Technical Manual

Boolean Expression Minimizer

Machine Problem 1

John Eron David Salongsongan
BS COMPUTER SCIENCE

Introduction

This guide is for the user to understand how the application works, and the methods within it.

How does the Tabulation Method work?

To understand BooM, you should know how the Tabulation method works, as it is the basis of the algorithm.

The Tabulation method can be divided into two steps:

1. Finding all **prime implicants** of the function; and
2. Using the prime implicants in the **prime implicant chart** to find the **essential prime implicants** and other prime implicants that are necessary to cover the whole function.

Step 1: Finding prime implicants

Input needed: Number of variables, minterms, don't cares(optional).

For demonstration, given $F(a,b,c) = \sum m(1,2,3,4,5) + d(0)$

1. Get the binary equivalent of every minterms and don't cares. Put them in a table, similar to the one below:

DECIMAL	a	b	c
0d	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1

Note: 'd' indicates that the number is a don't care.

After converting to binary, assign each of the binary bit to their corresponding variable assignment, from left to right. In the demonstration above, 'a' is assigned to the most significant bit while 'c' is assigned to the least significant bit.

2. Group them based on the number of 1s on their binary equivalent.

DECIMAL	GROUP	a	b	c
0d	0	0	0	0
1	1	0	0	1
2	1	0	1	0
4	1	1	0	0
3	2	0	1	1
5	2	1	0	1

3. Compare each member of the group to the members of the next group. If the binary of the member is only different by one bit, change that bit into '-', combine the binary, and place them in a new group in a new table. After that, marked the ones combined. Continue this until there is only one group left or you cannot compare anymore.

DECIMAL	GROUP	a	b	c
0d	0	0	0	0
1	1	0	0	1
2	1	0	1	0
4	1	1	0	0
3	2	0	1	1
5	2	1	0	1
0,1	a0	0	0	-
0,2	a0	0	-	0
0,4	a0	-	0	0
1,3	a1	0	-	1
1,5	a1	-	0	1
2,3	a1	0	1	-
2,5	a1	1	0	-
0,1,2,3	b0	0	-	-
0,1,4,5	b0	-	0	-

Note: Duplicated combinations of binary should be omitted before proceeding.

4. Collect the non-marked ones and proceed to the next part of the tabulation method.

Step 2: Prime implicant chart

Using the prime implicants (non-marked ones from the last step), create a prime implicant chart. To make this, create a column for the decimal combinations (prime implicants) and columns for each of the minterms (do not include the don't cares). Then, based on the digits of the prime implicants, put a mark on the column of the corresponding minterm.

For demonstration, let's use the prime implicants from the previous step and create a prime implicant table.

PRIME IMPLICANTS	1	2	3	4	5
0,1,2,3	X	X	X		
0,1,4,5	X			X	X

1. After creating the chart, mark the essential prime implicants. This can be gathered by marking the columns with only one mark. After that, mark the combinations that the marked minterms belong. Then, mark the column of the marked combinations.

PRIME IMPLICANTS	1	2	3	4	5
0,1,2,3	X	X	X		
0,1,4,5	X			X	X

In this example, 2, 3, 4, 5 are the columns with only 1 'X'. This makes (0,1,2,3) and (0,1,4,5) essential prime implicants. 1 is also marked because it is part of the group with the essential prime implicant.

2. After marking all the minterms, gather every prime implicants that have been marked on the chart and convert them into their variable forms. This can be done by converting the binary into variable form. '1' corresponds to the base form of the variable, '0' corresponds to the complement for of the variable, and '-' means ignore the variable and move to the next one.

PRIME IMPLICANT	a	b	c	VARIABLE FORM
0,1,2,3	0	-	-	a'
0,1,4,5	-	0	-	b'

Using the example before, (0,1,2,3), which corresponds to binary form (0 - -), was converted to a', and (0,1,4,5), which is (- 0 -) in binary form, was converted to b'.

3. Add the variables together to form the Sum-of-Product (SOP) form of the minimized expression.

Using the example in the last step, the SOP form is:

$$F(a,b,c) = \sum m(1,2,3,4,5) + d(0) = a' + b'$$

There are cases when the essential prime implicants did not handle all minterms or there are no essential prime implicant (all minterms have overlapped).

