Factoring Boolean functions using graph partitioning

Aviad Mintz, Martin Charles Golumbic

Department of Computer Science, Caesarea Rothschild Institute of Computer Science, University of Haifa, Haifa 31905, Israel

Received 7 March 2004; received in revised form 3 January 2005; accepted 14 February 2005
Available online 17 May 2005

Abstract
Factoring Boolean functions is one of the basic operations in algorithmic logic synthesis. Current algorithms for factoring Boolean functions are based on some kind of division (Boolean or algebraic). In this paper, we present an algorithm for factoring that uses graph partitioning rather than division. Our algorithm is recursive and operates on the function and on its dual, to obtain the better factored form. As a special class, which appears in the lower levels of the factoring process, we handle read-once functions separately, as a special purpose subroutine which is known to be optimal. Since obtaining an optimal (shortest length) factorization for an arbitrary Boolean function is an NP-hard problem, all practical algorithms for factoring are heuristic and provide a correct, logically equivalent formula, but not necessarily a minimal length solution. Our method has been implemented in the SIS environment, and an empirical evaluation indicates that we usually get significantly better factorizations than algebraic factoring and are quite competitive with Boolean factoring but with lower computation costs.

Keywords: Logic synthesis; Factoring Boolean functions; Graph partitioning

1. Introduction

Factoring is the process of deriving a parenthesized algebraic expression or factored form representing a given logic function, usually provided initially in a sum-of-products form.
(SOP) or product-of-sums (POS) form. For example, \( F = ac + bc + cde \) can be factored into the logically equivalent form \( F = c(a + b + de) \). In general, a logic function will have many factored forms. The problem of factoring Boolean functions into shorter, more compact logically equivalent formulae is one of the basic operations in the early stages of algorithmic logic synthesis. In most design styles (like CMOS design) the implementation of a Boolean function corresponds directly to its factored form. Generating an optimum factored form (a shortest length expression) is an NP-hard problem, thus heuristic algorithms have been developed in order to obtain good factored forms.

The earlier algorithms for factoring described in [1–3] were based on algebraic division while those in [22,32,5] were based on Boolean division. In this paper, we present a third approach based on graph partitioning, and which exploits an optimal algorithm for read-once functions. All of these algorithms start with a sum-of-products (SOP) or product-of-sums (POS) form.

**Example 1.** Consider the formula \( H \):

\[
H = adg + adh + bdg + bdh + eag + ebg + ecg + eh.
\]

Running algebraic factoring results in the following:

\[
H = e(h + cg) + (a + b)(eg + d(h + g)),
\]

while running our graph partitioning algorithm results in the formula:

\[
H = (d + e)(g + h)(a + b + e(c + h)),
\]

which is shorter by 2 variable symbols and 2 operations.

The algebraic and Boolean factoring algorithms are recursive and work using top down strategy: the factored form parse tree is built top down and at each iteration of the algorithm, a current leaf node is to be replaced by a sub-tree. The factoring of a function \( F \) is done by finding a divisor \( Q \) (algebraic or Boolean) and dividing \( F \) by it, yielding \( F = Q \cdot P + R \). Next iterations will factor \( P, Q \) and \( R \).

In contrast to this, our algorithm, called Xfactor which does factoring based on graph partitioning, also works in a top down fashion but it does not use any division, rather, it represents \( F \) as a sum of two sub-functions \( F = F_1 + F_2 \) or as a product of two sub-functions \( F = F_3 \cdot F_4 \) and selects one of those. Then it calls itself recursively with each sub-function.

The sub-functions for the sum or product are determined by building two edge weighted (cluster intersection) graphs \( G_F \) and \( G^*_F \) for the function \( F \) and applying graph partitioning to each of them. The better partition determines which sub-functions are used to continue the recursion.

A special case, but one which occurs frequently at the lower levels of the recursion are the read-once functions. A function is read-once if it can be factored into a formula in which each variable occurs only once [17–19,21]. For example, \( ab + bc' + bd \) is read-once, but \( ab + bc' + ad \) is not read-once. We note that read-once functions must be unate (monotone) meaning that a variable may not appear both in its complemented and noncomplemented form. The rich and beautiful theory of read-once functions, originally developed by Gurvich
plays a special role in our Xfactor methodology. One of the results used in [28], that follows immediately from the early work of Gurvich, is the following.

**Theorem 1.** For read-once functions, one of \( G_F \) or \( G^*_F \) must always be disconnected.

In [15] we described an efficient algorithm for recognizing and factoring read-once functions, called IROF, and showed that it produces the optimum factored form in almost linear complexity in the size of the input SOP or POS form.

Building on the read-once approach of examining \( G_F \) and \( G^*_F \) for connectivity, our motivation has been to extend this approach to general Boolean functions by partitioning these graphs into components with few edges connecting them, and thus generating better factored forms.

The paper is organized as follows. In Section 2, we define cluster intersection graphs and provide other background, then in Section 3, we describe the Xfactor algorithm and illustrate it on some examples. In Section 4, we examine Xfactor’s procedures more closely, and indicate the choices made to improve its performance. Next, in Section 5, we compare results of Xfactor with algebraic and Boolean factoring. Section 6 deals with details of the transformation of sum-of-products form to a product-of-sums form, and vice versa, and Section 7 deals with heuristics and analysis needed for speeding up this transformation which dominates Xfactor run-time.

### 2. Cluster intersection graphs and bipartitioning

The parse tree (or computation tree) of an SOP (resp., POS) is often regarded as a three level circuit with the root being the operation + (resp., *), the middle level nodes being the operation * (resp., +), and the literals labeling the leaves of the tree. The level one nodes partition the leaves (literals) into subsets which we will call clusters. In an SOP the clusters are the prime implicants or cubes; in a POS they are called the prime explicants. Finally, we recall that it is a straightforward but tedious exercise to transform an SOP form into an equivalent POS form or vice versa by applying the distributive laws of Boolean algebra and simplifying terms; moreover, this may have exponential complexity in time and space. This problem is known also by the name of dualization and is an NP-hard problem for general Boolean functions. In [11] it is shown that dualization is quasi-polynomial for the monotone (unate) Boolean functions. Because of the necessity in logic synthesis for such algorithms to handle general Boolean functions, there are a number of heuristic algorithms for dualization which are in use.

