Reflective Report

University of Nottingham Malaysia
School of Computer Science

COMP1028 – Programming and Algorithms
Group Coursework: Cyberbullying Text Analyzer in C

Group: KB cooperation

Group Members:
Lin Yiwei (20802133)
Cao Shiyu (20800646)
Deng Changhui(20793088)
Li Junhao(20806655)

GitHub Repository:
https://github.com/TobyYe1125/Text-Analyzer-programme-by-KB-Corperation.git

Video Recording Link:
https://github.com/TobyYe1125/Text-Analyzer-programme-by-KB-Corperation/blob/812f342bb925b82551d80bc0af39d40b116615eb/GCW%20-%20KB%20Corperation/Demo_KB%20Corperation.mp4

Module Convenor / Lecturer:
Dr. Simon Lau Boung Yew

Introduction

In this coursework, our group made a simple cyberbullying / toxic text analyzer in C. The program reads a text file, removes stopwords, detects toxic words based on a dictionary, calculates some basic statistics, and then generates a report file.

My main target was writing the core C code for loading words, sorting, detecting toxic words and generating the final report. In this way, I will describe my key design choices, the challenges I faced, how I tested and debugged the program, and what I learned from the whole process.

2. Key Design Choices

1) data structures

The main text is stored in a 2D array char word, where each row represents a single token.Stopwords are loaded into a similar array char stopwords.Toxic words are stored in char toxicList, with a parallel integer array int toxicFrequence for counting frequencies.

This design is not the most memory efficient, but it is simple and matches my current level.

2)functions and modularity

To keep the program more readable and in order, I separated the logic into several functions:

loadWords() – handles file opening, tokenization, normalisation and word counting.

loadStopwords() and removeStopwords() – implement stopword-based filtering.

countUniqueWords() – counts the number of unique words after stopword removal.

toxicWordCheck() – loads the toxic dictionary, detects toxic words and prints a frequency table.

sortingWords() – sorts the words alphabetically using Bubble Sort.

showTopWords() – shows the top N most frequent words.

generateReport() – writes the summary of the analysis into a report file.

I followed the coursework checklist and implemented these functions step by step. Every time one small part worked, I moved to the next one.

3) algorithm choices (sorting & statistics)

I used bubble logic because it is straightforward. And to calculate the result**,** I mainly used frequency counting and simple ratios:

toxic percentage = toxic words / total words × 100;

lexical diversity = unique words / total words.

These are simple to use but still give a clear idea how “toxic” or “diverse” a piece of text is.

3.Challenges Faced

1)lack of knowledge

The biggest challenge is my lack of C konwledge. The first time I saw this requirement, I was completely baffled because I had no idea what steps I needed to take to achieve these objectives. So, I previewed the later chapters in advance, covering pointers, string handling functions, structures, and file-related topics. This gave me a basic grasp of the requirements. Then I started experimenting step by step, debugging as I went. Whenever a current function couldn't solve a problem, I'd Google to see if another function could achieve the desired result. This process was incredibly long and time-consuming. Finally, through constant debugging like this, I completed the initial version.

2)file paths and opening files

Firstly, I faced "opened failed!" because my program was running in a different directory from the input files. I had to learn how to check the current working directory and make sure sampleWords.txt, stopWords.txt and toxicWords.txt are in the same folder as the .exe.

3)string handling and application

Dealing with punctuation and make the program correctly was difficult. I used ispunct() to cut off punctuation inside the word buffer, and tolower() to make everything to lowercase. I also had to be careful with buffer sizes (%39s with WORD_LONG = 40) to avoid overflow.

4)stopword removal without losing data

When implementing removeStopwords(), I needed to “compress” the words array in-place while skipping stopwords. I solved this by using a newCount index: every time I keep a word, I copy it to words and increase newCount.

5)avoiding division by zero and crashes

For empty or very short files, allWordCount can be zero. If I directly calculate result, the program will crash. So, I added checks in generateReport() and in the summary menu to handle this case and print 0.00% instead.

6)User input validation

When I initially used scanf("%d", &choice) without checking the return value, entering a letter would break the menu loop. I learnt to use int flag = scanf("%d", &choice); and clear the input buffer with while(getchar() != '\n') when the input is invalid.

4.Debugging and Testing Approach

To test and debug the program, I used the following strategies:

1) test files

I made a sampleWords.txt with some bad words mixed to check if the program detects them correctly.

2)error cases

Tried empty files – program didn't crash, just said 0 words, which was good. Also, a file with only stopwords, and after removal, everything was zero. Repeated some toxic words to check counting, and it did.

3)print-based debugging

I threw in extra printf() everywhere during coding, like printing the words array step by step. Helped me spot where things went wrong.

4)menu flow testing

I messed around with the menu, like typing letters instead of numbers, and it handled it without crashing, which made me feel better about the code.

5.Lessons Learned about problem-solving in C

From this project, I learned a lot about C programming. First, breaking code into functions like loadWords() makes debugging way easier – bugs are isolated. Second, strings and arrays need careful handling; I had overflows that taught me to check sizes. About sorting, Bubble Sort was okay for this, but next time I'll try something faster. Error handling is key too – without it, the program crashes on bad input. Overall, it showed me how basic stuff like loops and files can solve real problems like detecting cyberbullying, which is cool.

6.ReadMe Instructions (How to Run the Program)

1)place the following files in the same directory as the executable:

sampleWords.txt (input text file)
stopWords.txt (list of stopwords)
toxicWords.txt (list of toxic words)

Make sure the files are in the right place, or it'll complain

2)compile the program, for example using GCC:

```
gcc word_analyser.c -o word_analyser.exe
```

3)run the executable:

```
./word_analyser.exe
```

4)follow the on-screen menu:

Option 1: enter an input filename (e.g. sampleWords.txt) to load and preprocess the text.

Option 2: detect toxic words and print their frequency table.

Option 3: sort the processed words alphabetically.

Option 4: generate the summary report in analyse_report.txt.

Option 5: produce a summary on screen, including top 5 frequent words.

Option 6: exit the program.

If something goes wrong, check the file paths first – that's what got me stuck.

7. **GitHub and Video Link:**

https://github.com/TobyYe1125/Text-Analyzer-programme-by-KB-Corperation.git

https://github.com/TobyYe1125/Text-Analyzer-programme-by-KB-Corperation/blob/812f342bb925b82551d80bc0af39d40b116615eb/GCW%20-%20KB%20Corperation/Demo_KB%20Corperation.mp4