EC504

Project Report

Twitter Keyword Search

Team members:

Tao Zhang; Huize Hao; Junyi Tang

Github Link:

https://github.com/FaustineHui/EC504_Project

12-12-2021

I. Project Overview

Problem

The problem/goal for this project is to search the query words within all of the tweets in the input files by the user and output the top 10 similar tweets. Key deliverables include a command line user interface that lets inserting raw tweet files, and querying, the ability to retrieve relevant tweets to the query keywords, and the ability to display relevant tweets in correct importance order. A Graphical User Interface (GUI) is also needed for clearer illustration of the inputs and outputs.

Implementations

For this project, we try to figure out the similarity between tweets in our database and our users' input by distinguishing the same pattern they share. We set an array or vector as our database and use the getline command to input that data from the txt file which requires O(m) time. M is the number of tweets in datasets. After computing KMP algorithm, we need another m size array to store the frequency of each tweet. For the ranking part, we build up array based priority queue to use the Max heap algorithm and pop out the 10 most related tweets. Finally build up a GUI to input and display the final results.

Features

The first feature is command line user interface that lets inserting raw tweet files, and querying. In order to make the GUI interface relatively concise, we did not choose to add a file upload button to the UI interface. If you need to change the input database file, you only need to put the file in the project folder, and then find the corresponding location in the MyForm.h file to modify the file name. In this part, we use the iostream in C++ to complete the reading of the input file. And use vector to store tweets in the file.

The second feature is ability to retrieve relevant tweets to the query keywords. In this part, we have used two different methods to realize the precise search and fuzzy search of keywords respectively, which can be easily changed by comments in the code. For precise search, we use the str1.compare(str2) function in the C++ string library, so that we will search for the exact same word as the keyword. For fuzzy search, we use the KMP algorithm, so that we can search

for the same style of content as the keyword, such as searching for to, then potato will also provide a count for the frequency. Precise search is the main requirement of the project, and fuzzy search is the additional content we made ourselves.

The third feature is ability to display relevant tweets in correct importance order. In this part, we use a max heap. After counting the frequency for each tweets, we perform a heap adjustment and output the tweet content corresponding to the maximum ten frequencies in the heap.

II. Algorithm Explanation

Code & Time Complexity

KMP

```
int KMPsearch(string text, string pattern) {
    int n = text.size(); // read the length of pattern and text
    int m = pattern.size();
    int prefix[m]; // prefix table's length equal to length of the pattern

    computeprefix(pattern,m,prefix); // precompute the prefix table

    int i = 0;//start to do the pattern search
    int j = 0;
    int count = 0;//count the number of patterns found

    while (i<n){
        if (text[i]==pattern[j]){ // if a letter match, go to the next letter for both array
            i++;
            j++;
        }

        if (j==m){//if j exceed the last letter of the pattern, pattern found and go to the index
                  //referred by the prefix array
            count++;
            j = prefix[j-1];
        }

        else if (text[i]!=pattern[j] && i<n){
            if (j!=0) j = prefix[j-1]; // if not match, go to the index referred by the prefix array
            else i++; // if j=0; then i increments by 1
        }
    }
    return count;
}
```

Figure 1 KMP Code Snippet

```
void computeprefix (string pattern, int m, int*prefix){
    int len = 0; //length of the longest prefix suffix from the previous index

    prefix[0] = 0; //prefix array indexed at 0 is always set to 0

    int i = 1; //start to compute
    while (i<m){
        if (pattern[i] == pattern[len]){ // length of longest prefix suffix++
                                         //if the next letter matches the letter indexed at len+1
            len++;
            prefix[i]=len;
            i++;
        }
        else{
            if (len!=0) len = prefix[len-1]; // if not match, trace back len from the prefix array
            else {
                prefix[i]=0; // if not match then set it to 0 and continue working on the prefix array
                i++;
            }
        }
    }

}
```

Figure 2 Prefix Table Code Snippet

For searching algorithm in the project, we used Knuth–Morris–Pratt (KMP) to find the occurance of the words in query. The code uses the function computeprefix( ) to calculate the prefix suffix index table for each character in the pattern before traversing through the text (tweets). After the prefix table is made, it traverse through the text and counts the number of occurrences for each query word. The time complexity is O(n+m) where n is the total number of characters in tweets and m is the number of characters in query. As m is much smaller than n if n is large enough, the time complexity will approach O(n).

