Quine-McCluskey Calculator
Technical Guide

Caquilala, Cedric

Famadico, Nichol John

Overview

Quine-McCluskey method (also known as The Tabulation Method) resolves such difficulty in simplifying minterms of more than five (5) variables. This method has two parts: the first one is to find candidates known as prime implicants by thorough searching and comparing of all minterms, and the second one is to choose which of the prime implicants will yield an expression with the least number of literals.

This application simulates the said method. It is written using Java; thus, a Java Virtual Machine is needed so that it can run on any device. The source code is named *Main.java*. Inside it is a static main method, a static class *Minterm*, and other static methods necessary in carrying out the important operations in simplifying minterms using Quine-McCluskey method. The process is divided into four (4) parts: input, table 1, table 2, and output. Each of these processes will be described below.

Part I – Input

Initially, the program will ask the user to input the literals that will be simplified using Quine-McCluskey method. These literals must be a letter and no repetitions are allowed. The program will signal an error if an invalid input is encountered, then will ask the user to repeat such input again. Valid literals will be appended in the *literalsAL* (ArrayList).

Afterwards, the program will ask for the user to input the decimal representations of the minterms involved. These inputs must be integers greater than or equal to zero, and less than the value of allowed by the number of bits used (2^n). Again, the program will signal an error if an invalid input is encountered, then will ask the user to repeat such input again. Valid minterms will be stored in the *inputAL* (ArrayList).

The provided minterms will be converted into instances of the Minterm class. This class have the following fields and methods.

Minterm Class

Type/Return Type	**Fields/Methods**	**Description**
ArrayList<Integer>	name	Represents the minterm in its decimal form.

String	bits	Represent the minterm in binary form
boolean	mark	Represents whether the minterm has been marked while carrying out in Table 1 and Table 2 operations.
Constructor	Minterm(ArrayList<Integer> name, String bits)	Parameters: - ArrayList<Integer> name - String bits Creates an instance of Minterm class, sets the field values with the parameter. Sets the field *mark* to false.
List<Integer>	getName()	Return Value: Returns the object field *name*.
String	printName()	Return Value: Returns the String representation of field *name*.
String	getBits()	Return Value: Returns the field *bits*.
void	countOnes()	Counts ‘1’s in bits. Return Value: Returns the number of ‘1’s in bits.
void	setMark(boolean mark)	Parameters: boolean mark Sets the object field *mark* by the value of the parameter.
boolean	getMark()	Return Value: Returns the object field *mark*.
String	toString()	Returns the String representation of the Minterm object.

The minterm objects are then grouped and sorted according to the number of ‘1’s in their binary form through the *group()* method. This group will be also the main parameter for the next operations. Another collection of minterms will also be created named *primeImplicants* (ArrayList) which will store the prime implicants obtained for every iteration of the steps.

Part II: Table I

The first table of the Quine-McCluskey is primarily executed by the *checkBits()* method. This method is a recursive function that compares each minterm of each consecutive groups. It yields all the prime implicants and continues until there are no more matches can be made.

The *checkBits()* method starts by comparing each minterm of the first group with each minterm of the second group, then the second group with the third group, then the third group with the fourth group, and so on until it reaches the last group. It will attempt to check every bit of the minterm's binary form. If the two minterms has less than 2 differences, their *mark* (boolean) field will be marked as true. Afterwards, a new Minterm object will be instantiated, where the *name* field of the two compared minterms is combined, and the *bits* field is the binary form of both minterms, where the differing bit is labeled as '-'. These minterms are then added to a separate group and will be compared on the next iteration.

As the process of comparison soon finished, an object-container *newGroups* (ArrayList) will be created to hold the new groupings of the minterms. If there is an instance of a duplicate, the excess will be removed by calling *removeDupe()* method. All minterms of the original groupings with *mark* fields that are false are added to the object-container *primeImplicants* (LinkedList). The new groupings that are obtained will be printed in the console, followed by the prime implicants.

The method then calls itself by passing *newGroups* and *primeImplicants* as the parameters. The process repeats until there are no more comparisons can be made. The final *primeImplicants* (LinkedListList), which holds all the obtained prime implicants, will be used for the next step.

