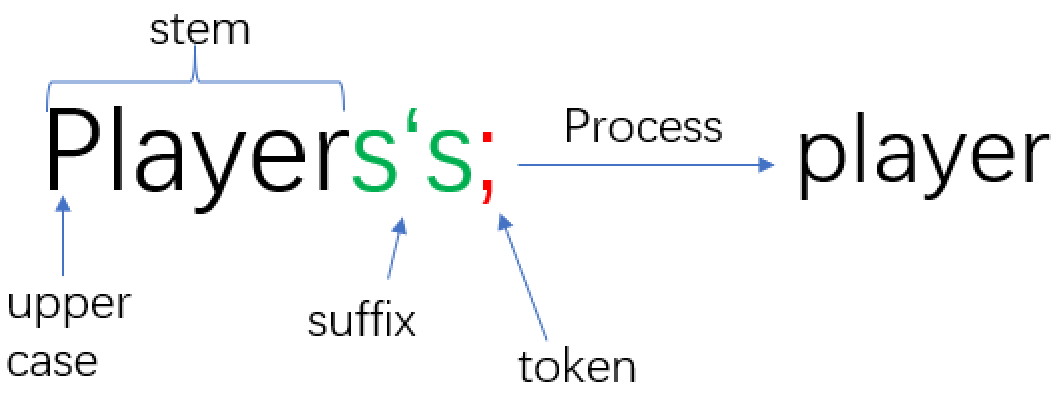
2.1 Generate Document Information

First, rather than read the documents immediately, we get and store the basic information of each document. For each document read from "Documents/", we get its title by read the first line. Notably, we also classify each document into four categories: comedy, history, tragedy and poetry (as it shows on http://shakespeare.mit.edu/). The category will be used in Query, allow users assign a certain category to narrow the range.

To be declared here, the structure of documents in our program is: we consider each piece of work a single document, rather than divide them by Acts and Scenes. So there are 42 document in total.

2.2 Word Process

Since words are read from documents according to white spaces separation, it may contain upper or lower cases, unexpected tokens and suffix. So we need to restore them to basic English words.



2.2.1 Transform to Lower Cases

We simply call a lib function to transform all English characters in the string into lower case.

2.2.2 Remove Tokens

We first delete tokens “ ‘ ”, which appear as ‘s, ‘d, ‘st, ‘n, o’. (not handle them in stem algorithm). Find ’ in the string and delete the later part. (if is o’, delete the former part);

Then we remove all tokens at the beginning and the end of the string, especially for tokens at the end of a sentence.

Note that we do not remove tokens in the middle of a word such as ‘-’ in “short-sighted”, it also means if the format of the input documents is not correct(such as missing a space at the begin of a sentence), it may cause mistakes.

2.2.3 Take the Stem

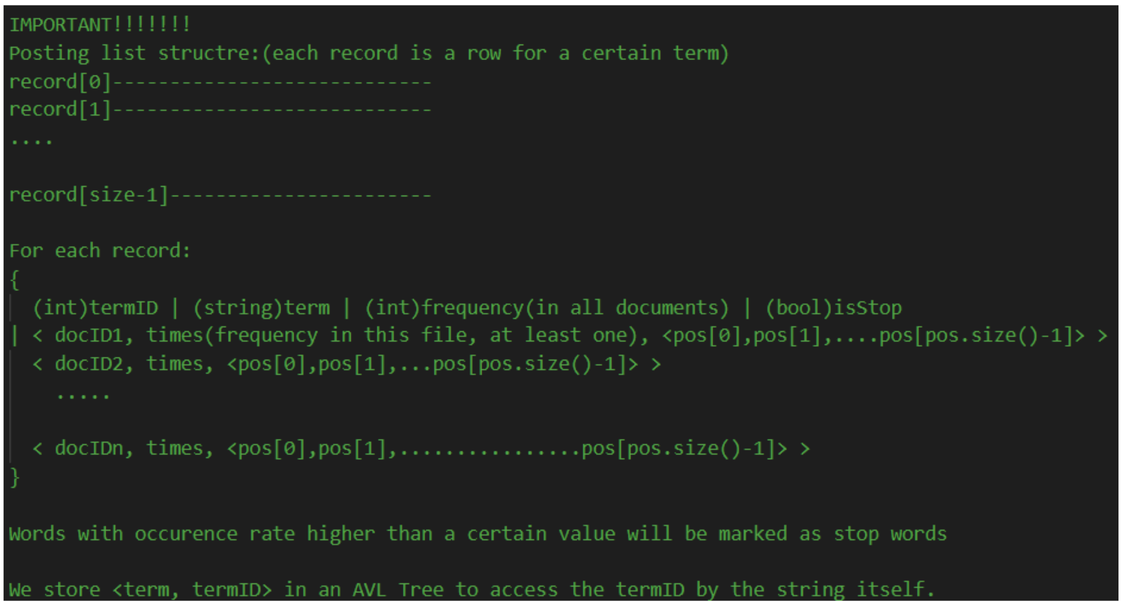
For a word that has removed tokens, we use stem algorithm to take the stem of the word. Because it is related to principles of lexicology, we refer to some materials in the Internet to implement it. (see References).

We need to specify that a stem word may not be a original word. For example, “create” will be turned into “creat”. But if does not matter, we will process the words that we input as well.

2.3 Index (including word count)

Index is the most important part of the search engine. For each word read from document, after process, we need to add them into the posting list. We need to determine the structure of the posting list, and design a data structure to access terms fast.

