An O(2^N) Expression Generator

Irvan Jahja / 13509099
School of Electrical Engineering and Informatics
Institut Teknologi Bandung

December 16, 2010

Abstract

In several disciplines of sciences there are needs to create systems which elicit behaviors that can be expressed in terms of logical expressions. This paper will give a method to obtain an expression that uses only not, and, or, and xor operations and is comparatively more efficient than the traditional sum-of-product or product-of-sum methods. This paper develops a strict bound on the number of such operators in term of N, the number of free variables that is in the expression.

Section 1 will talk about the motivation behind the research. Section 2 will talk about the terminologies used in the paper. Section 3 will formally presents the problem. Section 4 will revolve around the method used to obtain the corresponding expression. Section 5 will talk about the complexity. Section 6 compares the actual performance of this method to the generic sum-of-product method. Finally, Section 7 concludes the paper.

1. Introduction

The two most widely used methods when we talk about generating expressions in class are sum-of-product and product-of-sum. Both are very simple and greatly benefits from their simplicity. However, both often use much more operations than is perceived as acceptable (Section IV). This paper tries to perform a trade-off between these benefit. It gives a less simple method which will perform better than the usual simple methods.

The author conjectured that the extra complexity will be dominated by the benefits. In addition to the complexity that arises because of the complexity of the expression, there may be more delay that occurred in the hardware layer of the expression that may not coincide with what the designer had in mind. In addition, the algorithm given in this paper can be implemented easily within a computer so that the complexity of computing such expressions can be effectively hidden.

2. Terminology

This paper will use a notion that is loosely based on [1]. A proposition is a statement whose value is either true or false. We let 1 denotes a true proposition and 0 denotes a false proposition. Free variable is a proposition whose value is not known. A unary operator for the purpose of this paper will be a logical operator that operates on a single proposition, and whose result is another proposition. There are one unary operator that will be used in this paper : NOT (negation operator). For the sake of conciseness it will often be represented with the symbol ! (as in [2]). whose behavior is summarized below (A is a proposition).

A	!A
0	1
1	0

Table 2.1 : Truth table for negation.

A binary operator for the purpose of this paper will be a logical operator that operates on two propositions, and whose result is another proposition. There are three binary operators that will be used throughout the paper : OR (disjunction), AND (conjunction), and XOR

(symmetric difference). For the sake of conciseness they will be represented by the symbols (&, |, and ^) as in [2]. Their behaviors are summarized in the following tables (A and B are both propositions) :

A	**B**	**A & B**	**A | B**	**A ^ B**
0	0	**0**	0	0
0	1	**0**	1	1
1	0	**0**	1	1
1	1	**1**	1	0

Table 2.2 : Truth table for binary operators.

This paper will not discuss further about operator precedence, interested readers may consult [1]. Instead, we will extensively use parentheses to denote which operator precedes which so as to not find any ambiguity. The only exception is when there are both unary operator and a binary operator on the same level, the unary operator must be resolved first.

A truth table displays the relationships between the truth values of propositions [1].

3. Problem

Given the truth values for an expression (possibly in the form of a truth table), derive an expression that has the same truth values, contains only negation, conjunction, disjunction, and symmetric difference operators.

For example, suppose we are given the following truth table.

A	B	T
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.1 : Example truth table

One of the optimal solutions is A | B.

4. Solution

Without loss of generality, we will assume that there are at least two free variables in the expression (the case that there is only one or zero free variable is trivial). We will construct the expression recursively using induction-like steps. We start with the basis step :

BASIS STEP: 2 free variables

We construct the basis step by a case-by-case analysis.

Result when A&B is <AB>				Expression
<00>	<01>	<10>	<11>	
0	0	0	0	0
0	0	0	1	A & B
0	0	1	0	A & !B
0	0	1	1	A
0	1	0	0	!A & B
0	1	0	1	B
0	1	1	0	A ^ B
0	1	1	1	A | B
1	0	0	0	!A & !B
1	0	0	1	!A ^ B or A ^ !B
1	0	1	0	!B
1	0	1	1	A | !B
1	1	0	0	!A
1	1	0	1	!A | B
1	1	1	0	!A | !B
1	1	1	1	1

Table 4.1 : Basis step table

INDUCTIVE STEP:

Assume that an expression that contains N free variables can be formulated with this method. In an expression with (N+1) free variables, we pick one (X). Now we have N free variables and by the inductive step we can create any expression that we want that that contains only these free variables. We create two of these, Exp1 and Exp2 (the values picked will be

described later). Now, we construct the expression with (N+1) free variables as follows.

