CSCE 2301

Spring 2025

Dr. Mohamed Shalan

Quine-McCluskey algorithm

A systematic method for minimizing Boolean functions in digital logic design.

Rawan Muhammed, 900232389
rawan_khalid@aucegypt.edu

Doha Nour El-Din, 900232246
doha_deia@aucegypt.edu

Noor Emam, 900222081
Noortaytoy1@aucegypt.edu

Problem Definition

We are making a C++ program that would implement the Quine-Mcluskey algorithm, where our program will parse the file, as inputs, and properly perform the algorithm on the inputs to produce an output. Furthermore, our program contains a comprehensive error handling functionality that would handle any incorrect test cases that could be potentially defined as input files. Furthermore, in this report we aim to discuss the following: the used data structures and STL functions, testing code, the challenges that are associated with the program, and the contributions of each member at the end.

Data Structures Used

1. Vector

Used for storing tokens parsed from input files, where each inner vector contains a list of strings (tokens) related to a specific line or item.

2. Set

Stores unique integers (minterms) and ensures that duplicates are not stored. It's used in grouping minterms and prime implicants.

3. Map

Maps an integer (like an index or minterm) to a set of integers. It's used for storing the mapping between minterms and their corresponding groups.

4. unordered_set<int>

A hash table implementation that stores unique integers. Used for tracking essential prime implicants

5. bitset<32>

Used to represent a 32-bit binary number, converting minterms to binary strings.

Algorithms Used

1. File Input Parsing:

- **Algorithm**: The program reads and parses an input file containing Boolean expressions, which are separated by commas and might include special symbols like "don't care".
- **Approach**:
  - It reads the input line by line.
  - The first line is checked for validity (whether it contains a valid number of bits).
  - Subsequent lines are split into tokens based on commas.
- **Data Structures**:
  - **Vector of Vectors (vector<vector<string>>)**: Used to store all tokens read from the file.

2. Counting and Splitting of Terms:

- **Algorithm**: The terms in a Boolean expression are split based on the + symbol (OR operation). Each term is further processed to remove spaces and perform validation.

- **Approach**:
    - split_terms: Splits an expression string into terms by the + symbol and removes any spaces.
    - count_terms: Counts the number of terms in the expression by checking the number of + symbols and adding 1 for the last term.
- **Data Structures**:
    - **Vector (vector<string>)**: Used to store individual terms from the Boolean expression.

3. Generating Binary Strings for Prime Implicants (PI):

- **Algorithm**: Generates a prime implicant by comparing binary strings of minterms and replacing differing bits with **-** (indicating that the bit is irrelevant).
- **Approach**:
    - generateBinary: For each group of minterms that differ by only one bit, it combines them into a common binary string by replacing differing bits with -.
- **Data Structures**:
    - **Set (set<int>)**: Used to represent the group of minterms.
    - **Vector (vector<string>)**: Stores the binary string representations of minterms.

4. Finding Essential Prime Implicants (EPI):

- **Algorithm**: Identifies essential prime implicants, which are the implicants that cover a minterm that is covered by no other implicant.
- **Approach**:
    - findEssentialPIs: Iterates through all minterms and checks how many groups (prime implicants) each minterm belongs to. If a minterm appears in only one group, that group is marked as an essential prime implicant.
- **Data Structures**:
    - **Map (map<int, set<int>>)**: Used to map prime implicants to their corresponding minterms.

5. Removing Essential Prime Implicants (EPIs) and their Minterms:

- **Algorithm**: Once essential prime implicants are identified, this algorithm removes them and their covered minterms from further consideration.
- **Approach**:
    - removeEPIs: The program iterates through the groups of prime implicants and removes any implicant that is covered by an essential prime implicant.
- **Data Structures**:
    - **Set (set<int>)**: Used to track minterms covered by essential prime implicants.
    - **Vector (vector<string>)**: Used to store the prime implicants (strings).

6. Dominating Columns and Rows:

- **Algorithm**: Identifies and removes dominated columns or rows in the prime implicant chart. A column or row is dominated if it can be entirely covered by another one.