2.3.1 Structure of the Posting List



```
IMPORTANT!!!!!!!
Posting list structre:(each record is a row for a certain term)
record[0]----------------------------
record[1]----------------------------
....

record[size-1]-----------------------

For each record:
{
|  (int)termID | (string)term | (int)frequency(in all documents) | (bool)isStop
| < docID1, times(frequency in this file, at least one), <pos[0],pos[1],....pos[pos.size()-1]> >
|  < docID2, times, <pos[0],pos[1],...pos[pos.size()-1]> >
|    .....
|
|  < docIDn, times, <pos[0],pos[1],................pos[pos.size()-1]> >
}

Words with occurence rate higher than a certain value will be marked as stop words

We store <term, termID> in an AVL Tree to access the termID by the string itself.
```

We first use a dynamic linear table (using vector in STL for implementation) to store records(rows). Each record stores information of a single term. The row number of the posting list is also the index of record, and also the ID of the term in that record.

For each record, we store such information:

1. ID of the term

2. The term it self (a stem word string)

3. Frequency (how many times it appear in all documents)

4. isStop: if the term is a stop word or not.

5. Doc_pos = <docID, times, pos<int>> [doc_pos_size]

Doc_pos is a inner vector that stores the document that contains this term, and stores the occurrence times and positions of this term in this document.

Note:

Frequency: how many times it appear in all documents.

Times: how many times it appear in a certain document

Frequency is the sum of times theoretically.

While adding a word, we also tell the docID that the word comes from as well as its position in this document. Then we call findID function to find the ID of the word (also the row number of the list), then modify the frequency of inner vector in that record. If findID returns -1, if means this word is a new word and we create a new record in the list.

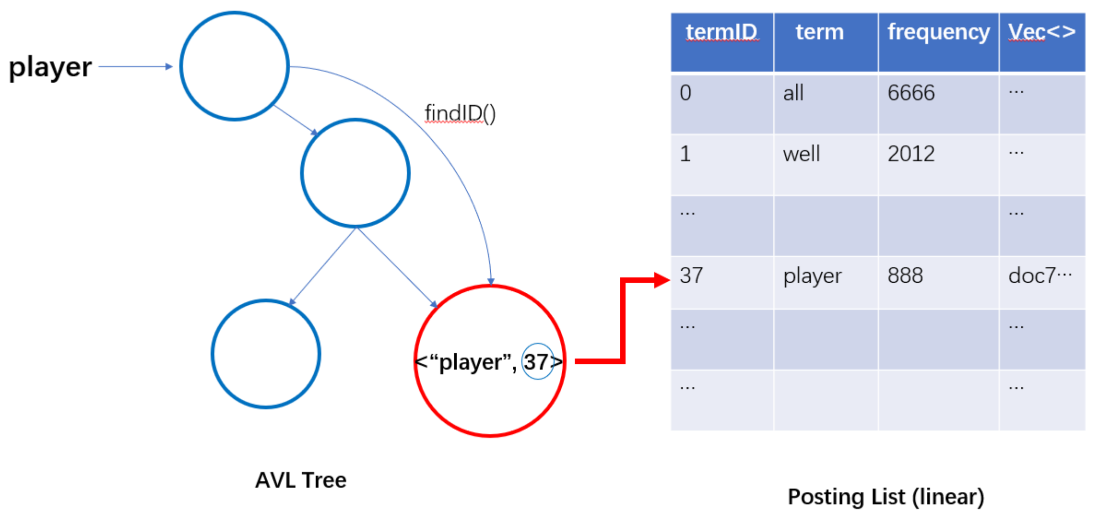
2.3.2 AVL tree for term access (*including bonus*)

"findID" is a frequently called function to find the position of the term according to the string itself. It is not only called while adding, but also called while querying. It's performance is highly related to the time cost for the search engine, so it's necessary to improve it.

At first we use linear scan and find it costs over 80 seconds to index all documents (with about 17776 distinct word). It obviously cannot handle 500 000 files and 400 000 000 distinct words in the problem requirement. Then we use map(<string term, int termID>) in C++ STL to implement it, time cost is reduced to about 0.9 second, pretty fast. But since it's a project and access term is a critical part, it's not good to directly use STL, so we design the AVL tree to do that.

In the nodes of AVL tree, we store pair<string term, int termID> (pair is a binary group) in it. Comparision is only made between the first element (string term), and we can find the node according to input word and return the termID from the node. The AVL tree is implemented by ourselves.

Of course we can also directly store all information in the records of the posting list into the AVL tree, it will save a little space but not much.



termID	term	frequency	Vec<>
0	all	6666	…
1	well	2012	…
…			…
37	player	888	doc7…
…			…
…			…

(example procedures for adding or searching, not real data)

2.3.3 Mark stop words

Although the problem description requires us not to include stop words in the posting list, we still do that, but just mark them.

But we have several reasons:

1. As the teacher said in class, in some situations we need to search stop words. For example when we search stop words, we can give a threshold for occurrence time. Only if the "times" in the doc_pos vector exceeds this value will the engine return the documents. It will be a extensible feature. (not implemented yet)

2. If we simply build a big table of stop words and check in the table. It may be not complete, especially for many old English words in Shakespeare's works. Instead, we can automatically check the frequency of a word and mark it as a stop word if the frequency is too high.

3. The number of stop words is not very big(about 112) compared with the number of

all distinct words(about 17776), so it doesn't occupy much space in the posting list. So we set a word-appearance-rate threshold (0.0013). If a word's appearance-rate is higher than it, it will be marker as a stop word in the record. We still store it, but we do not return them while searching.

2.4 Query

Generally, for each query, the program read a line of string including one or several keys (as words separated by spaces), then return the name of documents that containing all of the keys.

2.4.1 Extract Category

For each query, users can optionally assign the category by appending "=category" at the end of the searching terms; The format is:

key1 key2 key3 ... =category

2.4.2 Search Words and Do Threshold

First, we divide the line of key (a single word, phrase or sentence) into separated words. Then we preprocess the list of keys. Before passing the list of words (keys) to the posting list, we first erase some stop words so that they will not be searched. Pseudo code is like:

```
// now key is the list of keys
for i from 0 to key.size-1
    if key[i] is stop word
        remove key[i] from key
```

Then, we sort the rest of keys in ascending order by frequency, for the lower frequency means higher importance and we should get the several most important keys in the thresholding step. Since the number of keys won't be much big, we use selection sort, which has smaller constant and will be simpler and not slower.

