**CSCE2301 – Digital Design I**

**Spring 2023**

**Project 1:**

Quine-McCluskey Logic Minimization.

**Group members:**

Nadia Yassin, Adham Gouda, Ziad Amin.

**Quine-McCluskey Algorithm**

**Introduction:**

The Quine-McCluskey Algorithm is a method of simplifying Boolean algebra expressions. It was developed by Willard Van Orman Quine and Edward J. McCluskey in the early 1950s. This algorithm is widely used in computer science and engineering to simplify complex digital circuits. In this report, we will explore the Quine-McCluskey algorithm and its implementation in C++.

The Quine-McCluskey algorithm is a method of simplifying Boolean expressions. A Boolean expression is an expression made up of Boolean variables (variables that can only take the values True or False) and Boolean operators such as AND, OR, and NOT. A Boolean expression can be represented using a truth table, which lists all possible combinations of inputs and outputs.

In order to identify pairs of minterms—terms that contain all of the expression's variables—that differ by just one variable, the Quine-McCluskey algorithm compares pairs of minterms. Until no more pairs can be joined together, these pairs are combined to create a new term. As a consequence, the expression is simplified and has the fewest terms possible.

Quine-McCluskey method implementation in C++ is a multi-step process. The first stage is to generate a list of minterms in the expression. By building a truth table for the expression and finding the rows where the output is True, this list can be acquired.

The next stage is to group the minterms based on the number of 1s in their binary representation. For instance, a minterm falls into the category of minterms with two 1s if it has two 1s. 'Grouping' is the term used to describe this action.

The third step is to compare the minterms within each group and identify those that differ by only one variable. These pairs are combined to form a new term.

In the fourth stage, the new terms are subjected to another grouping operation. Similar to the prior grouping operation, this one takes into account the amount of 1s in the new terms' binary representation.

The procedure of comparing and combining is repeated in step five until no more pairs can be combined. When the algorithm reaches this point, a simplified expression is produced.

There are several advantages to using the Quine-McCluskey algorithm in C++. Firstly, it is a very efficient algorithm and can handle expressions with hundreds of variables. Secondly, the algorithm guarantees that the output is the minimum number of terms required to describe the expression. This can help reduce the size and complexity of digital circuits, resulting in faster and more efficient processing.

In conclusion, the Quine-McCluskey algorithm is a powerful tool for simplifying Boolean expressions. Its implementation in C++ involves several steps that are designed to identify pairs of minterms that differ by only one variable and combine them to form a new term. The resulting expression is guaranteed to be the minimum number of terms required to describe the input expression. Therefore, it is an important algorithm for any engineer or computer scientist working with digital circuits or Boolean algebra expressions.

**Validation function:**

For the first part of our project code, we had to write a function that takes in a Boolean function from the user and validate that it is in Sop form and that it only takes in letters from ‘’a’’ to ‘’z’’ as variables. The function could take 10 variables as input and validate it. First of all, we check if the string is empty or not, if it is we tell the user that the input is invalid. However, if the string is not empty we then proceed to check if all the characters in the Boolean function string are valid or not, meaning that they are a letter from a to z. The letters could either be small letters or caps. In addition, the string could also contain the ‘+’for the or, the “ ‘ “ for negation, th ‘^’ for xor, and “ . ” for the adiidtion. If the input given is out of the acceptable characters the function returns that the function given by the user Is invalid. We added a case for duplicated variables or operators so to avoid repeatition, for example: if a.b.b++c it is going to be a.b+c

Some examples of inputs that are valid:

-ab’c+bc

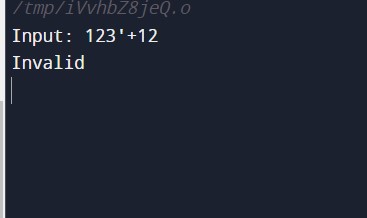
-b’cd’+cde

-a’b’cd+abcd

-a

The following two screenshots are for the

Accepted and invalid inputs.



The past valid test cases are valid for several reasons, the input is English letters from a to z making the characters in the acceptable range of character input. To add, the plus sign and the negation size are also accepted in the input and nothing else available is out of the range of acceptable characters. Spaces are also accepted when entering the Boolean expression in the function.

