EC504 Final Report-Twitter Keyword Searching

Leyang Yu, Zezhong Wang, Shidong Sun

1. Introduction

The project is to write a keyword search engine that reads a text file as twitter database (each line as a separate tweet) and provides a querying of tweets based on the keywords provided by the user. The program will provide 10 tweets in descending order based on keyword relevance. Also, in this project, it is required that some very efficient data structures and algorithms is to be chosen in order to make the program faster and to be able to handle all keywords correctly. The whole program is written in C++. This report contains the following parts: data structures and speed comparisons between them; complexity analysis for all data structures; GUI design and conclusion.

2. Data Structure

2.1 Data Structure Introduction

There are 4 data structures tested in this project. Some basic tools, including map, unordered_map and vector, are provided by C++ STL Library. (See Figure 1--4)

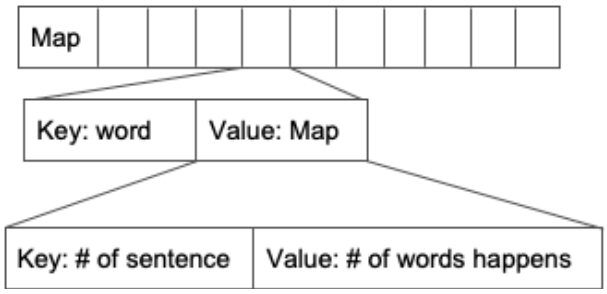

Figure 1. Map_Map Structure

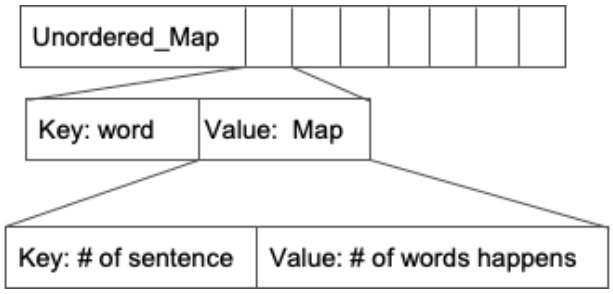

Figure 2.UOMap_Map Structure

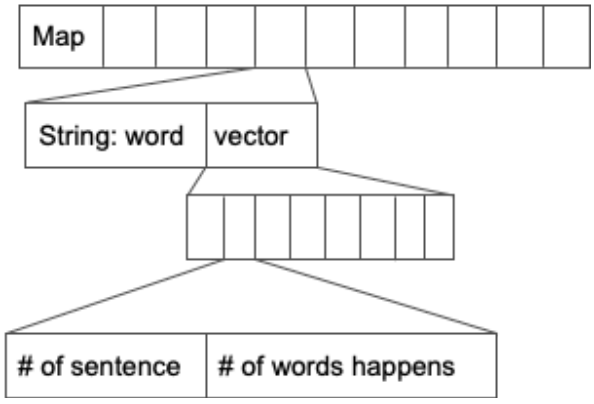

Figure 3. Map_Vector Structure

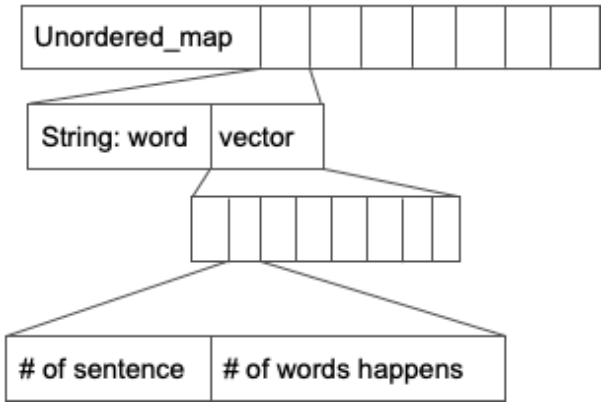

Figure 4. UOMap_Vector Structure

There is also a vector called ‘Vector A’ used to store all tweets which is not included in the figures because no operation is done to it except storing.
The 4 structures look similar but differ significantly in speed(will discuss later). The program builds the database as the following flow chart(See Figure 5, take the UOMap_Vector Structure as an example):

