Factoring Logic Functions Using Graph Partitioning

Martin C. Golumbic and Aviad Mintz

Department of Mathematics and Computer Science,
Bar Ilan University, Ramat Gan 69978, Israel

Abstract

Algorithmic logic synthesis is usually carried out in two stages, the independent stage where logic minimization is performed on the Boolean equations with no regard to physical properties and the dependent stage where mapping to a physical cell library is done. The independent stage includes logic operations like Decomposition, Extraction, Factoring, Substitution and Elimination. These operations are done with some kind of division (boolean, algebraic), with the goal being to obtain a logically equivalent factored form which minimizes the number of literals.

In this paper, we present an algorithm for factoring that uses graph partitioning rather than division. Central to our approach is to combine this with the use of special classes of boolean functions, such as read-once functions, to devise new combinatorial algorithms for logic minimization. Our method has been implemented in the SIS environment, and an empirical evaluation indicates that we usually get significantly better results than algebraic factoring and are quite competitive with boolean factoring but with lower computation costs.

1 Introduction

Algorithmic logic synthesis is usually done in two stages, the independent stage where logic minimization is performed on the Boolean equations with no regard to physical properties and the dependent stage where mapping to a physical cell library is done. The independent stage includes such logic operations as Decomposition, Extraction, Factoring, Substitution and Elimination. These operations are done with some kind of division (boolean, algebraic) [4].

The objective of factorization is to represent a boolean function in a logically equivalent factored form but with a minimum number of literals. This type of optimization will yield a minimum area taken by realization of this function. Algebraic algorithms for factorization are known [8, 9] and are widely used in commercial environments due to their speed. On the other hand, Boolean factoring [9] is not widely used because of its computational complexity even though it gives much better results.

In this paper, we investigate an algorithmic method for factoring that uses graph partitioning rather than division. Our method is a generalization of techniques for the so called *read-once* functions[1] [3], a special family of monotone Boolean functions, also known as *non-repeatable* tree (NRT) functions [6]. In our study of this method, we obtain better results than algebraic factoring in most test cases and very competitive results with Boolean factoring with less computation time.

2 The cluster intersection graphs of SOP and POS forms

A standard canonical form for representing a Boolean function is as a sum of products (SOP) also known as Disjunctive Normal Form (DNF). The formula $F_1 = aq + acp + ace$, in the example from the footnote, is in SOP form. Dual to this, is the canonical form known as product of sums (POS) also called Conjunctive Normal Form (CNF), which for our example would be $g = F_3 = a * (q + c) * (q + p + e)$. We assume that a given form is simplified, that is, a SOP is the sum of prime implicants of the function, and a POS is the product of prime explicants. A *prime implicant* (resp., *prime explicant*) is a minimal product (resp., sum) of literals whose truth implies the truth of the function and whose removal from the formula would change the function. In general, such an SOP or POS form is not unique, although for read-once functions it is unique.

The parse tree (or computation tree) of an SOP (resp., POS) may be regarded as a two level circuit with the literals labeling the leaves of the tree, the level one nodes being the operation $*$ (resp., $+$) and the root being the operation $+$ (resp., $*$). The level one nodes partition the leaves (literals) into subsets which

[1] A Boolean function f is called a *read-once function* if it has a factored form in which each variable appears exactly once. In logic synthesis, one traditional measure of the complexity of a logic circuit is the number of literals. In this sense, a read-once formula is the best possible since no variable is repeated. For example, the function $g = F_1 = aq + acp + ace$ is a read-once function since it can be factored into the read-once formula $g = F_2 = a(q + c(p + e))$. The reason that read-once functions are also known as *non-repeatable tree* functions is that the parse tree of a read-once formula has no variable repeated. Read-once functions have interesting special properties [3, 5] and according to [6] account for a large percentage of functions which arise in real circuit applications. They have also gained recent interest in the field of computational learning theory.

0-7803-5832-X /99/$10.00

we will call *clusters*. In an SOP the clusters are the prime implicants; in a POS they are the prime explicants. Finally, we recall that it is a straightforward but tedious exercise to transform an SOP form into an equivalent POS form by applying the distributive laws of Boolean algebra and simplifying terms, and this may have exponential complexity in time and space.

