ESE - 556 VLSI PHYSICAL AND LOGIC DESIGN AUTOMATION
Stony Brook University

PROJECT 3

DESIGN AND IMPLEMENTATION OF QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

Submitted to
Professor Alex Doboli
Department of Electrical and Computer Engineering

Project Members:
Suchetha Panduranga - 111463372
Manoj Kumar Gopala - 111679371

CONTENTS

LIST OF FIGURES

LIST OF TBALES

ABSTRACT

The minimization of logic gates is needed to simplify the hardware design area of programmable logic arrays (PLAs) and to speed up the circuits. The VLSI designers can use minimization methods to produce high speed, inexpensive and energy-efficient integrated circuits with increased complexity. Quine-McCluskey (Q-M) is an attractive algorithm for simplifying Boolean expressions because it can handle any number of variables. In the following paper, we present optimized Quine- McCluskey method that reduces the run time complexity of the algorithm by proposing an efficient algorithm for determination of Prime Implicants.

INTRODUCTION

One of the aims of VLSI Logic synthesis is to obtain a digital circuit which is optimal in relation to certain criteria. In other words, one of the steps of the synthesis consists of determining the simplest algebraic expression which can represent a switching function. In doing this, the simplest expression is usually sought in the classes of disjunctive and conjunctive normal forms. Simplification of Boolean expression to reduce the number of literals and gates is a practical tool to optimize programing algorithms and circuits. Area complexity is one of the most important criteria that have to be taken into account while working with simplification methods. Rapid increase in the design complexity and the need to reduce time-to-market have resulted in a need for computer- aided design (CAD) tools that can help make important design decisions early in the design process. However, to be able make these decisions early, there is a need for methods to estimate the area complexity and power consumption from a design description of the circuit at a high level of abstraction. A great number of methods has been developed to simplify the function in order to obtain its minimal normal form. Among them are algebraic, graphical and tabular methods. Minimization using Boolean algebra requires high algebraic manipulation skills and will become more and more complicated when the number of terms increases. Other methods like Karnaugh map provides the ordinary method for simplifying switching functions, although it is limited to the problems with five or less input variables.

When the number of variables exceeds six, the complexity of the map is exponentially enhanced and it becomes more and more cumbersome. Quine McCluskey method is more executable when dealing with larger number of variables and is easier to be mechanized and run on a computer. Quine (1952) and McCluskey (1956) suggested this method of simplification which is considered the most useful tabular procedure. This technique is also suitable for problems with more than one output. Besides, the Quine-McCluskey method is easier to be implemented as a computer program. This report presents the implementation of the Quine McCluskey algorithm that takes into account the Don’t-Care conditions and discusses the results that have been obtained. Section 2 describes the problem formulation. The algorithm in detail and the steps involved are described in section 3. Section 4 presents the software implementation of the algorithm while Section 5 discusses the evaluation of the implementation against different sets of input expressions. Section 6 is dedicated for remarks and conclusion of the topic. Like the K-map, the Q-M method seeks to collect product terms by looking for entries that differ only in a single bit. The only difference from a K-map is that we do it by searching, rather than mapping. The beauty of the Q-M method is that it takes over where the K-map begins to fail. The Q-M technique is capable of minimizing logic relationships for any number of inputs. The main advantage of this method is that it can be implemented in the software in an algorithmic fashion. But the disadvantage of this method is that the computational complexity still remains high.

RELATED WORK

In the 1930s, while studying switching circuits, Claude Shannon observed that one could also apply the rules of Boole's algebra in this setting, and he introduced switching algebra as a way to analyze and design circuits by algebraic means in terms of logic gates. The Duality Principle, also called De Morgan duality, asserts that Boolean algebra is unchanged when all dual pairs are interchanged. Modern electronic design automation tools for VLSI circuits often rely on an efficient representation of Boolean functions known as (reduced ordered) binary decision diagrams (BDD) for logic synthesis and formal verification. When simplification of Boolean expressions became most important for VLSI technology so many techniques were proposed. An algorithm was reported by Petrick (1959). This algorithm uses an algebraic approach to generate all possible covers of a function. A popular tool for simplifying Boolean expressions is the Espresso, but it is not guaranteed to find the best two-level expression (Katz, 1994). WWW-based Boolean function minimization technique was proposed by SP Tomaszewski (2003). In 2011, Solairaju and Periasamy mentioned a technique of simplification through K-map using object-oriented algorithm. Most of the previous method is dependent on number of variables and hence have not achieved much improved performance. Over the past two decades most of the problems in the synthesis, design and testing of combinational circuits, have been solved using various mathematical methods.

During the last two decades, Binary Decision Diagrams have gained great popularity as successful method for the representation of Boolean functions. Over the years, the number of nodes in a BDD has been used to assess the complexity of the Boolean circuit. Researchers in this area are actively involved in developing mathematical models that predict the number of nodes in a BDD in order to predict the complexity of the design in terms of the time needed to optimize it and verify its logic. Nemani, and Najm, proposed an area and power estimation capability, given only a functional view of the design, such as when a circuit is described only by Boolean equations. In this case, no structural information is known— the lower level (gate-level or lower) description of this function is not available. Of course, a given Boolean function can be implemented in many ways, with varying power dissipation levels. They were interested in predicting the minimal area and power dissipation of the function that meets a given delay specification. In this paper, they use “gate-count” as a measure of complexity, mainly due to the key fact observed by Muller, and also because of the popularity of cell-based (or library based) design.

In an early work, Shannon studied area complexity, measured in terms of the number of relay elements used in building a Boolean function (switch-count). In that paper, Shannon proved that the asymptotic complexity of Boolean functions was exponential in the number of inputs, and that for large number of inputs, almost every Boolean function was exponentially complex. Muller, demonstrated the same result for Boolean functions implemented using logic gates (gate-count measure). A key result of his work is that a measure of complexity based on gate-count is independent of the nature of the library used for implementing the function. Several researchers have also reported results on the relationship between area complexity and entropy of a Boolean function. These include Kellerman, empirically demonstrated the relation between entropy and area complexity, with area complexity measured as literal count. They showed that randomly generated Boolean functions have a complexity exponential, and proposed to use that model as an area predictor for logic circuits. However, the circuits tested were very small, typically having less than ten inputs.

PROBLEM STATEMENT

The problem with having a complicated circuit with many logic gates is that each element takes up physical space in its implementation and costs time and money to produce in itself. Circuit minimization is one form of logic optimization that can be used to reduce the area of complex logic in VLSI circuits.

Given: *An input text file consisting of decimal notations signifying the minterms and don't care conditions or the function that defines the functionality of the logic expression*

Goal: *Employ the tabular Quine McCluskey algorithm method to simplify the Boolean function and obtain its minimal normal form that results in reduced of the cost of the circuit*

Output: *A console output containing minimized expression with reduced product terms and prime implicants obtained from all iterations of the reduction*

QUINE MCCLUSKEY ALGORITHM

The algorithm was developed by Willard V. Quine and extended by Edward J. McCluskey. It is functionally identical to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a Boolean function has been reached. The idea employed is that when two terms contain the same variables differ only in one variable, they can be combined together and form a new term smaller by one literal. All the terms in the Boolean function are tested for possible combination of any two of them, after which a new set of terms that are smaller by one literal are produced and are further tested under the same procedures for further reduction. The same procedures will be repeated until no terms can be combined anymore. The irreducible terms are named prime implicants. Those prime implicants are used in a *prime implicant chart* to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function. The essential prime implicnats represent the final minimized expression.