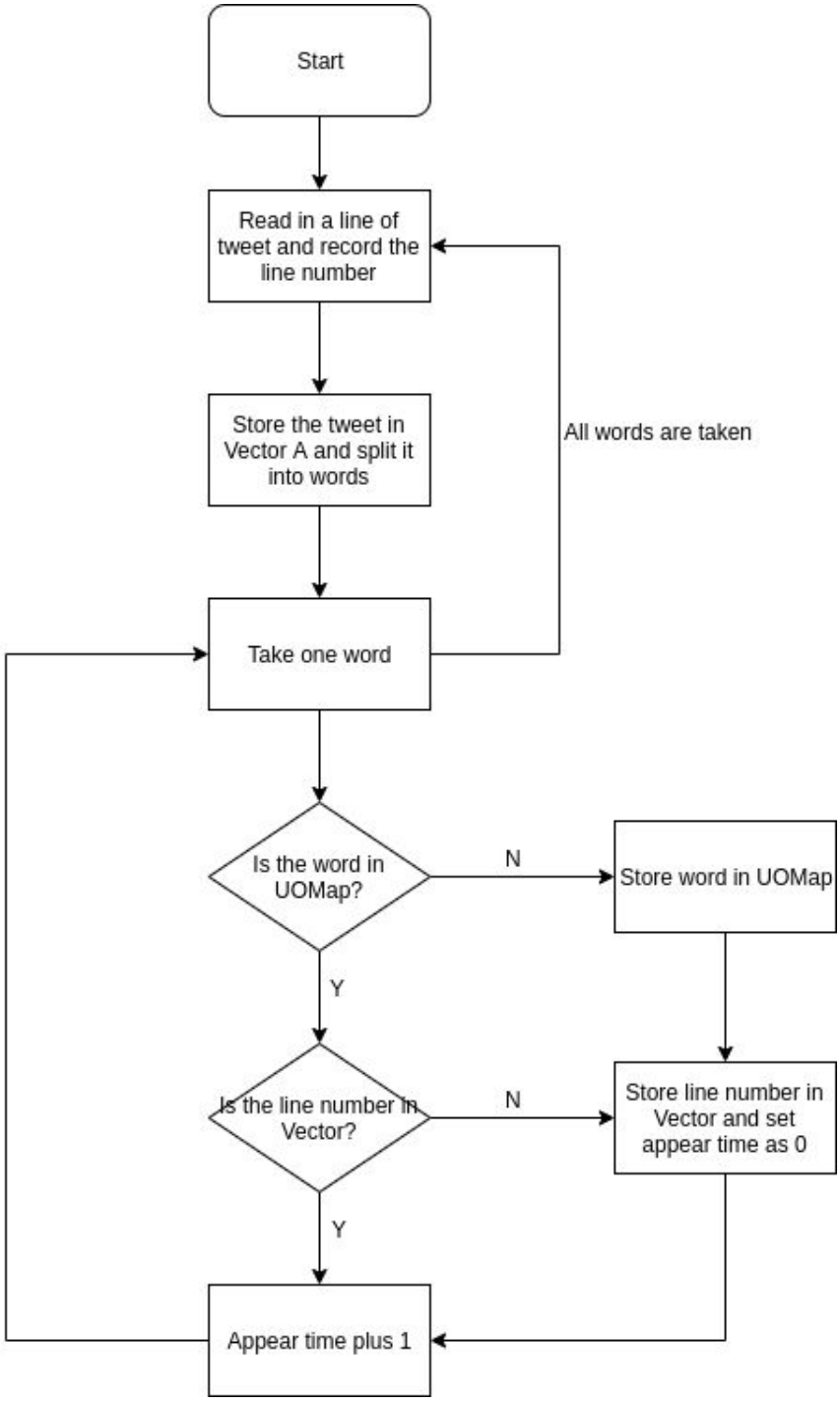

Figure 5. Flow chart of building database

Also, there is another data structure that is used for storing and sorting the results. It is a Heap consists of a Vector of Structures, as the graph below (See Figure 6):

