Quine-McCluskey Logic Minimization

Mohamed Ahmed, 900211305
Andrew Aziz, 900213227
Raef Hany, 900213194
Dr.Mohamed Shalan

March 18th, 2023

Program design:

There are three main constituents to the design of the Quine-McCluskey Logic Minimization program. The first part relates to understanding the function of the program and what the main target or desired result should be. Following that stage comes the process of breaking down the program into smaller functions that will eventually constitute an overview of the flow of data and process through which the final result is achieved. Finally, creating test data that will ensure each area of the program performs as desired. In the following section each constituent of the program design will be discussed in detail.

1- ***understanding the program:***

In this initial stage of the design, a clear understanding of the Quine-McCluskey algorithm is vital. Consequently, the inputs of the algorithm were established (in our case an Sop boolean function) that would be entered by the user. The program was then required to apply the QM algorithm to obtain several output results. To clarify further, the program needed to process inputs made by the user to create the corresponding truth table of the boolean function. This would then be utilized to identify the minterms and group them in the implication table (grouping of minterms is done according to the number of ones in the binary representation of each minterm). Following the creation of the implication table, every 2 consecutive groups would have their implicants compared against each other. This is done to identify if they can be grouped together or not(i.e helps identify if the implicant at hand is prime). After the implication table is complete, the Prime implicants need to be extracted and placed in the coverage chart, this would then be used to extract the essential prime implicants. Finally, after the described processing the desired outputs of the program would be acquired. Through this breakdown of the IPO of the program, a clear and precise understanding of the requirements is achieved, thus allowing us to proceed to the second phase of the program design.

2- ***program breakdown:***

In the second stage of the program design, the program structure, functions, and data structures are identified to create the flow of information and steps throughout the program. In this implementation of the Quine-McCluskey algorithm, a class was created to handle all instances of boolean functions entered. The main data structures used throughout the program are maps and vectors. Maps were utilized to help aid in the process of organizing implicants and creating tables according to specific key values. This would ensure the simplest method of applying the QM algorithm. Furthermore, maps possess low computational complexity when accessing elements, which would help bring down the overall complexity of the program.

After selecting the main layout of the program and data structures to be used, it was clear that the algorithm is interdependent and relies on outputs of previous stages,Through the analysis carried out in the previous stage of design, thus this would dictate which steps need to be carried out first.

Initially, The function would need to be validated to ensure that it's in the required form (SOP) and that only the allowed parameters were entered. (details of the validation function are discussed later on in the report). Following the validation of the boolean function a truth table would need to be generated, for this 2 functions were designed. The first is to generate the minterm values (i.e m1=0001) and the second is to generate the output of the corresponding minterm. The output of both functions would be a map corresponding to the truth table of the function entered. Following this stage, the canonical SOP and POS need to be extracted. To perform this step, 2 functions were created where each function would

utilize the truth table generated in the previous step to obtain the desired output (i.e SOP function would identify the minterms of the function while POS would identify the max terms). Following this analysis, the result would be concatenated into a string and displayed to the user. To obtain the prime implicants and generate their binary representation alongside the minterms they cover, 2 functions were assigned to this task. The first would be a utility function that compares strings (binary representation of minterms were treated as strings within the program and extracted from the truth table) and the second function would be the main function that generates the Prime implicants. In this main function, two fundamental phases take place. The first phase would group minterms according to the number of ones and form the implication table, while the second stage would initiate the comparison phase of each group to help identify the prime implicants. After the prime implicants are extracted and saved in a map, this is then passed on to the EPI function responsible for finding the essential prime implicants. The process of identifying said essential implicants was made much easier through the use of maps and the way they are organized.

Overview of the program flow:

1- user inputs boolean function
2- boolean function is validated
3- truth table is generated
4- canonical SOP and POS are extracted
5- Prime implicants are identified
6- essential prime implicants are extracted.

3- ***Test data:***

To ensure each function of the program was producing the correct output rigorous testing took place. After the completion of each function, sample test data with a known output was provided to the function and the output was monitored. Test data accounted for special cases and ensured that the program was able to deal with all possible scenarios. This process was repeated multiple times at different stages of the program to ensure that communication between the various functions and data structures was operating as desired. Furthermore, incorrect outputs or failure to handle specific cases helped aid the debugging process and the allocation of errors throughout the program.