Let \( F \) be a Boolean formula in SOP form for a function \( F \) over a set of variable \( V = v_1, v_2, \ldots, v_n \), and let \( F^* \) be the POS form of the same function \( F \). Let \( \mathcal{C} = C_1, C_2, \ldots, C_l \) denote the clusters of \( F \), and let \( \mathcal{D} = D_1, D_2, \ldots, D_m \) denote the clusters of \( F^* \). We define the literal cluster intersection (LCI) graphs \( G_F = (\mathcal{C}, E) \) and \( G^*_F = (\mathcal{D}, E^*) \) of \( F \) which have vertices corresponding to the clusters of \( F \) and \( F^* \), respectively, and there is an 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 \( G_{F^*} = G^*_F \). For example, the LCI graph \( G_H \) for the function in Example 1 is shown in Fig. 1 and the LCI graph \( G^*_H \) is shown in Fig. 2.
Intersection graphs are used extensively in a variety of combinatorial optimization problems [13,25]. For read-once functions, one of $G_F$ or $G^*_F$ must always be disconnected.

Peer and Pinter [28] defined cluster intersection graphs differently as they (implicitly) deal only with unate functions. In their definitions, the nodes of the graph are the clusters of $F$, but the edges connect clusters that have common variables. We will refer to the Peer–Pinter graph as the variable cluster intersection (VCI) graph. The function $H$ in Example 1 is unate, so the VCI and LCI graphs would be the same. However, in general, the VCI graph will have additional edges. For example, taking $C_i = ab$ and $C_j = cb'$ as two cubes of an SOP formula will give an edge $e = (i, j)$ in the VCI graph and no edge in the LCI graph since the literals $b$ and $b'$ are different. In general, the VCI graph will be much more dense than the LCI graph, although for read-once functions they are identical. For non-read-once functions, it is always more profitable to use the more sparse LCI graph in Xfactor.

We assign normalized weights to the edges of the literal cluster graphs as follows: For each edge $e = (i, j)$, corresponding to intersecting clusters $(C_i, C_j)$ or $(D_i, D_j)$, the weight $w_e$ is defined as the number of the common literals of $C_i$ and $C_j$ (or $D_i$ and $D_j$) in the LCI
The normalized weight is defined by $z_e = w_e^2 / (|C_i| \ast |C_j|)$ (or $z_e = w_e^2 / (|D_i| \ast |D_j|)$). A high value of the weight $w_e$ indicates many common literals which could be factored out together; the normalized weight $z_e$ tends to balance this against the size of the clusters, and thus against the number of disjoint literals. For example, taking $C_i = ab'c$ and $C_j = abdf$, of sizes 3 and 4 respectively, will give a weight of $w_e = 1$ in the LCI graph (a weight of $w_e = 2$ in the VCI graph). Using the normalized weight on each of the previous graphs will yield $z_e = 1/3 \ast 1/4 = 1/12$ in the LCI graph ($z_e = 2/3 \ast 2/4 = 1/3$ in the VCI graph).

A bipartition of a connected graph $G = (V, E)$ is a division of its vertices into two sets $A$ and $B$. The set of edges joining vertices in $A$ to vertices in $B$ is an edge separator that we shall denote by $\delta(A, B)$; the removal of these edges disconnects the graph into two or more connected components.

One goal in bipartitioning a graph is to minimize the number of edges cut by the partition, that is, the size of $\delta(A, B)$; or if the edges of the graph have weights associated with them, then partitioning may wish to minimize the sum $\sigma$ of the weights (here, modified weights) of $\delta(A, B)$. A second goal could be to balance the number of vertices in $A$ and $B$. Our heuristics for factoring will combine both of these goals. There are other variants of the graph partitioning problem including graphs with geometric coordinates, planar graphs, overlap graphs, etc. Surveys of graph partitioning can be found in [6,7,9,29].

Remark 1. The special case of read-once functions

Although one can use the LCI graph $G_F$ to handle read-once functions, it is much more efficient to use a different intersection graph model due to Gurvich [19,17], which we denote by $\Gamma(F)$. While $G_F$ has vertices that correspond to the clusters (i.e., the minterms or cubes) of the function, the graph $\Gamma(F)$ has vertices that correspond to the variables, and two vertices are joined by an edge if their variables appear together in some cluster. The two graphs are related, for example, $\Gamma(F)$ is disconnected if and only if $G_F$ is disconnected.

The Gurvich graphs are used in our companion paper [15] to recognize and factor read-once functions in a very efficient manner, and this is the subroutine used in XFactor as well. The main reason they work so wonderfully for read-once functions, is based on Gurvich’s result: If $F$ is read-once, then $\Gamma(F^*)$ equals the graph complement of $\Gamma(F)$. Thus, the expensive process for finding the dual form explicitly, is replaced by the cheap process of graph complementation. Unfortunately, we do not know any way to use the Gurvich graphs for the general factoring problem, which is why we had to move to our LCI graphs. There is a bit more to this story to get the almost linear complexity for read-once functions, but for further details, see [15].

3. General factorization using Xfactor

We now present our algorithm Xfactor for factoring an arbitrary Boolean function. (An earlier version was given in [14] and described more fully in [26].) It is based on an extension of the NRT algorithm of Peer and Pinter [27,28] for factoring read-once functions, although more efficient methods exist for read-once functions [17,15] using cograph recognition and normality checking, and these are indeed used at the final level of the recursion. The Xfactor
algorithm relies the availability of (i) subroutines S2P and P2S which transform an SOP form to an equivalent POS form and vice versa, and (ii) a bipartitioning algorithm to be applied to the LCI graphs $G_F$ and $G_F^*$. Xfactor 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 literal cluster intersection (LCI) graphs, $G$ and $G^*$. The algorithm then partitions these graphs into two parts $A$ and $B$ trying to minimize the separation cost $\sigma = \sum z_e \ (e \in \delta(A, B))$ over all partitions. Choosing the better separation, it performs one factoring step, and continues factoring each part recursively. When a branch of the recursion receives a read-once function to factor, which will always happen before reaching the literals, the algorithm completes that branch using the IROF algorithm described in [15].

3.1. Detailed description

The basic algorithm is presented in Fig. 3 and will now be described in detail. We use $T$ to represent both the formula being factored and its parse tree where the root represents the Primary Output of the function and the leaves represent the Inputs to the function. Initially, the form of $T$ is either SOP or POS. At the first step, the algorithm checks for a read-once 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 using the optimal routine IROF of [15]. Otherwise, it generates the weighted literal cluster intersection graph $G$ (ClusterGraphGen) and uses a heuristic graph bipartitioning algorithm (Separability) to calculate a partition and its separability value. The purpose of this is to recommend how to partition the function into two sub-functions for the next step of the recursion. A high separability indicates a high cost of partitioning. A separability of zero means that the graph is disconnected, and the partitioning is optimal. In case that the separability is not zero, Xfactor calculates the alternative form, generates its cluster intersection graph $G^*$ and calculates the separability of the dual function. Then Xfactor chooses the better of the two forms. Finally, the algorithm actually divides the tree elements (Xdecontract) and works with each subtree recursively.