After sorting, we calc the number of keys we should ignore and erase them from the list. Pseudo code:

```
// n is the number of keys now
reserve_num = floor(n * threshold) + 1; // +1: guarantee that there must be one word to be search
cut_num = n - cut;
erase the last cut_num keys from key
```

Now entering the most important part of Query: search words. The basic algorithm for get the list of documents that including all of the keys is: for each document, we set a counter initially 0. For each key, traverse its posting list for the ID of documents, plus these documents' counter by 1. After dealing with every key, documents that have their counters equal to the number of keys are the result.

To improve the efficiency and save the space, we didn't create counter for every document, only for the first key's document list, which has the lowest frequency. Because the final documents must include all keys, the list above must contain all of the results, while has a small number of documents to allocate space for counters. Pseudo code:

```
map<int, int> counter; // <ID, Times>
counter.insert(<docId,1> that docId in the posting list of key[0])
for i from 1 to key.size-1
    for each docID in the posting list of key[i]
        counter[docID]++
```

Chapter 3: Testing Results

3.1 Result of Index

After indexing, there are **112 stop words** being set (in the brackets are their frequency):

the(29147), and(26727), i(22616), to(19894), of(17980), a(14865), you(13883), my(12640), that(11673), in(11649), is(9444), not(8653), with(8062), for(7870), it(7866), me(7786), be(7777), his(7318), he(7048), your(7019), this(6715), but(6464), have(6164), as(5885), thou(5665), him(5350), so(5224), will(5155), what(4849), her(4245), thi(4123), all(4072), do(3979), by(3948), no(3851), lord(3748), we(3678), shall(3646), if(3597), are(3512), king(3287), come(3277), on(3276), thee(3258), our(3204), good(3005), sir(2929), now(2866), from(2794), love(2767), she(2701), they(2666), at(2577), which(2499), or(2491), let(2409), would(2383), more(2378), was(2373), here(2341), their(2326), then(2248), there(2220), how(2207), well(2189), make(2183), am(2179), o(2150), when(2136), enter(2088), them(2058), know(2046), say(2029), man(2021), hath(1992), like(1952), one(1940), than(1927), an(1852), upon(1811), go(1786), did(1715), may(1689), us(1663), yet(1656), were(1653), should(1627), see(1614), give(1574), must(1543), had(1514), whi(1472), such(1465), where(1419), out(1406), speak(1393), who(1384), take(1371), some(1367), these(1343), time(1328), first(1326), can(1285), too(1274), heart(1255), eye(1251), mine(1232), look(1213), most(1210), god(1205), hand(1200), think(1190).

Run the program for 3 times, the total time for indexing 42 documents are: 2.86s, 2.852s, 2.837s. (may differ among machines)

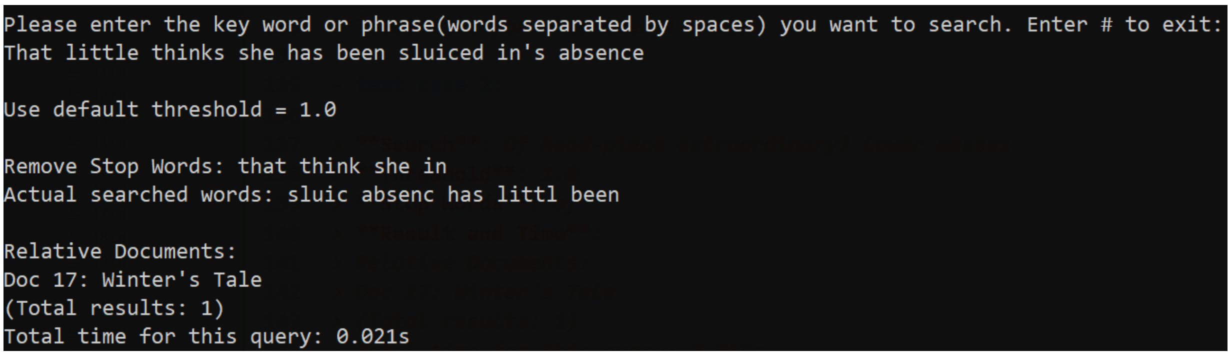
Total words number is **908548**, total distinct nonstop words number is **17664** (if we consider stop word it's 17776).

3.2 Result of Query

3.2.1 Test Correctness

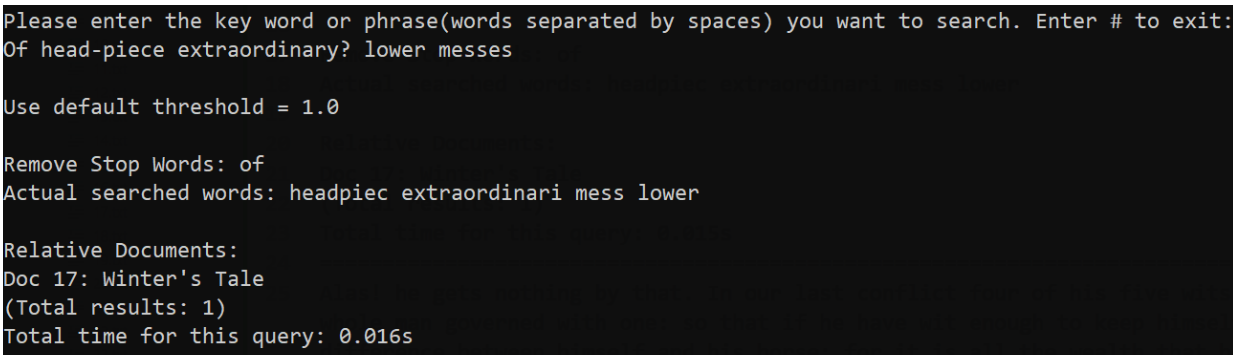
Take document17 as example, we search a line and check if the search engine returns the right result:

test case 1:



```
Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
That little thinks she has been sluiced in's absence

Use default threshold = 1.0

Remove Stop Words: that think she in
Actual searched words: sluic absenc has littl been

Relative Documents:
Doc 17: Winter's Tale
(Total results: 1)
Total time for this query: 0.021s
```

test case 2:



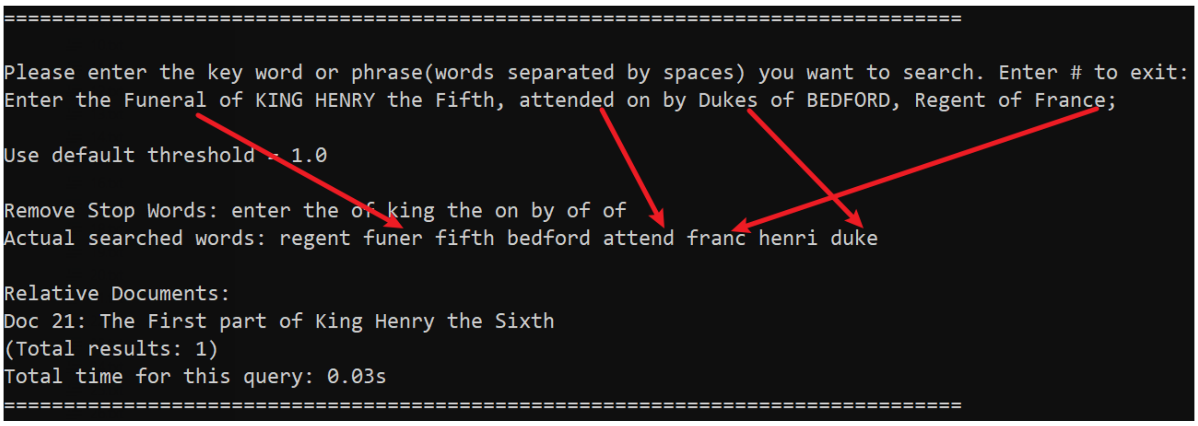
```
Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
Of head-piece extraordinary? lower messes

Use default threshold = 1.0

Remove Stop Words: of
Actual searched words: headpiec extraordinari mess lower

Relative Documents:
Doc 17: Winter's Tale
(Total results: 1)
Total time for this query: 0.016s
```

3.2.2 Test Stemming Words

The stemming words algorithm remove the suffix and extract the stem of each words to be searched. When process the word, we also transform all characters into lower cases, and remove punctuations.

test case 1:

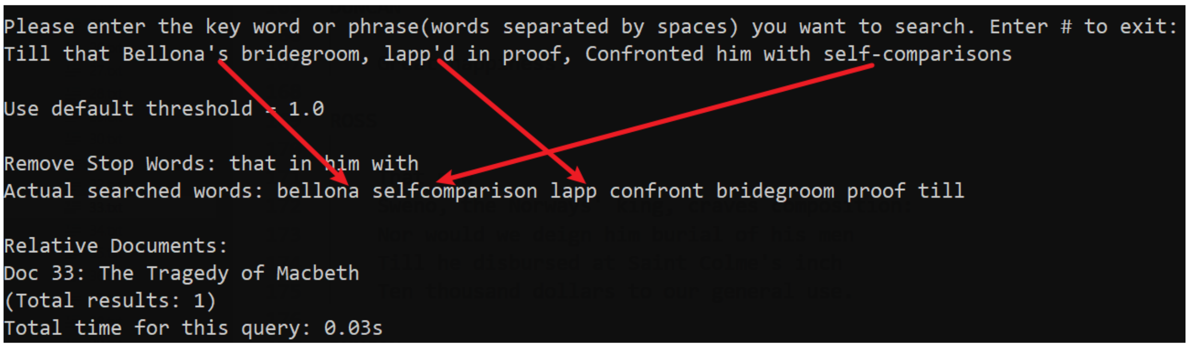
The red arrows shows the original and stemmed words. There are other words being transformed from upper case to lower case such as "BEDFORD" to "bedford", and punctuations removed such as "France;" to "franc".



```
================================================================================

Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
Enter the Funeral of KING HENRY the Fifth, attended on by Dukes of BEDFORD, Regent of France;

Use default threshold - 1.0

Remove Stop Words: enter the of king the on by of of
Actual searched words: regent funer fifth bedford attend franc henri duke

Relative Documents:
Doc 21: The First part of King Henry the Sixth
(Total results: 1)
Total time for this query: 0.03s
================================================================================
```

test case 2:

This case shows the process for " 's ", " 'd ", and " - " inside the word.



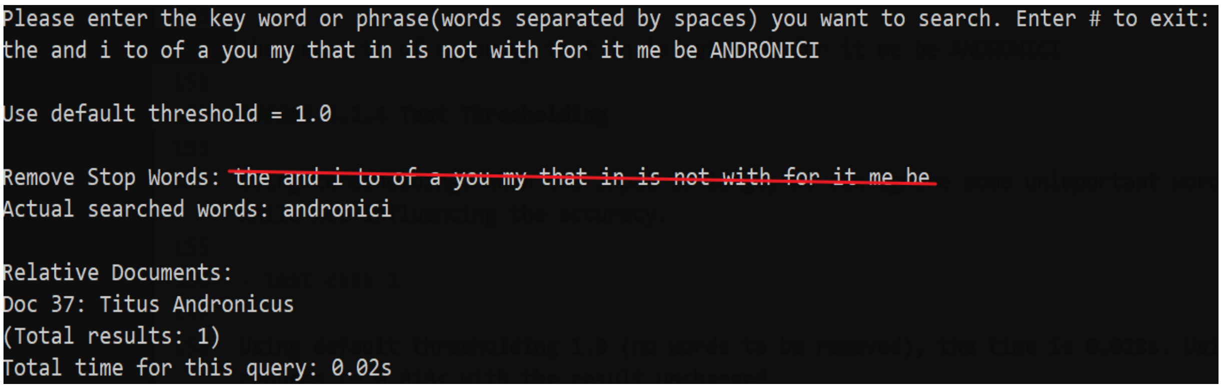
```
Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
Till that Bellona's bridegroom, lapp'd in proof, Confronted him with self-comparisons

Use default threshold - 1.0

Remove Stop Words: that in him with
Actual searched words: bellona selfcomparison lapp confront bridegroom proof till

Relative Documents:
Doc 33: The Tragedy of Macbeth
(Total results: 1)
Total time for this query: 0.03s
```