This phase was pivotal in verifying the functionality of the validation function, where a multitude of cases need to be taken into consideration and the function must be able to handle all such instances. Examples of test data included entering undefined operators such as @,$,% into the boolean function alongside numbers and double barred variables (eg.A``+B```. The program is able to identify that the expression is equivalent to A+B`).

Checkers needed to validate the input:-

Many checkers are needed to validate the input, and we put all these checkers in our validation function. Firstly, our validation function checks whether the ‘+’ sign exists at the beginning of the boolean expression or the end, and it also checks if there are two ‘+’ signs that come after each other. So, if any of the previous cases happens, our validation function will detect this as an error. However, if all of these

cases do not happen this means that the '+' sign exists in its correct position in the boolean expression. Secondly, it checks if the not sign (` ) exists at the beginning of the boolean expression or not. If it exists at the beginning, then our validation function will detect this as an error. Otherwise, the position of the not sign ( `) is correct in the boolean expression. Thirdly, it checks if the not sign (` ) comes after the '+' sign directly in the boolean expression and detects this as an error. Thus, all of the previous cases mainly focus on the position of the '+' sign and the not sign ( `) in the boolean expression. It will check whether they are at their correct position or not in the boolean expression.

Furthermore, our validation function also checks if there are any characters other than '+', not sign (`), and literals (alphabetical characters) in the boolean expression and detects this as an error. These characters may be a space in the boolean expression, a number, or any character which cannot exist inside a boolean expression. If our validation function detects any error of the previous errors in the boolean expression, it will require the user to enter another boolean expression, and the validation function will check this boolean expression again.

Instructions for how to build and run the program:-

We created a new C++ project and named it QM implementation. Then, we added a new header file to the project and named it bool_function.h. This header file contained the definition of the class that we created, which is called bool_function. We put the constructor to be public in this class, and we put all the functions that we need to accomplish the tasks of this project and some data structures that help us store the data that we need throughout the class to be private in this class.. We then add two source codes where one of them is the main, and the other one is the .cpp file where we put the implementation of the functions of the class in it, and we named this file bool_function.cpp.

It is very significant to call the functions in the constructor in a specific order. We first called the validate function, which will take the boolean expression from the user and validate it to check if it contains any errors or not. If it does not contain any errors, it will store this boolean expression in a string. Moreover, it will also store the literals of this boolean expression in a vector and a map which are very important for the other functions. Thus, the validation function must be called first. After calling the validation function, we call the function that will generate the truth table without its output. This function will use the vector that we create in the validation function to set the size of the columns of the truth table, and if we know the size of the columns, we will be able to set the size of the rows.

After that, we call the function that will generate the output of the truth table and store it in the truth table. This function mainly depends on the values of the truth table that were generated in the function that only generates the truth table without its output. Moreover, this function also depends on the

string that we store the boolean expression in it in the validation function. After calling all of the previous functions, it is very clear that our truth table is generated successfully with its output, so we call the function that will print it. We also call the function that will print the canonical sum of products and the function that will print the canonical product of sums. These two functions mainly depend on the values of the truth table that we generate and on the vector that we create in the validation function and store the literals in.

Then, we call the function that will generate the prime implicants and the minterms that they cover, and we put them in an unordered map where its key is the binary representation of the prime implicants, and the value is a vector of minterms covered by it. This function depends on the values of the truth table to be able to make the implication table. Moreover, we create an unordered map called minterms in this function where its keys are minterms (0,1,...), and its values are bools expressing the minterms that made the output of the function equal to 1. This map will be very beneficial for us afterward because it will enable us to know the minterms that are not covered by the essential prime implicants.

Finally, we call the function that will generate the essential prime implicants making use of the unordered map that we create in the function that generated the prime implicants. This map will enable us to detect the minterm that is only covered by one prime implicant. Then, we store the boolean expression of the essential prime implicants and the minterms that they cover in an unordered map. Furthermore, we will use the unordered map, which is called minterms that we created in the function that generated the prime implicant and made the value of the minterms covered by the essential prime implicants to be false. Thus, we will be able to print all the prime implicants that are not covered by the essential prime implicants.

contributions :

Mohamed : created the truth table generation and printing functions
Andrew: created the POS,SOP,truth table output functions.
Raef: Validation (with additions from Mohamed and Andrew).

Mohamed, Andrew, Raef: EPI,PI functions.