Note: This differs from the NRT algorithm of [28] in two ways: (a) they use the term literal and variable interchangeably 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, in which case their algorithm no longer applies.

Example 2. Consider an example of the algorithm for the following Boolean expression, taken from [22]:

$$B_1 = a'b'e + acd + ce.$$  

The function $B_1$ is not read-once since it is not unate. The parse tree $T$ and its cluster graph $G$ are given in Fig. 4. Since the cluster graph $G$ is connected, S2P is run on $T$ giving

$$B_1 = (a + e)(d + e)(a' + c)(b + c).$$  

The alternative parse tree $T^*$ and its cluster graph $G^*$ are given in Fig. 5.
Xfactor(T) {
    if (T is a read-once function) then
        return IROF(T)
    else
        G = ClusterGraphGen(T)
        partition0, sep0 = Separability(G)
        if (sep0 == 0) then
            T1, T2 = Xdecontract(T, partition0)
        else
            if (T is SOP) then T* = S2P(T)
            else T* = P2S(T)
            G* = ClusterGraphGen(T*)
            partition1, sep1 = Separability(G*)
            if (sep0 < sep1) then
                T1, T2 = Xdecontract(T, partition0)
            else
                T1, T2 = Xdecontract(T*, partition1)
            foreach (subtree Ti) do
                Xfactor(Ti)
            }
}

Here, it is clear that G* is disconnected and its disconnected parts represent the two subtrees T1 and T2 which are assigned to T on the next iteration.

Both subtrees, T1 representing \((a' + c)(b + c)\) and T2 representing \((a + e)(d + e)\) are read-once and are given by:

\[
T_1 : a'b + c, \quad T_2 : ad + e.
\]

Thus, the final result which includes six literals is

\[
B_1 = (ad + e)(a'b + c).
\]
Factoring $B_1$ with algebraic methods [3,1] results in seven literals by quick factor

\[ B_1 = a'b + c(ad + e) \]

and also seven literals by the good factor algorithm

\[ B_1 = acd + e(a'b + c). \]

**Example 3.** A larger example taken from MCNC benchmark [33] is the following:

\[
B_2 = aemstu + aeqstu + airstu + aqrstu + bfnst'u \\
+ bfgst'u + bjrst'u + bqrst'u \\
+ cgkos't'u + cgqs't'u + ckrs't'u + cqr's't'u \\
+ dhlpst'u + dhsqs't'u + dhrs't'u + dqrst'u + emr's'tu + eqr's'tu \\
+ fnr's't'u + fqr's't'u + gors't'u + gqrs't'u + hprs't'u + hqr's't'u \\
+ imq's'tu + iq'r's'tu + jns't'usr + jq'r's't'u + koq's'tu \\
+ kqr's't'u + lpq's't'u + lqr's't'u + mq'r's'tu \\
+ nqr's't'u + oq'r's't'u + pq'r's't'u, \\
\]
which has 220 literals. Running Xfactor algorithm results
\[ B_2 = u((pq' + hq + r)(lq' + dq + r')s' + (nq' + f q + r)(jq' + bq + r')s + t) \]
\[ ((oq' + gq + r)(kq' + cq + r')s' + (mq' + eq + r)(iq' + aq + r')s + t'), \]
which has 47 literals. Factoring it by good factor algorithm yields
\[ B_2 = u(s'(t'(lq'(r + p) + r'(h(q + p) + pq')) + d(r(q + l) + h(lp + q))) \]
\[ + t(kq'(r + o) + r'(g(q + o) + oq')) + c(r(q + k) + g(ko + q)))) \]
\[ + s'(t'(jq'(r + n) + r'(f(q + n) + nq')) + b(r(q + j) + f(jn + q))) \]
\[ + t(iq'(r + m) + r'(e(q + m) + mq')) + a(r(q + i) + e(im + q)))), \]
which has 79 literals.

4. Internal procedures

The Xfactor algorithm shown in Fig. 3 is constructed from several procedures which we present in this section. First, Xfactor uses the IROF which tests and parses a read-once function and is discussed in [15]. Until the recursion has reached the level of a read-once function, Xfactor uses ClusterGraphGen which generates the cluster graph and the Separability procedure which recommends how to partition the parse tree according to the bipartition of the cluster graphs. The other procedures used are P2S/S2P which converts forms, and Xdecontract that actually reorganizes the tree according to results given by the Separability procedure on both SOP and POS forms.

The ClusterGraphGen procedure takes an SOP formula or a POS formula and builds the cluster graph by examining each pair of cubes/sums in the SOP/POS formula. First it assigns a vertex to each cube/sum and then it connects a pair of vertices if there are common literals between the two cubes/sums. It also assigns a weight to each edge in the graph. The search for common literals between any two terms of the SOP/POS formula causes a complexity of \(O(k^2 \times c)\) where \(k\) represents the number of cubes/sums and \(c\) represents the cube/sum size.

The Xdecontract procedure reorganizes the generated parse tree of the function. It is activated as the last step of the Xfactor algorithm. Let \(M\) be the root node of the current parse tree which represents the function \(F\) and has children \(S_1, S_2, \ldots, S_k\). If Xfactor selects a bipartitioning where \(S_{a_1}, \ldots, S_{a_t}\) are vertices of one part and \(S_{b_1}, \ldots, S_{b_t}\) are the vertices of the other part, Xdecontract generates two nodes \(M_1\) and \(M_2\) which hold the same operation as \(M\) and links them to be children of \(M\), then moves the \(S_{a_i}\) nodes to be children of \(M_1\) and the \(S_{b_j}\) nodes to be children of \(M_2\). Thus, the complexity of the procedure is \(O(k)\). Fig. 6 shows the result of Xdecontract on the parse tree given in Fig. 5.