Input Processing

```
ifstream infile;
infile.open(argv[1]);
if(!infile){//error checking
cout << "Error opening file " <<endl;
return -1;
}
while (getline(infile,tweet)){ // read in the tweets
    if (tweet.empty()) continue;
    tweets.push_back(tweet);
    Ncount.push_back(0);
}
infile.close();

cout<<"Enter your query:"<<endl;
getline(cin,query);


for (int i=0;i<=query.size();i++){ //split the queryuntil pattern is null
    if (query[i] == ' ' || query[i] == '\0'){
    for (int j=0;j<Ncount.size();j++){
        Ncount[j] += KMPsearch(tweets[j],pattern);
}
    pattern = "";
}
    else pattern += query[i];
}
```

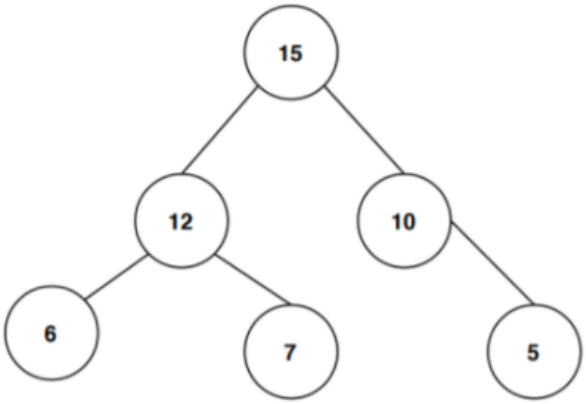
Figure 3 Input Processing Code Snippet

When the user compile the cpp files and type in the files with tweet in each line of the file, the program will first run an error-checking logic. After reading all the tweets, the program will ask the user to input the query. Then, it will split the query into words and run a KMP algorithm on each word. As a result, the complexity for KMP with this input logic will be O(k*n) where k is the number of words in the query and n is the total number of characters in tweets.

Ranking and Max Heap

Since we already get the frequency for every tweet with KMP algorithm, the key problem becomes how to organize the frequency and get the 10 most related tweets.

If we treat the whole question as outputting the 10 biggest numbers in an array, which makes it a simple sorting question. Since we do not need to sort the whole array, the max heap algorithm is one of the best options we get.



Algorithm 1: Max-Heapify Pseudocode

```
Data: B: input array; s: an index of the node
Result: Heap tree that obeys max-heap property
Procedure Max-Heapify(B, s)
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        largest = left;
    else
        largest = s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        largest = right;
    end
    if largest ≠ s then
        swap(B[s], B[largest]);
        Max-Heapify(B, largest);
    end
end
```

Fig 4. Sample max heap and pseudocode

Here are the sample data structure and pseudocode of building up a max heap. But we found a problem by simply implementing a max heap like the pseudocode. We can use max heap to sort and get the 10 biggest frequency but the indies will be messed up after sorting. In this case, we try to use the indices of the array as the reference to compare so that we can pop out the indices directly.

```
void heapify_down(int i)
{
    // get left and right child of node at index `i`
    int left = LEFT(i);
    int right = RIGHT(i);

    int largest = i;

    // compare `A[i]` with its left and right child
    // and find the largest value
    if (left < size() && original[A[left]] > original[A[i]]) {
        largest = left;
    }

    if (right < size() && original[A[right]] > original[A[largest]]) {
        largest = right;
    }

    // swap with a child having greater value and
    // call heapify-down on the child
    if (largest != i)
    {
        swap(A[i], A[largest]);
        heapify_down(largest);
    }
}

// Recursive heapify-up algorithm
void heapify_up(int i)
{
    // check if the node at index `i` and its parent violate the heap property
    if (i && original[A[PARENT(i)]] < original[A[i]])
    {
        // swap the two if heap property is violated
        swap(A[i], A[PARENT(i)]);

        // call heapify-up on the parent
        heapify_up(PARENT(i));
    }
}
```

Fig 5. Code of Max heap

Fig 5 shows the code we implement the max heap. The time complexity of the heapify process is O(n) and the pop out process is O(logn). Total time complexity is O(n+logn).

GUI

We use CLR package in C++ and we run it in Visual Studio 2019 on the Windows system. We first add the interactive buttons or text input boxes we want to the design interface, and the CLR will automatically generate related functions in the corresponding header files. Then we add the main code to the function of the corresponding button, and simply change the code structure, you can successfully run and generate the GUI interface.