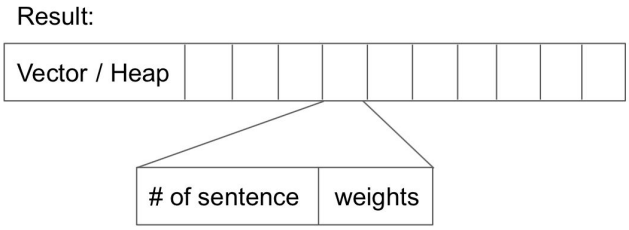

Figure 6. Result Data Structure

In searching for keywords, once we find a keyword, we loop through the inner data structure which contains the number of the sentence and how much time it occured in this sentence. Each time we know those data, we add the time it occured to the weights of this sentence. After finished finding, the vector will go through a heap sort based on the weights. After that, the 10 sentences which contain largest weights is the result.

2.2 Data Structure Comparison

The four data structures occupy the same memory space, but differ in time significantly. The following 2 figures illustrate the speed of building database and searching. (See figure 7 and 8)

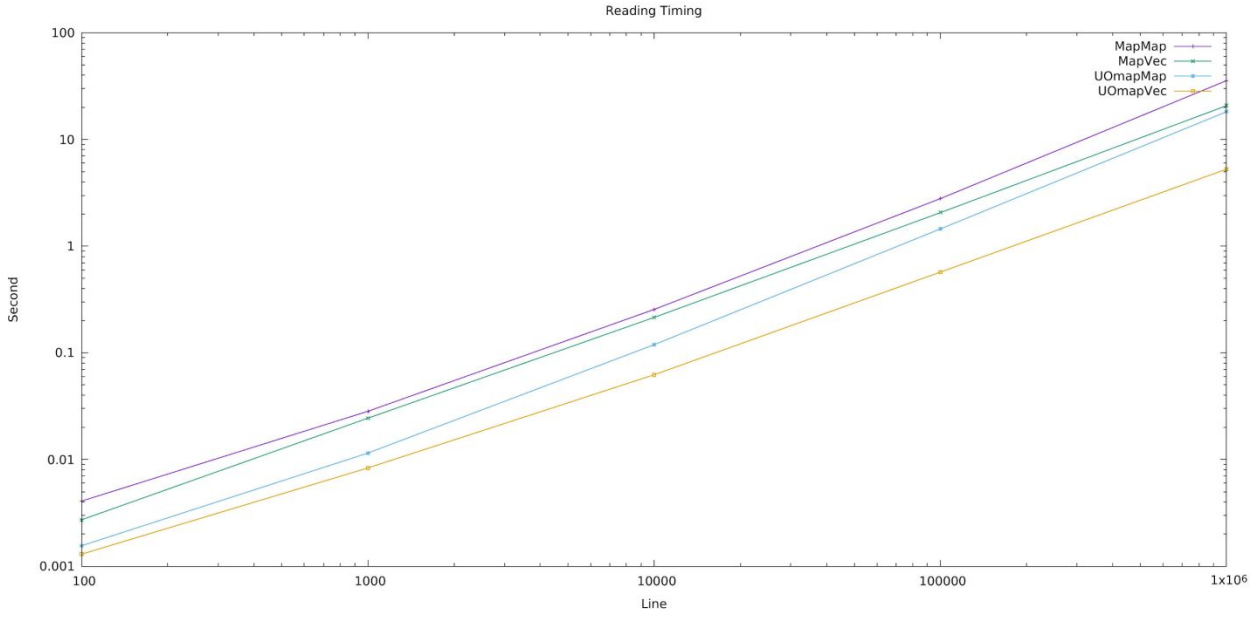

Figure 7. The speed of 4 data structures of reading file and building database 'Time vs File Size'