- **Approach**:
    - dominatingColumns: Compares columns of the prime implicant chart and removes those that are dominated by others.
    - dominatingRows: Similar to dominatingColumns, but operates on the rows of the prime implicant chart.
- **Data Structures**:
    - **Map (map<int, vector<set<int>>>)**: Maps minterms to their corresponding prime implicants.
    - **Vector (vector<set<int>>)**: Tracks the final groups of minterms after removing dominated rows or columns.

7. Prime Implicant Reduction:

- **Algorithm**: After the essential prime implicants and dominated rows/columns are removed, the program performs further reduction of prime implicants by checking if any implicant can be omitted without losing coverage of the minterms.
- **Approach**:
    - This part of the program uses heuristic methods like checking the coverage of minterms and their corresponding implicants to reduce the number of prime implicants.
- **Data Structures**:
    - **Map (map<string, set<int>>)**: Stores prime implicants and the minterms they cover.
    - **Vector (vector<string>)**: Stores the reduced set of prime implicants.

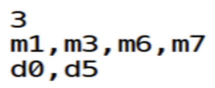
Our program design is split into 4 parts:

1- FileParser Module:

Responsible for reading the input file, and validating whether the file exists or not. It extracts the minterms, the bitsize, and the don't cares by using tokenization and STL functionalities.

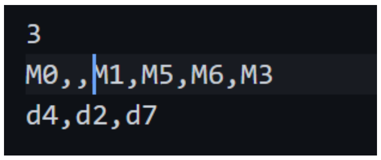
2- Error Handling:

The program that we implemented intends to solve a logic-minimization problem, where it will take a file written in the following format:



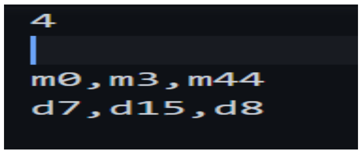
```
3
m1,m3,m6,m7
d0,d5
```

Where the first line represents the bit size, the second line represents all the minterms, and the third line represents all the don't cares. However, if the file input does not strictly follow this format, then our program will give you a file input error, where it will tell you that the input file format is not correct/ following the file format above.



```
3
M0,,M1,M5,M6,M3
d4,d2,d7
```

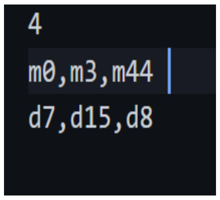
Before our program implements the Quine-Mccluskey algorithm, it checks whether the file format exists or not, then it will check whether the file format is written correctly or not. A function, implemented in our program, called the handleErrors function, does all of the error handling. Our error handling function is comprehensive, and extremely sensitive. However, there are some design decisions that we decided to take while designing this function. Firstly, if the file format example is in the following format:



```
4

m0,m3,m44
d7,d15,d8
```

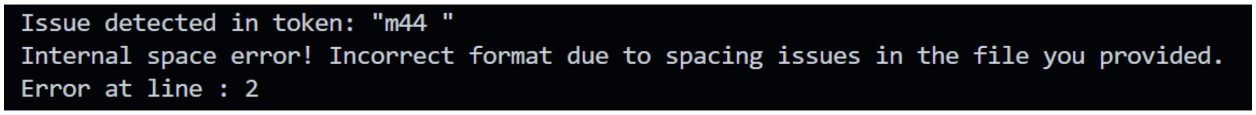
Then, we decided that the program would still work, as our functions are using the tokenize library, which is very effective in separating minterms/maxterms/don't cares based on the commas. Furthermore, the file format can also take this shape:



```
4
m0,m3,m44
d7,d15,d8
```

