University of Nottingham
UK | CHINA | MALAYSIA

COMP1028 Programming and Algorithm
Session: Autumn 2025
Group Coursework (25%)

Group Name	**Vamos**				
Group Members					
Name 1	**Muhammad Rafay Shahid**		**ID 1**	**20808856**	
Name 2	**Zuo Kaining**		**ID 2**	**20797433**	
Name 3	**Zuha Qasim**		**ID 3**	**20799766**	
Name 4	**Liu Guangbing**		**ID 4**	**20797468**	
Name 5			**ID 5**		
Marks **Total (100)**	Program Functionalities (65)	Bonus (5)	Code Quality (10)	Presentation / Demo (10)	Report / Documentation (10)
Submission Date					

1. Introduction

In an era where online harassment is a critical issue, our goal was to build a tool capable of processing massive datasets efficiently to detect toxic language. Unlike basic solutions that might rely on slow, nested loops to identify offensive terms, our project implements advanced data structures specifically the **Aho Corasick algorithm** (figure 1.0) and **Hash Tables**(figure 1.1) to ensure our analysis is computationally efficient (O(N)) and scalable.

2. Key Design Choices

2.1 Data structures used:

2.1.1 Hash Table (Uthash library)

We used the **Uthash** macro library to store toxic words, stop words, and unique word counts.

- **Justification:** We chose a Hash Table over a Binary Search Tree (O(log n)) or Linked List (O(n)) because word lookup occurs for *every single token* in the input text.
- **Complexity:** A Hash Table provides average O(1) time complexity for insertions and lookups. This constant time access is critical for performance when processing large corpora like chatlogs.csv.

2.1.2 Aho Corasick Trie

We implemented a Trie with failure links (Aho Corasick automation) for fast multi pattern matching.

- **Justification:** A naive approach using strstr for every toxic word would result in a time complexity of roughly O(N * K * L) (where N is text length, K is number of toxic words, L is avg word length). As the dictionary grows, this becomes exponentially slower.
- **Complexity:** The Aho Corasick algorithm scans the text in a single pass with O(N + M + Z) complexity (where M is total pattern length and Z is matches found), making it independent of the dictionary size during the search phase.

2.2 File Handling Strategy

The program automatically detects file types:

- **.txt files:** Processed line by line using fgets to keep memory usage low, preventing stack overflow on large files.
- **.csv files:** The program parses the header to count columns, then prompts the user to select a specific column (e.g., "comment_text") for analysis. This ensures we don't waste resources analyzing irrelevant metadata like IDs or Dates.

2.3 Tokenization & Text Normalization

We implemented a custom normalize_inplace function. Instead of creating new string copies (which increases memory fragmentation), it modifies the buffer directly to:

1. Convert characters to lowercase.
2. Remove punctuation and extra whitespace.
3. Filter out stop words (checked against the Hash Table).
4. **New:** Detect sentence terminators (., !, ?) to calculate average sentence length stats.

2.4 Toxic Word Detection Strategy

We categorize words into **MILD** and **SEVERE** types.

1. **Loading:** Words are loaded from mild_words.txt and severe_words.txt into the Hash Table and Trie.
2. **Scanning:** The program traverses the Trie. If a node has a valid output (match), it uses that string as a key to increment the counter in the Hash Table.
3. **Dynamic Updates:** Users can now add new toxic words via the menu, which triggers a rebuild of the Trie to ensure the new word is immediately detectable.

2.5 Sorting & Reporting

We utilize Uthash's built in HASH_SORT function, which implements **Merge Sort (**O(Nlog(N))**)**. This is stable and efficient for linked lists. We provide:

- Top N Toxic Words (with ASCII bar charts for visualization).
- Top N Most Frequent Words (General vocabulary also with ASCII bar charts).

2.6 Limitations

While efficient, our keyword based approach has inherent limitations:

1. **Context Blindness:** The system cannot detect context. A phrase like *"This is not bad"* is flagged as mild toxicity because it contains "bad", despite being a positive sentiment. This leads to false positives.
2. **Memory Overhead**: The Aho Corasick Trie allocates an array of 256 pointers for every node (for the ASCII alphabet). For a sparse dictionary, this results in significant memory usage compared to a compressed Trie or DAWG (Directed Acyclic Word Graph).
3. **Exact Matching Only:** The system does not handle obfuscation well. Variations like "stup1d" or "b@d" will be missed unless explicitly added to the dictionary.

2.7 User Interface Design

The user interface is simple and command line based. The program first asks the user if he wants to analyse a file or exit the program, if the user enters 1 the program asks the user for a file name. Then the program processes the file and provides the user with a few options on what to do. The user can then choose any of those options or choose to analyse another file.

2.8 Future Imrpovements

If we had more time, we would implement:

1. **Machine Learning Integration:** Replacing the dictionary lookup with a Naive Bayes or Logistic Regression classifier to handle context and probability, reducing false positives.
2. **Multi threading:** Using pthreads to process large files in chunks. One thread could read the file while others process tokens, utilizing multi core CPUs.
3. **Fuzzy Matching:** Implementing Levenshtein distance checks to detect obfuscated toxic words (e.g., detecting "h3ll" as "hell").