Some examples of inputs that are not accepted by the function:

-a32+def’

-(a+b+c)(a+c)

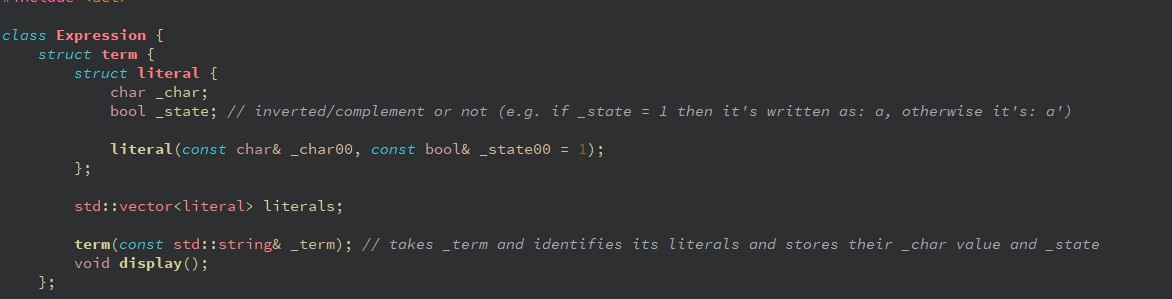
-(a)(b)

-123’ + 12

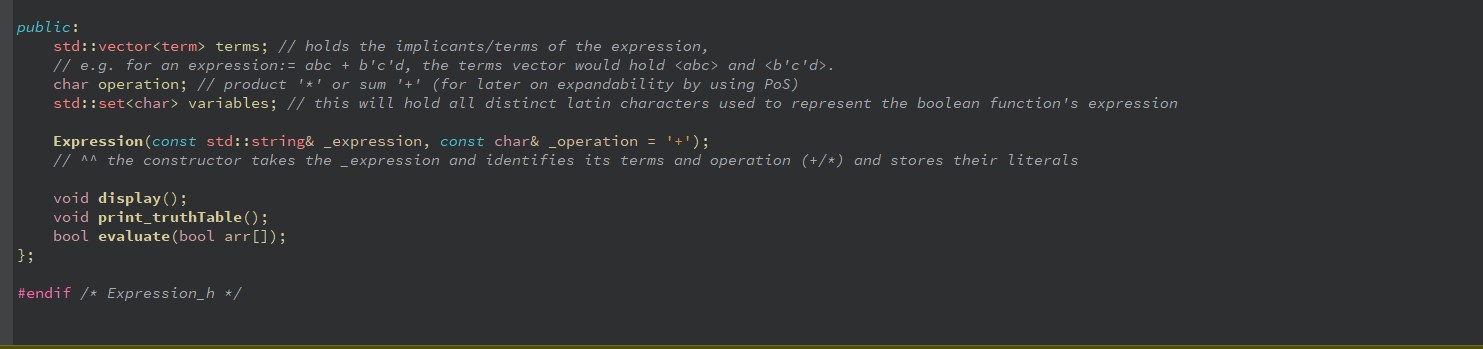
The invalid test cases demonstrated above are not accepted for several reasons. First of all, in the first test, the input contained numbers that are not accepted as input. Furthermore, “(a+b+c)(a+c)” is also not accepted as the function only accepts functions in the form of Sop and not Pos. “(a)(b)” is also the same case as the one before it.

Finally, 123’+12 is also not accepted as input due to the characters being not in the accepted range.

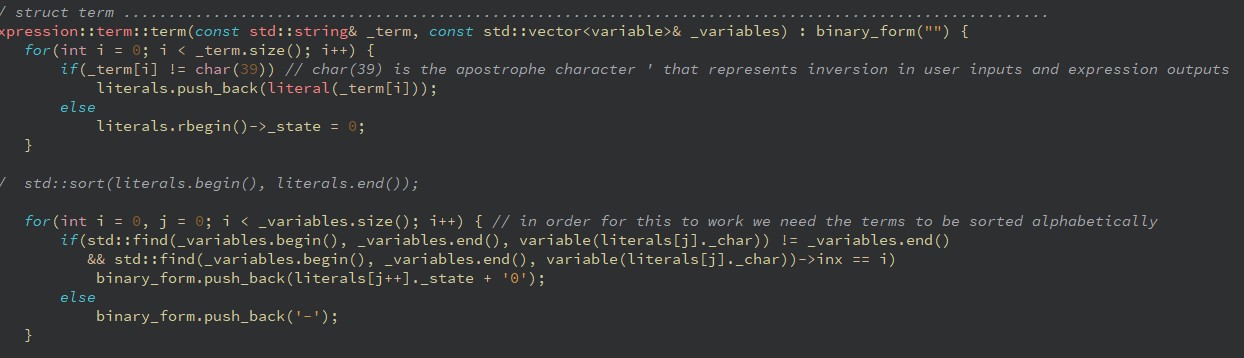
Here is a screenshot of the function.



