Advanced Data Structures and Algorithm Analysis Project-2

Roll Your Own Mini Search Engine

Author Names

Date: 2019-03-16

Chapter 1: Introduction

In this experiment, we have to segment the words through the inverted file index learned in the class, then enter the keywords and search the files.

The requirements of this experiment are as follows:

1. Count the words in the Shakespeare collection and find out the "noisy" words, and we need to set the threshold to judge.

2. Create our inverted index for Shakespeare's anthology and not the above-mentioned "noisy" words.

3. Write a query program for the inverted index table, and the query program can return the document number containing the word or phrase input by the user.

Chapter 2: Data Structure

In this project, we use the hash-table to reflect on every word. And our hash-table node recorded the page, last word, line and the next node.

```
1.  struct HashNode {
2.      string page;
3.      string formerword;
4.      int line;
5.      HashNode* Next;
6.  };
```

And we construct a class document, and it has lots of public and private variable, function and map, in order to solve the problem. And each intention of the variable and the function we have commented under them. And it is our fundamental data structure. Here is our major functions in Document

```
1.  class Document {
2.  public:
3.      void Files_getOrigin(void);
4.      void SearchWord(string name);
5.      string Query(void);;
6.      void Searchphrase(string name);
```

```
    void SearchKey(string name);
    void SearchSentence(string name);
private:
    int word_Camp(const string word);
    void Replace_annotation(string* content);
    void Document_LoadOrigin(const string path, const string name);
    vector<string>number_wordscamp;
    unordered_map<string, HashNode>HashTable;
    void HashInsert(string page, int line, struct HashNode* Header, string formerword);
};
```

Chapter 3 : Algorithm Specification

In this part we will introduce our main algorithm and how we deal with it.

We have already download all of the poems and articles of William Shakespeare, so we should read all the things by using the document.files_getorigin to get names and id of all the files.

In this function, which contains another important function—"Document::Document_LoadOrigin (const string path, const string name)", have already help us count the number of the stop word and the all other words.

```
void Document::Document_LoadOrigin(const string path, const string name)
{
    int word = 0;
    ifstream Origin_File("./"+path+"/" + name);
    while (getline(Origin_File, file_content))
    {
        split(split_string, file_content, is_any_of(" "));
        for (int i = 0; i < split_string.size(); i++)
        {
            if (!stop_word(split_string[i]))
                            HashInsert(name, word, &Poslocator->second, formerword);
            else
                stop_words++;
        }
```

```
    }
}
```

And we perform word segmentation, first pre-processing, that is, turn all punctuation into spaces, and change all uppercase and lowercase to lowercase. In the file and word creation process, we use the hash table insert to project the same words (except stop words) into the hash table, and then each word forms a linked list in the slot. We implemented a single word and multiple words and keywords. Our stop words are listed below which originates from the common word and the top 150 words with highest frequency.

```
vector<string> stopword_Camp = { "the","i","I","to","of","you","my","that","it","not",
    "were","was","he","helena","and","but","have","had","whice","must","what","how",
    "however","why","them","they","their","your","us","yet","base","but","fair","such",
    "if","issues","london","plebeians","unhappy","wellworthy","when","which","writ","Key"
    "had","love","so","him","her","with","wa","me","Her","|","are","if","end","which"
    "is","too","not","be","your","for","on","in","thee","an","am","sonnet","Sonnet"
};
```

Finally, we used unordered_map (the underlying implementation is a hash table) for searching and other operations. Due to the characteristics of the hash table, we can see that we have a long time in the setup, but the search operation is fast, almost O(K), where K Is the number of words to be found.

Chapter 4: Testing Results

The percentage represents the progress of loading the files, which could take a

while. The time cost may differ due to the hardware. After testing, three PCs has different loading time.

It is shown that the files are loaded successfully.

The test cases are as follows:

search word [key]	The name of files which word "key" appears and its position are output respectively. Files with few key word's appearance are omitted.
search word [boost]	If words that never appears are searched, Not found will be returned.
search phrase [blunt invention]	Search for a specific fixed phrase, the name of the file and the phrase's position in the file will be returned.
search key word [apple]	In key word mode, the most related files will be returned and sorted.
search key word: [invention apple account]	The pages that are most related are output. Our search engine only output at most ten most related files.
search key combination: [China Russia America]	The pages that are most related are output. But there are less than ten related files, then just output the related ones. Here are the results where "China", "Russia" and "America" are searched for respectively.
search key sentence old world could say this	Search for where the sentence exactly appears.
We have put the screenshot of the running outputs in the directory.	

Chapter 5: Analysis and Comments

The time complexity of creating the index is O(N). We use undirected map to create the index, which is realized based on hash table. Each hash node is a list. Each list represents the same English word. So there would be M lists if there were M distinct words.

For each English word, insert it to the head of the corresponding list. And the time complexity of find the correct list is O(1). Because there are N words in our data set, it takes O(1) to create the index.

The time complexity of searching is O(M). M is the number of distinct words. In the hash table in the undirected map, there are M distinct words. To find a specific word in the index, a traversal can do the job. And there are M lists representing M distinct words, as a result, it takes O(M) to search a word.

After testing, we understand that the threshold greatly affects the results of searching. If two key words merely appears together or one of them rarely appears, we would hardly get any research pages. But with thresholding, we can minimize the decrease of precision and recall rate brought by the noisy words. While thresholding, we don’t use a specific value as a threshold but achieve the effect of thresholding through certain procedural.

When searching with multiple keys, our search engine calculates how many times each key word appears, then divide the average times with it. For each file, the sum of the key words multiplies the number of them. Then output the at most ten files with the biggest values, if not zero. In this way, wo make sure that majority of the outcome pages contains the words of frequent appearance and have the words of low frequency as much as possible.

In brief, our search engine has its pros and cons and has many flaws, we still need time to perfect it. Boost and dirent are both really useful, so we strongly recommend that if you want to program with c++, please find an opportunity to give it a try.

References

[1] Jeremy Hylton, “The Complete Works of William Shakespeare”, http://shakespeare.mit.edu/