For example, given $F(a, b, c, d) = \sum m = (1,2,3,4,5,6)$

The prime implicant chart formed from this is shown below:

PRIME IMPLICANTS	1	2	3	4	5	6
1,3	X		X			
1,5	X				X	
2,3		X	X			
2,6		X				X
4,5				X	X	
4,6				X		X

As you can see, there are no column with only 1 'X'. In this case, there are two ways in dealing with this. One is doing trial-and-error, which becomes quite a tedious task in larger groups. Another one method is using Petrick's method. Although it is also hard at larger groups like the trial-and-error method, but at least this is a more systematic way and easier to implement in computer programs than trial-and-error.

Petrick's Method

Note: This method should be done after removing(marking) the essential prime implicants to reduce the number of steps.

This method is a systematic way of getting the possible combinations of prime implicants that can handle all the unmarked minterms.

The steps of doing this is as follows:

1. Label each of the row of the prime implicant table.

For example:

LABEL	PRIME IMPLICANTS	1	2	3	4	5	6
A	1,3	X		X			
B	1,5	X				X	
C	2,3		X	X			
D	2,6		X				X
E	4,5				X	X	
F	4,6				X		X

2. Form the product of sums. Marked rows will be added and columns will be multiplied.

Using the table from the last step, the formed formula will be:

$$(A + B)(C + D)(A + C)(E + F)(B + E)(D + F)$$

3. Apply the distributive law to transform the expression into its sum of products form. Additionally, apply the Idempotence Law $(A + A = A \text{ and } A \cdot A = A)$ and Absorption Law $(A + AB = A)$.

Using the product of sum from the last step, its sum of product is:

Note: The POS form is rearranged so that less step is needed to transform the expression into SOP form.

$$(A+B)(A+C)(B+E)(C+D)(D+F)(E+F)$$

$$((A+C)A+(A+C)B)((C+D)B+(C+D)E)((E+F)D+(E+F)F)$$

$$(A+AC+AB+BC)(BC+BD+CE+DE)(DE+DF+EF+F)$$

$$(A+BC)(BC+BD+CE+DE)(F+DE)$$

$$((BC+BD+CE+DE)A+(BC+BD+CE+DE)BC)(F+DE)$$

$$(ABC+ABD+ACE+ADE+BC+BCD+BCE+BCDE)(F+DE)$$

$$(BC+ABD+ACE+ADE)(F+DE)$$

$$((BC+ABD+ACE+ADE)F+(BC+ABD+ACE+ADE)DE)$$

$$BCF+BCDE+ABDF+ABDE+ACEF+ACDE+ADEF+ADE$$

$$ADE+BCF+BCDE+ABDF+ACEF$$

4. Each product in the SOP form is a possible minimization solution. To get the shortest expression, count the number of literals per term, and the term/s with lowest number of literals will probably be the shortest expression.

In the last example, the ones with the shortest number of literals are ADE and BCF.

5. Convert the result from step 4 into their variable form. This can be done by doing the following:
   1. Look to the labeled prime implicant chart.
   2. Convert each literal to its prime implicant equivalent, then to its binary equivalent.
   3. Add each one in the term.

Using the last example:

- ADE will be converted to $(1,3) + (2,6) + (4,5)$, then it will be converted to $A'B'D + A'CD' + A'BC'$
- BCF will be converted to $(1,5) + (2,3) + (4,6)$, then it will be converted to $A'C'D + A'B'C + A'BD'$

Boolean Expression Minimizer

After discussing the Tabulation Method, which is the basis of the algorithm of the program, we will discuss the program itself.

Boolean Expression Minimizer (BooM) is a console application written in Java programming language. It implements the Quine-McCluskey Algorithm (Tabulation Method). The main task of the application is to generate the shortest possible Boolean expression given the minterms.

Features:

1. Accepts up to 26 variables.
2. Users can customize the variable assignment.
3. Accepts don't care conditions.
4. Prints out every minimized Boolean expression, in the case of multiple "shortest" minimized expression.

Components:

The application is composed of three classes, which are the following:

1. Main.java
   Contains the main() method. Contains the 'Options'.
2. Tabulation.java
   Contains the various methods that process the user's input into the minimized Boolean expression.
3. Utility.java
   Contains the various methods that aid the "main" process of minimizing.

Note: To better understand the list of methods, check the source code for reference.

Main class

It contains the main method of the program.

Methods within:

1. main()
   Contains the "Main menu" of the application. Uses a combination of while loop and switch statement to make "Options".

Tabulation Class

This class contains the methods used in the whole process of Tabulation.

Methods within:

1. Declared variables
2. Getters and Setters

3. getMaxMintermCount()
   This method sets the variable Tabulation.maxMintermCount into 2^n, where n is the variable count, and returns it.