Methods

Modifier Type, & Return	Method and Description
public static void	**checkBits(*ArrayList<LinkedList<Minterm>> groups, LinkedList<Minterm> prime*)** Carries out the operations of the Quine-McCluskey method recursively until there are no more groups to be made. It compares each minterm from two consecutive groups. It marks each minterm and combines them into a new grouping if they have at most one different bit. Minterms that are unmarked are added to the *prime* field. **Parameters:** *groups* = ArrayList of Minterm objects to compare and group if they have at most one different bit. *prime* = LinkedList of prime implicants / minterms that are unmarked.

public static ArrayList<LinkedList<Minterm>>	**removeDupe(*ArrayList<LinkedList<Minterm>> groups*)** Removes duplicates from each group. **Parameters:** *groups* = the ArrayList of groups of minterms to remove duplicates from. **Returns:** *noDupe* = the same ArrayList of groups of minterms without duplicates.
public static LinkedList<Minterm>[]	**group(*LinkedList<Minterm> input, int variables*)** Groups Minterms according to their number of '1' bits. **Parameters:** *input* = LinkedList of Minterm objects to be grouped *variables* = the number of bits to be used **Returns:** *group* = LinkedList of Minterm objects groups according to their number of '1' bits.
static String	**convert(*int n, int size*)** Converts an integer to binary of with a specified number of bits padded with zeroes if necessary. **Parameters:** *n* = integer to be converted *size* = number of bits **Returns:** *binary* = string of the converted binary of integer *n*.

Part III – Table 2

The second table of the Quine-McCluskey method is executed by the method *checkBitsAgain().* Unlike the previous method in Table 1, this is not recursive and has a return type of LinkedList<Minterm>. It has two parameters: *primeImplicants* (LinkedList<Minterm>) which will hold the prime implicants yielded after doing Table 1, and *input* (int[]) which will hold the integers that was entered by the user during the initial progress of the program. The elements of the *input* array will be collected to an object-container *inputs* (LinkedList<Integer>). Afterwards, the elements of this linked list will be sorted from least to greatest.

The method will then create a matrix named *marks* (char[][]) and two arrays namely *counter* (int[]) and *markers* (char[]). The matrix *marks* will store a table that represents the instances of a specific input integer in a specific prime implicant. The instances will be showed by the putting ‘x’ on the matrix. On the other hand, the array *counter* will hold a single row table which stores the total number of ‘x’s of each column of matrix *marks*. Lastly, the array *markers* will represent the integers that have only one occurrence (or a column of matrix *marks* with only one ‘x’) in the overall collection of the prime implicants. The character ‘/’ is used for this representation.

Afterwards, the method will iterate the elements of the LinkedList *primeImplicants* and *inputs*, if an element of the *inputs* has only one occurrence, its corresponding element in *markers* array will be marked by ‘/’. Then, the method will iterate the elements of the *primeImplicants* (LinkedList) again, and it will mark minterms containing the integers that have single occurrence by setting their *mark* fields as true. The object field *name* (ArrayList) of the minterm objects that have been marked as true will be iterated so that the corresponding elements of their names in the *markers* array will be marked by ‘/’ as well. After these iterations, the program already gathered the “essential” prime implicants.

Following the process above, the method will collect the remaining minterms that still have their field *marks* set to false and store them in the LinkedList *remainingImplicants,* and the remaining integers that still have no ‘/’ in their corresponding element in *markers* array will be stored in the ArrayList *remainingInputs*. The method will then iterate to choose the least number of remaining implicants by selecting a minterm that contains all, or most if not all, of the remaining inputs and changing their *mark* field to true. The iteration will stop if all the remaining inputs are successfully covered by the selected implicants.

The method will return a LinkedList of Minterms (LinkedList<Minterm>) containing all the final implicants gathered in the process.

Methods

Modifier Type, & Return	Method and Description
public static LinkedList<Minterm>	**checkBitsAgain(LinkedList<Minterm> primeImplicants, int[] input)** carries out the Table 2 operation of Quine-McCluskey method. Then return a collection of minterms that represents the final answer. **Parameters:** primeImplicants = (LinkedList<Minterms>) the prime implicants obtained from carrying out the Table 1 operation of Quine-McCluskey method. input = int[] an array of integers entered by the user during the initial progress of the program. **Returns**

	finalAnswer = (LinkedList<Minterm>) a collection of minterms representing the final answer.

Part IV – Output

An object in main method named *finalAnswer* (LinkedList<Minterm>) will reference to the object returned by *checkBitsAgain()* method. The minterms will be then converted into standard form through *toStandard()* method and will be printed into the console.

METHODS

Modifier Type, & Return	Method and Description
public static String	**toStandard(LinkedList<Minterm> finalAnswer, char[] literals)** converts the minterms into their standard form then return it as String. **Parameters** finalAnswer = (LinkedList<Minterm>) the group of minterms obtained after carrying out Table 1 and Table 2 operations. literals = (char[]) an array of characters entered by the user during the initial progress of the program. **Returns** standardForm = (String) the final answer in standard form.