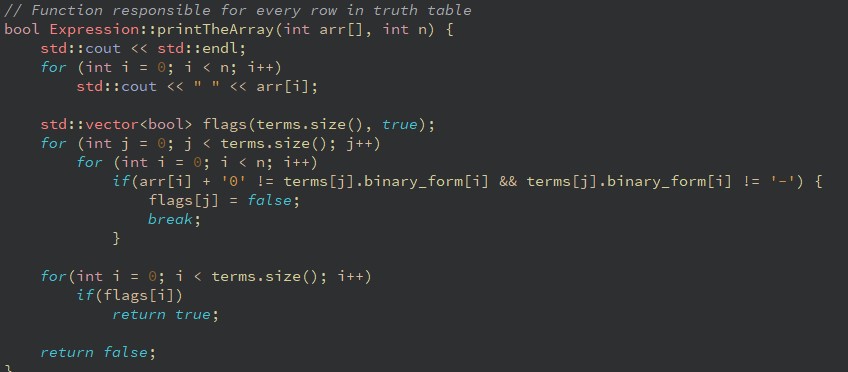
We have a class called Expression where we have struct term which also has struct literal. Inside struct literal, we have the character and the state of every character. Meaning that if the character has a negation sign beside it for example the state of the character which is stored as a bool will be 0. However, if there is no negation sign the character’s state will not be affected and it will stay as one which is the default because we initialized the state to one if there is no presence of the negation sign. “term(const std::string& \_term);” it takes every term and then identifies every literal to then store the characters along with their state (0 or 1). We then have a display function to print the information.



In the part above of the code we can see that we have an Expression constructor that takes in the expression entered by the user as a string and ignores the spaces to take all the characters entered whether they are terms or operations like addition or multiplication. We also have variables that will store the Latin characters and their index as they are input by the user. So for example, if the user inputs “a’b + cd” a will have an index of 0, b will have an index of 1 followed by 2 and 3 for c and d respectively. Moreover, we also have “char operation” that will help us later on when coding the parts of the Pos and Sop. It also helps as some sort of detection to the number of variables for when we create the rows for the truth table by performing for example (rows = 2^ number of variables).



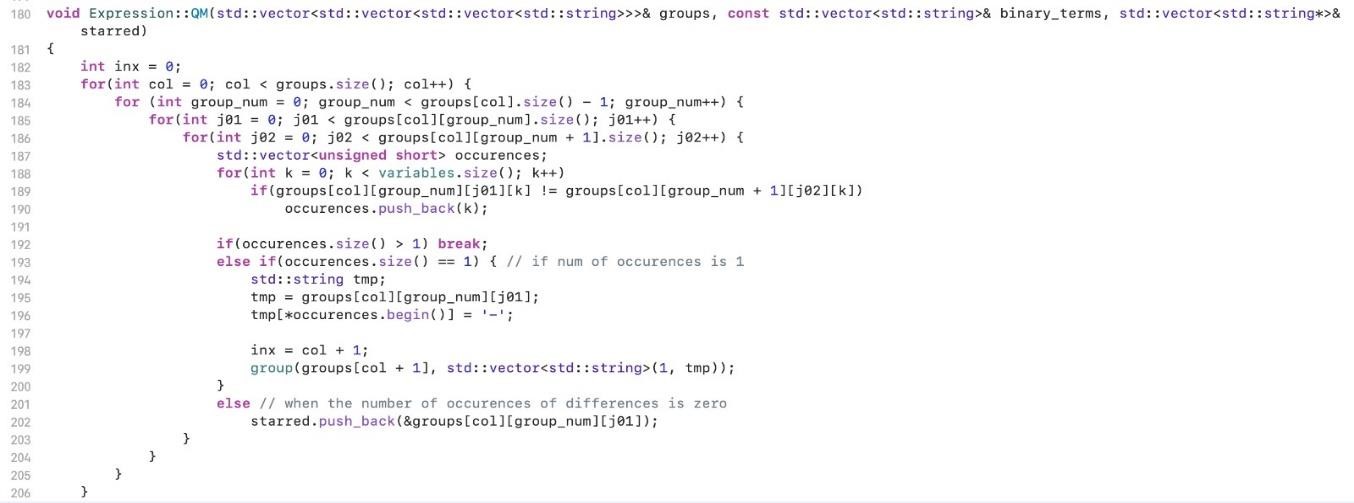
Here we added the attribute called binary form, this binary form will store the form of the binary representation and it’s a string that stores either 0,1 or a dash ‘-‘. So for example if we have the term c’, the binary form stored will be - - 0. So that we can compare between the binary representation and the binary string in the truth table.