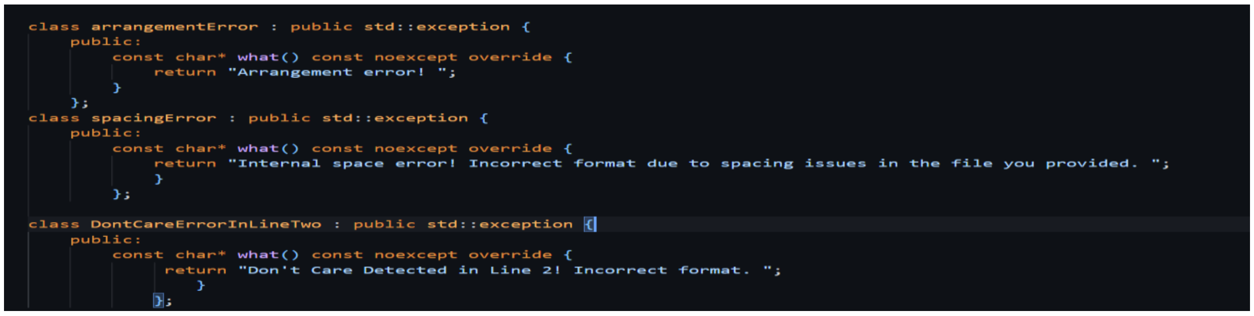
Where there could be a space between the bitsize line/ minterm or maxterm line / don't care line. This is because our handleErrors function uses the getline library, which can ignore any spacing errors between the lines, hence making the program work. However, some of the issues associated with this function, and the challenges we encountered in this program is the sensitivity of this function. For instance, if the user would input in a following file format:

It would throw an error, called the "internal space error," which indicates that there's some incorrect spacing inside some tokens.



```
Issue detected in token: "m44 "
Internal space error! Incorrect format due to spacing issues in the file you provided.
Error at line : 2
```

However, if the file format is strictly following the file format above, then the program will work as planned. This is one of the problems associated with our error handling function, as it is extremely sensitive to any changes in spacing, other than the commas and the empty lines. To demonstrate that our error handling file works, I will put a non-comprehensive list of example incorrect test cases as inputs to our program, to see whether our program can put up with the incorrect format test cases. This will be written down at the appendix. Please note that some of these incorrect test cases were also devised with the help of LLM and Gen-AI tools such as chatGPT and Claude. Furthermore, the ErrorHandle function uses supporting functions like the exception classes seen in the first few lines of the code.



```
class arrangementError : public std::exception {
    public:
        const char* what() const noexcept override {
            return "Arrangement error! ";
        }
    };
class spacingError : public std::exception {
    public:
        const char* what() const noexcept override {
            return "Internal space error! Incorrect format due to spacing issues in the file you provided. ";
            }
        };

class DontCareErrorInLineTwo : public std::exception {
    public:
        const char* what() const noexcept override {
             return "Don't Care Detected in Line 2! Incorrect format. ";
                }
            };
```

Algorithm Modules Overview:

1. **GroupAndCombine Module:**
   Implements the core functionality of the Quine-McCluskey algorithm by grouping minterms and recursively combining them to form prime implicants.
2. **PrimeImplicant Class:**
   Represents a binary implicant with its coverage and whether it has been used in further combinations.
3. **Essential Implicants Selector:**
   Extracts all essential prime implicants, ensuring all minterms are covered. For any uncovered ones, it invokes the Petrick method.
4. **PetrickMethod Solver:**
   Implements a symbolic multiplication-based approach to handle multiple combinations that can cover the remaining minterms.

Testing:

The primary objective of testing was to ensure the program's correctness, reliability, and performance. The tests were designed to ensure the algorithm's functionality by verifying that all the functions behave as expected under normal and edge-case scenarios, and boundary conditions by testing extreme or unexpected inputs to check for robustness and proper error handling. Also, the testing process included unit tests for each function, including minterm grouping, binary conversion, prime implicant generation, essential prime implicant selection, removing correct dominating columns and dominated rows, Petrick algorithm, and finally conversion to lettered expressions, including all possible minimized expressions. All the tests are conducted using Google tests, and the expected output in the Google test is generated by this online solver https://geeekyboy.github.io/Quine-McCluskey-Solver/ , after verifying that this website follows the correct steps to reach this final expression. The implementation was tested on a range of input sizes from 3-variable to 8-variable Boolean expressions, but it is designed to solve up to 32-variable Boolean expressions. To test that the algorithm works as expected in both cases of maxterms and minterms, some tests include maxterms, while others include minterms.