---

3. Challenges Faced and Lessons Learned

3.1 CSV Parsing & Quotes

- **The Problem:** Our initial strtok implementation failed on CSVs because it split strings by comma , blindly. This broke fields like "Hello, world" into two separate invalid tokens.
- **The Solution:** We wrote a custom state machine parser that tracks an inside_quotes boolean flag. Commas encountered while inside_quotes is true are treated as literal characters rather than delimiters.
- **Lesson:** Standard library functions like strtok are often insufficient for real world data formats; custom logic is required for robustness.

3.2 Aho Corasick Failure Links

- **The Problem:** Shorter words were sometimes skipped when they appeared as suffixes of longer words.
- **The Solution:** We had to carefully debug the BFS queue logic used to build failure links, ensuring that if a character match fails, the pointer traverses back up the failure link to the longest possible suffix match, rather than resetting to the root.

Appendices

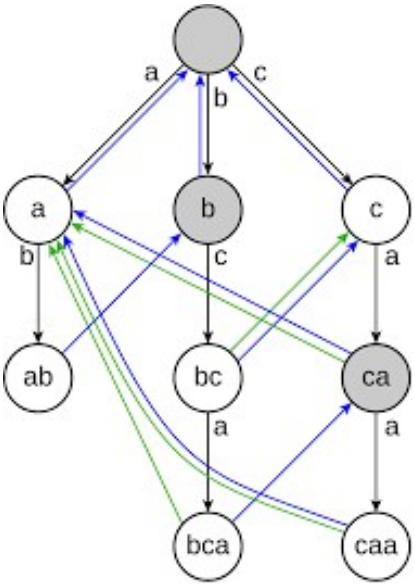

Figure 1.0

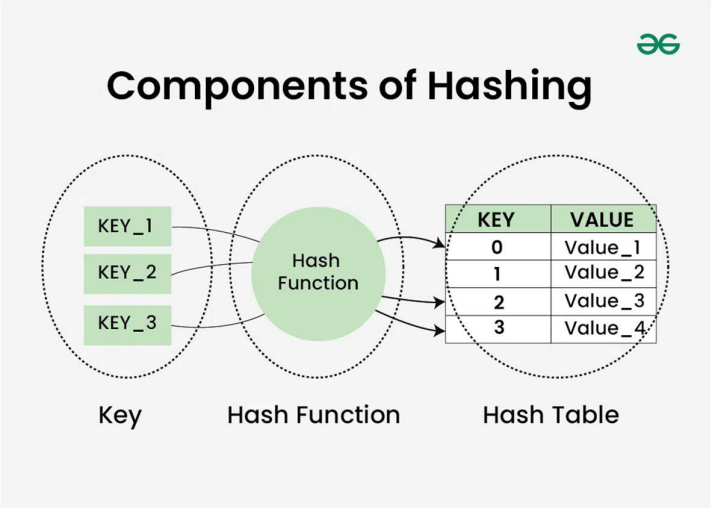

Figure 1.1

Appendix A: ReadMe

Required Input files

All required files to run the program are stored in the data folder. This includes:

- severe_words.txt
- mild_words.txt
- stopword.txt

Test Files

- chatlogs.csv
- train.csv

Required Software to Compile the Program

- C compiler (GCC/MINGW)
- CMake

Steps to Compile and Run the Program

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
cd ..
cmake --build build
./build/prog_ass1.exe
```

This is from the github repository, copy these commands after cloning the repository. This will automatically compile and run the code. In case the build directory already exists and is initialized, you only need to run "cmake –build build" from the project root.

```
mkdir build
cd build
cmake ..
```

For MacOS and linux, please try running the these commands to initialize the build directory, Afterwards the commands are the same as windows.

The program should only be run from the Project root (not from build, data, or src directory) as the relative path to the toxic and stop word dictionaries are hardcoded.

GitHub Link & Video Link

GitHub Repository: https://github.com/r0pc/prog_ass1.git

Video Presentation Link:
https://drive.google.com/file/d/1NDxeEc7K4ua_bx9GgcMtH0HhePNI75G0/view?usp=sharing

Appendix B: Marking Scheme Summary (Optional)

Criteria	Description / Notes
Text Input & File Processing	The program can take both txt and csv files as input and process them efficiently
Tokenization & Word Analysis	The program normalizes a string, stores total number of words/ characters and maintains a HashMap to manage unique words
Toxic Word Detection	Uses the Aho Corasick algorithm and HashMaps to efficiently detect toxic words in the file.
Sorting & Reporting	Uses the built in mergesort function in UTHash library to efficiently sort the hashmap.
Persistent Storage	Whenever the user request a output in a file we append the analysis_report.txt with a record, mention the file analyzed and store its respective data.
User Interface	Clean console UI with clear instructions and is easy to use, outputs a ascii bar chart for the top n elements.
Error Handling & Robustness	Every time we initiate a syscall we check the respective

	pointer and handle the error accordingly
Code Quality & Modularity	Use of multiple c files, header files and functions
Bonus Features	Can analyze a string and find a substring that is a match of any length (1 word, 2 words, 3 words, ….).
Presentation / Demo	Presentation is 9 minutes long, contains subtitles and gives a high explanation of the code while showing all the functions of the program
Report / Documentation	

Note: This may help to ease the marker on having an overview of your entire project in a glance.

References:

https://troydhanson.github.io/uthash/
https://cp-algorithms.com/string/aho_corasick.html
https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/