Expression = (Exp1 & X) | (Exp2 & !X) *(e4.1)*

All that left is to pick the values of Exp1 and Exp2 in such way that the expression above corresponds to the truth values of the original expression. There are 2^(N+1) entries in this truth table. Now we group these entries into 2^(N) pairs, where each pair consists of two entries that differs only at the value of X. We call the other non-X part is A. It can be seen that A differs for all the pairs for otherwise there will be multiple identical entries in the truth table. We can then associate a value for both Exp1 and Exp2 independent between pairs (they corresponds to A for each pair, which we have claimed to be distinct). Now, there are four cases.

X	Entry_Result	Exp1	Exp2
0	0		0
0	1		1
1	0	0	
1	1	1	

Table 4.2 Exp1 and Exp2 case table

By the inductive step, we can create both Exp1 and Exp2. Hence, the method is complete.

5. Complexity

We will now bound the number of both unary and binary operations found in the expression generated by the method above.

(Definition 5.1) B(N) : Upper bound on the number of binary operators for an expression with N free variables.

(Definition 5.2) U(N) : Upper bound on the number of unary operators for an expression with N free variables.

Clearly, B(2) = 1 *(e5.1)* and U(2) = 2 *(e5.2)*. Now, we will find B(N) through the inductive step in Section 4. The following recurrence relation can then be deduced.

B(N+1) <= 2 * B(N) + 3 *(e5.3)*
U(N+1) <= 2 * U(N) + 1 *(e5.4)*

Solving the recurrence relation, we find

B(N) <= 2^(N) – 3 *(e5.5)*
U(N) <= 3 * 2^(N-2) – 1 *(e5.6)*

Summing up, the total number of operators are bounded by :

O(N) <= 2^(N) + 3 * 2^(N-2) – 4 *(e5.7)*

Now, we will compute the average number of binary operators and unary operators found in the expression generated by this method. We have the same recurrence relation as in (e5.1) and (e5.2) but with different basis. For completeness, we will restate all of them :

(Definition 5.3) EB(N) : Expected number of binary operators for an expression with N free variables.

(Definition 5.4) EU(N) : Expected number of unary operators for an expression with N free variables.

EB(2) = 10/16 *(e5.8)*
UB(2) = 10/16 *(e5.9)*
EB(N+1) <= 2 * B(N) + 3 *(e5.10)*
EU(N+1) <= 2 * U(N) + 1 *(e5.11)*

Solving them we have

EB(N) <= 29/32 * 2^N – 3 *(e5.12)*
UB(N) <= 13/32 * 2^N – 1 *(e5.13)*

Finally, the expected total number of operators is :

EO(N) <= 21/16 * 2^N – 4 *(e5.14)*

6. Performance Comparison

We will compare their performance through both the worst case and expected number of operators. We have calculated both values for our method in Section 5. The calculation of both for the sum-of-product is simpler and below we their values.

Worst Case

SoP_O(N) = N * 2^N + 0.5 * N * 2^N + (2^N-1) *(e6.1)*

Expected

SoP_EO(N) = N * 2^N * 0.5 + 0.25 * N * 2^N + (0.5 * 2^N – 1) *(e6.2)*

It can be seen that in both case our method exceeds the performance of the regular Sum-of-Product approach. Our algorithm outperforms the regular Sum-of-Product approach by a factor of N.

7. Conclusion

We have presented an algorithm that can be used as an alternative to generating an logical expression. We also realize that our algorithm is not easy to compute without a computer. However, although at the time of the writing of this paper we haven't realized this idea, we realize that it is entirely feasible to author a computer program that can implement this method.

References

[1] Rosen, Kenneth H. "Discrete Mathematics and Its Applications Fifth Edition." 2003. McGraw-Hill

[2] Stroustrup, Bjarne. "The C++ Programming Language Special Edition". 1997. Addison Wesley