Boolean Function Minimization Using the Quine-McCluskey Algorithm

CSCE230102

Dr. Dina Mahmoud

Section 2

Group Members: Salma El-marakby, Marina Nazeh and Mikhael Khalil

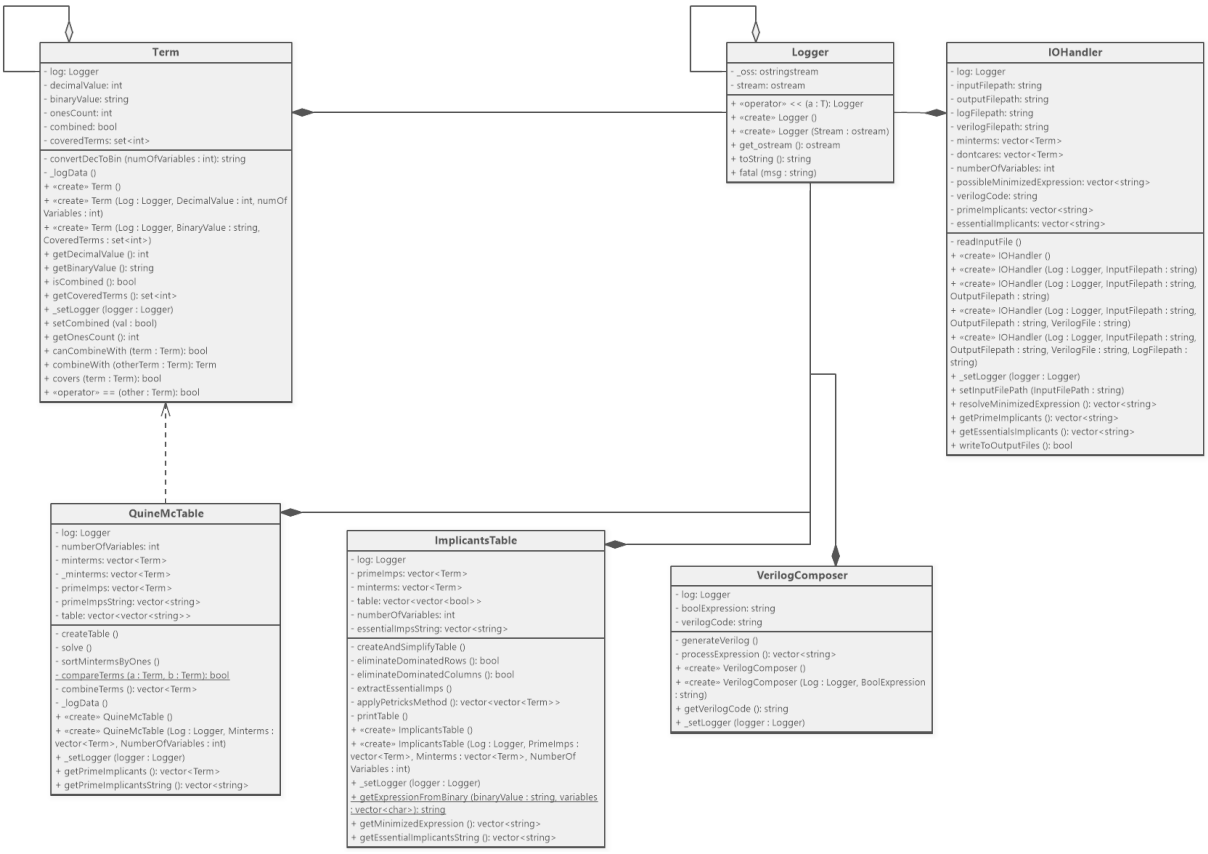
Table of contents

1. Introduction

Boolean function minimization is a crucial process in digital logic design, and aims to simplify Boolean expressions while preserving their circuit's functionality. Digital circuits can become complex quickly, with each additional logic gate added which causes propagation delay, increases power consumption, and increases the size of a circuit physically. By minimizing Boolean expressions, we can reduce the number of required logic gates, leading to lower costs, improved performance due to reduced delay, and enhanced reliability by making circuits contain less errors and easy to debug (*Logic minimization in FE electrical exam*). We've studied the K-map method and Quine-McCluskey for minimizing boolean expressions. The Quine–McCluskey method offers a significant advantage over the K-map method, particularly for functions with more than five variables. While K-maps depend on pattern recognition of adjacent cells, this becomes difficult as the number of input variables increases (*Introduction to digital systems: Modeling, synthesis, and simulation using VHDL*). However, Quine–McCluskey uses a tabular approach based on the adjacency theorem to reduce minterms. This makes it more effective for larger functions where K-maps become harder or even impossible to use. This project aims to develop an efficient boolean function minimization tool using the Quine-McCluskey method using C++. The program will generate and display all prime implicants (PIs). Using this data, the program will identify essential prime implicants (EPIs). In cases where multiple minimized solutions exist, the program will employ Petrick's Method to identify and display all possible minimal expressions (*Prime implicant simplification using Petrick's method - technical articles*). The final output will provide an optimized Boolean function. The program offers several key features that make it a reliable tool for Boolean minimization. It begins by reading and validating input from a structured text file, supporting both minterms and maxterms along with don't-care terms. The program then identifies and displays all prime implicants (PIs), showing their binary representations and the terms they cover. Using these PIs, it determines the essential prime implicants (EPIs) and highlights any uncovered minterms. To achieve Boolean minimization, the program solves the Prime Implicant Table, ensuring an optimized expression. When multiple solutions exist, it applies Petrick's Method to find and display all possible minimal expressions. Additionally, the program can generate Verilog code for the minimized function, aiding in digital circuit design. It supports Boolean functions with up to 20 variables, making it suitable for complex logic circuits. Its user-friendly output presents results in a structured format, including Boolean expressions, truth tables, and binary representations. Moreover, it includes input validation, providing error detection and reporting for invalid input files.

2. Program Design

Class Diagram:



The I/Ohandler class reads input from a file in a structured format to ensure accurate processing of Boolean expressions. The input file consists of three lines: the first line contains the number of variables, the second line lists the minterms (denoted by 'm') or maxterms (denoted by 'M'), and the third line contains the don’t-care terms (denoted by 'd'). Each term (minterm, maxterm, or don’t-care) is separated by commas, and the values must be properly formatted as integers. The program validates the input by checking if the number of variables is a valid positive integer. If any input is invalid, such as a non-integer, both minterms and maxterms are provided together, or invalid term format, the program handles errors by displaying appropriate error messages. This error handling ensures that only valid input data is processed, preventing crashes or incorrect outputs during execution.

The Term class is the class responsible for some helper functions that are used later on in other classes that solve the function minimization using the Quine-McCluskey method. For converting minterms, maxterms, and don’t-cares into binary representation, we have: string convertDecToBin(int numOfVariables). To help with grouping the terms with respect to the number of ones in each term in its binary representation, we use: int getOnesCount(), which counts the number of ones in each binary

number, and later the binary numbers with the same 1s count are grouped together. Combining the terms together means to check the binary expressions in adjacent groups after grouping them from the previous step and loop to compare each two members of adjacent groups, checking if there is only 1-bit difference between them, then they can be combined together. We use: bool canCombineWith(Term &term) which returns true if the term provided can be combined. If it can be combined, we use: Term combineWith(Term &term) to combine them by replacing the 1-bit difference between them with a hyphen ('-'). Finally, to track the covered terms by the combined terms, we have: bool covers(Term &term). Other data structures used in the class include set<int>, which is used to store and track the covered terms (minterms or maxterms) for each term, ensuring efficient reference when combining terms. bitset<20> is used for converting decimal values to binary, supporting up to 20 variables. string is used to store and manipulate binary representations of terms. Additionally, std::count() from the <algorithm> header is utilized to efficiently count the number of 1s in the binary string during grouping.

The QuineMcTable class creates the Quine-McCluskey table and solves it to get the PIs to be used later on. In the QuineMcTable class we have: void createTable(), which creates an adjacency matrix of the size of minters (if the input was Maxterms, it was converted to minterms in the class before) with puts a ‘x’ between the minterms that can be combined together. For the: void solve() function, it calls the sortMintermsByOnes() for grouping the minterms (ensuring that minterms are sorted in ascending order based on the number of 1s in their binary representation). The sortMintermsByOnes() function uses the std::sort() which is typically implemented using the Quicksort algorithm. Then, it enters a loop where it repeatedly calls combineTerms(), which attempts to combine terms. Using the std::find() algorithm in it to make sure no duplicates are found. If no new terms are created, the loop terminates, which indicates that the prime implicants have been found. The combineTerms() function keeps track of the terms that were not combined to return them as the prime implicants. It uses vector<bool> to flag the combined ones to consider the non-flagged terms.

The ImplicantsTable class is responsible for constructing and simplifying the Prime Implicants Chart to find the essential prime implicants and obtain the minimized Boolean expression. It manages a table that maps prime implicants to minterms, applies dominated row and dominating column elimination, and finally uses Petrick’s Method to determine the minimal cover. The class contains multiple helper functions like: void createAndSimplifyTable(), which initializes the table, and marks which prime implicants covered which minterms. The functions bool eliminateDominatedRows() and bool eliminateDominatedColumns() iteratively remove unnecessary rows and columns by checking for dominance relationships. Where if we have a column that covers all the prime implicants that other rows covered, it can be eliminated. However, for the rows, if what a row covers was covered by another row completely, it can be eliminated to simplify. If further simplification is needed, vector<vector<Term>> applyPetricksMethod() which returns a 2D Boolean table represents the prime implicants chart. The function constructs a Boolean product-of-sums expression and finds the minimal covering set of prime implicants. The final minimized expression is generated using getMinimizedExpression(), which converts

binary terms into Boolean expressions.

The VerilogComposer class translates a minimized Boolean function into Verilog code, describing the corresponding digital circuit using basic logic gates. It begins by processing the Boolean expression into sub-expressions, representing each term in a sum-of-products form. The generated Verilog module consists of a declaration for inputs , an output X for the final result, and wire declarations for each sub-expression. Each sub-expression is implemented using AND gates, where the inputs are either the variables or their negations, the negation of a variable (denoted by ' in the Boolean expression) is implemented in Verilog using the ~ operator. For example, a term like A'B would be translated into the Verilog code and(_1, ~A, B);.. An OR gate is then used to combine the outputs of all the AND gates to produce the final output. The structure used to construct the Verilog code is: module declaration The Verilog code starts with a module declaration, where the inputs are the variables from the Boolean expression, and the output X represents the result of the entire Boolean function. The inputs are derived from the distinct alphabetical variables in the Boolean expression, sorted in alphabetical order. These inputs are declared in the input section, wire and gate declarations Each sub-expression of the Boolean function is represented by a wire. The wires are named _1, _2, ..., _n and are used to connect the AND gates that generate each term of the function, finally the endmodule keyword.

3. Challenges and Issues Encountered

One of the key challenges faced in this project was determining all possible minimized solutions when not all prime implicants were essential after eliminating dominated rows and columns in the Prime Implicants Chart. For instance, if a chart contained four implicants—two of which were essential while the other two occupied the same row—there would be multiple valid solutions. The minimized expression could either include the two essential implicants along with the first repeated row or the second repeated row. To systematically address this issue, Petrick’s Method was implemented. This method provided a structured approach to finding all possible combinations of non-essential implicants, ensuring that no valid solution was overlooked. By incorporating Petrick’s Method, the program effectively handled cases with multiple minimized expressions, enhancing its accuracy and reliability in Boolean function minimization.

4. Program Testing and Validation

The testing phase of the Boolean Function Minimization project plays a crucial role in ensuring the correctness, reliability, and robustness of the system. This report presents the testing strategies employed, detailing the unit tests implemented for different modules, their objectives, methods, and expected outcomes. The tests were conducted using the Google Test (gtest) framework to validate the functionality of key components, including IOHandler, ImplicantsTable, QuineMcTable, Logger, and Term. Each of these components plays a fundamental role in the system, and their individual validation ensures that the overall Boolean function minimization process operates correctly.

The IOHandler class, responsible for managing file input and output operations, was tested through multiple unit tests. One such test, ConstructorReadsInputFile, was designed to verify that the constructor correctly reads the input file and processes the Boolean expression. This was done by creating a temporary file containing minterms and don't-care terms and initializing an instance of IOHandler. The expected outcome was that the resolved Boolean expression matched the predefined result, "AB + C." Additionally, the SetInputFilePath test ensured that updating the input file path correctly reflected in the processed output, while HandlesMaxterms confirmed the correct handling of maxterms, expecting the minimized expression to be "A." Furthermore, WriteToOutputFiles validated that the minimized expressions were correctly written to an output file, ensuring that the file was non-empty and contained accurate results.

The ImplicantsTable class, which constructs and simplifies the prime implicants table, was tested to verify its ability to compute minimized Boolean expressions. The BooleanFunction1 test checked whether a basic input case was processed correctly, expecting an output of "A'B'D + A'B'C." Further tests, BooleanFunction2 and BooleanFunction3, examined the class's ability to handle additional prime implicants and complex cases with multiple valid expressions. Each of these tests assessed the accuracy of the getMinimizedExpression() function in producing correct results based on different sets of minterms and implicants.

The QuineMcTable class, implementing the Quine-McCluskey algorithm for determining prime implicants, was also subjected to rigorous testing. The DefaultConstructor test ensured that an empty table initialized correctly, while the ConstructorWithMinterms test verified that the class correctly processed minterms upon initialization. The Solve test focused on the accuracy of prime implicant computation by calling the getPrimeImplicants() function with various minterm sets and confirming the expected outputs.

To monitor program execution and debugging, the Logger class was tested to validate its functionality. The DefaultConstructor test ensured that logging was initialized properly, while the ConstructorWithStream test verified that logging worked with external streams. Additional tests, such as LoggingMultipleValues and LoggingWithManipulators, confirmed that multiple values could be logged accurately and that newline manipulators were handled correctly.

The Term class, which represents Boolean function terms and their transformations, underwent several tests to ensure its proper functionality. The DefaultConstructor test validated that an empty term was correctly initialized, while DecimalConstructor and BinaryConstructor confirmed the accurate conversion of decimal values into binary representations. The CanCombine and CannotCombine tests ensured that terms with a single differing bit were correctly identified as combinable, while those with multiple differing bits were not. Lastly, the CombineWith test checked that two combinable terms correctly merged, replacing differing bits with a '-' character.

The results of the testing phase were highly successful, with all unit tests passing, confirming that the

IOHandler, ImplicantsTable, QuineMcTable, Logger, and Term classes function as expected. Edge cases, such as handling maxterms and different input formats, were addressed to enhance the system's robustness. Moving forward, further improvements include adding unit tests for private functions in ImplicantsTable and implementing additional test cases for functions like sortMintermsByOnes() and combineTerms() in QuineMcTable. These enhancements would further strengthen the reliability of the system. Overall, the comprehensive unit testing conducted affirms that the Boolean function minimization process is correctly implemented using the Quine-McCluskey algorithm, ensuring accurate file handling, logical processing, and output generation.

5. User Guide: How to Build and Run the Program

This project is a C++ console application designed to minimize Boolean functions using the Quine-McCluskey method. The program reads a Boolean function from an input file, identifies prime implicants (PIs) and essential prime implicants (EPIs), and outputs the minimized Boolean expression. Additionally, it can generate a Verilog module based on the minimized result.

The app can be used from the console as follows:

./QuineMcCluskeySolver [-v] [-d <dump_log_file>] [-o <output_file>] [--verilog <verilog_file>] <input_file>

You can download and use prebuilt binaries from the GitHub Releases page.

CLI Switches/Flags

- **<input_file>**: Input file with the Boolean function (required)
- **-v** : Verbose output
- **-d <dump_log_file>** : File to dump intermediate solving steps and final output
- **-o <output_file>** : File to write the minimized expression
- **--verilog <verilog_file>** : File to write the generated Verilog module

The input file must contain three lines:

1. The number of variables
2. The list of minterms or maxterms (prefixed with m or M, separated by commas)
3. The list of don't-care terms (prefixed with d, separated by commas)

    3

    m1,m3,m6,m7

    d0,d5

This represents a Boolean function with 3 variables, minterms {1, 3, 6, 7}, and don't-care terms {0, 5}.

The output includes:

- Prime Implicants and their coverage
- Essential Prime Implicants
- All possible minimized Boolean expressions
- Generated Verilog module implementing the minimized function

Sample output:

Prime Implicants:

[01] A'B' covers: 0 | 1

[02] AB covers: 6 | 7

[03] C covers: 1 | 3 | 5 | 7

Essential Implicants:

[01] C covers: 1 | 3 | 5 | 7

[02] AB covers: 6 | 7

Possible Minimizations:

[01] F = AB + C

Verilog Output :

```
`timescale 1ns / 1ps

// Boolean Expression: AB + C

module fn(

   input A, B, C,
```

```
    output X
    );

    wire _1, _2;

    and(_1, A, B);
    and(_2, C, B); // Note: logic based on generated expression

    or(X, _1, _2);

endmodule
```

For Developers: Building from Source

To build the application from source:

1. Clone the repository
2. Navigate to the project directory
3. Compile the code using CMake:

```
cmake.exe --build cmake-build-profile_name --target QuineMcCluskeySolver
```

(CMake and a C++ compiler must be installed)

For more optimized and cross-compiled builds:

```
zig build --release
```

(Requires the Zig compiler to be installed)

Run the Application

./QuineMcCluskeySolver [-v] [-d <dump_log_file>] [-o <output_file>] [--verilog <verilog_file>] <input_file>

6. Conclusion

This project successfully implemented Boolean function minimization using the Quine-McCluskey algorithm, effectively reducing Boolean expressions and generating essential outputs such as prime implicants and minimized expressions. The integration of Petrick's Method allows for handling multiple minimized solutions, while the Verilog code generation feature enhances applicability in digital circuit design.

Through this process, key lessons were learned, including the importance of efficient data structures for managing large-scale inputs, the need for systematic debugging in Boolean function operations, and the necessity of handling edge cases to ensure accurate reductions.

Looking ahead, future enhancements could include developing a graphical user interface (GUI) for more intuitive input handling, supporting multi-output Boolean functions, and integrating an interactive K-Map visualization to complement the tabular method. These improvements would further refine the program's usability and expand its practical applications.

Contributions of Each Team Member:

Salma EL-marakby: "Term "class, 10 test cases and half of the report/readme file

Marina Nazeh: The "Quine-McCluskeytable" class, 10 test cases and the second half of the report/readme file

Mikhael: `ImplicantsTable` class, the CLI, GoogleTest integration, and the Zig build file

AI Usage by Each Team Member:

- **Salma El-Marakby**: Utilized AI to outline the primary functionalities required for the class term, ensuring a structured approach to project development.

  Prompt to AI:
  🟢 *"Can you help me outline the key functionalities I need to implement for this term class's project? I want a structured approach that covers all essential components."*

- **Marina Nazeh**: Used AI to resolve coding errors and receive recommendations for suitable data structures, such as using a set instead of a vector for storing prime implicants to prevent duplicates. Additionally, AI helped in recalling specific syntax, including sorting and finding expressions like: sort(minterms.begin(), minterms.end(), QuineMcTable::compareTerms);, and find(newTerms.begin(), newTerms.end(), combinedTerm); These contributions improved code efficiency and ensured the correct implementation of the algorithm.

  Prompt to AI:
  🟢*"What does that error mean? And how to resolve it, 'passing the error' "*

  🟢*"What is a better data structure than the vector to be used to store prime implicants in a Quine-McCluskey solver in c++?"*

  🟢*"What is the syntax of the std::sort, and the std::find functions and what to pass to them in c++?"*

- **Mikhael**: Used AI to identify the appropriate method for obtaining all possible combinations of the Prime Implicants Chart, specifically Petrick's Method. Additionally, it provided a clear explanation of how the method works, aiding in its implementation.

  Prompt to AI:
  🟢 *"What's the best way to find all possible combinations of prime implicants in a Quine-McCluskey algorithm? I heard Petrick's Method is used — how does it work?"*
  🟢 *"Can you explain Petrick's Method and how to implement it in code?"*

Work Cited

"Logic Minimization in FE Electrical Exam." *Group (4)*, 3 Mar. 2025.

"Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL." *O'Reilly Online Learning*, Wiley, Accessed 1 Apr. 2025.

"Prime Implicant Simplification Using Petrick's Method - Technical Articles." *All About Circuits*, Accessed 1 Apr. 2025.