4.1. P2S and S2P

The two procedures P2S and S2P were implemented as one procedure so called P2SS2P. It receives a function given by a parse tree and yields a parse tree that represents its dual (SOP/POS). Hachtel and Somenzi [20] describes a method to convert SOP to POS and vice versa using the distributive law and Shannon’s Expansion:
Theorem 2.

\[
F(x_1, x_2, \ldots, x_n) = x'_1 F(0, x_2, \ldots, x_n) + x_1 F(1, x_2, \ldots, x_n) \\
= (x'_1 + F(1, x_2, \ldots, x_n))(x_1 + F(0, x_2, \ldots, x_n)).
\]

They did not state an exact algorithm but they mentioned that the conversion calculated by their recommendation is not guaranteed to be minimized and even if the original formula was minimized the resulting formula is not minimized. It was also noticed that the conversion is not linear in the number of literals in the original formula and it is very sensitive to the order of literals that are expanded.

The same results happen when using just the distributive law. The results have to be minimized otherwise taking a minimized formula and converting it to the alternative form will yield a non-optimized formula. The order of using the distributive law directly influences the size of intermediate terms (same as the order of using Shannon’s Expansion). We will discuss our version of P2SS2P using logic minimization techniques in Section 6.

4.2. Separability and graph partitioning

The separability procedure evaluates the bipartition connectivity of the literal cluster graph using partitioning algorithms and a heuristic function of the weights of the edges which connect the two parts (so called the separability value).

Among the partitioning algorithms we used and compared, is the level-structure partitioning method [29]. That algorithm chooses a starting vertex \( v \), and a breadth first search (BFS) from \( v \) is used to partition the vertices into levels.

The Greedy Graph Growing Partitioning algorithm proposed by Karypis and Kumar [23] is another algorithm we used as a basis of comparison. The algorithm starts with a starting vertex \( v \) and grows a region (growing region) around it in a greedy fashion as in
level-structure partitioning but the order of the vertices is not BSF but is driven by a gain function.

A different partitioning algorithm is IBM90, proposed by Golumbic et al. [16]. That algorithm works as follows: At each step the algorithm chooses two vertices (u and v) which are connected by the heaviest edge (e = (u, v)) and merges them to one new vertex (wuv). During the merge, the edge e connecting the two vertices is eliminated and each pair of edges ((u, x), (v, x)) connecting these two vertices to another vertex x are merged by summing their weights. All other edges remains the same. (Non-edges are assumed to have weight zero.) This process stops when there exist only two vertices and one edge which represents the bipartition of the whole graph.

At the last stage of the partitioning, one invokes the Fiduccia–Mattheyses (FM) version [10] of the well-known Keringhan–Lin (KL) algorithm which is described in [24]. It is used as a refinement algorithm after running the partitioning algorithms.

The Separability procedure returns the separability value and a list of vertices which are included in one part (while all the other vertices belong to the other part).

5. Comparison with other factoring methods

In this section, we will compare Xfactor’s results with the Algebraic and Boolean methods. In Section 5.1, we consider the quality of the factorization and in Section 5.2, we will check the run-time of the Xfactor algorithm and compare it to run-time of the Algebraic and Boolean methods.

5.1. Factorization performance analysis

Tables 1–3 include Xfactor results compared with Algebraic factoring and Boolean factoring. Table 1 includes examples from [1,14]; Tables 2 and 3 include examples from the MCNC benchmark [33] and as they appeared in [14] and [32]. All figures in the tables are the number of literals. The rows include different tests and the headers indicate

<table>
<thead>
<tr>
<th>Test</th>
<th>SOP</th>
<th>QF</th>
<th>GF</th>
<th>BF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ofactor</td>
<td>36</td>
<td>28</td>
<td>28</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>Pfactor</td>
<td>24</td>
<td>20</td>
<td>20</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Qfactor</td>
<td>23</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Rfactor</td>
<td>48</td>
<td>12</td>
<td>12</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Sfactor</td>
<td>23</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Tfactor</td>
<td>23</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>177</td>
<td>101</td>
<td>97</td>
<td>83</td>
<td>83</td>
</tr>
</tbody>
</table>
Table 2
Xfactor versus algebraic and Boolean methods—Test set #2

<table>
<thead>
<tr>
<th>Test</th>
<th>SOP</th>
<th>QF</th>
<th>GF</th>
<th>BF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>293</td>
<td>167</td>
<td>164</td>
<td>159</td>
<td>161</td>
</tr>
<tr>
<td>f51m</td>
<td>250</td>
<td>146</td>
<td>144</td>
<td>—</td>
<td>143</td>
</tr>
<tr>
<td>misex1</td>
<td>122</td>
<td>89</td>
<td>86</td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>rd53</td>
<td>140</td>
<td>70</td>
<td>70</td>
<td>68</td>
<td>82</td>
</tr>
<tr>
<td>sao2</td>
<td>489</td>
<td>196</td>
<td>185</td>
<td>174</td>
<td>171</td>
</tr>
<tr>
<td>z4ml</td>
<td>124</td>
<td>49</td>
<td>46</td>
<td>—</td>
<td>48</td>
</tr>
<tr>
<td>Total</td>
<td>1418</td>
<td>717</td>
<td>695</td>
<td>—</td>
<td>693</td>
</tr>
</tbody>
</table>