Experimental Results:

The implementation successfully minimized all tested Boolean functions using the Quine-McCluskey method. The tests conducted on the project demonstrated that the system generally performed as expected, and results were consistent with expected outputs in all test cases, confirming the correctness of the algorithm. For the results, we managed to present all the minimized expressions in the standard Sum of Products (SOP) format to facilitate the process of generating Verilog code for the final expressions. Thus, the outputs mainly present all minimized Boolean expressions for the function in addition to Verilog code for each of these expressions.

Challenges and Potential Improvements:

- Petrick's Method: Translating Petrick's method from theoretical form into executable code was non-trivial. Boolean expression simplification in symbolic form required nested loops and a lot of debugging time to debug logical errors that initially existed in our implementation of the algorithm.

Problems in Our Program

One key challenge was handling Boolean expressions provided in maxterm form. To handle them, we have excluded the maxterms and don't care about the total number of possible combinations which is (2^n), where n is the number of variables. Supporting up to **20 variables** is required, which means working with up to: 2^20=1,048,576. So, storing and checking this many combination efficiently was a major concern because this leads the time complexity to be O(2^n+(m+d)). If we use a hash set, it will be O(2^n).

How to Run the Program:

Requirements: *C++ compiler (e.g., g++) , Terminal or IDE (like Clion, Visual Studio or Code::Blocks)*

Clone the repository, present at the following link: https://github.com/CSCE2301/dd1-s25-project1-prime-implicants.git Then, go to the directory where you cloned this link, and simply just compile and run the following command: g++ -o quine_mccluskey main.cpp, then ./quine_mccluskey <input.txt> where this input.txt would be the name of the file that contains the bitSize, minterms, and the don't cares. The file format and error handling is discussed in more depth in the functionalities part.

Contributions of Group Members

Doha: I implemented a function to read input from a file, then I classified the terms of the input into minterms and don't care terms. After that, I converted these terms into binary expressions. I also generated prime implicants after classifying terms according to their number of ones. I also implement the function of a function to remove essential prime implicants, in addition to a function for the Petrick method to get all the possible minimized expressions. Then, I generated all possible minimized final expressions after converting them to letters. I also contributed to the first part of handling the Verilog module for all final expressions. Finally, I fixed some errors in the code to correct the final expressions to the expected ones.

Rawan: I was responsible for handling the conversion of Maxterms into Minterms and ensuring that the logic for processing the Maxterms was correctly implemented. I also worked on detecting the Essential Prime Implicants (EPIs), developing the algorithm to accurately identify these from the list of prime implicants. Additionally, I contributed to the Bonus section by implementing part of the Verilog module generation. Furthermore, I improved the program's readability by organizing the code into modular functions, making the code more structured, easier to maintain, and simpler to debug. In addition, I implemented the automated test to automatically test all the test cases at the same time.

Noor: Implemented the error handling function, designed the domination column and rows, and integrated it into the main function. Designed the correct and incorrect test cases, and did a part of the bonus, involving outputting and mapping wires as outputs coming from AND and OR using Verilog primitive gates. Also, contributed to fixing the errors in the code when the output is different from the expected one, especially edited the dominations.

References

We used different resources to implement this algorithm. The main source we used is the slides for Dr. Shalan, we followed the steps of the algorithm written in these slides to implement our code. We also used a video to understand the Petrick method, so we can implement its function. The link to the video is this, https://www.youtube.com/watch?v=EkgPsTH4kLU.

Acknowledgment of AI Usage

ChatGPT was sometimes used to check the accuracy of some functions, especially the dominations, when the output is different from the expected one, and it highlights to us what may be wrong in the function logic, then we edited the function accordingly. It was also used to understand how to do automated tests without using Google tests. One of the used prompts is "In Clion, how to automatically test all the output of the 10 input text files". We used this prompt to only understand the way of doing the tests, but we didn't use it to get the code.