(Note: You don't need to specify query threshold. It's just for testing. Threshold is set to 1.0(search all words) by default, you can modify it in PostingList.cpp)

3.2.3 Test Stop Words

Ignore stop words will reduce the words to be searched when query, while do little influence to the accuracy. the follow case shows that we ignore stop words from query.

test case:



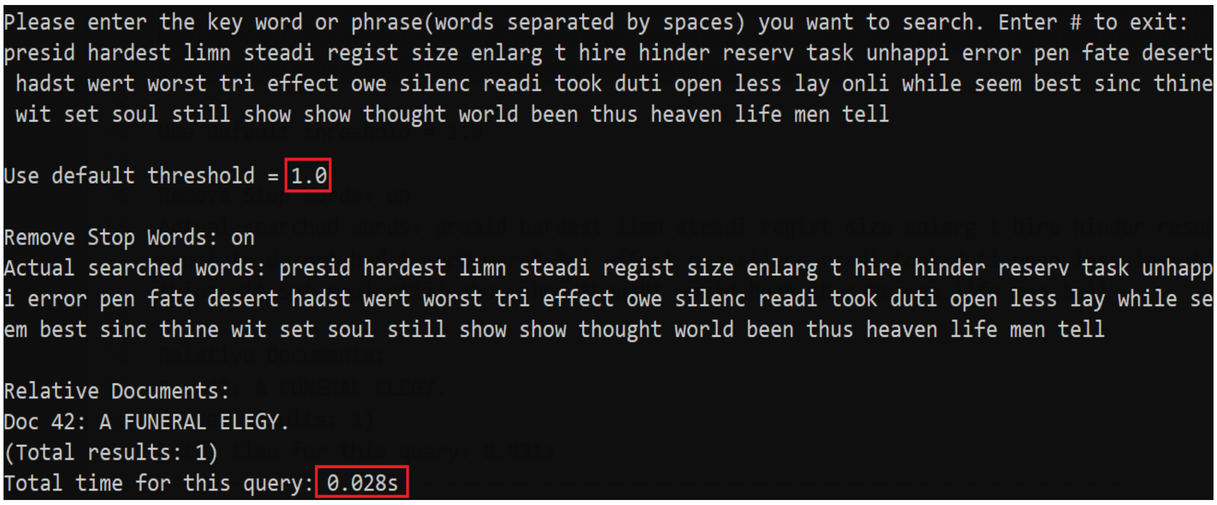
```
Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
the and i to of a you my that in is not with for it me be ANDRONICI

Use default threshold = 1.0

Remove Stop Words: ~~the and i to of a you my that in is not with for it me be~~
Actual searched words: andronici

Relative Documents:
Doc 37: Titus Andronicus
(Total results: 1)
Total time for this query: 0.02s
```

3.2.4 Test Thresholding

Using thresholding, when the input is large, we can ignore some unimportant word to improve the search speed while not influencing the accuracy.

test case

Using default thresholding 1.0 (no words to be removed), the time is 0.028s. Using thresholding 0.1, the time reduces to 0.013s with the result unchanged.



```
Please enter the key word or phrase(words separated by spaces) you want to search. Enter # to exit:
presid hardest limn steadi regist size enlarg t hire hinder reserv task unhappi error pen fate desert
 hadst wert worst tri effect owe silenc readi took duti open less lay onli while seem best sinc thine
 wit set soul still show show thought world been thus heaven life men tell

Use default threshold = 1.0

Remove Stop Words: on
Actual searched words: presid hardest limn steadi regist size enlarg t hire hinder reserv task unhapp
i error pen fate desert hadst wert worst tri effect owe silenc readi took duti open less lay while se
em best sinc thine wit set soul still show show thought world been thus heaven life men tell

Relative Documents:
Doc 42: A FUNERAL ELEGY.
(Total results: 1)
Total time for this query: 0.028s
```

Chapter 4: Analysis and Comments

4.1 Complexity Analysis

The findID function costs the most of time. While the posting list has N records (meaning that there are N nodes storing <term, ID> in the AVL tree), it will take O(logN) to access a term.

From the global angle, if we have M words in all documents to read and there are N distinct words in total, it will take us O(MlogN) to index all documents, while searching for a word still takes O(logN).

Of course operations on the record take time as well, but it's usually small compared to index and access term.

4.2 Comparing With Other Methods

4.2.1 Linear Search (vector in C++)

Before using AVL Tree to support the search of words, we tried linear search by vector first. At this time, stemming words were not used, and the index time was about 80s. Then we replaced vector with map, the run time reduced to about 0.9s. Finally, we used the AVL Tree implemented by our group leader, the time finally reduced to about 0.7s.

4.2.2 Hash

We don't choose hash to implement the search of words because the current index and query time is good enough for these documents, and hash requires dynamic hashing, which is really complex to implement.

Why it requires dynamic hashing? Because the number of distinct words is unknown before the index is completed. If we use fixed size of hash table, it may perform bad or even out of space if the size is too small, or it may take too much space if the size is too large.

4.3 Current Drawbacks and Extensible function

4.3.1 Punctuation In-between

Normally, the process of word convert words with a '-' in-between to a consecutive word without '-'. However, we don't judge whether the linking punctuation is a single '-'. If the space after a ',' or '.', the two words will be converted to a whole word.

For example, the two words "father" and "O" are considered a single word fathero: To solve this problem, the special case should be judged, or the stemming words algorithm should be improved more completely.

4.3.2 Consecutive Search

The query now is search each word and find documents that include all of the words. However, the simple treatment can't get a consecutive retrieval, which means that the words are treated as an entirety, the order and the consistency are considered.

To implement consecutive retrieval, the position of words in the posting list should be considered. The problem to be solved are:

1. How to judge two words are next to each other, since the position is on the basis of character rather than words.

2. How to deal with stop words, since they probably link other words.

4.3.3 Threshold for Documents

Though not required, there is a need for document threshold. We can sorted documents by weight (should be defined and calculated first), and only search some top documents. If the number of documents are larger, the threshold is meaningful.

What's more, we can use different document threshold for different search mode. For example, if users want the Recall higher and don't mind taking more time, we use high threshold to search more documents.

Appendix: Source Code:

(codes for AVL tree not included)

```
#include<iostream>
#include<fstream>
#include<string>
#include<algorithm>
#include<time.h>
#include"PostingList.cpp"
#include"StemLib/porter2_stemmer.cpp"
using namespace std;
#define DOCNUM 42//total number of documents
#define CATEGORY_NUM 4//number of different categories(not including
"all")

PostingList myPostingList;//the posting list
string docTitles[DOCNUM];//titles(the first line for a document)
string docCategory[DOCNUM];//categories(5 categories: all, comedy,
history, tragedy, poetry)
string categories[CATEGORY_NUM]=
{
    "comedy",
    "history",
    "tragedy",
    "poetry"
};

//function declarations
string dec2_to_string(int n);//turn an integer between 0 and 99 to two
character string (such as 1->"01")
```

```
bool isLowerCase(char c);//determine if c is a lowwer case English character
bool isWhite(char c);//detect white
bool isCategory(string cat);//determine if c is a name of existed catogery

void RemoveTokens(string &word);//remove tokens at the front and end of the word, remove ' in it.
void TakeStem(string &word);//restore the stem of input word(refer to codes in the Internet)
string ProcessWord(string &word);//romove tokens, case transfer, stem algorithm.(allowed to refer to the Internet)
vector<string> ProcessPhrase(string phrase);//turn a phrase represented by a string with space into individual word strings

void GenerateDocInfo();//generate document information: title and category
void ReadDoc(int docID);//read a certain document word by word, add into posting list after processing them
void Index();//create inverted file index
void DisplayInterface();//display interface
void Query();//do query



//function implementations
string dec2_to_string(int n)
{
    string ret="00";
    if(n<0||n>=100)
    {
        cout<<"Error:Out of range"<<endl;
    }
    else if(n<10)
    {
        ret[1]=n+'0';
    }
    else
    {
        ret[0]=n/10+'0';
        ret[1]=n%10+'0';
    }
    return ret;
}
bool isLowerCase(char c)
```

```
{
    return c>='a'&&c<='z';
}
bool isWhite(char c)
{
    return c == ' ' || c == '\t' || c == '\n';
}
bool isCategory(string cat)
{
    string temp;
    temp="all";
    if(cat==ProcessWord(temp))
        return true;
    for(int i=0;i<CATEGORY_NUM;i++)
    {
        temp=categories[i];
        if(cat==ProcessWord(temp))
            return true;
    }
    return false;
}

void RemoveTokens(string &word)
{
    int l=word.length(),pos;
    //delete 'xxx....
    while((pos=word.find('\''))!=-1)
    {
        l=word.length();
        if(pos==1&&word[0]=='o')
            word.erase(0,2);
        else
            word.erase(pos,l-pos);
    }

    //delete token in the front
    while(word.length()>=1 && !isLowerCase(word[0]))
    {
        word.erase(0,1);
    }
    //delete token at the end of a sentence
    while(word.length()>=1 && !isLowerCase(word[word.length()-1]))
    {
        word.erase(word.length()-1,1);
    }

}
void TakeStem(string &word)
{
    Porter2Stemmer::trim(word);
    Porter2Stemmer::stem(word);
}
string ProcessWord(string &word)
{
    transform(word.begin(),word.end(),word.begin(),::tolower);//transfor
m into lower cases
    RemoveTokens(word);//delete tokens(not including tokens inserted
into the word)
    TakeStem(word);//stem algorithm
    return word;
}
vector<string> ProcessPhrase(string phrase)
{
    vector<string> ret;
    string word;
    int pos, wordStart;

    pos = 0;
    while (pos < phrase.length()) {
        while (pos < phrase.length() && isWhite(phrase[pos])) pos++;
        wordStart = pos;

        while (pos < phrase.length() && !isWhite(phrase[pos])) pos++;

        if (wordStart < pos)
        {
            string word=phrase.substr(wordStart, pos-wordStart);
            ProcessWord(word);
            if(word.length()!=0)
                ret.push_back(word);
        }
    }
    return ret;
}

void GenerateDocInfo()
{
    for(int i=1;i<DOCNUM;i++)//category info
```

```cpp
    {
        if(1<=i&&i<=17)
            docCategory[i-1]=categories[0];
        else if(18<=i&&i<=27)
            docCategory[i-1]=categories[1];
        else if(28<=i&&i<=37)
            docCategory[i-1]=categories[2];
        else if(38<=i&&i<=42)
            docCategory[i-1]=categories[3];
        ProcessWord(docCategory[i-1]);
    }
    for(int i=1;i<=DOCNUM;i++)//title info
    {
        string docName="Documents/"+dec2_to_string(i)+".txt";
        ifstream ifs(docName);
        if(!ifs.is_open())
        {
            cout<<"Error:Can not read document "<<i<<endl;