Table 3
Xfactor versus algebraic methods—Test set #3

<table>
<thead>
<tr>
<th>Test</th>
<th>SOP</th>
<th>QF</th>
<th>GF</th>
<th>XF</th>
<th>Test</th>
<th>SOP</th>
<th>QF</th>
<th>GF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>9symml</td>
<td>326</td>
<td>182</td>
<td>161</td>
<td>114</td>
<td>cm163a_q</td>
<td>23</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>alu2_k</td>
<td>343</td>
<td>143</td>
<td>140</td>
<td>96</td>
<td>cm163a_r</td>
<td>31</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>alu2_l</td>
<td>1082</td>
<td>345</td>
<td>317</td>
<td>192</td>
<td>cm163a_s</td>
<td>39</td>
<td>15</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>alu2_o</td>
<td>559</td>
<td>181</td>
<td>168</td>
<td>86</td>
<td>cm163a_t</td>
<td>47</td>
<td>17</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>alu4_o</td>
<td>345</td>
<td>148</td>
<td>147</td>
<td>96</td>
<td>cm85a_l</td>
<td>104</td>
<td>26</td>
<td>26</td>
<td>16</td>
</tr>
<tr>
<td>alu4_p</td>
<td>996</td>
<td>360</td>
<td>348</td>
<td>192</td>
<td>cm85a_m</td>
<td>144</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>alu4_v</td>
<td>128</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>cm85a_n</td>
<td>104</td>
<td>26</td>
<td>26</td>
<td>16</td>
</tr>
<tr>
<td>b9_a1</td>
<td>56</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>frgl_d0</td>
<td>780</td>
<td>119</td>
<td>111</td>
<td>42</td>
</tr>
<tr>
<td>b9_d1</td>
<td>94</td>
<td>33</td>
<td>22</td>
<td>17</td>
<td>majority</td>
<td>13</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>b9_i1</td>
<td>55</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>mux</td>
<td>220</td>
<td>79</td>
<td>79</td>
<td>47</td>
</tr>
<tr>
<td>c8_r0</td>
<td>98</td>
<td>24</td>
<td>24</td>
<td>20</td>
<td>pcle_y</td>
<td>30</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>c8_s0</td>
<td>112</td>
<td>26</td>
<td>26</td>
<td>22</td>
<td>pcle_z</td>
<td>36</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>c8_t0</td>
<td>126</td>
<td>28</td>
<td>28</td>
<td>25</td>
<td>pcle_a0</td>
<td>42</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>cm150a</td>
<td>81</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>pcle_b0</td>
<td>48</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>cm162a_o</td>
<td>29</td>
<td>16</td>
<td>16</td>
<td>13</td>
<td>pcle8_r0</td>
<td>60</td>
<td>29</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>cm162a_p</td>
<td>36</td>
<td>18</td>
<td>18</td>
<td>14</td>
<td>sct_d0</td>
<td>44</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>cm162a_q</td>
<td>43</td>
<td>20</td>
<td>20</td>
<td>16</td>
<td>term1_r0</td>
<td>368</td>
<td>70</td>
<td>69</td>
<td>68</td>
</tr>
<tr>
<td>cm162a_r</td>
<td>50</td>
<td>22</td>
<td>22</td>
<td>18</td>
<td>term1_s0</td>
<td>374</td>
<td>72</td>
<td>71</td>
<td>70</td>
</tr>
<tr>
<td>Total</td>
<td>7691</td>
<td>2475</td>
<td>2370</td>
<td>1686</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the different methods. As before, SOP means Sum of Product, QF/GF/BF are short for Quick/Good/Boolean factoring and XF is for Xfactor factoring.

Tables 1–3 indicate that our Xfactor usually gives better results than the Algebraic methods, except for rd53 test where Xfactor results failed. Similarly, Xfactor also gives comparative results to Boolean methods (except for rd53 test). We sought to find an explanation for this.
Table 4  
RD tests—close look

<table>
<thead>
<tr>
<th>Test</th>
<th>SOP</th>
<th>QF</th>
<th>GF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd53_0</td>
<td>20</td>
<td>14</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>rd53_1</td>
<td>80</td>
<td>28</td>
<td>28</td>
<td>46</td>
</tr>
<tr>
<td>rd53_2</td>
<td>40</td>
<td>28</td>
<td>28</td>
<td>24</td>
</tr>
</tbody>
</table>

A closer look at the rd53 test where Xfactor failed is given in Table 4 (where each line represents a different formula). Xfactor fails to give better results on 1 of the 3 formulas (rd53_1); on all other formulas, Xfactor gives better results (rd53_0 and rd53_2). These poor results on rd53_1 occur due to the special structure of this formula, which is based only on XOR operations. It causes a symmetric cluster graph and none of its minimum cuts leads to a minimum factored form representation.

Summing all results from Tables 1–3 gives 3293 literals for QF, 3162 for GF and 2462 for XF.

5.2. Run-time

In Section 4, we discussed the different procedures that Xfactor uses. There we mentioned, that only the P2SS2P conversion procedure is not polynomial (on \( N \), the number of literals of given parse tree) while all the other procedures are polynomial. This causes Xfactor to spend most of its CPU time on that procedure. Algebraic factoring algorithms are quite fast because they rely on polynomial procedures only. Boolean factoring usually are not polynomial but there is not much information about the different implementations of these kinds of algorithms (Table 4).

We compared our Xfactor (XF) run-time with those of algebraic factoring (QF and GF) and Boolean factoring (BF). In Table 5, we give the results for the larger benchmark. In Table 6, we estimated the run-time of Boolean factoring from [32] (Boolean factoring is not public) calculated from the statistics reported in [32] by taking simple ratios between their results compared with algebraic factoring. The figures in these tables are in CPU seconds while running on a Pentium1—200 Mhz.

Tables 5 and 6 indicate that XF consumes 2–12 times more CPU time than the algebraic methods (all comparisons which appear on other papers are for GF). It also indicates that for the bigger tests like alu2_1, alu2_o, etc. the CPU time ratio is lower (2–4) while for small tests like 5xp1, f51m, cm150a, etc. the CPU time ratio is higher (6–10). There are exceptional cases where this ratio is more than 20, like i2, term1_r0, and term1_s0. From our results we learned that for these tests, most of the CPU time is spent on form conversions using P2SS2P (70–80%) and a large portion of these conversions are not used (50–70%).

Table 6 also compares XF run-time with BF run-time. The results do not give a clear preference. There are cases where XF runs faster (5xp1, misex1, sao2) and other cases that BF runs faster (f51m, rd53, z4ml).
Table 5
Run-time: Xfactor versus algebraic methods