Let F be a Boolean formula in SOP form for a function g over a set of variable $V = v_1, v_2, \ldots, v_n$, and let F' be the POS form of the same function g . Let $C = C_1, C_2, \ldots, C_m$ denote the clusters of F , and let $D = D_1, D_2, \ldots, D_m$ denote the clusters of F' .

We define the *cluster intersection (CI) graphs* $G_F = (C, E)$ and $G'_F = (D, E')$ of F which have vertices corresponding to the clusters of F and F' respectively and edge between C_i and C_j (or D_i and D_j) if they contain a common literal. By abuse of notation, we allow $G_{F'} = G'_F$. In our example, the graph G_{F_1} is a triangle, and $G_{F_3} = G'_{F_1}$ is a single edge plus an isolated vertex. Intersection graphs are used extensively in a variety of combinatorial optimization problems [2].

The Peer and Pinter [6] NRT algorithm and our new XFactor algorithm rely on the availability of subroutines S2P and P2S which transform an SOP form to an equivalent POS form and vice versa, and use the graphs G_F and G'_F .

3 General Factorization using XFactor

The generalized algorithm Xfactor which is proposed here for factoring an arbitrary boolean function is based on a modification of the NRT algorithm [6] used as a heuristic together with a graph partitioning algorithm. The algorithm receives an SOP (Sum Of Product) or POS (Product Of Sum) form of a boolean function and builds the Factored Form recursively. At each step it considers both forms (SOP and POS), constructing their weighted cluster intersection graphs, G and G' having vertices corresponding to the clusters with an edge between two clusters C_i and C_j assigned a weight of w if C_i and C_j share exactly w literals (not simply variables). We will call G and G' the *cluster graphs* for short. The algorithm then tries to partition these graphs into two parts A and B according to a heuristic criterion (*separation*), for example, minimizing the sum S of the weights of all edges between the two parts, or minimizing the quotent $S \;/\; |A||B|$. Choosing the optimum heuristic separation, it performs one factoring step of the form, and continues factoring each part recursively until it reaches all the leaves. In the case that the function happens to be read-once, the algorithm reduces to the NRT algorithm, and produces the repetition-free formula, since it is known that exactly one of G and G' must be disconnected, hence the graph partitioning is trivial.

The basic algorithm is presented here in Figure 1. The algorithm works on a general Boolean function represented as a rooted tree T, starting from the root (Primary Output) progressing down to the leaves. Let T be the parse tree of the formula being factored (either SOP or POS). At the first step the algorithm checks for a read-once (NRT) function (which includes trivial cases such as a sum or product of literals) and returns the factored form of the function if it is a read-once (R_Factor). Otherwise, it continues and calculates the separability of the tree which is the heuristic function which works on the weighted cluster graphs. Its purpose is to recommend how to partition the tree. A high separability indicates a high cost of partitioning. Then Xfactor chooses the better of the two forms. Finally, the algorithm actually divides the tree elements and works with each subtree recursively. *Note:* This differs from the NRT algorithm of [6] in two ways: (a) they use the term literal and variable interchangably since they (implicitly) deal only with monotone boolean functions, and (b) since our functions are not necessarily read-once, both graphs G and G' are likely to be connected. Moreover, we use a new polynomial method for the read-once (NRT) subroutine (Golumbic, Mintz and Rotics, in preparation).

The Xfactor Algorithm

```
Xfactor(T) {
  if T is a read-once function
    then return R_Factor(T);
  else
    if T is SOP then T' := S2P(T)
    else T' := P2S(T);
    construct the cluster graphs G and G'
      corresponding to T and T' ;
    sep0 := separability(G);
    sep1 := separability(G' );
    if sep0 < sep1 then use T partitioning
      into subtrees T_i according to the separability
      of G;
    else use T' partitioning into subtrees T_i
      according to the separability of G';
    for each subtree T_i do
      Xfactor( T_i);
}
```

Figure 1: The Xfactor Algorithm

Consider an example of the algorithm for the following Boolean expression:

$$Q = abg + acg + adf + aef + afg + bd + be + cd + ce$$

The expression is represented by the tree in Figure 2. The related cluster graph is also shown. Here it seems that the graph is heavily connected (hard to divide) and using the separability function S above will yield a value relatively big compared to the number of literals.