            return;
        }
        else
        {
            getline(ifs,docTitles[i-1]);
        }
        ifs.close();
    }
}
void ReadDoc(int docID)
{
    string docName="Documents/"+dec2_to_string(docID)+".txt";
    ifstream ifs(docName);
    if(!ifs.is_open())
    {
        cout<<"Error:Can not read document "<<docID<<endl;
        return;
    }
    string word;
    int pos=0;//position of a word

    while(ifs>>word)
    {
        pos++;
        ProcessWord(word);
        if(word.length()!=0)
        {
            myPostingList.add(word,docID,pos);
        }
    }
    ifs.close();
}
void Index()
{
    double start=clock();
    for(int i=1;i<=DOCNUM;i++)
    {
        cout<<"\n-------------------Indexing Document "<<i<<"-----------
--------"<<endl;
        ReadDoc(i);
    }
    myPostingList.MarkStopWords();//mark stop words in the list. not 
searching them
    cout<<"Total time for indexing "<<DOCNUM<<" documents: "<<(clock()-
start)/1000<<"s"<<endl;
    cout<<"Total words number: "<<myPostingList.getAllWordNum()<<endl;
    cout<<"Total distinct nonstop words: 
"<<myPostingList.getDistinctNonstopWordNum()<<endl;
}
void DisplayInterface()
{
    cout<<"\n--------------------------------------Shakole---------------
----------------------"<<endl;
    for(int i=0;i<2;i++)
        cout<<"+                                                            
                   +"<<endl;
    cout<<"+  Welcome to use Shakole search 
engine                                         +"<<endl;
    cout<<"+  You can add \"=category\" at the end of the searched key 
to query by category  +"<<endl;
    cout<<"+  All Categories: comedy, history, tragedy, poetry.(\"all\" 
by default)          +"<<endl;
    for(int i=0;i<2;i++)
        cout<<"+                                                            
                   +"<<endl;
}
void Query()
{
    while(true)
    {
        string key;
        cout<<"\nPlease enter the key word or phrase(words separated by
spaces) you want to search. Enter # to exit:"<<endl;
        getline(cin,key);
        if(key=="#")
        {
            cout<<"Thanks for using Shakole!"<<endl;
            return;
        }
        else//search
        {
            vector<string> words;
            string category="all";
            int pos=key.rfind('=');
            if(pos!=-1)
            {
                category=key.substr(pos,key.length()-pos);
                ProcessWord(category);
                if(!isCategory(category))
                {
                    cout<<"No such category!"<<endl;
                    continue;
                }
                key=key.substr(0,pos);
            }

            words=ProcessPhrase(key);//input word will be processed as
well
            vector<int> result_docs;
            result_docs=myPostingList.search(words);//search in the
posting list!

            cout<<"Relative Documents:"<<endl;
            int resultNum=0;
            for(vector<int>::iterator
it=result_docs.begin();it!=result_docs.end();it++)
            {
                if(category=="all"||docCategory[*it-1]==category)
                {
                    resultNum++;
                    cout<<"Doc "<<*it<<": "<<docTitles[*it-1]<<endl;
                }
            }
            if(resultNum==0)
```

```
                cout<<"Can't find any result! (maybe stop words)"<<endl;
            else
                cout<<"(Total results: "<<resultNum<<")"<<endl;
        }
    }
}


int main()
{
    GenerateDocInfo();//create the document title table(not necessary in fact) and their category.
    Index();//create the inverted file index for all documents
    DisplayInterface();//display the interface
    Query();//do query

    return 0;
}
```

```
//Implemention of a posting list, maintaining an AVL Tree in it to access terms

/*
IMPORTANT!!!!!!!
Posting list structre:(each record is a row for a certain term)
record[0]----------------------------
record[1]----------------------------
....

record[size-1]-----------------------

For each record:
{
  (int)termID | (string)term | (int)frequency(in all documents) | (bool)isStop
| < docID1, times(frequency in this file, at least one), <pos[0],pos[1],....pos[pos.size()-1]> >
  < docID2, times, <pos[0],pos[1],...pos[pos.size()-1]> >
    .....

  < docIDn, times, <pos[0],pos[1],...............pos[pos.size()-1]> >
}
```

```
Words with occurence rate higher than a certain value will be marked as
stop words

We store <term, termID> in an AVL Tree to access the termID by the
string itself.
*/

#pragma once
#include<iostream>
#include<fstream>
#include<string>
#include<vector>
#include<time.h>
#include<map>
#include"myAVLTree.cpp"

#define DOCNUM 42
using namespace std;

//after sorted in ascending order of frequency, only this rate of input
words will be searched
const double global_threshold_query=1.0;
//if a word occurence rate is more than this value, it's marked as a
stop word, not participate in serching
const double global_stop_word_rate=0.0013;

struct _doc_pos
{
    int docID;//document ID
    int times;//times of a certain word that occurs in this document
    vector<int> pos;//occurence position
};

typedef struct _doc_pos doc_pos;
struct _record
{
    int ID;//term ID, also the row number of the record
    string term;//(word)
    int frequency;//occurence times of term
    bool isStop;//if the term is a  stop word
    vector<doc_pos> occurence;//docID and the position where term
appears at this document
};
typedef struct _record record;
```

```cpp
class PostingList
{
private:
    int size;//number of records(also number of all distinct words)
    int wordNum;//number of all words(not distinct)
    int stopWordNum;//number of stop words
    vector<record> records;//records vector
    AVLTree wordTree;//this is an AVL tree that stores words and their
ID.

public:
    PostingList();
    int getDistinctNonstopWordNum();//return the number of distinct
nonstop words
    int getAllWordNum();//return the number of all words
    int findID(string term);//find the ID(or row number in posting list)
of a term
    void add(string word,int docID,int pos);//add a word
    void MarkStopWords();//mark stop words in the list, do not search