Prime Implicants: A literal is any variable or its negation in the expression. A product term is implicant of a function if the function has the vaule 1 for all minterms of the product term. For function of n variables, implicants may contain n or less literals. The most basic implicants are the minterms. Each minterm of a switching function represents the implicant of that function which covers it on only one vector.

Essential Prime Implicant: Prime implicant that is able to cover an output of the function which is not covered by any combination of prime implicant called essential prime implicant.

Cost of a Circuit: The cost of a logic circuit can be usually expressed as a number of gates plus the total number of inputs to all gates in the circuit. Here it is implied that input variables are available in the right and complementary form. That is, *Cost of circuit = number of gates + total number of inputs to all gates in the circuit.* The main agenda of minimizing logic expressions is to attain reduced Boolean space that leads to reduced circuit cost.

Quine-McCluskey method is based on the procedure of grouping applied to every two minterms which differ only by the value of one variable. Two main parts in the Quine-McCluskey algorithm are -

- Finding all prime implicants of the function.
- Use those prime implicants in a prime implicant table to find the essential rimie implicants of the function and other prime implicants that provide the coverage pf the function with minimum cost.

The following points discusses the steps involved in minimizing a logic expression using Quine McClusky.

Step 1 Tabulate all the minterms of the function by their binary representations.
Step 2 Arrange the minterms into groups according to the number of 1's in their binary representation.
Step 3. Compare each minterm in a group with each of the minterms in the group below it. If the compared pair is adjacent (i.e., if they differ by one variable only), they are combined to form a new term. The new term has a 'X' in the position of the eliminated variable. Both combining terms are checked off in the original list indicating that they are not prime implicants.
Step 4. Repeat the above step for all groups of minterms in the list. This results in a new list of terms with 'X's in place of eliminated variables.
Step 5. Compare terms in the new list in search for further combinations. This is done by following step 3. In this case a pair of terms can be combined only if they have 'X's in the same positions. As before, a term is checked off if it is combined with another. This step is repeated until no new list can be formed. All terms that remain unchecked are prime implications.
Step 6. Select a minimal subset of prime implicants that cover all the terms of the original Boolean function.

Example-1

We can consider solving the equation

$$\begin{aligned} f(A,B,C,D,E) \\ &= \sim A\sim B\sim C\sim D\sim E \ + \ \sim A\sim B\sim C\sim DE \ + \ \sim A\sim B\sim CD\sim E \ + \ \sim AB\sim C\sim DE \ + \ \sim AB\sim CDE \\ &+ \ \sim ABC\sim D\sim E \ + \ \sim ABC\sim DE \ + \ AB\sim CDE \ + \ ABC\sim D\sim E \ + \ ABC\sim DE \end{aligned}$$

The minterms are first organized as below

$$f(A,B,C,D) = \sum m\,(0,1,2,9,11,12,13,27,28,29)$$

They are then grouped according to the number of 1's contained in each term, as specified in step 2. This results in list 1 of Figure 1. In list 1, terms of group 1 are combined with those of group 2, terms of group 2 are combined with those of group 3, and so on, using step 3. The next step is to compare the two terms in group 2 of list 1 with the two terms in group 3. Only terms 1(00001) and 9(01001) combine to give 0X001; all other terms differ in more than one variable and therefore do not combine. As a result, the second group of list 2 contains only one combination. The process of combining terms in adjacent groups is continued for list 2. This results in list 3. These correspond to the prime implicants of the Boolean function and are labeled PI1, . . ., PI7.

	List 1			List 2			List 3		
	Minterm	ABCDE		Minterms	ABCDE		Minterms	ABCDE	
Group 1	0	00000	✓	0,1	0000x	PI_2	12,13,28,29	X110x	PI_1
Group 2	1	00001	✓	0,2	000x0	PI_3			
	2	00010	✓	1,9	0x001	PI_4			
Group 3	9	01001	✓	9,13	01x01	PI_5			
	12	01100	✓	9,11	010x1	PI_6			
Group 4	13	01101	✓	12,13	0110x	✓			
	11	01011	✓	12,28	X1100	✓			
	28	11100	✓	13,29	X1101	✓			
Group 5	29	11101	✓	11,27	X1011	PI_7			
	27	11011	✓	28,29	1110x	✓			

Table 1. Visualization of Quine McClusky algorithm of example1

The final step of the proedure is to find a minimal subset of the prime implicants which can be used to realize the original function. The complete set of prime implicants for the given function can be derived from Figure 1 are

$$(BC{\sim}D \;+\; {\sim}A{\sim}B{\sim}C{\sim}D \;+\; {\sim}A{\sim}B{\sim}C{\sim}E \;+\; {\sim}A{\sim}C{\sim}DE \;+\; +{\sim}AB{\sim}D{\sim}E \;+\; {\sim}AB{\sim}CE \;+\; B{\sim}CDE)$$

In order to select the smallest number of prime implicants that account for all the original minterms, a prime implicant chart is formed as shown in Figure 2. A prime implicant chart has a column for each of the original minterms and a row for each prime implicant. For each prime implicant row, an X is placed in the columns of those minterms that are accounted for by the prime implicant. To choose a minimum subset of prime implicants, it is first necessary to identify the essential prime implicants. A column with a single X indicates that the prime implicant row is the only one covering the minterm corresponding to the column; therefore, the prime implicant is essential and must be included in the minimized function. The minterms covered by the essential prime implicants are marked with asterisks. The next step is to select additional prime implicants that can cover the remaining column terms. This is usually done by forming a reduced prime implicant chart that contains only the minterms that have not been covered by the essential prime implicants.

	0	1	2	9	11	12	13	27	28	29
PI_1*						X	X		X	X
PI_2	X	X								
PI_3*	X		X							
PI_4		X		X						
PI_5				X			X			
PI_6				X	X					
PI_7*					X			X		

Table. Prime Implicant chart of example1

Therefore, the minimum sum of-products equivalent to the original function is

$$f(A,B,C,D,E) = PI1 + PI3 + PI4 + PI7$$

$$f(A,B,C,D,E) = X110X + 000X0 + 0X001 + X1011$$

$$f(A,B,C,D,E) = BC{\sim}D \ + \ {\sim}A{\sim}B{\sim}C{\sim}E \ + \ {\sim}A{\sim}C{\sim}DE \ + \ B{\sim}CDE$$

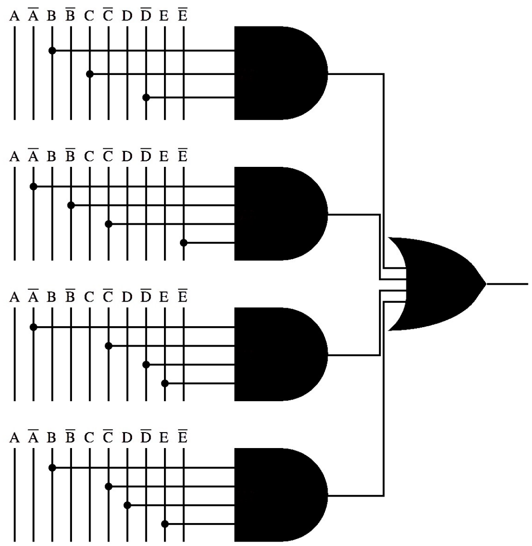

Figure 1: Circuit Representation of example 1

Example-2

The following example demonstrates the implementation of the algorithm for a function with minterms in decimal notation with don't care conditions.

$$f(A,B,C,D) = \sum m\ (2,3,7,9,11,13) + \sum d\ (1,10,15)$$

The procedure is similar to the previous example. Minterms are grouped according to the number of 1's contained in each term, as specified in step 2. This results in list 1 of Figure 3. In list 1, terms of group 1 are combined with those of group 2, terms of group 2 are combined with those of group 3, and so on, using step 3. The next step is to compare the two terms in group 2 of list 1 with the two terms in group 3. The process of combining terms in adjacent groups is continued for list 2. These correspond to the prime implicants of the Boolean function. The final step of the procedure is to find a minimal subset of the prime implicants which can be used to realize the original function. The don't care terms are treated as required minterms while finding prime implicants. The don't care columns are omitted when forming the prime implicant chart. The minimum sum of-products equivalent to the original function is

$$f(A,B,C,D) = {\sim}BC \ + \ CD \ + \ AD$$

	Minterms	ABCD		Minterms	ABCD		Minterms	ABCD	
Group1	1	0001	✓	1,3	00x1	✓	1,3,9,11	x0x1	
	2	0010	✓	1,9	x001	✓	2,3,10,11	x01x	
Group2	3	0011	✓	2,3	001x	✓	3,7,11,15	xx11	
	9	1001	✓	2,10	x010	✓	9,11,13,15	1xx1	
	10	1010	✓	3,7	0x11	✓			
Group3	7	0111	✓	3,11	x011	✓			
	11	1011	✓	9,11	10x1	✓			
	13	1101	✓	9,13	1x01	✓			
Group4	15	1111	✓	10,11	101x	✓			
				7,15	x111	✓			
				11,15	1x11	✓			
				13,15	11x1	✓			

Table 1. Visualization of Quine McClusky algorithm of example 2

	2	3	7	9	11	13
(1,3,9,11)		X		X	X	
*(2,3,10,11)	X	X			X	
*(3,7,11,15)		X	X		X	
*(9,11,13,15)					X	X

Table2. Prime Implicant chart of example 2

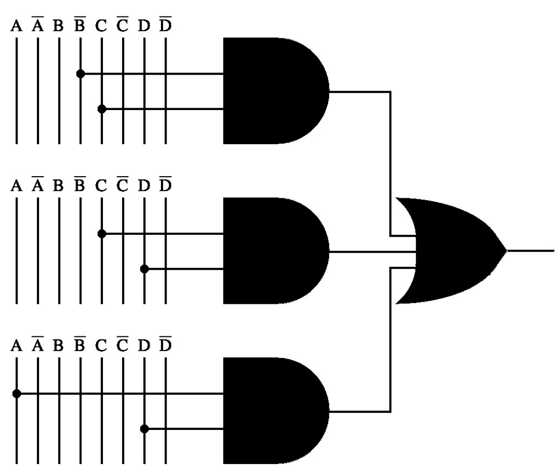

Figure 1: Circuit Representation example 2

SOFTWARE IMPLEMENTATION OF QUINE MCCLUSKEY

The Quine McCluskey algorithm is implemented using C++ language. The input is read from a text file and the format of the input follows a description -

- The input string that is to be parsed is of the form

$$\$\{function - name\}(Variables\ divided\ by\ ',' ..) = \$\{TERMS\} + \$\{TERMS\} + \cdots$$

- All the white spaces are ignored.
- Default character to invert a variable is ~
- Accepts variables from A to Z and a to z

The data structures used and the issues involved in the process of software implementation of the algorithm are discussed below.

Input

The logical expression entered is parsed initially and validated against the pre-defined set of rules to confirm that the input entered is of the right format. The canonical form of expression is then converted to a sum of products (SOP) expression and transformed into binary format and arranged into different groups according to the number of 1s in the minterm representation. To realize this, we employed a vector data structure with property type *logical_term <property_type>* to store the input. The function *make_std_spf()* parses the input to a standard SOP expression. We designed a function *make_min_table()* that exploits the idea of dynamic tables to create a compression table for storing the binary formats of the minterms of the expression.

Finding the prime implicants

Each minterm is compared with larger minterms in the next group down. If they differ by a power of 2 then they pair-off. The digit being canceled is replaced by ‘X’. A subsequent compression table is formed containing the minterms paired that are smaller by one literal and is used for further compression. This procedure is repeated until no terms can be paired anymore. The remaining irreducible terms will end up being the prime implicants. We designed dynamic tables with a control loop for storing the newly generated sets of terms with reduced literals. The function *compress_impl()* is invoked for compression of tables and returns a true boolean expression for every prime implicant found. It falsifies once the iterations exhaust and no implicants are found. The terms that are being cancelled during compression are replace by ‘X’ terms while being stored for further processing.

Finding the essential prime implicants

In the previous step, the program would have found all the possible sets of prime implicants that cover the remaining minterms using nested for loop. The number of layers of the nested for loop depends on the number of the remaining minterms. However, the number of the remaining minterms depends on the input of the program, which is unknown when writing the program. To solve this problem, a dynamic recursive function is used. After going through all the possible sets of prime implicants, a simple for loop will be utilized to find the set containing the least number of prime implicants. The simple implicants in this set will be the rest essential prime implicants. The function *simplifier()* finds the essential prime implicants and simplifies the logic expression.

Accounting for Don't Cares

When the Boolean function includes don't cares, they are taken into consideration to generate the complete set of prime implicants. Since determining the essential prime implicants requires only minterms/maxterms, don't care terms are not listed as column headings in the compression table for determining the essential prime implicants.

Display

We attempt to display the results consisting of the prime implicants resulting from all iterations of the reduction along with the levels of compression the minimization technique has employed to arrive at the acceptable solution. And this is done by invoking *print_truth_table()* that prints the expression in its true sop form and *get_prime_implicants()*, that prints the final prime implicants list. The *compress_table()* function, in itself, prints every iteration of compression and the marked terms during minimization. The steps involved in the design of the software implementation of the algorithm is consolidated in the form of a flowchart to visualize the control flow of the software implementation of the logic minimization algorithm. The figure is referred to as Figure 3.

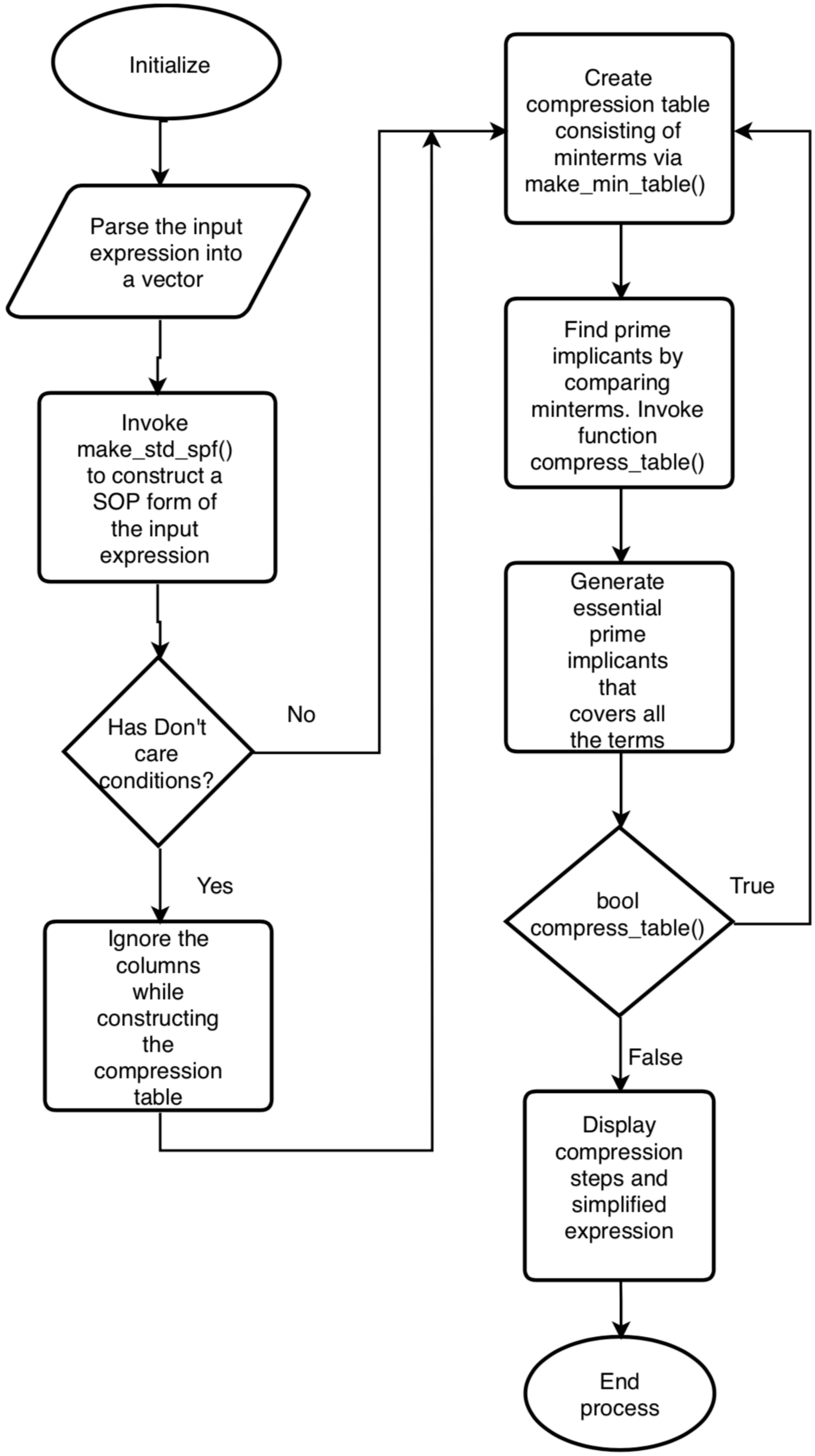

Figure 3. Flowchart for software implementation of Quine McCluskey

EVALUATION

With an adequate program, it is possible to achieve minimization of switching functions with maximum 10 input variables and maximum 50 product terms. The evaluations were carried out on a MacOS High Sierra with 2.3GHz dual-core Intel Core i5, Turbo Boost of 3.6GHz, with a memory of 8GB of 2133MHz LPDDR3. In order to check for exact operation, the program has been tested on minimization of several functions, each including a different number of input variables and corresponding running times have been recorded. The results tested were then checked by hand. In all cases good results were obtained and no discrepancy was found.

Table 5 shows the resultant minimized expression obtained with a Quine McCluskey algorithm implemented using C++ along with the running times for each set of inputs.

No. of literals	Logic expression	Minterm expansion	Reduced logical expression	Execution time (seconds)
4	$f(A,B,C,D) = \sim AB\sim C + A\sim B\sim C + AB\sim C\sim D + A\sim BC\sim D + BC\sim D + \sim ABCD + A\sim BCD + \sim A\sim BD$	$\Sigma m(1,3,4,5,6,7,8,9,10,11,1$	$f' = \sim AD + B\sim D + A\sim B$ $f' = \sim BD + \sim AB + A\sim D$	4.96
5	$f(A,B,C,D,E) = \sim A\sim B\sim C\sim DE + \sim A\sim BC\sim DE + \sim A\sim BCDE + \sim AB\sim CDE + A\sim B\sim C\sim DE + A\sim BC\sim DE + A\sim BCDE + ABCDEF$	$\Sigma m\ (1,5,7,11,17,21,22,31) + \Sigma d\ (0,4,14,30)$	$f' = \sim B\sim DE + \sim A\sim BCE + ACD\sim E + ABCD + \sim AB\sim CDE$	8.2

No. of literals	Logic expression	Minterm expansion	Reduced logical expression	Execution time (seconds)
6	$f(A,B,C,D,E,F) = \sim A\sim B\sim C\sim D\sim E\sim F + \sim A\sim B\sim C\sim DE\sim F + \sim A\sim B\sim CDE\sim F + \sim A\sim B\sim CDE\sim F + \sim A\sim B\sim CDEF + \sim A\sim BCDE\sim F + \sim A\sim BC\sim D\sim E\sim F + A\sim BC\sim D\sim EF + \sim A\sim BCD\sim E\sim F + \sim A\sim BCDEF + \sim A\sim BC\sim DE\sim F$	$\Sigma\, m(0,2,4,6,7,8,10,12,14,15 + \Sigma\, d\ (1,5,11)$	$f' = A\sim BC\sim D\sim EF + \sim A\sim B\sim D\sim F + \sim A\sim BC\sim F + \sim A\sim BDE$	22.34
7	$f(A,B,C,D,E,F,G) = \sim A\sim B\sim C\sim DE\sim FG + \sim A\sim B\sim CDEFG + \sim AB\sim CDE\sim F\sim G + A\sim BC\sim DE\sim F\sim G + A\sim BC\sim DE\sim FG + ABCDEFG$	$\Sigma\, m(5,15,44,84,85,127) + \Sigma\, d\ (55,63,106,125)$	$f' = A\sim BC\sim DE\sim F + BCDEFG + \sim A\sim B\sim C\sim DE\sim FG + \sim A\sim B\sim CDEFG + \sim AB\sim CDE\sim F\sim G$	35.71
8	$f(A,B,C,D,E,F,G,H) = \sim A\sim B\sim C\sim D\sim E\sim F\sim G\sim + \sim A\sim BC\sim D\sim E\sim F\sim GH + A\sim BC\sim D\sim E\sim F\sim GH + ABCDEFGH$	$\Sigma\, m(0,33,161,255) + \Sigma\, d\ (22,237)$	$f' = \sim BC\sim D\sim E\sim F\sim GH + \sim A\sim B\sim C\sim D\sim E\sim F\sim G\sim + BCDEFG$	48.32

Table 5. Evaluation Results

The table also gives the execution times of different sets of inputs. It was found that the execution time of the algorithm exponentially increases with the increase in the number of literals and the complexity of the logical expression. The plot figure 4 visualizes the variation of execution time with the number of literals.

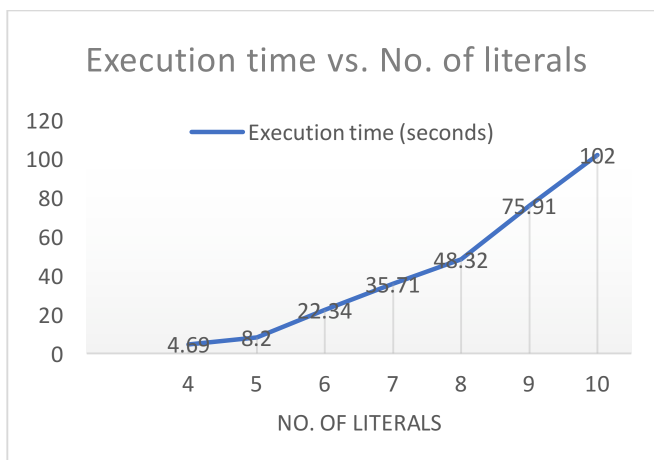

Figure 4. Run time for Quine McCluskey

In QM method each minterm of each group is compared with minterm from a previous group and the total number of comparisons (Z) between minterms in worst case is given by $Z = \Sigma\, C_i^n . C_{i+1}^n$ where 'n' is the number of variables. The plot Figure 6 shows the complexity variation with increase in variables in the expression.

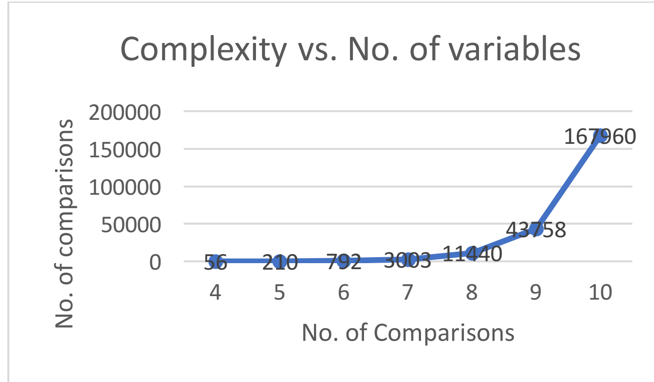

Figure 5. Complexity for Quine McCluskey

CONCLUSION

The evaluations presented for the Quine-McCluskey algorithm for logic gate minimisation has shown that the program developed is reliable and fast. This program minimizes switching functions with maximum 10 variables. The code can easily be widened to correspond to the functions whose number of variables is restricted by memory of the computer. Development and incorporation of user interface is also suggested for future work.

BIBILOGRAPHY

1. E. J. McCluskey, "Minimization of Boolean Functions," Bell System Technical Journal, vol. 35, no. 5, pp. 1417-1444, 1956.

2. S. Tani, K. Hamaguchi, and S. Yajima, "The Complexity of the Optimal Variable Ordering Problems of a Shared Binary Decision Diagram," IEICE Transaction on Information and Systems, Vol. 4, pp. 271-281, 1996.

3. M Karnaugh, "The map method for synthesis of combinational logic circuits," Trans AIEE, Commiin & Electron, vol 72, no 1, pp 593-598, 1953.
4. M Morris Mano, Digital Logic and Computer Design, Prentice Hall of India 2004 Edition, pp 72-112.
5. Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions

6. O. Coudert, J.C. Madre, H. Fraisse, A New Viewpoint on Two-Level Logic Minimization, Proc. 30th Design Automation Conference, Dallas, TX, USA, pp. 625–630, June 1993.

7. M. Karnaugh, The Map Method for Synthesis of Combinational Logic Circuits, AIEE Transactions on Communications & Electronics, Vol. 9, pp. 593–599, 1953.

8. E. Morreale, Recursive Operators for Prime Implicant and Irredundant Normal Form Determina- tion, IEEE Transactions on Computers, 1970.

APPENDIX

SOURCE CODE FOR ALGORITHM IMPLEMENTATION

Quine_mccluskey.cpp

```
#include <iostream>
#include <set>
#include <algorithm>
#include <stdexcept>
#include <numeric>
#include <cmath>
#include "logical_expr.hpp"
#include "quine_mccluskey.hpp"

using namespace std;
using namespace logical_expr;

namespace quine_mccluskey {


typedef simplifier::property_type property_type;
typedef simplifier::term_type term_type;
typedef simplifier::set_type set_type;
typedef simplifier::table_type table_type;

// Make standard sum of products form
const logical_function<term_type>& simplifier::make_std_spf() {
   stdspf_.clear();
   arg_generator<> generator(0, std::pow(2, func_.term_size()), func_.term_size());
   for( auto arg : generator )
      if( func_(arg) )
         stdspf_ += logical_term<term_mark>(arg);
   return stdspf_;
}

const table_type& simplifier::make_min_table() {
   table_[0].resize(func_.term_size() + 1, set_type());
   for( auto term : stdspf_ )
      table_[0][term.num_of_value(true)].push_back(term);
   return table_[0];
}

void simplifier::compress_table(bool printable) {
   for( ;; ) {
      if( printable )
         cout << get_current_level() + 1 << "-level compression:" << endl;
      if( !compress_impl(printable) ) break;
```

```
    }
    for( auto table : table_  )
        for( auto set : table )
            for( const logical_term<term_mark> &term : set )
                if( !property_get(term) )
                    prime_imp.push_back(term);
    make_unique(prime_imp);
}

const vector<logical_function<term_type>>& simplifier::simplify() {
    vector<int> next_index(prime_imp.size());
    std::iota(next_index.begin(), next_index.end(), 0);
    // bool: wether simplifying finished  int: loop number that simplifying took
    std::pair<bool, int> end_flags = { false, 0 };
    for( int i = 1; i <= prime_imp.size(); ++i ) {
        if( end_flags.first && end_flags.second < i )
            break;
        do {
            logical_function<term_type> func;
            auto log_func_comp
                = [&](const logical_function<term_type> &f){ return f.is_same(func); };
            for( int j = 0; j < i; ++j )
                func += prime_imp[next_index[j]];
            if( func == stdspf_  &&
                std::find_if(simplified_.begin(), simplified_.end(), log_func_comp)
                  == simplified_.end() ) {
                simplified_.push_back(func);
                end_flags = { true, i };
            }
        } while( next_permutation(next_index.begin(), next_index.end()) );
    }
    return simplified_;
}

void simplifier::add_table(const table_type& table) {
    table_.push_back(table);
}

void simplifier::clear_table() {
    for( int i = 0; i < table_.size(); ++i )
        table_[i].clear();
}

template<typename T>
void simplifier::make_unique(vector<T> &vec) {
    for( auto it = vec.begin(); it != vec.end(); ++it ) {
        auto rm_it = remove(it + 1, vec.end(), *it);
        vec.erase(rm_it, vec.end());
```

```
    }
}

// Try to find prime implicants
// Return true while trying to find them
// Return false if it finished
bool simplifier::compress_impl(bool printable) {
    table_type next_table;
    next_table.resize(func_.term_size(), set_type());
    int count = 0;
    for( int i = 0; i+1 < table_[min_level_].size(); ++i ) {
        for( int j = 0; j < table_[min_level_][i].size(); ++j ) {
            for( int k = 0; k < table_[min_level_][i+1].size(); ++k ) {
                try {
                    // Throw an exception if could not minimize
                    auto term = onebit_minimize(table_[min_level_][i][j], table_[min_level_][i+1][k], false);
                    if( printable )
                        cout << "COMPRESS(" << table_[min_level_][i][j] << ", " << table_[min_level_][i+1][k] << ")
= " << term << endl;
                    if( std::find_if(next_table[term.num_of_value(true)].begin(),
next_table[term.num_of_value(true)].end(),
                                [&](const term_type &t){ return t.is_same(term); }) ==
next_table[term.num_of_value(true)].end())
                        next_table[term.num_of_value(true)].push_back(term);
                    ++count;
                    // Mark the used term for minimization
                    property_set(table_[min_level_][i][j], true);
                    property_set(table_[min_level_][i+1][k], true);
                } catch( std::exception &e ) {}
            }
        }
    }
    if( count ) {
        ++min_level_;
        add_table(next_table);
    }
    return (count ? true : false);
}


}

quine_mccluskey.hpp
#ifndef QUINE_MCCLUSKEY_HPP
#define QUINE_MCCLUSKEY_HPP


#include <iostream>
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
#include <set>
#include <algorithm>
#include <stdexcept>
#include <cmath>
#include "logical_expr.hpp"


namespace quine_mccluskey {

using namespace std;
using namespace logical_expr;

//
// logical function simplifier
//
// How to simplify:
//  [*] In the case which simplifier is constructed with logical_function
//      In this case, constructor will prepare to simplify a function
//      1. compress_table()     // Compress the compression table
//      2. simplify()           // simplify the function and get simplified
//  [*] In the case which simplifier is default-constructed
//      1. set_function()       // set a target function
//      2. make_std_spf()       // make a standard sum of products form
//      3. make_min_table()     // create a compression table
//      4. same as the case above
//
class simplifier {
public:

    typedef term_mark property_type;
    typedef logical_term<property_type> term_type;
    typedef vector<term_type> set_type;
    typedef vector<set_type> table_type;

    simplifier() : min_level_(0)
        { add_table(table_type()); make_min_table(); }
    explicit simplifier(const logical_function<term_type> &function) : min_level_(0), func_(function)
        { add_table(table_type()); make_std_spf(); make_min_table(); }
    ~simplifier() {}

    void set_function(const logical_function<term_type> &func) { func_ = func; }
    int get_current_level() const { return min_level_; }
    const logical_function<term_type>& get_std_spf() const { return stdspf_; }
    const set_type& get_prime_implicants() const { return prime_imp; }

    // Make standard sum of products form
    const logical_function<term_type>& make_std_spf();
    const table_type& make_min_table();
```

```
    void compress_table(bool printable = false);
    const vector<logical_function<term_type>>& simplify();

private:
    void add_table(const table_type& table);
    void clear_table();
    template<typename T>
    static void make_unique(vector<T> &vec);

    // compress compression table
    // return true while trying to compress
    // return false if compression finished
    bool compress_impl(bool printable = false);

    int min_level_;
    logical_function<term_type> func_, stdspf_;
    vector<logical_function<term_type>> simplified_;
    vector<table_type> table_;
    set_type prime_imp;
};



}   // namespace quine_mccluskey


#endif  // QUINE_MCCLUSKEY_HPP
```

logical_expr.hpp

```
#ifndef LOGICAL_EXPRESSION_HPP
#define LOGICAL_EXPRESSION_HPP


#include <iostream>
#include <string>
#include <sstream>
#include <utility>
#include <iterator>
#include <algorithm>
#include <stdexcept>
#include <cmath>
#include <boost/regex.hpp>
#include <boost/format.hpp>
#include <boost/tokenizer.hpp>
#include <boost/call_traits.hpp>
#include <boost/algorithm/string.hpp>
#include <boost/dynamic_bitset.hpp>
#include <boost/io/ios_state.hpp>
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
#include <boost/optional.hpp>
//#include <boost/logic/tribool.hpp>

//
// namespace for Logical Expression
//
namespace logical_expr {


using namespace std;

// Don't care
/*thread_local*/static const boost::optional<bool> dont_care = boost::none;
// static const boost::logic::tribool dont_care = indeterminated; better than optional<bool>

// expr_mode is not used now.
enum expr_mode { alphabet_expr, verilog_expr, truth_table };

// argument generating iterator for logical_function
class arg_gen_iterator {
public:
    typedef boost::dynamic_bitset<> value_type;
    typedef arg_gen_iterator this_type;
    arg_gen_iterator(int width, int val)
        : width_(width), current_val_(val), value_(width, val) {}
    this_type& operator++() {
        value_type tmp(width_, current_val_ + 1);
        value_.swap(tmp);    // never throw any exceptions
        ++current_val_;
        return *this;
    }
    this_type operator++(int)
        { this_type before = *this; ++*this; return before; }
    this_type& operator--() {
        value_type tmp(width_, current_val_ - 1);
        value_.swap(tmp);
        --current_val_;
        return *this;
    }
    this_type operator--(int)
        { this_type before = *this; --*this; return before; }
    bool operator<(const this_type &it) const
        { return (it.width_ == width_ && current_val_ < it.current_val_); }
    bool operator>(const this_type &it) const
        { return (it.width_ == width_ && current_val_ > it.current_val_); }
    bool operator==(const this_type &it) const
        { return (it.width_ == width_ && current_val_ == it.current_val_); }
    bool operator!=(const this_type &it) const
```

```
        { return !(*this == it); }
    const value_type& operator*() const { return value_; }
private:
    const int width_;
    int current_val_;
    value_type value_;
};

// Argument generator for logical_function using Iterator (default: arg_gen_iterator)
template<typename Iterator = arg_gen_iterator>
class arg_generator {
public:
    arg_generator(int nbegin, int nend, int width)
        : begin_(width, nbegin), end_(width, nend) {}
    const Iterator& begin() const { return begin_; }
    const Iterator&  end()  const { return end_; }
private:
    const Iterator begin_, end_;
};


//
// * String to be parsed has to be in the following form:
// * ${Function-Name}(Variables-divided-by-',' ...) = ${TERMS} + ...
// * White spaces will be ignored
// * Default character to invert a variable is '~' (first template parameter)
// See README for more information about parsing
// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
template<char inverter = '~', bool escape = true, expr_mode mode = alphabet_expr>
class function_parser {
public:
    typedef std::pair<string, vector<string>> result_type;
    function_parser() {}
    explicit function_parser(const string &expr, const char first) : expr_(expr), first_char_(first) {}
    ~function_parser() {}

    void set_expression(const string &expr) { expr_ = expr; }
    const string& get_expression() const { return expr_; }
    const string& function_name() const { return func_name_; }

    result_type parse() {
        auto untokenized = scanner();
        auto token = tokenizer(untokenized);
        boost::optional<char> undecl = use_undeclared_vars(token.second, token.first);
        if( undecl )
            throw std::runtime_error(
                (boost::format("expr: Using undeclared variable, %c") % *undecl).str()
            );
```

```
        if( !is_sequence(token.first, first_char_) )
            throw std::runtime_error("expr: used variables are not sequence");
        return std::move(token);
    }

    vector<string> scanner() {
        if( expr_.empty() )
            throw std::runtime_error("expr: Expression is empty, aborted");
        string expr_with_nospaces = boost::regex_replace(expr_, boost::regex("\\s"), "");
        boost::regex reg(
            (boost::format("([A-Za-z_-]+)\\((((\\s*[A-Za-z],)*)([A-Za-z]))\\)=(((%1%?[A-Za-z])+\\+)*((%1%?[A-
Za-z])+))$")
                % (escape ? string{'\\', inverter} : string{inverter})).str(),
             boost::regex::perl
        );
        boost::smatch result;
        if( !boost::regex_match(expr_with_nospaces, result, reg) )
            throw std::runtime_error("expr: Input string does not match the correct form");
        func_name_ = result[1];
//                          decl-vars  decl-terms
        return vector<string>{result[2], result[6]};
    }

    static result_type tokenizer(const vector<string> &untokenized) {
        std::vector<string> terms;
        typedef boost::char_separator<char> char_separator;
        boost::tokenizer<char_separator>
            var_tokenizer(untokenized[0], char_separator(",")), term_tokenizer(untokenized[1],
char_separator("+"));
        ostringstream oss;
        for( auto token : var_tokenizer )   oss << token;
        for( auto token : term_tokenizer )  terms.push_back(token);
        return std::make_pair(oss.str(), terms);
    }

    static bool is_sequence(const string &vars, char first_char) {
        if( vars[0] != first_char )
            throw std::runtime_error("expr: declare terms which starts with not specified char");
        auto previous = vars.begin();
        for( auto it = ++vars.begin(); it != vars.end(); ++it ) {
            if( *it != static_cast<char>(*previous + 1) )
                return false;
            previous = it;
        }
        return true;
    }

    static boost::optional<char> use_undeclared_vars(const vector<string> &terms, const string &vars) {
```

```
        for( auto term : terms )
          for( auto used_var : term )
            if( used_var != inverter && vars.find(used_var, 0) == string::npos )
              return (used_var);
        return boost::none;
    }

private:
    string expr_, func_name_;
    char first_char_;
};


//
// Term properties
//
// Type Requirements:
//  [*] typedef value type
//  [*] Have set() and get() member functions
//  [*] Default constructible
//  [*] Has swap() member fuction that never throws any exceptions for expcetion-safe
//

// Term Property template class for POD types
template<typename T, T DefaultValue>
class term_property_pod {
public:
    typedef T value_type;
    term_property_pod() : value_(DefaultValue) {}
    typename boost::call_traits<T>::param_type get() const { return value_; }
    void set(typename boost::call_traits<T>::param_type value) { value_ = value; }
    void swap(const term_property_pod<T, DefaultValue> &property) noexcept(true)
        { std::swap(value_, property.value_); }
private:
    value_type value_;
};

typedef term_property_pod<int, 0> term_no_property;
typedef term_no_property term_dummy;
typedef term_property_pod<char, ' '> term_name;
typedef term_property_pod<bool, false> term_mark;
typedef term_property_pod<unsigned int, 0> term_number;
//

//
// class: logical term
//
template<typename Property_ = term_dummy>
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
class logical_term {
public:
    typedef boost::optional<bool> value_type;
    typedef boost::dynamic_bitset<> arg_type;
    typedef logical_term<Property_> this_type;
    typedef std::size_t size_t;
    typedef Property_ property_type;

    logical_term() {}
    logical_term(int bitsize, const value_type &init = logical_expr::dont_care)
        : term_(bitsize, init) {}
    template<typename Property>
    explicit logical_term(const logical_term<Property> &term)
        { construct_from(term); }
    explicit logical_term(const arg_type &arg) {
        for( int i = arg.size() - 1; 0 <= i; --i )
            term_.push_back(arg[i]);
    }

    template<typename Property>
    void construct_from(const logical_term<Property> &term)
        { term_ = term.term_; }

    template<typename Property>
    void swap(logical_term<Property> &term) noexcept(true) {
        term_.swap(term.term_);
        property_.swap(term.property_);
    }

    size_t size() const
        { return term_.size(); }

    bool size_check(const arg_type &arg) const
        { return (size() == arg.size()); }

    bool is_same(const this_type &term) const
        { return (term.term_ == term_); }

    template<typename Property>
    bool size_check(const logical_term<Property> &term) const
        { return ( size() == term.size() ); }

    size_t num_of_value(bool value) const {
        return static_cast<size_t>(std::count_if(term_.begin(), term_.end(),
                [value](const value_type &b){ return (b != dont_care && *b == value); }));
    }

    size_t diff_size(const this_type &term) const {
```

```
    if( !size_check(term) )
        throw std::runtime_error(size_error_msg);
    size_t diff_count = 0;
    for( int i = 0; i < size(); ++i )
        if( term_[i] != term[i] )
            ++diff_count;
    return diff_count;
}

bool calculate(const arg_type &arg) const {
    if( !size_check(arg) )
        throw std::runtime_error(size_error_msg);
    bool ret = true;
    for( int i = 0; i < term_.size(); ++i )
        ret = ret && (term_[i] == dont_care ? true : arg[term_.size()-1-i] == term_[i]);
    return ret;
}

bool operator()(const arg_type &arg) const
    { return calculate(arg); }

value_type& operator[](int index)
    { return term_[index]; }
const value_type& operator[](int index) const
    { return term_[index]; }

template<typename Property>
bool operator==(const logical_term<Property> &term) const {
    if( !size_check(term) ) return false;
    arg_generator<> gen(0, std::pow(2, size()), size());
    for( auto it = gen.begin(); it != gen.end(); ++it )
        if( calculate(*it) != term.calculate(*it) )
            return false;
    return true;
}

template<typename Property>
friend typename logical_term<Property>::property_type::value_type
    property_get(const logical_term<Property>& term);

template<typename Property>
friend void property_set(logical_term<Property>& term,
    const typename logical_term<Property>::property_type::value_type &arg);

template<typename Property>
friend std::ostream& operator<<(std::ostream &os, const logical_expr::logical_term<Property> &bf) {
    boost::io::ios_flags_saver ifs(os);
    for( auto b : bf.term_ ) {
```

```
            if( b ) os << std::noboolalpha << *b;
            else    os << 'x';
        }
        return os;
    }

private:
    static const std::string size_error_msg;
    vector<value_type> term_;
    property_type property_;
};
template<typename Property>
const string logical_term<Property>::size_error_msg = "target two operands are not same size";


// Create a logical_term with Property parsed from expr
template<typename Property = term_no_property, char Inverter = '^'>
logical_term<Property>
parse_logical_term(const string &expr, int bitsize, char const first_char = 'A') {
    logical_term<Property> term(bitsize);
    for( auto it = expr.begin(); it != expr.end(); ++it ) {
        bool value = true;
        if( *it == Inverter ) {
            ++it; value = false;
        }
        term[*it - first_char] = value;
    }
    return term;
}

//
// Setter and getter functions of term property
//
template<typename Property>
typename logical_term<Property>::property_type::value_type
    property_get(const logical_term<Property>& term)
{ return term.property_.get(); }

template<typename Property>
void property_set(logical_term<Property>& term,
    const typename logical_term<Property>::property_type::value_type &arg)
{ term.property_.set(arg); }


//
// Minimize the different 1bit of term a and b
//
template<typename Property>
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
logical_term<Property> onebit_minimize(const logical_term<Property> &a, const logical_term<Property> &b)
{
   if( a.size() != b.size() || 1 < a.diff_size(b) )
      throw std::runtime_error("tried to minimize a term which has more than 1bit different bits");
   logical_term<Property> term(a);
   for( int i = 0; i < term.size(); ++i )
      if( term[i] != b[i] )
         term[i] = dont_care;
   return std::move(term);
}

// Return minimized term which has pval as its property value
template<typename Property>
logical_term<Property> onebit_minimize(
      const logical_term<Property> &a,
      const logical_term<Property> &b,
      const typename logical_term<Property>::property_type::value_type &pval
   )
{
   logical_term<Property> term = onebit_minimize(a, b);
   property_set(term, pval);
   return std::move(term);
}


//
// class: logical function
//
template<typename TermType>
class logical_function {
public:
   typedef std::size_t size_t;
   typedef TermType value_type;
   typedef boost::dynamic_bitset<> arg_type;
   typedef typename vector<value_type>::iterator iterator;
   typedef typename vector<value_type>::const_iterator const_iterator;
   typedef logical_function<TermType> this_type;

   logical_function() {}
   explicit logical_function(const TermType &term) { add(term); }
   ~logical_function() {}

   iterator begin()             { return func_.begin(); }
   const_iterator begin() const { return func_.begin(); }
   iterator end()               { return func_.end(); }
   const_iterator end()   const { return func_.end(); }

   void swap(logical_function<TermType> &func) noexcept(true)
```

```
    { func_.swap(func.func_); }

int size() const
    { return func_.size(); }
size_t term_size() const {
    if( func_.empty() ) return 0;
    return func_[0].size();
}
void add(const TermType &term)
    { func_.push_back(term); }

void add(const this_type &func) {
    vector<TermType> tmp(func_);
    for( auto term : func )
        tmp.push_back(term);
    func_ = std::move(tmp);
}

void clear()
    { func_.clear(); }

bool calculate(const arg_type &arg) const {
    bool ret = false;
    for( auto term : func_ )
        ret = ret || term(arg);
    return ret;
}

bool is_same(const this_type &func) const {
    if( size() != func.size() )
        return false;
    for( const value_type &term : func_ )
        if( std::find_if(func.begin(), func.end(),
            [&](const value_type &t){ return t.is_same(term); })
                == func.end() )
            return false;
    return true;
}

bool operator()(const arg_type &arg) const
    { return calculate(arg); }
value_type& operator[](int index)
    { return func_[index]; }
const value_type& operator[](int index) const
    { return func_[index]; }

// The expression like "term + term = func" is not allowed
const logical_function operator+(const TermType &term) {
```

```
        logical_function ret(*this);
        ret += term;
        return ret;
    }

    const logical_function operator+(const logical_function<TermType> &func) {
        logical_function ret(*this);
        ret += func;
        return ret;
    }

    logical_function& operator+=(const TermType &term)
        { add(term); return *this; }
    logical_function& operator+=(const logical_function &func)
        { add(func); return *this; }
    friend ostream& operator<<(ostream &os, const logical_function &bf) {
        for( auto term : bf.func_ )
            os << term << " ";
        return os;
    }

    template<typename Property>
    bool operator==(const logical_function<logical_term<Property>> &func) const {
        arg_generator<> gen(0, std::pow(2, func.term_size()), func.term_size());
        for( auto it = gen.begin(); it != gen.end(); ++it )
            if( calculate(*it) != func.calculate(*it) )
                return false;
        return true;
    }

private:
    vector<TermType> func_;
};


}   // namespace logical_expr


template<typename Property>
logical_expr::logical_function<logical_expr::logical_term<Property>> operator+
    (const logical_expr::logical_term<Property> &first, const logical_expr::logical_term<Property> &second) {
    logical_expr::logical_function<logical_expr::logical_term<Property>> ret(first);
    ret += second;
    return ret;
}


#endif  // LOGICAL_EXPRESSION_HPP
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

main.cpp

```
#include <iostream>
#include <cstring>
#include <utility>
#include <stdexcept>
#include <cmath>
#include <cstdlib>
#include <boost/program_options.hpp>
#include "logical_expr.hpp"
#include "quine_mccluskey.hpp"
#include <fstream>
#include <string>

using namespace std;

unsigned int start_s=clock();
// the code you wish to time goes here

template<typename Property>
void print_term_expr(const logical_expr::logical_term<Property> &term,
          char first_char = 'A', char inverter = '~')
{
   for( int i = 0; i < term.size(); ++i ) {
      if( term[i] == false )  cout << inverter;
      if( term[i] != logical_expr::dont_care )
         cout << static_cast<char>(first_char + i);
   }
}

template<typename TermType>
void print_func_expr(
      const logical_expr::logical_function<TermType> &func,
      char first_char = 'A', const string &funcname = "f", char inverter = '~')
{
   cout << funcname << " = ";
   for( auto it = func.begin(); it != func.end(); ++it ) {
      print_term_expr(*it, first_char, inverter);
      if( it + 1 != func.end() )
         cout << " + ";
   }
   cout << endl;
}

template<typename TermType>
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
void print_truth_table(
      const logical_expr::logical_function<TermType> &f,
      char first_char = 'A', const string &funcname ="f"
   )
{
   cout << "Truth Table: ";
   print_func_expr(f, first_char, funcname);
   for( char c = first_char; c != first_char + f.term_size(); ++c )
      cout << c;
   cout << " | " << funcname << "()" << endl;
   for( int i = 0; i < f.term_size() + 6; ++i )
      cout << ((i == f.term_size() + 1) ? '|' : '-');
   cout << endl;
   logical_expr::arg_generator<> generator(0, std::pow(2, f.term_size()), f.term_size());
   for( auto arg : generator )
      cout << arg << " |  " << f(arg) << endl;
}

int main(int argc, char **argv)
{
    ifstream infile;
   int exit_code = EXIT_SUCCESS;
   try {
      bool print_process = true;
      char first_char = 'A';
      constexpr char inverter = '~';

      //
      // Parse command line options
      //
      using namespace boost::program_options;
      options_description opt("Options");
      opt.add_options()
         ("quiet,q", "never print the information of the process of simplifying")
         ("first-char,c", value<char>(), "specify a character of the first variable used for input expression")
         ("help,h", "display this help and exit");
      variables_map argmap;
      store(parse_command_line(argc, argv, opt), argmap);
      notify(argmap);
      if( argmap.count("help") ) {
         std::cout << opt << endl;
         return EXIT_SUCCESS;
      }
      if( argmap.count("quiet") )
         print_process = false;
      if( argmap.count("first-char") )
         first_char = argmap["first-char"].as<char>();
```

QUINE-MCCLUSKEY LOGIC MINIMIZATION ALGORITHM

```
    // Input a target logical function to be simplfied from stdin
    if( print_process )
      cout << "This is the Quine-McCluskey simplifier"   << endl
          << "Enter a logical function to be simplified"   << endl
          << "    (ex. \"f(A, B, C) = A + BC + ~A~B + ABC\" )" << endl
          << "[*] Input: " << flush;
    string line;

    infile.open ("/Users/suchethapanduranga/Downloads/Quine-McCluskey-master/sample/in4.txt");

      getline(infile,line); // Saves the line in line.
      cout<<line; // Prints our STRING.

    infile.close();

    //getline(cin, line);

    // Parse input logical expression and return tokenized
    logical_expr::function_parser<inverter, true> parser(line, first_char);
    auto token = parser.parse();
    // Create a logical function with logical_term<term_mark>
    typedef quine_mccluskey::simplifier::property_type PropertyType;
    typedef quine_mccluskey::simplifier::term_type TermType;
    logical_expr::logical_function<TermType> function;
    for( string term : token.second )
      function += logical_expr::parse_logical_term<PropertyType, inverter>(term, token.first.size(),
first_char);

    // Create a simplifier using Quine-McCluskey algorithm
    quine_mccluskey::simplifier qm(function);
    if( print_process ) {
      cout << endl << "Sum of products form:" << endl;
      print_truth_table(qm.get_std_spf(), first_char);    // Print the function in sum of products form
      cout << endl << "Compressing ..." << endl;
      qm.compress_table(true);                              // Compress the compression table
      cout << endl << "Prime implicants: " << endl;
      for( const auto &term : qm.get_prime_implicants() ) {      // Print the prime implicants
        print_term_expr(term, first_char);
        cout << "  ";
      }
      cout << endl << endl << "Result of simplifying:" << endl;
    }
    else
      qm.compress_table(false);

    for( const auto &func : qm.simplify() )        // Simplify and print its results
      print_func_expr(func, first_char, parser.function_name() + "\'");
    unsigned int stop_s=clock();
```

```
        cout << "Run time of this operation: " << (stop_s - start_s)/double(CLOCKS_PER_SEC)*1000 <<
"seconds"<< endl;

    }

    catch( std::exception &e ) {
        cerr << endl << "[-] Exception: " << e.what() << endl;
        exit_code = EXIT_FAILURE;
    }
    return exit_code;
}
```