Quine–McCluskey Logic Minimizer

Digital Design Project CSCE 2301

The American University In Cairo

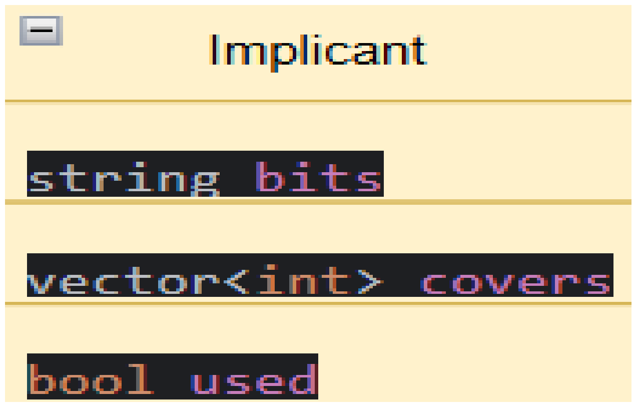
Prepared for
Dr. Cherif Salama

By
Omar Abdel Motalb
Seif eldine Ramy Mostafa
Ali Ahmed Alkholy

November 15, 2025

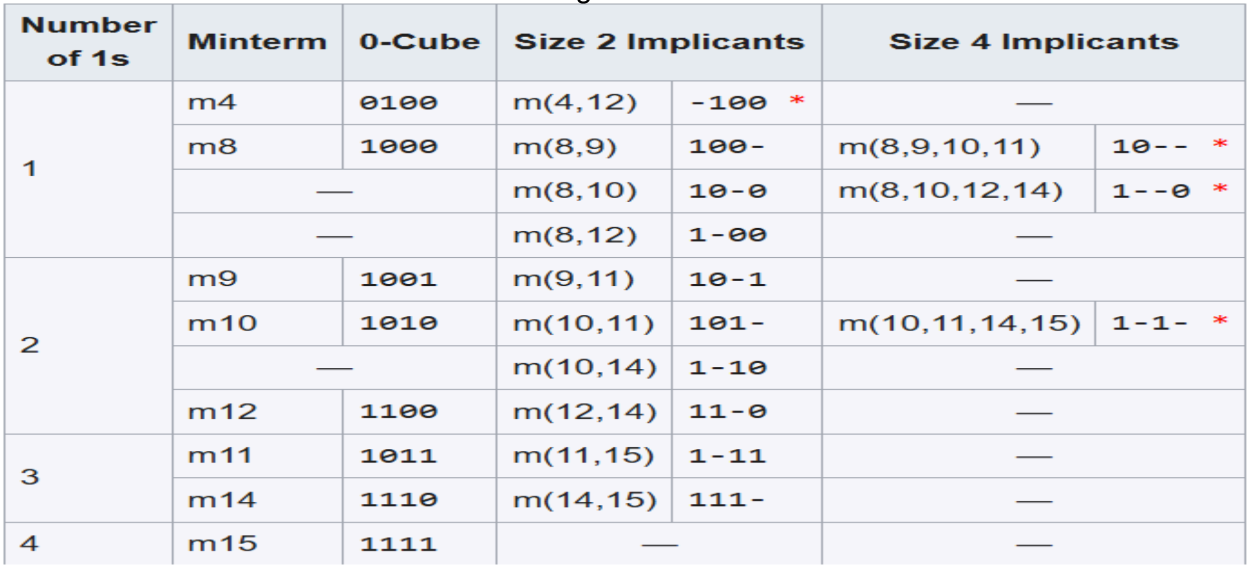
1. Program Design and Algorithms:

In this project, we implemented a systematic method for minimizing the Boolean expressions using the Quine-McCluskey algorithm. The program is divided into 3 parts, the header file “q_m.h”, the source file “q_m.cpp” and “main.cpp”. The header file contains the essential function definitions and data structures required for the minimization process. The main data structure used in the program is a struct named “Implicant”, which represents a candidate term in the minimization process. An Implicant stores a string named “bits” for its binary representation, a list of minterms or don’t care terms it covers and a flag that showcases whether it has been combined in an iterative minimization process to find the prime implicants which is going to be discussed in further detail later in the report. For better visualization, refer to the figure which showcases the Unified Language Model of Implicant. The header file also declares a set of helper functions and qm_minimize which is the main bulk of the program that will be discussed later. The main serves as the entry point of the program. It accepts input text files that contain Boolean functions specification whether it is minterms or maxterms. The main function delegates the actual minimization work to “qm_minimize” which orchestrates the input file reading, performing the algorithm and outputting information. The rest of the report is delegated on this function and its helper functions.



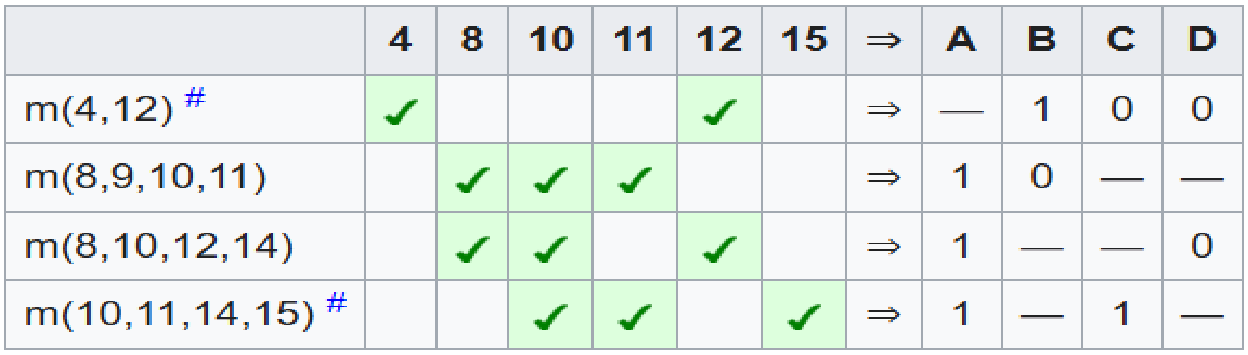
The first stage of the “qm_minimize” function is to handle the reading of the input file. The program opens the file given to the main and reads the first line to determine the number of variables in the Boolean function. It also ensures that the number is not zero or a negative number. The program then reads the second line of the file which contains either a list of minterms (SOP) or maxterms (POS). Each term is parsed, cleaned and converted into an integer where it is stored in a list for later use. A flag is also set whether the file is a maxterm or a minterm file for later importance. We followed that by error handling making sure the input of variables and terms is valid. Checking for maxterms was done in code to make it easier to compute one type of problem which was chosen to be minterms so it was linked to a converter if any Maxterms were used. With the general use of ‘term’ it was easier to later on implement all the problems done on minterms/maxterms once instead of writing separate cases for each. Integer bit operations are also used later in the algorithm for combining implicants. Parsing begins by reading the number of variables and removing whitespace. The program reads the final line which contains the don’t care terms that are extracted in the same method as the second line. The program then validates the input given by ensuring all the terms are within the range of [0, 2^(var)-1]. The program checks the flag, if the variables are minterms nothing happens, otherwise the program gets the complement of the given terms by iterating through all the variables and excluding the maxterms terms. After this normalization, the program combines the minterms with the don't care terms in a single list. With this set, the initial implicant set is formed and the program converts each term and don't care into a binary string using a simple Binary Expansion Algorithm. The binary string is stored into the Implicant's bits and the corresponding term is stored in the Implicant list of covers. The used flag is initialized as false until the combine and minimize algorithm begins. After converting all the terms into implicants, the list called "current" holds all the implicants and current will be fed the iterative prime implicant generation that follows in the main Quine-McCluskey algorithm. The program then starts the systemic combination of the implicants to identify all the prime implicants. It mirrors the tabulation process described in the formal Quine–McCluskey procedure as the loop compares each implicant pairwise and combines them if they differ in 1 bit using the first helper function "combine". Combine checks if two implicants are different in only one bit and if they are merged into a new implicant containing '-" in the place of the bit difference , both implicant used to create this possible prime implicant is true and the new implicant is added to a list "next" created in the algorithm for possible further combinations. The helper function implements the single-difference combining rule which is the fundamental mechanism that reduces Implicants in Quine–McCluskey. This iterative loop continues until the used flag is false i.e. any implicant not used in a combination and can not be further combined is a prime implicant. These uncombinable implicants are pushed in a list called primes provided that they are not duplicates. The boolean variable called "again" controls the loop after there are no further combinations possible, the algorithm reaches convergence and the loop ends. This ends the first main part of the algorithm where prime implicants are made and stored in a list in the program. To further understand the algorithm, we traced one of our tests called Test 6 in Figure 1 which can be defined as F(A,B,C,D)=∑m(4,8,10,11,12,15) + d(9,14).

Figure.1



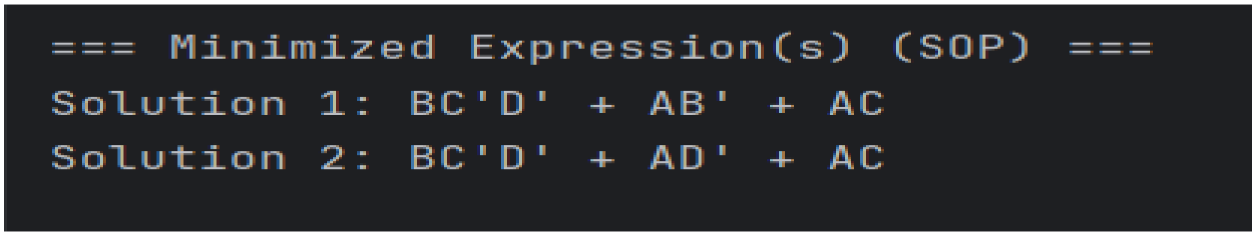
Number of 1s	Minterm	0-Cube	Size 2 Implicants		Size 4 Implicants	
1	m4	0100	m(4,12)	-100 *	—	
	m8	1000	m(8,9)	100-	m(8,9,10,11)	10-- *
	—		m(8,10)	10-0	m(8,10,12,14)	1--0 *
	—		m(8,12)	1-00	—	
2	m9	1001	m(9,11)	10-1	—	
	m10	1010	m(10,11)	101-	m(10,11,14,15)	1-1- *
	—		m(10,14)	1-10	—	
	m12	1100	m(12,14)	11-0	—	
3	m11	1011	m(11,15)	1-11	—	
	m14	1110	m(14,15)	111-	—	
4	m15	1111	—		—	

For verification and debugging, the program prints all discovered prime implicants along with the specific terms they cover. To support later stages of expression formatting, conversion, and evaluation, a set of helper functions is implemented to turn implicants into human readable expressions. The function implicantToExpr converts a bit string representation into a Boolean expression in either SOP or POS form depending on the original input format. Similarly, the implicantToVerilogTerm function converts each implicant into its Verilog format using the correct symbols and logic operators. Also, the helper function literalCount is used later in the algorithm to count how many actual literals an implicant has and ignores other don't care bits.This part of the program takes the raw input terms and organizes them so the next simplification steps can happen. By comparing, combining, and formatting implicants automatically, it creates the base needed for minimization and makes sure the rest of the algorithm works with a clean and optimized set of prime implicants. After the prime implicants have been chosen and scoped by the program, the code constructs a prime implicant table. This table essentially shows how many prime implicants cover which minterm. In our implementation, the number of prime implicants and minterms are stored in two variables of type integer called R and C. Then the table which will contain the prime implicants is made with dimensions RXC and initialized with rows of zeros as a starting point. This table will provide a matrix representation that simplifies further analysis for essential prime implicants and the application of the Petrick's method later in the code. The construction of the table happens by a double loop, where the inner loop iterates over each prime implicant and checks each minterm. If a prime implicant covers a minterm, the table will mark one for the ith minterm for the jth prime implicant. After constructing the table, the program identifies the essential prime implicants. The essential prime implicants are prime implicants that cover at least one minterm uniquely. Essential prime implicants are a must when making the final boolean expression since they cover a variable that can not be removed or suspended. The program identifies the Essential prime implicants by looping in the table that is already constructed and counting how many prime implicants cover a minterm. If a minterm is covered by only one prime implicant then the implicant is essential and then it is added to a new vector that will store the essential prime implicants. This part of the program is essential for the efficiency of the Quine-McCluskey algorithm since the essential prime vector is added at the end of every minimal solution as the solution must contain these implicants which will eliminate unnecessary combinations. In addition to that, this step will put the framework for handling uncovered minterms using Petrick's method which will be further discussed in this report. To be able to derive the final solution and Boolean expression from a set of prime implicants obtained from the program-implemented Quine-McCluskey method, the process involves identifying the essential prime implicants, which is already covered and explained, and handling the uncovered minterms using Petrick's method. The program proceeds as follows, the code uses the table previously constructed (Figure 3) where rows represent the Prime implicant bits and the columns are the minterms to cover. This table is the basis for all further reductions. The algorithm iterates over each minterm and counts the number of Prime implicants covering it and marks them as covered. If some minterms are uncovered, the program uses the Petricks's method to find all combinations of Prime Implicants that cover them. The algorithm goes as follows, each uncovered minterm produces a list of Prime Implicants that can cover it, let their name be coverers. These lists are combined systematically using sets in order to generate all the possible combinations that can be identified as a solution to cover all the minterms. Solutions are then filtered by picking only those with the minimum number of Prime implicants and then refined by picking only those with the fewer total literals by using a helper function called “literalCount” which by convention counts the number of literals in an implicant. The final few filtered refined sets that came out from Petrick’s method are then added to the Essential prime implicants. If multiple solutions exist, they are all saved in the set to be printed out. The minimized expressions are printed out in the form that they are read as by either SOP or POS the program by first using the helper function “varName” which will convert the index to a human-readable expression which is used in a grander helper function “implicantToExpr” which will convert an implicant’s bit representation into the subsequent Boolean expression string either SOP or POS form ready to be read by a human. The function skips any “-” characters in a string of bits since they represent the “don’t care” condition. If a bit is zero i.e. not checked, the corresponding variable gets negated. In contrast, if the bit is 1 i.e. checked, the variable is not negated. The variables are concatenated directly to form a product term. The function performs the previous operation if the expressions were minterms (SOP), it is inverted in the maxterms expression. For bit 1 in a string literal of bits, the variable is negated and vice versa. The function finally bridges the gap between by adding a plus sign between every prime implicant computed in the function. For better understanding and visualization, refer to Test 6 in Figure 2 and the program output in Figure 3 for final expression.



	4	8	10	11	12	15	⇒	A	B	C	D
m(4,12) #	✓				✓		⇒	—	1	0	0
m(8,9,10,11)		✓	✓	✓			⇒	1	0	—	—
m(8,10,12,14)		✓	✓		✓		⇒	1	—	—	0
m(10,11,14,15) #			✓	✓		✓	⇒	1	—	1	—

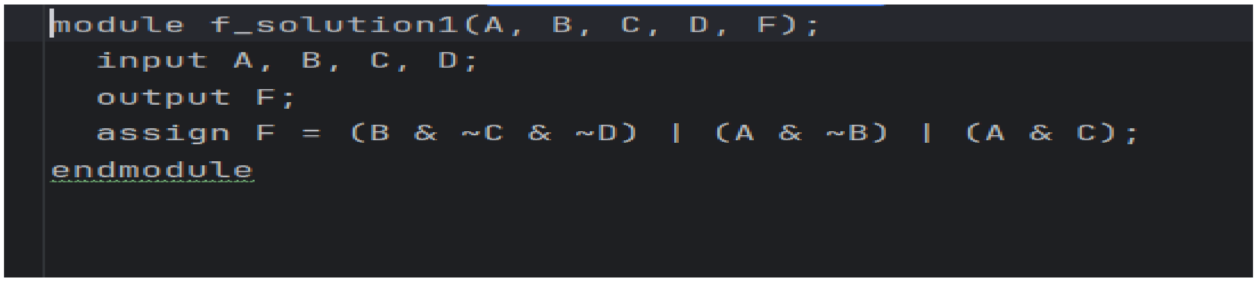
Figure.2



```
=== Minimized Expression(s) (SOP) ===
Solution 1: BC'D' + AB' + AC
Solution 2: BC'D' + AD' + AC
```

Figure.3

After deriving the minimized Boolean expressions, the program generates synthesizable Verilog modules which correspond to each minimum solution set. For every minimized solution, the program creates the input declarations by using the helper function “varName”. The output declaration is then created to a single variable F which will be assigned to the minimized expression already derived by the program. The program first builds the assigned statement by first converting each implicant to a Verilog expression using the essential helper function “implicantToVerilogTerm” which is similar to the function “implicantToExpr” function in terms of negation of the variable or not, but it differs in the form of combining expressions. For the minterms, it uses AND to combine variables and OR to combine the entire expression while in the maxterms it inverts the usage of the gates. After the construction of the assigned expression, the verilog file is constructed. This final stage bridges the gap between theoretical expressions and practical hardware design ready for implementation. For further reference, check Figure 4.



```
module f_solution1(A, B, C, D, F);
  input A, B, C, D;
  output F;
  assign F = (B & ~C & ~D) | (A & ~B) | (A & C);
endmodule
```

Figure 4

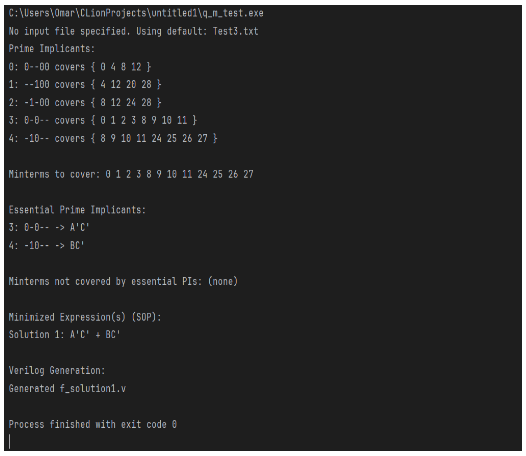
To conclude this part of the report, we implemented the Quine-McCluskey method for minimization of systematic boolean functions. The approach we took consisted of several key steps in the bulk function the “qm_minimize” as well as helper functions where the essential prime implicants are defined using the Petrick’s method. The represented code provides a simple, organized and complete method to minimize Boolean expressions while also being suitable for hardware implementations to come.

2. Challenges:

The challenges we faced while we were implementing the program were trying to avoid duplicate implicants that are formed from combining the terms at the start of the program. Ensuring the correctness of the combination logic as well as Pertricks method by tracing the algorithm on simple test cases that will be discussed in the Testing section. Finally, we had a problem with creating verilog files and parsing them correctly.

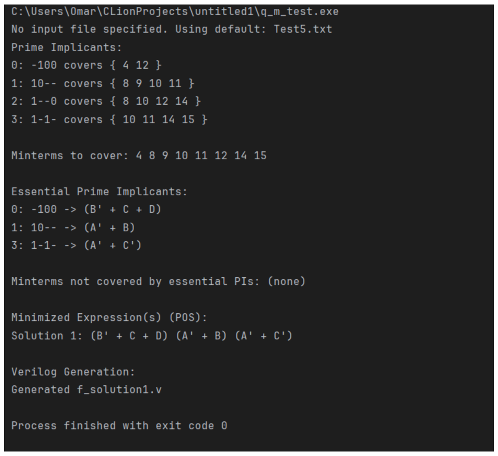
3. Testing:

To validate the correctness and robustness of our program, we used 10 test cases each equipped with a challenge for our program to handle. For test case one, two and three, the program should be able to handle SOP expressions covering minterms of different increasing variable sizes while ignoring the don’t cares (Figure 5). We confirmed the results by tracing the algorithm by hand. For test case 4 and 5, we challenged our program to handle max terms and give us the Product of Sums expression (Figure 6). We traced test case 6, which is the exact replica of 5 but minterms, while discussing the algorithms in the program. For test case 7 to 10 we were pushing the program limits by having up to 20 variables in the test case to see how the program would handle it (Figure 7).



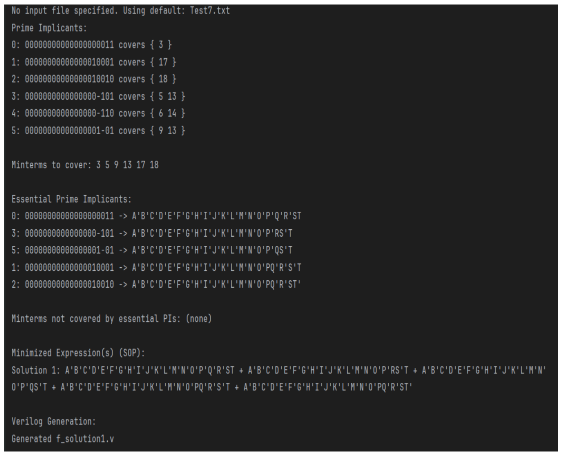
```
C:\Users\Omar\CLionProjects\untitled1\q_m_test.exe
No input file specified. Using default: Test3.txt
Prime Implicants:
0: 0--00 covers { 0 4 8 12 }
1: --100 covers { 4 12 20 28 }
2: -1-00 covers { 8 12 24 28 }
3: 0-0-- covers { 0 1 2 3 8 9 10 11 }
4: -10-- covers { 8 9 10 11 24 25 26 27 }

Minterms to cover: 0 1 2 3 8 9 10 11 24 25 26 27

Essential Prime Implicants:
3: 0-0-- -> A'C'
4: -10-- -> BC'

Minterms not covered by essential PIs: (none)

Minimized Expression(s) (SOP):
Solution 1: A'C' + BC'

Verilog Generation:
Generated f_solution1.v

Process finished with exit code 0
```

Figure.5



```
C:\Users\Omar\CLionProjects\untitled1\q_m_test.exe
No input file specified. Using default: Test5.txt
Prime Implicants:
0: -100 covers { 4 12 }
1: 10-- covers { 8 9 10 11 }
2: 1--0 covers { 8 10 12 14 }
3: 1-1- covers { 10 11 14 15 }

Minterms to cover: 4 8 9 10 11 12 14 15

Essential Prime Implicants:
0: -100 -> (B' + C + D)
1: 10-- -> (A' + B)
3: 1-1- -> (A' + C')

Minterms not covered by essential PIs: (none)

Minimized Expression(s) (POS):
Solution 1: (B' + C + D) (A' + B) (A' + C')

Verilog Generation:
Generated f_solution1.v

Process finished with exit code 0
```

Figure.6



```
No input file specified. Using default: Test7.txt
Prime Implicants:
0: 00000000000000000011 covers { 3 }
1: 00000000000000010001 covers { 17 }
2: 00000000000000010010 covers { 18 }
3: 0000000000000000-101 covers { 5 13 }
4: 0000000000000000-110 covers { 6 14 }
5: 0000000000000001-01 covers { 9 13 }

Minterms to cover: 3 5 9 13 17 18

Essential Prime Implicants:
0: 00000000000000000011 -> A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'P'Q'R'ST
3: 0000000000000000-101 -> A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'P'RS'T
5: 0000000000000001-01 -> A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'P'QS'T
1: 00000000000000010001 -> A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'PQ'R'S'T
2: 00000000000000010010 -> A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'PQ'R'ST'

Minterms not covered by essential PIs: (none)

Minimized Expression(s) (SOP):
Solution 1: A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'P'Q'R'ST + A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'P'RS'T + A'B'C'D'E'F'G'H'I'J'K'L'M'N'
O'P'QS'T + A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'PQ'R'S'T + A'B'C'D'E'F'G'H'I'J'K'L'M'N'O'PQ'R'ST'

Verilog Generation:
Generated f_solution1.v
```

Figure.7

4.Instructions to build the program:

After downloading the zip file from the git repository, compile the program with `g++ main.cpp q_m.cpp -o qm_minimizer` on the terminal and prepare a text file specifying the number of variables, minterms/maxterms, and optional don't-cares in three lines separate or use the built in test cases. Run with `./qm_minimizer` (uses default file) or `./qm_minimizer <filename>`. The program outputs minimized expressions and generates corresponding Verilog modules.

5.Problems/remaining issues in the program:

The program does not have any interface or GUI so it is less interactive and user friendly. Despite the program being able to handle 20 variables and even above it, it slows down due to the increasing time complexity of Petrick's and Quine-McClusky algorithm. The program currently does not perform input validation beyond basic formatting; incorrect or malformed files can cause abrupt termination.

6.Contributions:

Omar ElHussein Abo El Makarem Abdel Motalb: Implementation of Petrick's method, helper functions responsible for converting the literals to readable boolean expressions and verilog as well as extracting essential prime implicants and managing the Github Repository.
Seif eldin Ramy Mostafa: Implementation of the combining of implicants algorithm and extracting the prime implicants.
Ali Ahmed Alkholy: Implementation of file reading and handling maxterms and minterms files.