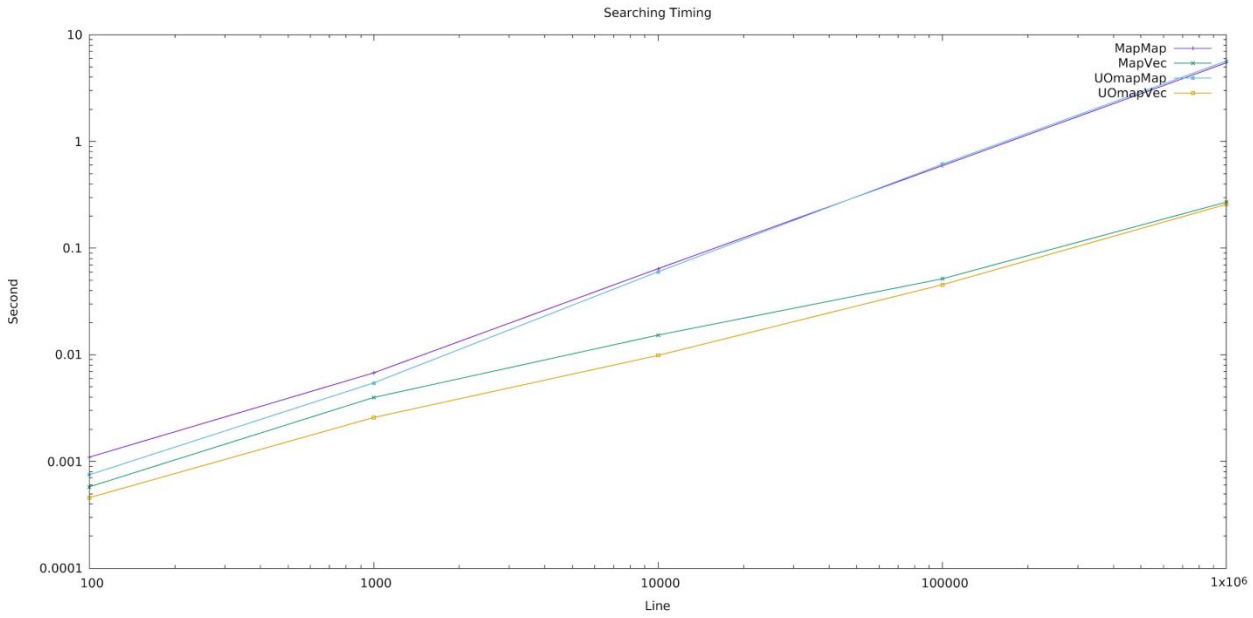

Figure 8. The speed of 4 data structures of searching for keywords 'Time vs File Size'

	Map<Map>	UNO_Map <Map>	Map <Vector>	UNO_Map <Vector>
Set up database(s)	38.171726	17.891907	21.352259	5.508170
Search(s)	6.133707	5.758840	0.301352	0.270565
Sort(s)	0.282422	0.260048	0.272540	0.267386

Table 1. The speed of 4 data structures running with 1M lines of inputs

2.3 Comparison Analysis

By analyzing the figures, we can have the following conclusions:

1. When building database, unordered_map has a significant advantage over map. This is because map has O(logN) complexity while unordered_map has O(1) average complexity.
2. Vector is overall faster than map when building the database and searching. Consider that searching through map and vector both takes O(n) time, the advantage of vector should come from its continuous storage in cache, thus can hardly have a cache miss when getting data. Map, however, has a much higher possibility of cache miss.
3. When searching, unordered_map's advantage over map is not clear because of higher possibility of collisions. Although unordered_map has an average complexity of O(1), its worst case complexity can be O(n) which makes it slower than map.