Here in print the array we have an algorithm that compares between the binary form and the binary string permutation in the truth table. The output then taken into the generate all binary string function.

The updated constructor for term, is responsible for translating the term into a binary representation which could be found in the block of the for loop.

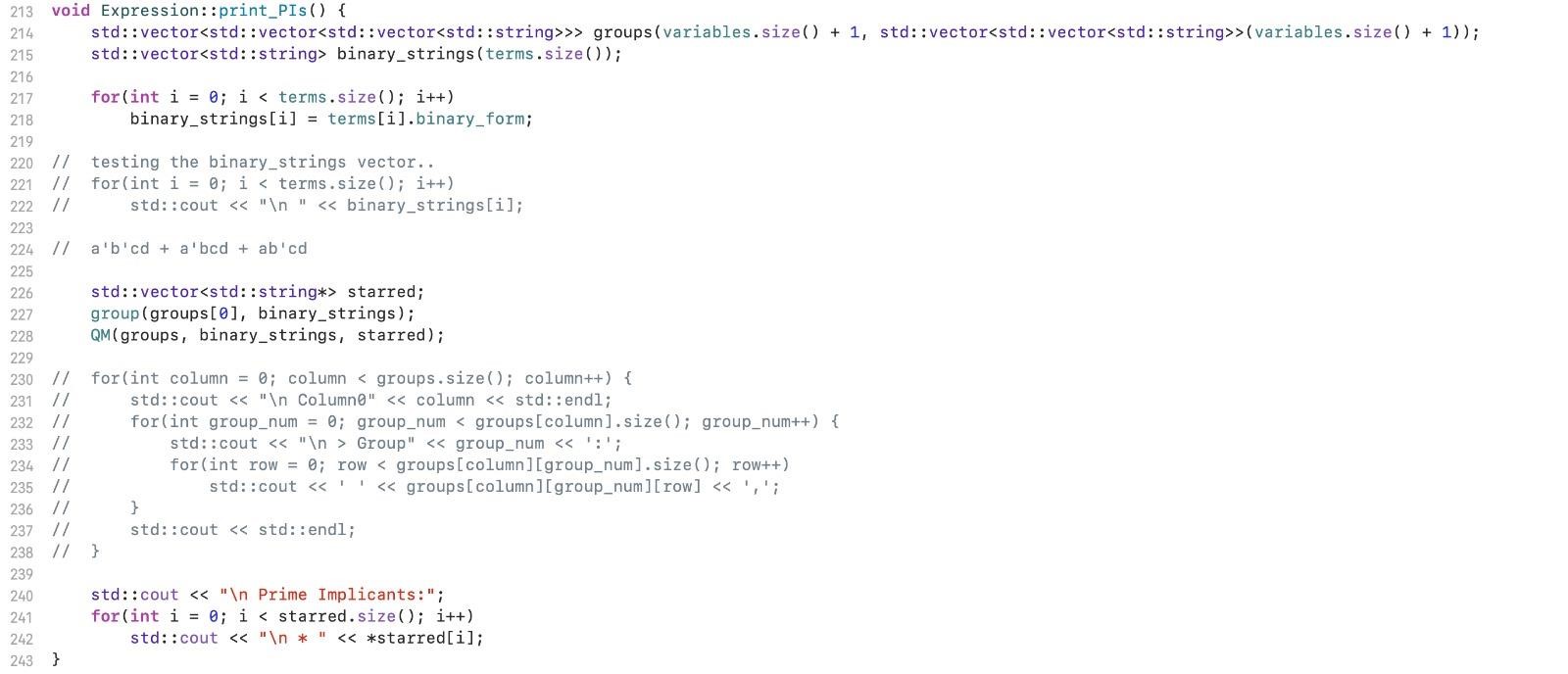
In the next part there is the QM function that sorts each row and group the in the next function which is ‘group’.



The group function takes the input and sort it based on the number of ones in each minterm and group them, so there is a group for minterms that include only one in the minter, then another sorting an d grouping for functions with 2 1s in the input, etc.



Then the print function prints what is done for the prime implicants after the grouping and sorting of the combinations and the repeateions.



And a final output of the function is:

