The American University in Cairo

Quine-McCluskey Minimizer

A C++ Program

Amal Fouda, Eslam Tawfik, Nour Selim

Fall 2023

Project Description:

- The project is intended to do the following:

  1. Read in (and validate) a Boolean function using the SOP and POS.
  2. Print the truth table, canonical SoP and PoS, generate and print the prime implicants
  3. Print the essential prime implicants (EPIs) with the non covered minterms
  4. Solve the PI table and print the minimized Boolean expression
  5. Print the Kmap
  6. Draw the logic circuit of the minimized function

Point 1:

- The program uses a class called "Implicant" which we used to call objects from. We begin by entering a Boolean expression, which goes through a validation checker to make sure that the entered expression is valid. If the expression is invalid, the program terminates.
- bool isValid(char c): checks if the string is a lowercase letter if not it return false
- bool checkSOP(string str) returns true or false based on the validation check done by the function. The function checks what comes before and after the literal. It can only accept if there is a complement or a plus sign. This function is called in 'check' to validate the input.
- bool checkPOS(const string& expression) returns true or false based on the validation check done by the function. The function checks for the order of a PoS expression as we know it; it has an opening parenthesis and must end the literal with a closing parenthesis which is checked by this function. It also allows for complements and does not accept any other irrelevant character.
- bool check(const string& expression) prints true if the expression is SoP or PoS, and prints out SoP or PoS. If true is returned, the if statement in the main function executed the rest of the program, if false, the program prints out 'invalid input' and terminates.

Point 2:

- vector<bool> dec_to_bin(int num, int num_of_bits) this function turns each row index into its binary representation in a suitable size that matches all other rows' indices.
- vector<string> separator_SOP => this function separates the expression based on the + sign into smaller expressions.
- vector<string> separator_POS => this function separates the expression based on the)(sign into smaller expressions.
- bool calculate_expression_SOP => to calculate a product term based on the values of the literals passed to it.
- bool calculate_expression_POS=> to calculate a product term based on the values of the literals passed to it.
- bool calculate_function => based on the type of the expression (SOP or POS) it uses the above functions to calculate the output of the whole expression for the given values of the literals.
- vector<char> NumberOfLiterals(string str)=> it returns the literals in the expression.

- string bool_string(vector<bool> v)
- vector<Implicant> settingToclass(map<int, vector<bool> > minterms)
- vector<Implicant> truth_table_generator(string s) => this is the main function in part 2 that loops over all the rows of the truth table and generate values for the literals at this row and then calculate the output of the expression at this specific row using calculate_function.

Point 3:

- Class Imlicant: this class has three components vector of integer to save the minterms, string binary to save the binary representation, and bool combine to help while tracking and combining the implicants.
- vector<Implicant> settingToclass(map<**int**, vector<**bool**> > minterms): this function takes a map(key, vector<bool>). This map is filled by the truth table, then it assign the inputs in this map to the vector of objects, so we can use to to extract the prime implicants.
- int countOnes(string binaryLiteral): this function takes a binary number in the form of a string and counts the number of ones in the binary number, then return the number of ones.
- string replace_diff(const string& literal1, const string& literal2): it takes two binaries and finds the difference between them and replaces it with "-", then returns a new binary number with the replaced value.
- bool logic_diff(const string& a, const string& b): this function takes two binary values and returns true if the logic difference between them is one.
- vector<vector<Implicant> > find_groups(vector<Implicant> Tawfik): this function takes vector<Implicant>, and this a vector of objects. In this function, we try to group binary numbers depending on their number of ones, then put these objects in 2d vector of class Implicant. The binary numbers are assigned to the 2d vector depending on the number of ones. When we know the number of ones, we put this binary number in the row that matches its number of ones. Then, it returns the 2d vector.
- vector<Implicant> handling(string str1, string str2, string new_binary, vector<Implicant> v1): this function merges two implicants represented by their binary strings (str1 and str2) to produce a new implicant with a binary representation (new_binary). It then updates a vector of implicants (v1) by marking the original implicants as combined and adding the newly merged implicant if it doesn't already exist. This function helps in combining and tracking the implicants that can be merged to simplify the function more.
- vector<Implicant> prime_Impicants(vector<Implicant> Tawfik): this function is designed to find out the prime implicants. This function starts by the while loop, and in this loop we call find_group function to create the 2D vector. Within the nested loop, the function utilizes logic_diff to check if two implicants can be combined. Specifically, if two implicants from adjacent groups differ in only one bit position, they are suitable for combination. Then the replace_diff function is called to generate a new binary for the newly merged implicant by replacing the difference with " -". Then the function calls the handling function to update the Tawfik list with this new implicant and the minterms that have been marked as combined. Once it finishes, it filters the vector of objects by removing the minterms in which its combined bool is true from the vector. Then, it identifies the prime implicants by checking if its combined bool is false and if it matches the count size. If these conditions are achieved, we push this implicant to another vector of objects called extract_primes. Then, it deletes these prime implicants from the primary vector of objects. Then we update the count by multiplying by 2, because every time two implicants are combined, the number of indices they cover is doubled. Finally, we return the vector extract_prime.

Point 4:

- After obtaining the vector that has the prime implicants, we call a function that finds the essential prime implcants. This is done by checking the minterms for each prime implicant; if a minterm is covered once, then this cover is essential. Afterwards, we use a different function to convert the binary string into an expression, and return a vector<int> that includes the minterms covered by the essential prime implicant.
- vector<int> findEssentialPrimeImplicants(vector<Implicant>& amal) returns a vector of integers that have the minterms covered by essential prime implicants, which is a vector we later on use in point 5. This function takes vector<Implicant> amal, which has all the prime implicants after going through the implication table. It counts the number of times a minterm is present in all of the PIs. If a minterm appears only once, then this minterm belongs to an essential prime implicant. We track this counting through using a map, where the map stores the minterm and the number of times it shows up. Lastly, we push all the indices from the implicant into essentialMintermsAmal, which is the vector we send to the next point.
- string binaryToExpression(string s) is a function that recieves a string that has the binary value of a literal (or in this case, the implicant) and returns the boolean expression. It neglects the ‘ -’ part of the binary value and goes on to the next literal based on alphabetical order. This function is called in the findEssentialPrimeImplicants to display the essential PIs as boolean expressions.

Point 5:

- vector<int> mintermsnotcoveredbyessential(vector<int>nour, vector<Implicant>primes): this function takes the vector of integers that has the minterms that are covered by essential implicants and the vector of objects of prime implicant. It loops over the vector of objects of prime implicants, gets out the minterms covered by them, and pushes them in another vector of integer called vector<int> p. Then, it loops over the vector that has the minterms covered by essential prime Impicants and vector p that has all minterms covered by prime implicants. When it finds a minterm that it isn’t covered by the essential, it pushes it to another vector of integers called prin. Then, it uses a sort and unique built-in function in vectors to remove duplicates. Finally, it prints it out.
- void print_nonessentialminterms(vector<**int**>a): this function prints out the function that are not covered by essential covers by taking a vector of integers from mintermsnotcoveredbyessential. Then, it check if the vector is not empty, it print the vectro, otherwise it print that no minterms aren’t covered by essential prime implicants.

Point 6:

- We create a function called findEssentialPrimeImplicant_bonus to begin the 3 step heuristic method to apply the last step for Quine-McCluskey. This function is different than the findEssentialPrimeImplicant; as previously mentioned, findEssentialPrimeImplicant returns a vector of minterms. This function – findEssentialPrimeImplicant_bonus returns vector<Implicant> so that we can send this new vector onto a function that deletes the essential prime implicant and keeps the non-essentials. This achieves step 1 out of 3. N
- Next, we call function eliminateDominatingColumns, which calls a Boolean function that checks if a column is dominating. If it is, the column is removed. The third and final step is to call function eliminateDominatedRows, which follows similar design to step 2; it calls a boolean function that checks is a row is dominated, and if it is, it is removed. After all three functions are called, we call a print function that prints the minimized boolean expression.

Point 7:

- Int Number_Literals(string str): this function takes a string that represents the boolean expression and returns the number of literals in the boolean expression.

- Void Kmaps_print(string str, vector<int>minterms): this function takes the boolean expression and a vector of integers that has all the minterms from the truth table. Firstly, we check if the size of the literals is in the allowed range; if not, it terminates the function. If the size is within the range, we create if condition and determine the number of rows and columns depending on the size. If it is of size two, we have two rows and two columns. If it is of size three, we have two rows and four columns. If it is of size four, we have four rows and columns. Then, it initializes kmap[row][col] with zero. Then we create an array called grey code={0,1,3,2} to help in reversing the order in the Kmap. Then, it loops over the minterms vector and determines the position of the row and the column of each minterm. If we want to know the row position of the minterm, we divide the minterm by the number of columns (m/col). If we want the column position, we find the remainder of the minterm and the number of columns (m%col). After finding the position of the minterm, we assign it by one in the Kmap. Finally, we print the Kmap.

Point 8 :

- Void display_circuit(expression)=> This function turns the expression it gets into wavedrom code integrated inside html code and stores that into a string that is to be written later into an html file.
- Open_url()=> this function opens the html file into the browser displaying the circuit design.

Problems in the program

- While the program does provide the required specifications, it begins to slow down after entering 5 variable Boolean expressions. We found out that our computers may crash due to the complexity of the computation.
- Function kmaps_print can not show the covers on the Kmap; it only shows the minterms.
- The bonus point on applying the 3-step heuristic QM was a challenge; although the logic of the function was fine, we were unable to seamlessly implement this specification.

How to build and run the program:

- Making sure that you are using c++ compiler because this code is codded using c++
- Enter a valid boolean expression it has to be in the form of SOP or POS
    1. Valid boolean expression has to follow:
        a. Has to be in in lowercase letter from a to z
        b. No complement symbol (’) before the variable
        c. Characters are not allowed, whether they are symbols such as (-, _)
        d. Numbers are not allowed as input
- Then, run the program and you will find the truth table, minterms & maxterms, canonical SOP &POS, prime implicants, essential emplicants, minterms not covered by essential, and the Kmap.

Contributions by each group member

- Amal Fouda:
    1. Implemented part 3, this part find the prime implicant by taking the minterms that are generated in the truth table and binary representation in a map(int,vector<bool>). Then, it converts the information in the map to vector of object, so it can find out the prime implicats
    2. Implemented part 5, this part print the minterms that are not covered by essentials by taking vector of integers from the the function that find essential implicants. Then, print them.
    3. Implemented part 7, this part print out the kmap. It takes vector of integer that has the minterms and boolean expression. Then, it represent this minterms on the Kmap.

- Eslam Tawfik:
    1. Implemented part 2 completely using the truth table generator function along with various functions that displays the truth table and the canonical SOP and POS.
    2. Worked on part 6 by creating a function that prints out the coverage table and then finds and extracts the essential prime implicants then deletes them and their corresponding minterms from the table. Then I call Nour's function to complete the 3 heuristic steps.
    3. Implemented part 8 completely by creating an html file and a function that writes to it wavedrom code format integrated into html file and opening that file into the browser to display the logic circuit of the minimized expression.
- Nour Selim:
    1. Implemented part 1 completely using the checkSOP, checkPOS and check functions to validate the input expression. These functions get called in part 2 to validate whether the expression was in PoS or SoP.
    2. Implemented part 4 completely using the findEssentialPrimeImplicants and binaryToExpression. The function findEssentialPrimeImplicants returns the expression of the essential prime implicants, and the binaryToExpression is called inside the fiindEssentialPrimeImplicants function to print out the boolean expression of the binary string.
    3. Worked on part 6 along with Tawfik; created functions isDominatedRow and isDominatingColumn, which are called in eliminateDominatingColumn and eliminateDominatedRow.