4. getCombinedTerms()
   This method returns the combination of minterms and dontCares int[] (combined through use of combinedArr[], a method from Utility class).

5. startTabulation()
   Calls the methods to start the process.

6. initialize()
   Initializes the declared variables.

7. getNeededInputs()
   Method that asks the user for input of variable count, variables (in another method), minterms, and don't cares.

   As shown in the code, each of them has specific permissible values. For variable count, it is only allowed to input from 1 to 26. For the minterms and don't cares, it is only allowed to input numbers from 0 to $2^n - 1$, where n is the variable count. The user also needs to press ENTER per input. In case of an invalid input like using letters for the variable count or minterms, a message will show up and let the user try again. For the minterms and don't cares part of the "input gathering", the user can "stop" by typing 'x' (lowercase letter 'x'). The minterms and don't cares will be combined using getCombinedTerms(), and this will be used in the prepInitialTable() method.

8. getVariableAssignment()
   Method that let the user choose if the variable assignments will be customized or not.

   As shown in the code, there are two choices here: customizing the variables, or sticking to the default. For the default, the user will also choose if the variables will be in uppercase or lowercase. If the user chose to customize, the user will be asked to from left to right (the first variable assignment asked will be for the most significant bit). Like in the previous method, if the user entered an invalid input, like non-single letter variable, a number, or a duplicate (Note: lowercase and uppercase letter are treated as one letter; e.g. 'A' and 'a' are considered the same), a message will be shown and ask the user to try again.

9. prepInitialTable(int varCount, int[] minterms)
   Parameters:
   - varCount – number of variables
   - minterms – array of minterms

   Method that creates the table of minterms and their binary equivalents per row.

   This method corresponds with the "1." in "Step 1: Finding Prime Implicants" (look at page 1 of the Technical Manual for reference). Each of the minterm (input) is converted to its binary equivalent. Then, it is added to a List<int[]> with an indicator (serves as the mark) and the minterm it represents. After the whole process of putting everything in a "table", it calls the method groupedInitialTable() with the ungrouped table and the variable count as parameters.

List<List<int[]>> ungroupedTable			
List<int[]> row	int[] indicator	int[] minterm	int[] binary
List<int[]> row	int[] indicator	int[] minterm	int[] binary

Figure 1: Visual representation of the ungrouped initial Table

   Note: The table is arranged from smallest to largest minterm and still ungrouped based on the number of '1's present in the binary form of the minterm.

   To get access to each row, java.util.List.get(int index) is used with the index being the row you want to access. To access a column in a row, you need to use another get() to access the data. For example, if you want to access the binary of the first row, you can use List<List<int[]> ungroupedTable.get(0).get(2), since the first row is in index 0 of the List and the binary is located on index 2.

10. groupedInitialTable(List<List<int[]>>ungrouped, int varCount)
    Parameters:
    - ungrouped – the table that will be divided into groups
    - varCount – number of variables

Method that groups the ungrouped table of minterms-binary by the number of the ‘1’s in the binary.

The method corresponds with “2.” in “Step 1: Finding Prime Implicants” (look at page 2 of the Technical Manual for reference). The number of ‘1’s will be counted for each of the binary values of the minterms using the method count1s(int[] binary), which is under the Utility class. The row is added to the group when the number of ‘1’s is equal to the index that the loop is in. The loop will run for variable count + 1 times, as there are that number of groups that can be formed. After the each grouping, if the List<List<int[]>>>is not empty, it will be added to List<List<List<int[]>>> initialGroups (a declared variable).

List<List<List<int[]>>> initialGroups			
List<List<int[]>> group	List<int[]> row		
group.get(0)	int[] indicator	int[] minterm	int[] binary
	int[] indicator	int[] minterm	int[] binary
group.get(n)	int[] indicator	int[] minterm	int[] binary
	int[] indicator	int[] minterm	int[] binary

Figure 2: Visual representation of the Grouped table (initialGroups)

11. step1(List<List<List<int[]>>> group)
    Parameters:
    - group – list of minterm-binary that is grouped based on the number of ‘1’s present in the binary

    Method that compares the current group to the next group, then make a new group until there are less than 2 groups left.