The equivalent POS form of that Boolean function will be:

$$Q = (a+e+d) * (a+b+c) * (b+c+f) * (d+e+g)$$

which has the representation in Figure 3 along with the related

cluster graph. Here the graph is more easily separated and the minimum cut will be 1 (returned by the separability function).

Now when the POS is chosen we can partition: $Q = T_1 * T_2$, where

$$T_1 = (a+e+d) * (d+e+g) \; and \; T_2 = (a+b+c) * (b+c+f)$$

And now we will deal with each subexpression recursively. The equivalent SOP form of T_1 is

$$T_1 = ag + d + e$$

is read-once, which concludes this branch. Similarly, the equivalent SOP form of T_2 is

$$T_2 = af + b + c$$

which is also read-once, and this finishes the factoring procedure. Figure 4 shows the function in its final factored form $Q = [(a*g)+d+e]*[(a*f)+b+c]$. This is best possible since the function is not read-once.

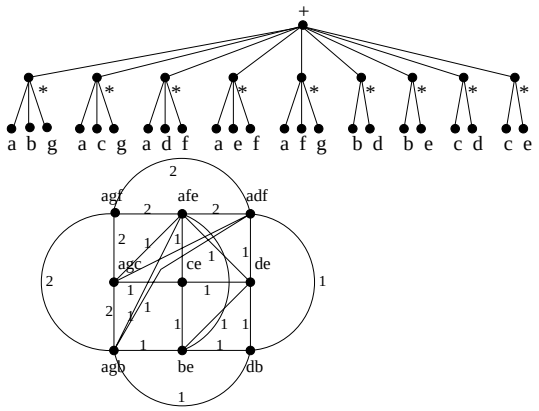

Figure 2: Sum of Product Representation of Q

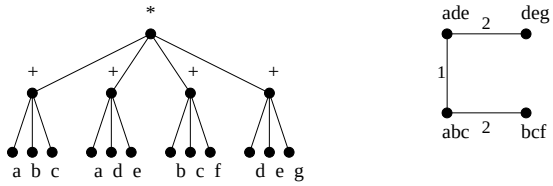

Figure 3: Product of Sum Representation of Q

4 Results

The Xfactor procedure was implemented within the SIS environment [1, 7] and was run with a modest collection of examples. The following five examples from [9] illustrate the results

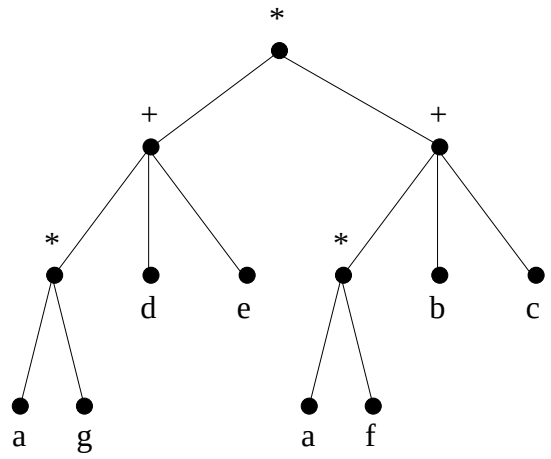

Figure 4: The Factored Form of Q

of having been run with different algorithms available in SIS and our current version of Xfactor. Let

$$O = a'b'c'd'e'f' + abcdef + (a'b' + c'd' + e'f') * (ab + cd + ef)$$

$$P = (a'b' + c'd' + e'f') * (ab + cd + ef)$$

$$Q = (ag + e + d) * (af + c + b)$$

$$R = (a+b) * (c+d) * (e+f) + (a'+b') * (c'+d') * (e'+f')$$

$$S = a'((c'+e') * (ef'gh' + d) + (c'h + d'e) * (f'+e))$$

The results of running Xfactor on these examples are summarized below:

	SOP	QF	GF	BF	XF
O	36	28	28	24	27
P	24	20	20	12	12
Q	23	12	11	8	8
R	48	26	22	16	12
S	23	14	14	14	14

The column SOP represents the number of literals in the sum of product representation. For example, the SOP representation of Q is

$$Q = abg + acg + adf + aef + afg + bd + be + cd + ce$$

which has 23 literals. QF is short for Quick Factor, which is an Algebraic factoring in SIS that works with zero order kernels only. GF represents Good Factor which is an Algebraic factoring

that works with all orders of kernels [9], easily implemented in SIS. BF represents Boolean factoring of [9] and XF represents our Xfactoring with graph partitioning.

