Project 1 Report: Quine-McCluskey Logic Minimization

Andrew Antoine - Mostafa Elfaggal - Kirolous Fouty

The American University in Cairo

CSCE 230/2301 - DD1-SP23

Dr. Mohamed Shalan

16/03/2023

Author Note

This paper is prepared for the CSCE 2301 course instructed by Professor Mohamed Shalan

Mostafa B. Elfaggal
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
mostafaelfaggal@aucegypt.edu

Andrew A. *Ishak*
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
andrew625@aucegypt.edu

Kirolous A. Fouty
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
kirolous_fouty@aucegypt.edu

OUTLINE

Project 1 Report: Quine-McCluskey Logic Minimization

I - Introduction

This program is an implementation of the Quine-McCluskey Logic Minimization algorithm in C++. It was implemented in a set of a few classes; each one executes a specific independent task to ensure the program's functionality. The following program was divided into five major sub-problems. The first one is responsible for reading the inputs as a string and extracting all the needed data used by the program, validating all types of errors that could be inputted. The second sub-problem consists of creating the truth table of the function and generating the canonical SoP as well as the canonical PoS. The third sub-problem is about computing and printing all prime implicants; moreover, showing the minterms covered in binary representation. The fourth task includes obtaining and printing all the essential and non-essential prime implicants. Lastly, the program contains generating the minimized boolean expression of the function. Such a program is designed to be efficient for any number of variables and operate irrelevant of the variable count; however, it has yet to be tested beyond ten variables, and time spent to get optimization of the function cannot be guaranteed.

II - Classes

A - The Boolean Function Class

1) Overview of Class Design

The Boolean Function class is mainly designed to describe a boolean function. It could be generated either through its usual constructors, where we define minterms and don't care if they are described, or by providing an SoP string for the SOPString class to handle and generate the necessary data for the Boolean Function class to operate. Many other classes make use of the Boolean Function class, where this class is essentially the backbone of the project. To begin with, all terms in the program could only be set to a value of 0 or 1, or X(a don't care term); thus, we defined an enum named 'boolean value,' which consists of OFF, which represents the Value of 0, ON which represents the Value of 1 and X which represents the Value of the don't care term. The BooleanFunction class has three private variables: a vector of strings named 'variables' to list the names of all the variables listed in order of significance, a vector of Boolean Value called 'terms' that behaves like the truth table such that the ith element is the ith row in the truth table and has its Value of Boolean Value, and a string named expression to save the SoP expression that was imputed by the user or by any in computation in the program.

2) The Constructors

Moreover, the class has eight overloaded constructors, all designed to take many different combinations of inputs from the user, which adds ease and flexibility to the program usage. These constructors can take as parameters: an integer variable as Count which explains how many variables are there in the function, a vector of strings as Variable Names to specify the names of each Variable such as {"A", "B", "C", "D"}, a vector of integers as Minterms, so we can specify directly the minterms numbers and finally a vector of integers as Don't Care terms. These constructors can take as parameters any combinations of these variables. The primary constructor used in the program takes a string as 'SoP' such as BooleanFunction f("A+B'+C"). Each constructor of them is designed to take as many inputs from the user as possible and to initialize some other variables in the program that are necessary for the program functionalities, such as the "setVariableName()" function, the "setMinterms()" function, the "setTerms()" function; all these functions are only used in the inner functions of the class.

3) The Operators

Moreover, the class contains three operator functions, two of which are created to compare functions with each other, the "==" and "!=" operator functions, and returns a value of true or false. The third operator function, " [ ] " takes an index as a parameter and returns the boolean Value from the truth table from the terms vector. All index-parameterized functions validate the input; of course, the <stdexcept> library is included in all program classes to throw exceptions whenever needed.

Furthermore, the class contains setters and getters functions to ensure smooth inner workings of the program, such as: "getTermsCount()," "getTerms()," "getMinterms()," "setTerm()," etc...…

4) The Printing Functions

The second central part of this class consists of printing which is divided into four functions: printing the truth table itself, printing the truth table letters, printing the SoP, and printing the PoS. The primary function "printTruthTable()" mainly uses another function called "DECtoBIN," which stands for decimal to binary. The such function uses the term count Variable to know how many terms there are and uses it as the number of inputs. Then, it loops till 2^(inputs) to generate the number of all the possible combinations of inputs. Last, it converts these decimal numbers to binary numbers to be shown in the truth table. Printing the letters function extracts from the expression variable all the different variables and whether they are followed by a bar or not to know if they are inverted. It can also use all the overloaded constructors to check the inputted letters immediately. Then, it follows these letters with their order of significance to be printed in the truth table.