    The method corresponds with “3.” in “Step 1: Finding Prime Implicants” (look at page 2 of the Technical Manual for reference). This method is a recursive one and will finish when the size of the parameter is less than 2 (there are only one group left or there are no group at all). The method compares the binary current row in the group to every row in the next group (compared using the method is1Difference()), and if the binary have only one difference, the minterms will be combined (using the method combineArr[]) and the index of the bit with one difference will be marked with ‘-1’ (instead of ‘-‘). Lastly the indicator of the current row and the row it is compared to will be marked (marked from ‘-1’ to ‘0’). Also, duplicates will not be added to the group (there is a method that checks this). After comparing the whole group, it will be added to the step1Table.

    Note: There is a method for displaying the step1Table. If the user wants to see it, just remove the “//” in front of the line “Utility.showTable(getStep1Table());”.

List<List<List<List<int[]>>> step1Table				
	List<List<int[]>> group	List<int[]> row		
step1Table.get(0)	step1Table.get(0).get(0)	int[] indicator	int[] minterm	int[] binary
		int[] indicator	int[] minterm	int[] binary
	step1Table.get(n).get(0)	int[] indicator	int[] minterm	int[] binary
		int[] indicator	int[] minterm	int[] binary
step1Table.get(n)	step1Table.get(n).get(0)	int[] indicator	int[] minterm	int[] binary
		int[] indicator	int[] minterm	int[] binary
	step1Table.get(n).get(n)	int[] indicator	int[] minterm	int[] binary
		int[] indicator	int[] minterm	int[] binary

Figure 3: Visual representation of the step1Table

12. setupStep2(List<List<List<List<int[]>>>> table)
    Parameters:
    - table – the table (from step 1) with appropriate markings

    Method that gathers every row with unmarked indicators and put them into step2Table.

    The method corresponds with "4." in "Step 1: Finding Prime Implicants" (look at page 2 of the Technical Manual for reference) and the initial step in "Step 2: Prime Implicant Chart" (look at page 3 of the Technical Manual for reference). The method checks every row and gathers the unmarked indicators (int[] indicator with '-1' as its element). If unmarked, the minterm combination and its binary combination will be added to step2Table with an indicator and a checklist of present minterms (not including the don't cares), which are filled up using a loop.

    Note: There is a method for displaying the step2Table. If the user wants to see it, just remove the "//" in front of the line "Utility.display(getStep2Table());".

List<List<int[]>> step2Table				
	List<int[]> row			
step2Table.get(0)	int[] indicator	int[] minterms	int[] binary	int[] checklist
step2Table.get(n)	int[] indicator	int[] minterms	int[] binary	int[] checklist

Figure 4: Visual representation of step2Table

13. step2(List<List<int[]>> table)
    Parameters:
    - table – the table generated from setupStep2

    Method that gathers the essential prime implicants by counting the number of marks per index.

    The method corresponds with "1." In "Step 2: Prime Implicant Chart" (look at page 3 of the Technical Manual for reference). The method checks if the minterm is an essential prime implicant by counting the number of marks per minterm, and get the ones with only one mark. The essential prime implicants will be marked and the binary that have the minterm will be added to the essential list, and it will be also marked in the checklist. For the unmarked, it will be added to another table. The unmarked rows will be added to another table, this will be used if there are still unused minterm. If all minterms are already covered it will proceed to the next step, else it will call the method petrick().

14. petrick(List<List<int[]>> table)
    Parameters:
    - table – the non-essential prime implicants

    Method that implements the Petrick's Method.

    This method corresponds to the Petrick's Method (look at page 4 of the Technical Manual). Each of the row in the table with be given a label. Then the POS will be generated by using the checklist, the index of the unused minterms will be gathered and these indexes will be checked if the checklist in the row is marked. Those that are marked in a column will be added, and these will be multiplied. After gathering the POS, this will be converted to SOP (and simplified). Then the shortest term will be converted back and added to the essential prime implicants from step2 (if multiple terms, each of them will be added), then added to listOfPossible.