<table>
<thead>
<tr>
<th>Test SOP</th>
<th>QF</th>
<th>GF</th>
<th>XF</th>
<th>Test SOP</th>
<th>QF</th>
<th>GF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>9symml</td>
<td>326</td>
<td>0.06</td>
<td>0.33</td>
<td>1.63</td>
<td>cm163a_q</td>
<td>23</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu2_k</td>
<td>343</td>
<td>0.07</td>
<td>0.39</td>
<td>2.06</td>
<td>cm163a_r</td>
<td>31</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu2_l</td>
<td>1082</td>
<td>0.31</td>
<td>13.27</td>
<td>13.55</td>
<td>cm163a_s</td>
<td>39</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu2_o</td>
<td>559</td>
<td>0.10</td>
<td>1.60</td>
<td>2.72</td>
<td>cm163a_t</td>
<td>47</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu4_o</td>
<td>345</td>
<td>0.05</td>
<td>0.54</td>
<td>2.08</td>
<td>cm85a_l</td>
<td>104</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu4_p</td>
<td>996</td>
<td>0.30</td>
<td>10.71</td>
<td>13.25</td>
<td>cm85a_m</td>
<td>144</td>
<td>0.01</td>
</tr>
<tr>
<td>a lu4_v</td>
<td>128</td>
<td>0.01</td>
<td>0.03</td>
<td>0.10</td>
<td>cm85a_n</td>
<td>104</td>
<td>0.01</td>
</tr>
<tr>
<td>b9_a1</td>
<td>96</td>
<td>0.01</td>
<td>0.01</td>
<td>0.06</td>
<td>frg1_d0</td>
<td>780</td>
<td>0.14</td>
</tr>
<tr>
<td>b9_d1</td>
<td>94</td>
<td>0.02</td>
<td>0.03</td>
<td>0.06</td>
<td>i2</td>
<td>577</td>
<td>0.11</td>
</tr>
<tr>
<td>b9_i1</td>
<td>55</td>
<td>0.01</td>
<td>0.02</td>
<td>0.06</td>
<td>majority</td>
<td>13</td>
<td>0.01</td>
</tr>
<tr>
<td>b9_z0</td>
<td>48</td>
<td>0.01</td>
<td>0.01</td>
<td>0.14</td>
<td>mux</td>
<td>220</td>
<td>0.07</td>
</tr>
<tr>
<td>c8_r0</td>
<td>98</td>
<td>0.01</td>
<td>0.01</td>
<td>0.10</td>
<td>pcle_y</td>
<td>30</td>
<td>0.01</td>
</tr>
<tr>
<td>c8_s0</td>
<td>112</td>
<td>0.01</td>
<td>0.02</td>
<td>0.11</td>
<td>pcle_z</td>
<td>36</td>
<td>0.01</td>
</tr>
<tr>
<td>c8_z0</td>
<td>126</td>
<td>0.01</td>
<td>0.02</td>
<td>0.13</td>
<td>pcle_a0</td>
<td>42</td>
<td>0.01</td>
</tr>
<tr>
<td>cm150a</td>
<td>81</td>
<td>0.02</td>
<td>0.04</td>
<td>0.62</td>
<td>pcle_b0</td>
<td>48</td>
<td>0.01</td>
</tr>
<tr>
<td>cm162a_o</td>
<td>29</td>
<td>0.01</td>
<td>0.01</td>
<td>0.06</td>
<td>pcler8_r0</td>
<td>60</td>
<td>0.01</td>
</tr>
<tr>
<td>cm162a_p</td>
<td>36</td>
<td>0.01</td>
<td>0.01</td>
<td>0.07</td>
<td>sct_d0</td>
<td>44</td>
<td>0.01</td>
</tr>
<tr>
<td>cm162a_q</td>
<td>43</td>
<td>0.01</td>
<td>0.01</td>
<td>0.07</td>
<td>term1_r0</td>
<td>368</td>
<td>0.03</td>
</tr>
<tr>
<td>cm162a_r</td>
<td>50</td>
<td>0.01</td>
<td>0.01</td>
<td>0.09</td>
<td>term1_s0</td>
<td>374</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 6
Run-time: Xfactor versus algebraic and Boolean methods

<table>
<thead>
<tr>
<th>Test SOP</th>
<th>QF</th>
<th>GF</th>
<th>BF</th>
<th>XF</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>0.06</td>
<td>0.11</td>
<td>1.06</td>
<td>0.94</td>
</tr>
<tr>
<td>f51m</td>
<td>0.04</td>
<td>0.10</td>
<td>0.62</td>
<td>0.98</td>
</tr>
<tr>
<td>misex1</td>
<td>0.05</td>
<td>0.05</td>
<td>1.05</td>
<td>0.42</td>
</tr>
<tr>
<td>rd53</td>
<td>0.02</td>
<td>0.05</td>
<td>0.13</td>
<td>0.60</td>
</tr>
<tr>
<td>sao2</td>
<td>0.08</td>
<td>0.30</td>
<td>2.17</td>
<td>2.18</td>
</tr>
<tr>
<td>z4ml</td>
<td>0.01</td>
<td>0.04</td>
<td>0.03</td>
<td>0.27</td>
</tr>
</tbody>
</table>

6. P2SS2P conversion

In this section, we will first describe our P2SS2P algorithm and then explain the minimization procedure.

6.1. P2SS2P description

Our P2SS2P algorithm begins with a straightforward approach using only the distributive law and SIS [31] procedures. Here, $T^n$ (where $n$ is an integer) denotes a parse tree, $T^0$ represents a node which is the root of the parse tree and $T^n_i$ represents all other nodes in the parse tree. The algorithm is given in Fig. 7.
P2SS2P(T₀^0) \{
  \textbf{if} (T₀^0 \text{ function is Sum}) \textbf{then} \\
  T₀^1 = \text{Negate}(T₀^0) \\
  \textbf{else} T₀^1 = T₀^0 \\
  M = \text{Collapse}(T₀^1) \\
  T₀^2 = \text{MakeParseTree}(M) \\
  \textbf{if} (T₀^0 \text{ function is Sum}) \textbf{then} \\
  T₀^3 = \text{Negate}(T₀^2) \\
  \textbf{else} T₀^3 = T₀^2 \\
  \textbf{foreach} T_i^3 \text{ child of } T₀^3 \\
  T_i^3 = \text{DeMorgan}(T_i^3) \\
\}

Fig. 7. The P2SS2P algorithm.

The idea of this algorithm is the following: given a root node (T₀^0) of a parse tree (which represents an SOP or POS formula) P2SS2P uses Collapse to convert a POS tree to a single node which is given by an SOP form and has a type of a complex node (a complex node is a node that holds a function, not just an AND/OR operator). The Collapse procedure converts a POS parse tree to one node which is given in SOP form using the distributive law. To assure a POS parse tree as an input to the Collapse procedure, a preliminary step is needed for the SOP parse tree input, in this case P2SS2P negates the function using DeMorgan’s laws (Negate procedure)

\[(P₁ + P₂)' = P₁' P₂'.\]

In order to produce a tree from the Collapse result, the P2SS2P runs MakeParseTree which constructs an SOP parse tree from the complex node. Then it negates it again if necessary (in case of SOP to POS transform) and uses DeMorgan’s laws to eliminate unnecessary not operators.