3. Complexity Analysis

In order to do Complexity analysis, there are some basic information that needs to be introduced.

3.1 Time Complexity of STLs

In this project, some predefined STLs was used to help accelerate the run-time complexity of the entire program, the STLs are some well-documented libraries that was embedded in the C++ Language Libraries. In our project, we are mainly using the following STLs to accelerate the program and help structuralize our data in the memory:

3.1.1 Map

Map is a data structure based on Red-Black Tree, and in each node there is a key and a value. As an STL, the key and the value can be any other data structures in C++, even another STL. The way to use a map is that it can store a pair of key and value in to a node, and user can search for a specific key and the program can return an iterator pointing to that key, and user can quickly find the linked value. The basic idea is that it can use a Red-Black Tree to accelerate the search to O(logN), and the time to construct the map is O(NlogN)

3.1.2 Unordered_map

Unordered_map is a data structure that can do exactly the same as a map, however instead of a Red-Black Tree, it is using Hashmap to store the key and the value. So the time to search for something in the unordered_map is O(1) for best case and O(N) for worst case, which depends on the hash function that it uses.

3.1.3 Vector

Vector is a data structure that can store only value in a node, and it stores value in a continuous space in memory, so if we want to search for a specific value it takes O(n) to do it. However, in the case that the user wants to access a node with a known position in the vector, then it takes only O(1) to do it.

3.1.4 Looping through STLs

In our algorithm, in the step where the weight of each sentence is to be calculated, the program needs to loop through the inner data structure. In this case, the map and vector was selected as the inner data structure. Although both data structures require O(n) to loop through, it takes more time and resource to loop through a tree than a continuous space, so vector could be way faster than map in this case.

3.2 Time complexity of Heap

In this project, we implemented a Heap when sorting the weight of each sentence, in order to sort them and keep the speed fast, we implemented a compare(struct, struct) function to do the comparison between sentence nodes. Also, this ensures that when retrieving the most weighted sentences, the speed could be O(1) since we stores the number of the sentence within the structure. Together with the time required to do heap sort, the total time complexity should be O(NlogN).

3.3 Time Complexity of different data structures and algorithms

So when we put all the complexities together, we can calculate the complexity of all different data structures and algorithms as below, W means the number of words in our database, S means number of sentences in our database, Q means the number of words in the key words the user input to search for:

	Map<Map>	Unordered_map <Map>	Map<Vector>	Unordered_Map <Vector>
Build Database (Avg.)	O(W*logW*logS)	O(W*logS)	O(W*logW)	O(W)
Build Database (Worst Case)	N/A	O(W2*logS)	N/A	O(W*W)
Search (Avg.)	O(Q*S*logW)	O(Q*S)	O(Q*S*logW)	O(Q*S)
Search (Worst Case)	N/A	O(Q*S*W)	N/A	O(Q*S*W)
Sort	O(S*logS)	O(S*logS)	O(S*logS)	O(S*logS)

Table 2. The time complexity of 4 different data structures

4. GUI

In our GUi design, we used Qt for our GUI program.

1. We need to define our GUI graph which includes the interface rectangular layout. We add the function “Text Edit”, “Text Line”, “Time Edit” and “Button” in our GUI.
2. Second, we need to define these functions one by one which well build the connection slot with their next step. For example, we defined the “Button Cliked” slot that connected to reading file. When you clicked the button, it will execute reading file function. The reading function will pick the file up from particular address.
3. Then, we also defined our “Text Edit” with the same function. This function will allow our algorithm file could read the content in “Text Edit”. Then we build the “Time Connection” which means that we transfer our time show from terminal into our GUI.
4. Because we defined the timing show in our algorithm, we just need to rebuild this function and transfer the function from our terminal show to our GUI show.
5. FInal step, it is something similar to step 4. We also finished our function to transfer our terminal results show into our GUI result show.
6. The final GUI is in following graph. When you click the “Open File” button, the program will let you choose one file which you want to read it. Then the reading file will be road by our algorithm file and the GUI will show the reading time at the right corner. Then, you type your keyword and click the search button. The result will show in the result rectangular.

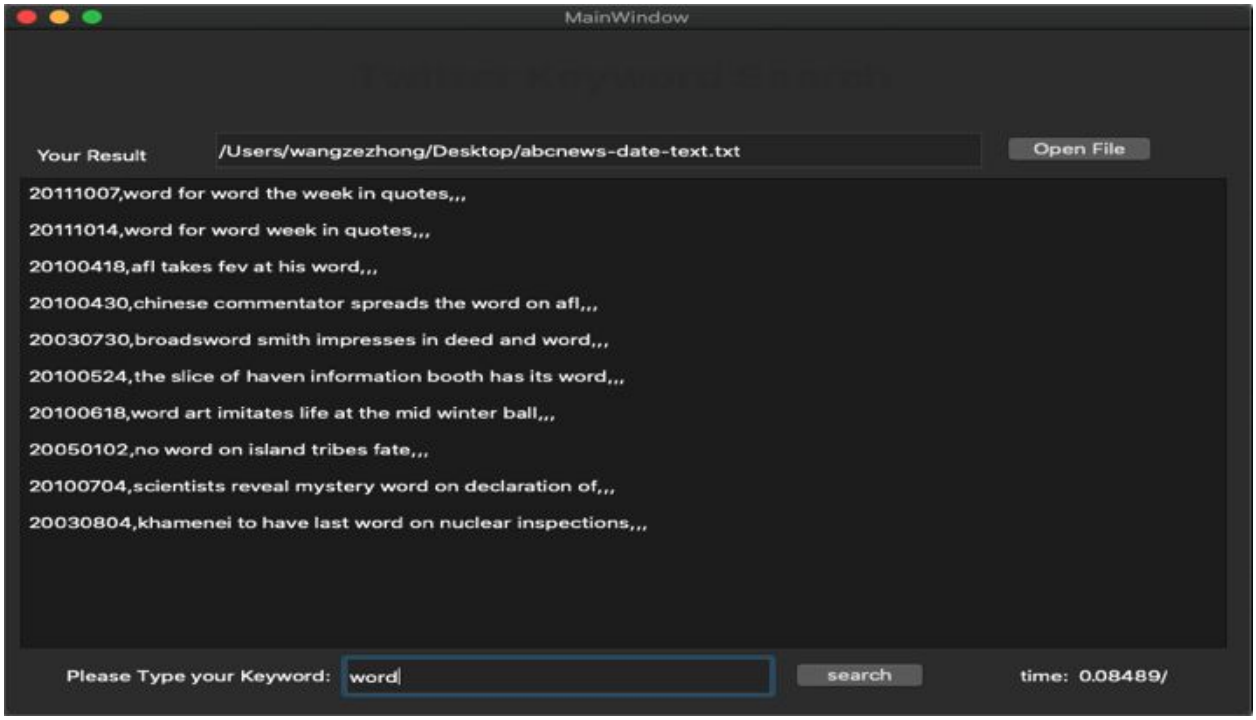

Figure 9. GUI

5. Summary

In our project, we designed a series of data structures and corresponding algorithms to do Twitter Keyword Search trying to do it quickly and efficiently. In order to make setting up database and searching keywords faster, we designed four different data structures based on inverted index to accelerate the inserting and searching process. After doing complexity calculation and testing on real machine, we figured out that using the unordered_map as outer structure and vector as inner structure is faster.

Also, we have used a Heap to store the weight of each sentence together with it's sentence number, which does not only accelerates the process of sorting the results but also accelerates the process of accessing the sentence. What's more, we have also developed a GUI to make all this process easier to interact with the user, and make the user easier to use this application.

In conclusion, the best structure to store database is the unordered_map as outer structure and vector as inner structure, and the structure to store the result is a heap, and a GUI to make the UI fancier.