Minimizing Boolean Functions: The Quine-McCluskey Method

Group members:

Kirollos Ehab Zikry

Mohammad Yahya Hammoudeh

Rawad Batrawi

On October 31, 2023

Report on *Digital Design Project 1*

Instructor: *Dr. Mohammad Shaalan.*

Abstract

The code uses the Quine-McCluskey algorithm to minimize Boolean functions. It accepts as input a Boolean function in the Product of Sums (PoS) or Sum of Products (SoP) form, and outputs the minimized Boolean expression. During and in addition to this task, the code creates a truth table, finds all prime implicants, and divides them into essential and non-essential categories. It also creates HTML files that can be used to visualize logic circuits and Karnaugh Maps (Kmaps).

1. Introduction

In many areas of computer science and engineering, especially in the design and optimization of digital circuits, boolean function minimization is a crucial task. As circuits get more complex, it becomes more and more necessary to simplify them; simpler boolean (logic) circuits carry many benefits: lower transistor cost, less logic gates, easier readability, less power requirements, quicker outputs and more effective connections, and all in all lower costs in all forms from electrical to financial.

The Quine-McCluskey algorithm, named after computer scientists Edward J. McCluskey and Willard V. Quine, is one of the most well-known and trusted algorithms for boolean simplifications. Given its procedurally executed and exact nature, outputs can be predicted, and it can be easily extrapolated to many different sizes of variable circuits (as demonstrated through this project!) However, like many algorithms, the QM algorithm gets quite costly in terms of space at much larger circuits, one of the unfortunate limitations of such an efficient algorithm.

This report describes a program that uses this QM algorithm to minimize Boolean functions. The program, written in C++, finds the minimized Boolean expression, creates logic circuits and KMaps as HTML files to help visualise and clarify the circuits, hopefully providing a comprehensive tool for understanding and using Boolean minimization in practical situations, and in variable circumstances.

The report will go into detail about the program's design, going over the data structures and algorithms that were employed. It will also list all current issues and restrictions, describe the testing procedures used, and offer comprehensive guidelines for compiling and executing the program.

2. Program Structure

2.1 Architectural Overview

Both extensibility and modularity were considered in the program's design, for multiple reasons, especially for ease of extrapolation and implementation or expansion, given the segmentation and organisation. It is divided into multiple main functions, each in charge of a particular step in the simplification. The HTML/JavaScript Code Generator, the Quine-McCluskey Algorithm Engine, the Karnaugh Map Generator, and the Input Parser are some of these components. There are also several smaller aid functions to make many different tasks more streamlined and efficient.

2.2 Input Parser

The user-provided Boolean function (the input of which is slightly altered to automatically run the test cases) must be read and validated by the parser. It transforms the function into a format that the Quine-McCluskey Algorithm Engine can understand. The parser verifies that the function follows the permitted Boolean expressions and looks for syntax errors.

2.3 Quine-McCluskey Algorithm Engine

This is the program's central component, where the real minimization happens. This algorithm simplifies while efficiently storing and managing minterms, prime implicants, essential prime implicants, variables and more using a variety of data structures, including vectors and maps.

2.4 Karnaugh Map Generator

Following minimization, a Karnaugh Map is produced around the simplified function. This graphic representation confirms the outcome and helps visualise and understand how the minimization was accomplished.

2.5 JavaScript and HTML Code Generator

This part is in charge of producing JavaScript and HTML code that can render the logic circuit diagram and the Karnaugh map in their visual forms. Similarly to the above section, this section aims to clarify and help visualise the circuits for analysis and/or understanding.

2.6 Code Organization

For easier reading and maintenance, the code is divided into several functions and structs. Logically related functions are grouped together, and related functionalities and data members are conveniently encapsulated in a struct – for example, the classification of essential and non-essential prime implicants is in its own section, and all characteristics of an implicant are grouped together under one singular struct. Some operators are also overloaded to help accommodate these modules.

2.7 Efficiency

In order to efficiently handle Boolean functions with a reasonable number of variables, the program is optimized for speed and memory usage. However, functions with a lot of variables may cause the program to perform less well due to the Quine-McCluskey algorithm's inherent complexity, as well as the programming being only up to par with our personal knowledge and abilities to varying extents.

2.8 Variable Naming and Type Safety

Whenever possible, type-safe constructs are employed to avoid type-related errors. The use of descriptive variable names improves readability and maintainability by making the code self-explanatory. Additionally, the code is implemented with the idea of completely and entirely preventing incorrect values or situations to occur, but there is still code to deal with these situations in the case that they do arise.

2.9 Future Expansion

The current architecture makes extensibility simple. Future iterations may incorporate a graphical user interface, support other minimization algorithms, and don't-care terms, among other features.

3. Issues and Restrictions

3.1 Scalability

While still being quite extensible and expandable, the inherent nature of the QM algorithm with such a high complexity with larger and larger numbers of variables makes it both more inefficient and quite costly in terms of both space and time at extremely large values, but it remains efficient for smaller sets.

3.2 Error Handling

Although the program has some basic error handling—particularly for syntax errors in the Boolean function used as input—it does not have extensive mechanisms for handling more complex problems or inputs that are not completely expanded in either PoS or SoP form. It cannot handle edge cases such as self-contradictory functions or circular dependencies between variables, for instance.

3.3 Code Duplication

There are some code blocks in the current version of the program that are repeated with only slight modifications over several iterations and stages of the work process. This increases the size of the codebase and creates possible points of failure because every modification must be applied to every section. Additionally, with multiple people working on this project in parallel, there were several methods that were developed in parallel and in differing ways, and which were integrated in different methods and functions continuously.

3.4 Inconsistencies in Code Style

While the style of the code is generally consistent, there are a few small deviations, like using both cout and std::cout. These are also a result of the point stated above with multiple people programming in differing styles and mannerisms. While not diminishing the functionality of the program, such inconsistencies can make the code more difficult to read and update.

3.5 Lack of User Interface

Right now, the program runs via the terminal console, which might not be the most intuitive method for all users, particularly for more visual aspects such as the tables and maps. Although it hasn't been used yet, a graphical user interface would increase the program's accessibility, implemented through QT or other tools for example.

3.6 Restricted Formats of Output

As of right now, the program can produce logic circuit diagrams and Karnaugh Maps in HTML and JavaScript code. It does not, however, support other formats, such as JSON, XML, or plain text, which might be helpful in other contexts.

3.7 Type Safety

The program still uses some less type-safe components, such as representing minterms with character vectors, despite efforts to employ type-safe constructs. As a result, these small inefficiencies may lead to problems in the future depending on the differing implementations of other functions, etc.

3.8 Testing Limitations

The software has been put through some basic testing to make sure it works, but neither formal verification techniques nor extensive, rigorous testing have been applied to it. This leaves open the chance of unidentified flaws or restrictions. Additionally, in order to submit this program with its own automatic test cases, as stated a handful of points above, the manual entry has had to be subverted to allow for this automatic testing. However, the functionality structure is there.

3.9 Documentation

The extant documentation, encompassing external guides and inline comments, is not comprehensive. This could present difficulties for users or potential programmers attempting to comprehend the inner workings of the program as well as for future development. Moreover, once again returning to the multiple programmers point, differing explanations of different parts of the program provide quite some inconsistency in understanding its functionality.

We hope to further push this program to new heights and improve upon all of these issues.

4. Program Testing

4.1 Unit and Method Testing

Each function was tested individually and separately with a variety of inputs to put it through the loops and string it out.

4.2 Integration Testing

The entire program was tested with complex Boolean functions to ensure that all components work in harmony, and there was continuous debugging and programming-time outputs and connection cases between different methods.

5. Build and Run Instructions

- Clone the Repository - https://github.com/kirollos21/DD-Project1
- Simply Compile, and Run
- Circuits and other files are generated as each case and function is simplified.

In summary

All things considered, the application functions as a strong tool for Boolean function minimization with such a powerful and efficient algorithm, and it provides a number of expandable and modulated features, such as the ability to generate Kmaps and render logic circuit diagrams. Based on the Quine-McCluskey algorithm, it offers a precise and effective method for minimizing Boolean functions in a variety of circumstances. However, as the earlier sections have shown, there are some shortcomings in the program and room for development.

Task allocation:

Task 1: Kirollos

Task 2: Kirollos

Task 3: Rawad (a first draft that was written by Kyrillos using a different approach was scrapped and is commented out)

Task 4: Rawad

Task 5: Rawad

Task 6: Mohammad Yahya, with bug fixing by Rawad

Task 7: Mohammad Yahya, Kirollos Zikry (the translation to HTML was by Kirollos, the older code is commented out was written by Yahya)

Task 8: Mohammad Yahya (Kirollos fixed an issue with the HTML and the NOT gate)
Testing, test case runs, and structure: Rawad

Report written by Mohammad Yahya Hammoudeh and Rawad