As an example, let

\[F = ab + cd.\]

The parse tree T₀^0 of F is given by the following preorder description

\[(+(-ab)(-cd))\]
and where \( T_0 \) is the root node which holds the OR operator. The P2SS2P first negates \( F \) (SOP form) and yields a tree \( T^1 \) with root node \( T_0^1 \) and where \( T^1 \) represents \( F' \)
\[
(\cdot(ab)'(\cdot cd)').
\]

Then P2SS2P uses Collapse to transform a POS form to SOP form yielding a complex node named \( M \):
\[
M = (a' + b') \cdot (c' + d') = a'c' + a'd' + b'c' + b'd'.
\]

MakeParseTree generates a tree from the complex node \( M \), thus the tree \( T^2 \) is given by
\[
(+(-a'c')(a'd')(\cdot b'c')(\cdot b'd')).
\]

Then P2SS2P uses Negate to get back \( F \), and yields \( T^3 \)
\[
(\cdot(a'c')'(\cdot a'd')(\cdot b'c')(\cdot b'd')).
\]

At the last step, P2SS2P uses DeMorgan’s Laws to eliminate all unnecessary not operators yielding the final result:
\[
(+ac)(+ad)(+bc)(+bd)).
\]

In the case that the input to P2SS2P is given in POS form, no negate is necessary and no use of DeMorgan’s Laws is needed (last step).

6.2. P2SS2P minimization step

Within our basic algorithm, we also used the Espresso minimizer with do not care set (see [30]) which runs on the complex node \( M \) and minimizes it. The Espresso minimizer minimizes PLA (two level logic) and is discussed in [30,8,20]. On every iteration of Xfactor, two new nodes are built by the Xdecontract procedure (appearing in Fig. 3 as \( T_1 \) and \( T_2 \)). These nodes represent two functions \( (F_1 \) and \( F_2) \) given both in SOP form or POS form. In cases where the support of \( F_1 \) is not disjoint with the support of \( F_2 \), further minimization can be made. Each function \( F_i \) can be minimized while the other function is part of its do not care set.

As an example, we will run the Xfactor algorithm on \( P \):
\[
P = abc'd' + abe' f' + a'b'cd + a'b'ef + cde' f' + c'd'ef.
\]

At first, \( P \) is represented by a parse tree whose its root is \( T \), and is not a ROF (read-once function) so \( G \), the cluster graph is generated, and the alternative POS form of \( P \) is computed
\[
P^* = (b + d + e)(b + c + f)(a + d + f)(a + c + e)
\]
\[
(b' + d' + e')(b' + c' + f')(a' + d' + f')(a' + c' + e').
\]

and assigned to \( T^* \). An appropriate cluster graph \( G^* \) is also built. At this point we can notice that \( G^* \) graph is disconnected (sep1 = 0), which causes Xdecontract to generate two nodes \( T_1 \) and \( T_2 \) that represents the following \( P_1 \) and \( P_2 \) functions
\[
P_1 = (b + d + e)(b + c + f)(a + d + f)(a + c + e),
\]

and where \( T^0 \) is the root node which holds the OR operator. The P2SS2P first negates \( F \) (SOP form) and yields a tree \( T^1 \) with root node \( T^1_0 \) and where \( T^1 \) represents \( F' \)
and
\[ P_2 = (b' + d' + e')(b' + c' + f')(a' + d' + f')(a' + c' + e'). \]

Next, Xfactor is recursively called with \( T_1 \) (now assigned as \( T \)). \( T_1 \) is not a ROF, and again a connected cluster graph is generated. P2SS2P generates the alternative form of \( T_1 \):

\[ P\star_1 = ab + cd + ef + ace + adf + bde + bcf. \]

Here we can see that the support of \( P\star_1 \) is equal to the support of the previous \( P_2 \) and \( P = P\star_1 P_2 \). Thus, we can use \( P_2 = 0 \) (or \( P'_2 \)) as a do not care set of \( P\star_1 \) during the minimization process.

Here \( P_1\star_{DC} = (P_2)' = ace + adf + bde + bcf \)

and
\[ P_1\star_{ON} = P\star_1 P_1\star_{DC}' = ab + cd + ef. \]

Minimizing \( P\star_1 \) gives
\[ P\star_1 = ab + cd + ef, \]
which is a read-once function. Accordingly, \( P\star_2 \) is minimized with \((P\star_1)'\) as its do not care set.

II: \[ P\star_2 = a'b' + c'd' + e'f' + a'c'e' + a'd'f' + b'd'e' + b'c'f' \]

and
\[ (P\star_1)' = (ab)'(cd)'(ef)' = a'c'e' + a'c'f' + a'd'e' + a'd'f' + b'd'e' + b'd'f' + b'c'e' + b'c'f'. \]

Taking \( P_2\star_{DC} = (P\star_1)' \) and \( P_2\star_{ON} = P\star_2(P_2\star_{DC})' \) gives
\[ P\star_2 = a'b' + c'd' + e'f', \]
which is again a read-once function. The complete Xfactor algorithm on \( P \) gives the optimal solution of
\[ P = (ab + cd + ef)(a'b' + c'd' + e'f'). \]

Factoring \( P \) using only the Espresso minimizer (with no use of the DC set) will give worse results. The minimizer cannot minimize (I) and (II) without the DC set and the result of factoring \( P \) would be given by
\[ P = ((a + c)(b + d) + e)((a + d)(b + c) + f) \]
\[ ((a' + c')(b' + d') + e')(a' + d')(b' + c') + f'). \]

In the case where \( P_1 \) and \( P_2 \) are two products \( (P = P_1 + P_2) \), \( P_2 \) will be the DC set of \( P_1 \) and vice versa.
7. Reducing the run-time

We already mentioned that the P2SS2P procedure consumes most of the CPU time because it has non-polynomial complexity. This procedure is run unless the current node represents a read-once function or it represents a tree whose separability is zero. In order to reduce the overall run-time of Xfactor, we developed a method to identify and eliminate certain runs of redundant P2SS2P.

7.1. Redundant P2SS2P runs

In Fig. 8, we describe the percentage of CPU time of the P2SS2P (from the total CPU time) as a function of the number of literals in the input formula. We took only the runs that yield the best results and filtered out all the formulas with less than a 100 literals. The x-axis varies from 100 literals to 1100 literals. The y-axis varies from 0 to 100 percent.

Fig. 8 shows a big variation between different P2SS2P runs. There are cases where P2SS2P consumes almost no CPU time and there are cases where P2SS2P consumes more than 80% of the total CPU time. Still, bigger formulas spend more CPU time in P2SS2P. For formulas at the range of 100 to 200 literals, P2SS2P consumes around 30% of the CPU time, but for formulas of more than 700 literals, P2SS2P usually consumes more than 50% (60% on the average) of the total CPU time. For cases of thousands of literals, it is reasonable to expect that this percentage will go higher.