From the above table we can see that the results of XF were always better than the algebraic methods QF and GF and quite comparable to BF (it is better in R but worse in O).

The next table includes examples taken from the MCNC benchmark (1991), where *decrease* = $(1-\mathrm{XF}/\mathrm{GF})100$. From this table we can see that XFactor almost always produced results better than the algebraic methods Quick factor and Good factor, especially for the larger examples, with 22.4% improvement on average and 36.0% improvement on weighted average. In the cases where it produced worse results, the variance was less than 10% and since the problems were small, the designer would normally run all the algorithms to choose the best. Boolean factoring could not be run on these larger examples because of its expected high computation cost and unavailability.

	SOP	QF	GF	XF	*decrease*
i2	577	228	228	209	8.4%
mux	220	79	79	47	40.5%
b9_a1	56	12	12	13	-8.3%
b9_d1	94	33	22	24	-9.1%
b9_i1	55	14	14	12	14.3%
b9_z0	48	19	19	20	-5.3%
c8_r0	98	24	24	20	16.7%
c8_s0	112	26	26	22	15.4%
c8_t0	126	28	28	24	14.3%
cmb_r	51	37	37	12	67.6%
9symml	326	178	163	83	49.1%
alu2_k	343	143	140	97	30.7%
alu2_l	1082	345	317	263	17.0%
alu2_o	559	181	168	69	58.9%
alu4_o	345	148	147	97	34.0%
alu4_p	996	360	348	263	24.4%
cm85a_l	104	26	26	16	38.5%
cm85a_m	144	17	17	17	0%
f51m_44	41	23	23	25	-8.7%
f51m_45	82	42	42	42	0%
f51m_46	60	34	34	31	8.8%
f51m_47	39	24	22	23	-4.5%
frg1_d0	780	119	111	43	61.3%
z4ml_25	68	22	20	20	0%
cm162a_o	29	16	16	12	25.0%
cm162a_p	36	18	18	15	16.7%

5 Conclusions

We have presented a new algorithm for factoring Boolean functions which has many advantages over current methods. The use of our weighted cluster intersection graph appears to be a novel approach within logic synthesis. The algorithm Xfactor is a generic version of the type of algorithms we are developing in this project. There are many graph partitioning algorithms which are good candidates for our separability function and alternative implementations with varying computational complexity. We have implemented our method under SIS, and have compared it to known factoring algorithms, algebraic and boolean. Our results are usually significantly better than algebraic factoring and quite comparable to Boolean factoring.

6 Bibliography

References

[1] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, **MIS: A multiple-level logic optimization system**, *IEEE Transaction on Computer Aided Design, vol CAD-6 No-6, 1062-1081* (November 1987).

[2] M.C. Golumbic, **Algorithmic Graph Theory and Perfect Graphs**, Academic Press, 1980.

[3] V. Gurvich, **Criteria for repetition-freeness of functions in the algebra of logic**, *Soviet Math. Dokl. Vol 43, No 3* (1991), 721–726.

[4] G.D. Hachtel and F. Somenzi, **Logic Synthesis and Verification Algorithms**, *Kluwer Academic Publishers* (1996).

[5] M. Karchmer, N. Linial, I. Newman, M. Saks and A. Wigderson, **Combinatorial characterization of read-once formulae**, *Discrete Math. 114* (1993), 275–282.

[6] J. Peer and R. Pinter, **Minimal decomposition of Boolean functions using non-repeating literal trees**, Proc. Int'l. Workshop on Logic and Architecture Synthesis, *IFIP TC10 WD10.5* (Grenoble, December 1995), pp. 129–139.

[7] UCB, **SIS: A System for Sequential Circuit Synthesis,** *UCB - Electronic Research Library M92/41* (1992).

[8] J. Vasudevamurthy and J. Rajski, **A method for concurrent decomposition and factorization of Boolean expressions,** *Proceedings of the International Conference on Computer Aided Design, pp. 510-513* (1990).

[9] Albert Ren Rui Wang, **Algorithms for Multi Level Logic Optimization,** *Ph.D. Thesis, University of California, Berkeley* (1989).