Digital Design I

Quine-McCluskey Logic Minimization

Project Report

Reem Said, Gehad Ahmed, and Islam Mahdy

10-23-2022

Dr. Mohamed Shaalan

Contents

1 Introduction

The Quine-McCluskey (QM) algorithm is a tabulation method used to minimize any Boolean. It was developed in 1956 by Willard V. Quine and Edward J. McCluskey. In terms of functionality, it is very similar to the Karnaugh mapping method, however, the QM method is faster and more efficient especially if the number of variables is more than five variables. The QM method has mainly two steps; find all prime implicants, then use a table with all the prime implicants to find the essential prime implicants and other prime implicants that are necessary to cover the function using rows and columns dominance. We implemented the QM algorithm in C++ with a time complexity ($2^n n^2$) using maps and vectors.

2 Program Design

Our focus was to try and implement the algorithm in the most efficient way. We decided on using maps instead of vectors because it reduces the time complexity from $(2^n n)^2$ to ($2^n n^2$).

2.1 User Input and Validation

The program reads directly from a file written by the user which contains three lines: the first line specifies the number of variables, the second line is for the minterms, and the last line should specify the don't care terms (if there is any). Before the program calls the QM algorithm, it validates the input to make sure that there are no repeated terms and that the minterms' and don't cares' values don't exceed the maximum value for a minterm. For example, if the number of variables is 4 then the values for minterms and don't cares shouldn't be greater than 15.

2.2 Implicant Class

This class contains 4 attributes: binary representation, the name corresponding to its binary, the number of variables, and a set of covered terms. The main feature of this class would be its parameterized constructor `Implicant(int variables, int n)`. This constructor initializes all the minterms and don't cares as one-term implicants. For example, if a function f(a,b,c,d) has the term 8 as a minterm, it will call the constructor `Implicant(4, 8)`, which will create an implicant with a binary representation equal to 1000, AB'C'D' as a name, and a set of covered terms that contains the number 8. This class also has a method that generates the name by reading the binary representation, a method that changes a decimal number to its equivalent binary number, a method called pad that fills the binary number with zero from the most significant bit according to the number of variables and bits, and other methods that will be mentioned later in the QM algorithm.

2.3 QM Algorithm Class

This class's most important attributes are a `map<int, bool> Terms` to store all of the minterms and don't cares, true means that it is a minterm and false means it is a don't care, `map<Implicant, bool, comparatorImp> Implicants` to store the implicants after merging, true if it is a PI, a vector for the prime implicants (PI), a vector for the essential prime implicants (EPI), and a vector for the reduced terms. QM class has a method `void reduce()` that has the main algorithm of the program, a method to get the PIs, a method to get the EPIs, and a method to reduce the Boolean function.

2.3.1 The Main Algorithm `reduce()`

This function iterates over all the implicants in the map and changes each '0' bit at a time to '1' and if it finds an implicant with a binary number similar to the changed binary this means that the two implicants can be merged into one implicant with the changed bit as a '-'. It only gets out of the loop when the implicants can't be merged further.

While(the merged map is not empty){

create a newly merged map;

for each implicant in the merged map **do**{

for each '0' bit in the implicant binary number **do**{

change the bit to '1' [1]

check if the changed binary number exists in the rest of the map then

create a new binary number by replacing the bit with '-'[2]

retrieve the set of covered terms of both implicants and merge them

add the new implicant with the new binary number and the new set of covered terms to the newly merged map

}

[1] There is a function called changebit in the Implicant class that takes an index as a parameter and changes the bit at that index.

[2] There is a function called replace_complements in the Implicant class that takes an index as a parameter and changes the bit at that index to '-'.

}

Merged map = newly merged map

}

2.3.2 The Populate PI Function

This method iterates over the implicants map and checks the bool value of each implicant if it is false then the implicant is a PI, and it gets added to the prime implicants vector. Then, it checks if there are any implicants in the PI vector that only cover don't care terms to remove it from the vector.

2.3.3 The Populate EPI Function

The populate EPI function takes the vector of minterms as a parameter and creates a map that will hold the frequency of each minterm. It checks all the covered terms in each implicant in the PI vector and increments the frequency of the corresponding term in the map. Then, it checks the map to get all the minterms that have a frequency equal to 1 which means that it is covered by only one implicant, an EPI. After getting the EPI vector, we check the minterms that aren't covered by the EPIs and try to find the biggest prime implicant that covers these minterms but doesn't cover an EPI at the same time.

3 Problems

There is one problem with this program is the very high time complexity which makes it difficult to use it for a large number of variables (more than 20). However, this is expected considering the multiple iterations we do to check each implicant and each term used in the function. It takes approximately one minute to solve 104 test cases of Boolean functions.

4 Program Testing

We created a test case generator that produces 100 test cases; each test case has three lines randomly generated: the first one is the number of variables, the second one is the minterms, and the third one is the don't care terms. To make sure that these generated values are valid, we used the data type set.

5 How to Build The Program

There is a repository on GitHub that contains all the files needed to build this program. Clone the files from GitHub to your desired compiler, then run the code. There is also an option where you can put

your own values. After running, you can open the file titled “testValidOutput.out” to find the results of the generated test cases.

6 Contribution

We all worked on the QM main algorithm together; however, each one took a couple of small tasks and did it on their own as listed below:

Reem:

1. Wrote the code for
    a. Reading and validating a Boolean function
    b. Obtaining and printing the EPIs
    c. Printing the minterms that are not covered by the EPIs
2. Wrote the project report

Gehad:

1. Wrote the code for
    a. Solving the PIs and reducing the Boolean function
    b. Generating the test cases
2. Debugged major errors in the program

Islam:

1. Designed the general form of the program (classes, methods, ... etc.)
2. Wrote the code for
    a. Generating and printing the PIs
3. Debugged major errors in the program