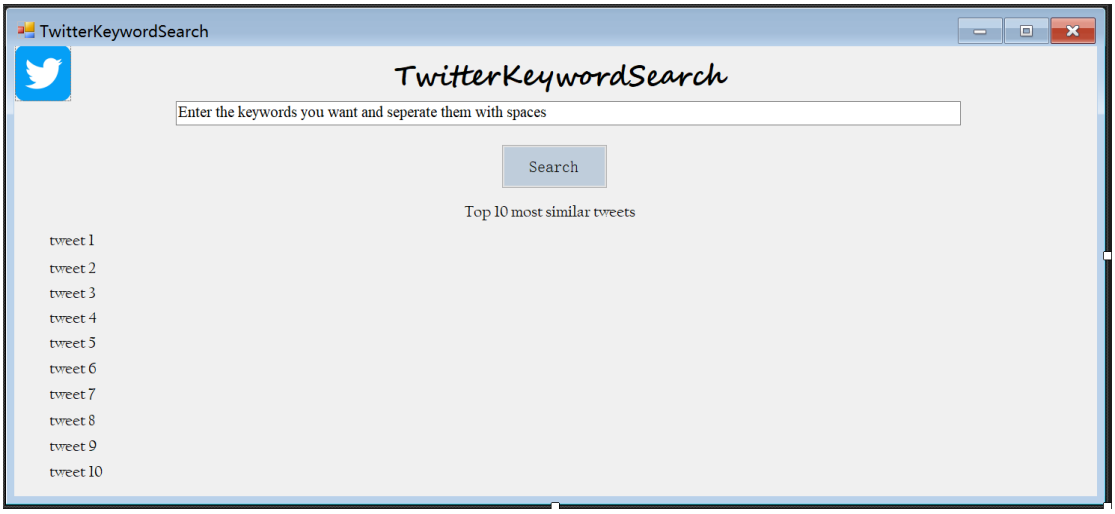


Figure 6: GUI interface

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    tweets.clear();
    Ncount.clear();
    ifstream infile;
    infile.open("abcnews.txt");
    while (getline(infile, tweet)) { // read in the tweets
        if (tweet.empty()) {
            continue;
        }
        tweets.push_back(tweet);
        Ncount.push_back(0);
    }
    infile.close();
    String^ keyword = textBox1->Text;
    query = ConvertToString(keyword);
    for (int i = 0; i <= query.size(); i++) { //split the queryuntil pattern is null
        if (query[i] == ' ' || query[i] == '\0') {
            for (int j = 0; j < Ncount.size(); j++) {
                Ncount[j] += KMPsearch(tweets[j], pattern);
            }
            pattern = "";
        }
        else pattern += query[i];
    }

    PriorityQueue Mheap; // Max heap
    Mheap.initize(Ncount);

    int k = 0;
    string out;
    if (k != 10) {
        if (!Mheap.isEmpty()) {
            out = tweets[Mheap.pop()];
            String^ temp = gcnew String(out.c_str());
            label4->Text = temp;
            k++;
```

Figure 7 The function of click the button

```
int KMPsearch(string text, string pattern) {
    int n = text.size(); // read the length of pattern and text
    int m = pattern.size();
    string textpart = "";
    int count = 0;
    for (int i = 0; i < n ; i++) { //split the queryuntil pattern is null
        if (text[i] == ' ') {
            if ((textpart.compare(pattern)) == 0) {
                count++;
            }
            textpart = "";
        }
        else textpart += text[i];
    }
    return count;
}
```

Figure 8: Precise search

In figure 6, I did not change the name of the function to let it easy to change it from precise search to fuzzy search. The KMP algorithm we used for fuzzy search has been shown above.

Compile Instructions

C++ Compiler

To run the program with terminal, go to the directory “main_code”. Then, type in the commend “g++ Twitter.cpp” and then type “./a.out (the file with tweets you want to search)”. Follow the instruction to enter the query and the top 10 similar tweets will be shown as the output.

GUI

We recommend running this program under windows system, of course, other systems can also be used, unless we use a different C++ version. Please download the EC504_project_final folder in Github, use Visual Studio to run the EC504_project.sln file, click the local Windows debugger button at the top of the interface, wait for the UI interface to pop up, enter the keywords you want to query, and wait for a while, the 10 with the highest match Tweets will be displayed on the interface. If you want to change the database file, please put the file into the EC504_project_final folder, in the MyForm.h file, use Ctrl+F to query infile, and change the file name in infile.open() to the new file name.

III. Work breakdown

Tao Zhang: KMP, Input logic, Organize and split the project work

Zehao Hui: GUI, Precise search part, Integrate everyone's code

Junyi Tang: Maxheap, Ranking

IV. Reference

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/