B - The SoP String Class

1) Overview of Class Design

The SoP String Class mainly constructs and validates the data imported to the Boolean Function Class. It constructs a string that represents the Sum of the Products string. The class proposes a variety of expression options to the program. This class has two jobs: validating the SoP string imputed before parsing it and extracting a boolean function. In its private section, it has variables to handle the input expressions such as a string "expression" variable, a string "expression_without_spaces" variable to handle the spacing in the inputting phase, a vector of strings "expression products" variable which breaks out the imputed expression whenever it meets a "+" sign to construct the separate product. Objects of this class can, using its functions, add new variables, minterms, and products, ensure the validity of the desired string, ensure that a given expression term is a product, and ensure that the alphabet given in an expression term is safe to use or not. An object of this class has a vital function, prepareFunctionsData, which focuses on extracting the minterms and adding them to the set of minterms. In order to properly maintain an object of this class, some helping functions help by returning the variable count, variable names, or even minterms. The function convertProduct also takes in the product as a string and returns a vector of boolean values accordingly. It also indicates whether this expression is zero or not using the variable isZero that is passed by reference in its parameter, which will be needed in a case like "ABA'," which is 0.

2) Validation is done through the following steps:

1- Ensure that all characters in a given string are ' '(space) or '+' or a lowercase alphabet letter "a b c … z" or an uppercase letter "A B C … Z"; spaces are also stripped in this step from the string
2- When the validation encounters a letter, uppercase or lowercase, it adds it as a new variable to the list of variables; thus, the class takes the letters not alphabetically but based on their order of occurrence.
The list of variables is handled in a set to avoid repeating variable names.

3- Following validation over products takes place, where the string is broken down at every '+' to represent every product. Then each product is validated such that no product starts with a " ' "and is not an empty string itself, which would indicate that the original string contained "++."

3) Post validation, the string is then converted to the boolean function class data members:

1- Every product is individually converted to a representing cover "1,0,-"
2- Through a recursive function is converted to a vector of integers (minterm indices) to be handed over to the boolean function class
3- During the conversion, some products could be evaluated as =0, such as when the product is "ABA'" which is equal to 0 and does not affect the function's minterms accordingly
4- The conversion process is done by converting every index which is 1 in the cover representation to 2^index and accumulating their Sum. For Instance, "10010" is equivalent to 2^4 + 2^1 = 18.
However, note that the cover contains "-" as well; thus, after converting the "1", we convert all possible combinations of the "-" to generate all possible Minterms of that given cover. For Example, "10-1-" is 2^4 + 2^1 + combinations = 18 + combinations, which amounts to 18, 19, 22, 23
5- The generated minterms are combined and placed into the boolean function class

C - The Implicants Class

1) Overview of the Class

The Implicant class describes an implicant, a cover in the Kmap system or the QM. An *implicant* is a function that describes a portion of the minterms of the original function. Thus, we have the class Implicant, which inherits from BooleanFunction and adds the needed data members and methods to describe an Implicant. An instance of this class usually contains the same set of variable names its original BooleanFunction has. The Minterms it represents are naturally the terms it covers. Its data members include a vector of booleanValues which is the cover, {1, 0, X, 0, etc.}, and two bool values representing whether the implicant is prime or not and whether it is essential or not. That is because later down the line, we found the need for them in QM, but we felt it was optional to separate them as their own classes.

2) The Setters and Getters Functions

The class has setters and getters for its data members, including getting the number of 1s in the cover (needed for the QM part 1 to sort in the right groups). It can also convert a cover to its respective Minterms used for specific constructors. The constructors include many different combinations of potential inputs. There is also a constructor which takes a single minterm index and converts it into its respective cover values (binary representation), which is used in generating the first column of the QM primes table.

The class can also compare itself to another instance where they can be merged if they have the same indices of don't cares and a difference of one 1. The compare function reduces the bit at which the single change takes place. Otherwise, it returns -1 if the two instances are not mergeable. Some operators, such as == and !=, were overloaded to compare two implicants based on their cover representation equality. Operator < was overloaded to compare Implicants in a way that both considers the number of minterms covered as well as which minterms. That operator was needed for data structures that made use of ranking for Implicants.

There is also a print function to produce the following format for a given implicant; for example, “10-1-(18, 19, 22, 23)”. Further, there is also toString to produce the product string, which looks like this for the example above “AB’D,” assuming the variables were A, B, C, D, E in the same order of significance.

Later we found the need to use unordered_set<Implicant>, which raised the need to overload the hash function for the Implicant class, which was built upon the hashing of the string representation of the cover.

D - The QM Class

The QM class is constructed to be composed of the other classes previously defined. It is an abstract class because it is only composed of a constructor that takes the function as an input and two void functions: “implicantsTable()” and “primesTable().” It is designed this way to abstract the complexity of the algorithm and breaks down its functionalities into smaller parts. It has in its private part only two data members: a BooleanFunction and a vector of Implicants, which is a defined data member used to store prime implicants named “PIs.” This class mainly calls the two classes, QM Implicants Table and QM Prime Implicants Table, in its two functions to start generating the tables and computing the rest of the algorithm. It also collects from both of them their outputs, such as the Prime Implicants, which are collected from the first table (Implicant to Primes table), and the Essential Prime Implicants as well as the non-essential minterms, covered by more than one prime implicant, from the second table (Primes to Essentials and Non-Essentials)