them
    vector<int> search(vector<string>key);//search a word or phrase
};

PostingList::PostingList()
{
    this->size=0;
    this->wordNum=0;
    this->stopWordNum=0;
}

int PostingList::getDistinctNonstopWordNum()//return the number of
distinct nonstop word
{
    return this->size-this->stopWordNum;
}

int PostingList::getAllWordNum()
{
    return this->wordNum;
}

int PostingList::findID(string term)//return the ID of a term. return -
1 if not exists
```

```
{
    int retID = wordTree.Find(term);
    return retID;
}

void PostingList::add(string word,int docID,int pos)
{
    this->wordNum++;
    int termID=findID(word);
    if(termID!=-1)//add into existed record
    {
        this->records[termID].frequency++;
        for(vector<doc_pos>::iterator 
it=this->records[termID].occurence.begin();it!=this->records[termID].oc
curence.end();it++)
        {
            if(it->docID==docID)
            {
                it->pos.push_back(pos);
                it->times++;
                return;
            }
        }
        doc_pos newDocPos;
        newDocPos.docID=docID;
        newDocPos.pos.push_back(pos);
        newDocPos.times=1;
        this->records[termID].occurence.push_back(newDocPos);
    }
    else//create new record(a new term)
    {
        record newRec;
        newRec.ID=this->size;
        newRec.frequency=1;
        newRec.term=word;
        newRec.isStop=false;
        doc_pos newDocPos;
        newDocPos.docID=docID;
        newDocPos.pos.push_back(pos);
        newDocPos.times=1;
        newRec.occurence.push_back(newDocPos);
        this->records.push_back(newRec);
        this->size++;
        wordTree.Insert(make_pair(newRec.term,newRec.ID));
    }
    return;
}

void PostingList::MarkStopWords()
{
    //cout<<this->wordNum*global_stop_word_rate<<endl;
    for(vector<record>::iterator
it=this->records.begin();it!=this->records.end();it++)
    {
        if(it->frequency>this->wordNum*global_stop_word_rate)
        {
            //cout<<it->term<<": "<<it->frequency<<endl;
            it->isStop=true;
            this->stopWordNum++;
        }
    }
}

vector<int> PostingList::search(vector<string> key)//return the vector
of document IDs of searched key
{
    vector<int> ret_docs;
    //scan first and erase stop words (vector dynamic deletion is a
little complicated to implement)
    vector<string>::iterator it=key.begin();
    while(true)
    {
        if(it==key.end())
            break;
        int termID=findID(*it);
        if(termID==-1)
        {
            return ret_docs;
        }
        else if(this->records[termID].isStop==true)//do not search stop
words
        {
            key.erase(it);
            it=key.begin();
        }
        else
            it++;
```

```cpp
    }

    //sort the input word vector "key" in ascending order according to 
frequency
    int l=key.size();
    if(l==0)
        return ret_docs;

    vector<int> freq;//freq[i] is the frequency of key[i] in all 
documents
    for(int i=0;i<l;i++)
    {
        int termID=findID(key[i]);
        if(termID==-1)
        {
            return ret_docs;
        }
        else
        {
            freq.push_back(this->records[termID].frequency);
        }
    }
    for(int i=0;i<l;i++)
    {
        int min=freq[i],minIndex=i;
        for(int j=i+1;j<l;j++)
        {
            if(freq[j]<min)
            {
                min=freq[j];
                minIndex=j;
            }
        }
        //swap
        int temp=freq[minIndex];
        freq[minIndex]=freq[i];
        freq[i]=temp;
        string ttemp=key[minIndex];
        key[minIndex]=key[i];
        key[i]=ttemp;
    }

    //threshold for searching
    int cut=(int)(l*global_threshold_query)+1;
    int cut_num=l-cut;
    while((cut_num--)>0)
    {
        key.pop_back();
    }

    //search
    //the first element is possible docID(key[0] determines). the second 
is how many terms in key appear in the document,
        //if the second equals to key.size(), it means this document 
contain all searched words
    map<int, int> docID_time;
    int termID=findID(key[0]);
    //first narrow the range of possible documents using key[0] (the 
term with the lowest frequency)
    for(vector<doc_pos>::iterator 
it=records[termID].occurence.begin();it!=records[termID].occurence.end(
);it++)
    {
        docID_time.insert(make_pair(it->docID,1));
    }
    for(int i=1;i<key.size();i++)
    {
        termID=findID(key[i]);
        for(vector<doc_pos>::iterator 
it=records[termID].occurence.begin();it!=records[termID].occurence.end(
);it++)
        {
            docID_time.find(it->docID)->second++;
        }
    }

    //return results
    for(map<int,int>::iterator 
it=docID_time.begin();it!=docID_time.end();it++)
    {
        if(it->second==key.size())//it means doc[it->first] contain all 
terms of key
        {
            ret_docs.push_back(it->first);
        }
    }
    return ret_docs;
}
```

References

[1] smassung, porter2_stemmer, https://github.com/smassung/porter2_stemmer

Author List

Guan Jiarui

1. Process the documents.

2. Construct the basic frame of the engine, including design of the structure of the posting list, the pattern to index, and the algorithm to search terms.

3. Build an AVL tree to access term fast to get bonus.

4. Add a new function: search according to category.

5. Write chapter 2 in the report and do the integration.

6. Do the presentation.

Qi Yue

1. Find and test the code doing word stemming.

2. Write a really simple function: ProcessPhrase (seperating sentence of query into words)

3. Do the most of the test.

4. Initially build the structure of the report and write: Chap.2 (2.1, 2.4), Chap.3, Chap.4 (4.3, 4.4). Some of them are further modified by our group leader.

Zheng Ming

1. Participated in the compilation discussion of stop Word decision

2. Completed the preparation of the background in part 1 of the report

3. Participated in the preparation of part of complexity

Declaration

We hereby declare that all the work done in this project titled "Shakole Search Engine" is of our independent effort as a group.

Signatures

管嘉瑞

齐越

[illegible]