15. Methods related to Petrick's Method
    a. shortestCombs(String[] a)
       Parameters: a – sum of product form of the simplified expression
       Method that returns the shortest terms in a sum of products expression.
    b. remove(String a)
       Method that applies the Idempotence Law. Duplicate terms will be removed from the String.
    c. absorp(String a)
       Method that applies the Absorptive Law. The string will be sorted by length and the lowest one will be grouped and compared to the rest (it is grouped too). Then the absorptive law will be applied, the nonrepeating ones will be added to a HashSet<String>, that will be converted into an expression after all the terms have been checked.
    d. contain(String a, String b)
       Method that checks if the a is on b.
    e. simplify(List<String> exp)
       A method that simplifies the expression into a SOP form. Each of the "members" of the List is multiplied to each other. This is a recursive method and will stop when the size of the parameter is only less than two (multiplication is not possible anymore). For this, the multiplication will happen by pairs. If the current "sum" doesn't have a "partner", it will be paired to an empty String. When the "stop" condition is reached, extra simplifications will be applied (remove() and absorp() are called).
    f. simplifyInd(String a, String b)
       A method that applies the Distributive Law and other Boolean algebra Laws. This method is used for multiplying expressions.
    g. removeDup(String a)
       Method that checks for duplicate characters in a term.
    h. simplifyFurther(String a)
       Method that applies Absorption Law. It is a recursive function that ends when if checkIfSimplest(String) returns true.
    i. checkIfSimplest(String a)

Method that checks is already the simplest form. It applies the Absorption Law. If the expression is eligible for Absorption Law, then it will return false, as it is not the simplest form when it is possible to apply the law.

j. binaryToSOP(List<List<int[]>> combs, char[] variables)
   Method that converts binary stored in combs into its SOP form.
k. printShortest(List<String> sop)
   Method that prints the shortest SOP form. This can be determined by the number of literals. In case of multiple shortest SOPs, everything will be printed and will be separated by lines.

Utility Class

This class contain the methods that helps the methods in the Tabulation class.

Methods within:

A. Methods that return Boolean
   Note: The methods here use counters, which counts the occurrences. Then the result of these counters will determine what will be the result.

1. isNumeric(String input)
   Method that checks if the string is composed of only numbers. If yes, the method return true, else return false.
2. isAlphabet(String input)
   Method that checks if the string is composed of only alphabets. If yes, the method return true, else return false.
3. isAlreadyThere(int[] arr, int input)
   Method that checks if the input is already included in arr.
4. alreadyThere(int[] a, List<int[]> b)
   Method that checks if a is already in b.
5. is1Difference(int[] a, int[] b)
   Method that checks if a and b only have one difference.

B. Methods that return int[]

1. initArray(int size)
   Method that initializes an int[] with its length equal to size. The array is filled up with '-1' as its elements.
2. findingSimilar(int[] a, int[] b)
   Method that gathers the similar elements from two int[].
3. findingDifference(int[] a, int[] b)
   Method that gathers the elements that is in a but not in b.
4. convertSetToIntArr(Set<Integer> a)
   Method that converts a Set<Integer> to an int[].
5. convertDecToBin(int decimal, int varCount)
   Method that converts a base-10 into base-2. Additionally, this method also adds zero in front if the length of the binary is less than varCount and place each element at a int[].
6. arraySorter(int[] arr)
   Method that sort arr into ascending order.

7. countChecks(List<List<int[]>> a, int size)
   Method that is specifically made to assist step2(). This method counts the number of marked indexes in the checklist of each row of the table (step2Table).
8. markedDifference(int[] a, int[] b)
   Method that is specifically made to assist step1(). This method checks every element of the int[]. If the element on the index of a and b is similar, it will be put to int[] marked, else '-1' will be the value of that index for marked. After the process, the method will return marked.
9. combineArr(int[] a, int[] b)
   Method that combines a and b into one single array, and then sort this combined int[]. The method will return the sorted combined int[].

C. Methods that return int
   1. getArraySize(int[] arr)
      Method that counts the number of non- '-1' elements in arr. The method will return this count.
   2. count1s(int[] binary)
      Method that counts the number of '1's in binary. The method will return this count.
   3. countDifference(int[] a, int[] b)
      Method that counts the number of differences per index of a and b. The method will return this count.
   4. elemCount(int[] a)
   5. Method that counts the number of elements in a by counting the number of times the for loop of iterating a happened. The method will return this count.

D. Other Methods
   1. display(List<List<int[]>> a)
      Method that is specifically made for step2Table. This method prints a (table).
   2. showTable(List<List<List<List<int[]>>>> a)
      Method that is specifically made for step1Table. This method prints a (table).

      Note: These two methods are commented out in the source code. Remove the "//" in front of them for the user to see the tables.

If you want to edit the source code, you can use a text editor, or better yet an IDE, as it is needed to be compiled. Access the 'src' folder for the source code. This concludes this manual. Thank you for using Boolean Expression Minimizer.

References

Wikipedia contributors. (2021, May 29). *Petrick's method*. Wikipedia.

https://en.wikipedia.org/wiki/Petrick%27s_method