E - The QM Implicants Table Class

The QM Implicants Table Class is used to build the first table, implicants to the primes table. This implicants table is a vector of columns where each column is a vector of ImplicantGroup_QM_ImplicantsTable, which is an unordered set of Implicants. The fundamental function of this class is the function computeColumn. This function takes the column index as its parameter and checks if it is within range. Building a column has two prominent cases. The first case is that it is the first column, which is of index 0; in this case, the column is built from scratch by using the BooleanFunction’s getTerms, and if a term’s output is ON or X, then it is considered a minterm and is inserted into the prime implicants group and is also inserted into the corresponding group in the table. By the corresponding group, it is meant that it corresponds to the number of 1’s in the Implicant. Furthermore, the second case is that the index is within range and not 0; in this case, the program refers to the previous column and starts comparing each Implicant with the implicants of the following group in that same column, and if they are mergeable, their attribute is changed to be set as a non-prime implicant instead of a prime one, then they are merged into a new implicant and inserted into the desired-to-be-constructed column in the proper group of 1’s. For this step, there is a function called isMergable that returns a boolean value according to the fact of whether the two given implicants can be merged or not. In this class, a function called “Compute” calls the function computeColumn to construct all the columns of the desired table. All Implicants are initially assumed to be prime when being constructed, and only later, in case of being merged, will they be set to being non-prime. To collect the primes from the table, we use an unordered_set< Implicant> to represent them externally from the actual table, where any new Implicant is set to be prime and added to which. Later if it merges with some other implicant, it is removed from the set; eventually, as the entire table is computed, this set would hold the real primes. This set of primes could be exported out of the class as a vector through a getter function.

F - The QM Prime Implicants Table Class

The QM Prime Implicants class' main functionality is to prepare the table of minterms and implicants, find essential prime implicants and the non-essential prime implicants, and computes the dominating covers. The class is broken down into a few tasks. Firstly, it creates the table as a vector of vectors of bool values, having a false value as a no cross and a true value as a X.
The class has a few helper functions such as removeRow, and removeColumn which take an index of a row or a column respectively and eliminates it from the table, as well as the PI or minterm from the data member which represents them in this class.
Secondly, the class has two void functions that are responsible for searching and analyzing the newly created table to find the essential and the non-essential prime implicants.
Essentials are found by looping over every column (minterm) and within that column if there is only one X, then that row is an Essential Implicant. Accordingly, the class removes that column and any other column this row has an X at which. Afterwhich, we remove that essential row itself.

Essential prime implicants were collected as they were found into a set of essential prime implicants named (EPIs). Before computing the rest of the table, we extract the non-essential prime implicants which are all implicants left in the table into a set named NonEPIs. Finally we collect the minterms not covered by essential prime implicants, which indicates that it was covered by two different prime implicants, at least, and collect them into a vector named NonEPIMinterms.
These three data structures are later exposed to the calling QM class.

To find non-essentials, we apply the 3 step heuristic algorithm, where we compute column and row dominations. Then find sub-essentials (essentials in the simplified table), or pick a random implicant (first implicant) in case all implicants are equally valuable, and after which repeat the steps to reach a new implicant to pick.

The denominations are done through two functions each of which loops over every column/row comparing to upcoming column/rows, so as to identify dominations based on the Xs. If dominations are found, dominating columns/dominated rows are removed.

Finding essential implicants at this step, is done by passing a false value parameter to it to represent that we aren't looking for actual essentials but sub-essentials so as to not include them in the set of essentials which would be exported later to the QM class.

G - The Tester Class

This class was used for testing purposes. A struct called TestCase has an input, expected result, and expected outcome. The expected result is whether it is expected to run successfully or fail; the expected outcome is the actual output. The void function Tester takes a vector of test cases as its parameter and loops over them all to test and displays whether there was a failure or a non-matching outcome.

Project 1 Report: Quine-McCluskey Logic Minimization

III - Problems Faced

None to be noted so far 😀

IV - Building and Running the Program

The program has been run using the g++ compiler. The command is as follows (assuming the files are the only cpp files in the working directory):
g++ *.cpp -o main
The previous command complies, to run the program use "./main.{extension}" where extension will be replaced by the appropriate exteSOPToFunctionnsion per operating system:-
For windows: "./main.exe"
For linux: "./main"
For mac: "./main"

When running, you will be prompted to either pick "c", to input a custom sop string input attempt, or pick "p" to run the predefined test cases.
In the custom input option, type your sop string in the terminal, if there are any errors in the input, the program will crash with an error of invalid input.
In the predefined input option, the program will run each test case and ask you if you wish to continue before running the next test case. To continue to the test case press enter.

V - Contribution

Mostafa: BooleanFunction, Implicant class, SOPString class, and overall code clean up
Andrew: Automated Testing, QM Primes Table (part 2), and overall code bug fixing
Kirolous: BooleanFunction, QM Implicants Table (part 1), and Report