Thus, it is clear that in order to reduce the CPU time of the algorithm we need to reduce the number of P2SS2P calls, especially at the early stages of the recursion. Moreover, by analyzing the Xfactor algorithm, we can identify cases where the P2SS2P calls are redundant, and Xfactor continues with its initial form (sep0 < sep1) and not with the P2SS2P generated form.

In Fig. 9 we describe the binary tree that is built during the Xfactor execution and which represents the factored form. Each node has an operation (AND or OR) except for the leaves that represent literals. This tree is different from the final factored form tree, it has nodes...
whose parent has the same operation as themselves. In all these cases, the P2SS2P run is redundant. Fig. 9 shows the binary tree and the factored form tree of $P$, where $P$ is given by

$$P = de' + cd'e + ce'fg'h' + c'd + d'ef' + c'f'h.$$ 

The (I), (II) and (III) symbols mark the redundant runs of P2SS2P ((I) represents the initial redundant conversion from SOP to POS which was not needed).

Fig. 10 shows the percentage of the redundant P2SS2P runs as a function of the number of literals in the input formula. Here again the $x$-axis varies from 100 to 1100 literals. The $y$-axis is given in percents of total P2SS2P runs.

Fig. 11 shows the percentage of the redundant P2SS2P CPU time over the total P2SS2P CPU time. The values on the $x$-axis and $y$-axis are the same as in Fig. 10 but the meaning
is different. There might be cases where there are few redundant P2SS2P runs, but those consume most of the CPU time of all P2SS2P runs and vice versa.

Figs. 10 and 11 show different cases of redundant P2SS2P usage. There are cases where there is no redundancy (0 in both graphs) and there are cases where 50% of the P2SS2P runs are redundant. Alternatively, we have cases where 60% and more of the P2SS2P CPU time is redundant. Still, we can identify a common behavior of both graphs: bigger formulas have more redundant runs (and usually more redundant CPU time) than the smaller ones. Most of the cases of no redundancy appear in 100 to 150 literals.

In order to reduce the P2SS2P redundant runs, Xfactor should identify cases where P2SS2P is going to be redundant (before P2SS2P is run or even during a P2SS2P run). There are two cases of redundant P2SS2P runs. One, is when a small form is converted to a very large form, which has a high separability value. For example,

\[ F = (af + b + c)(ag + d + e) \]

is small compared to its equivalent form

\[ F = afg + adf + aef + abg + acg + bd + be + cd + ce. \]

The other, is where the two forms (SOP and POS) are quite the same size (literal count), and their separabilities are quite similar. In these cases, it is very difficult to identify a redundant run and more research is needed. Unfortunately, the second case is common, and most of the redundant runs belong to this group.

The identification of the first case is done during the P2SS2P run. In Section 4.1, we noted that the Collapse procedure converts a POS form to an SOP form using the distributive law (and is the heart of the algorithm, except for the minimization step). This procedure is implemented as a loop over the sums of the POS form. In each iteration the intermediate SOP is multiplied by one sum.
To avoid wasting CPU time on generating large redundant forms, first we added to this procedure a limit for the number of literals in the generated SOP form. This limit is a function of the literal count of the input form. Exceeding this number causes the P2SS2P to stop and thus not convert the form.

### 7.2. Analysis of redundant P2SS2P

We investigated whether we can filter redundancy P2SS2P runs without loosing performance by limiting the number of cubes built in the alternative form during the P2SS2P process. The chosen threshold is a linear factor on the number of cubes (sums) of the current form, A P2SS2P run will be stopped if during its process it exceeds this threshold.

For each P2SS2P execution (redundant or essential) done during the factoring process of each test, the ratio

\[
\frac{\text{Number of cubes in the alternative form (before simplify)}}{\text{Number of cubes in the current form}}
\]

has been measured.

We ordered all these ratios of essential P2SS2P executions on one list (called essential list) and of the redundant P2SS2P executions on another list (called redundant list).

We discovered that most of the time (501/720–69.6%) the redundant list was totally covered by the essential list. Thus, there was no way to find a redundant P2SS2P filter without loosing performance. There were cases where we found a partial cover (124/720–17.2%), meaning that only part of the redundant P2SS2P could be filtered out (The higher ratios of the redundant list were bigger than the highest ratio of the essential list). There were also cases where we found a solution to filter all the redundant P2SS2P runs (95/720–13.2%), the redundant list was not covered by the essential list.

Table 7 summarizes our results. The test column includes test names, the XF column gives the Xfactor number of literals, the Reg. (Regular) column gives the CPU time running Xfactor with no filter, the Fast column the CPU time running Xfactor with a filter, the Filter column indicate whether a part or all of the redundant P2SS2P was filtered out and Coef. is the filter ratio value.

The results given in Table 7 show that the run-time of one test, vg2_1 was dramatically reduced (from more than 40 –0.34 s), and the run-time of several other tests were significantly reduced (by tens of percents) like, sao2_3, vg2_4, vg2_6, and mux. Other tests were only slightly reduced (a few percents) and some had slightly larger run-times. The last group are very small tests—less than 25 literals in its factored form; the overhead of the added routines and just a single occurrence of a filtered P2SS2P added to their run-time. The tests whose run-times were only slightly reduced, where influenced by a low percentage of filtered P2SS2P (usually only some of these redundant P2SS2P where filtered out). It is also clear that the filter’s ratio value is not identical for all tests. Still the major impact of this filter is to shorten the time of extremely long runs (like vg2_1).
Table 7
Run-time: fast Xfactor versus regular Xfactor

<table>
<thead>
<tr>
<th>Test</th>
<th>XF</th>
<th>Reg.</th>
<th>Fast</th>
<th>Filter</th>
<th>Coef.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd53_0</td>
<td>12</td>
<td>0.06</td>
<td>0.08</td>
<td>Part</td>
<td>2</td>
</tr>
<tr>
<td>rd53_2</td>
<td>24</td>
<td>0.15</td>
<td>0.19</td>
<td>Part</td>
<td>2</td>
</tr>
<tr>
<td>sao2_3</td>
<td>34</td>
<td>0.32</td>
<td>0.25</td>
<td>Part</td>
<td>3</td>
</tr>
<tr>
<td>vg2_1</td>
<td>56</td>
<td>&gt;40.00</td>
<td>0.34</td>
<td>Part</td>
<td>8</td>
</tr>
<tr>
<td>vg2_4</td>
<td>42</td>
<td>1.49</td>
<td>0.56</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>vg2_5</td>
<td>14</td>
<td>0.05</td>
<td>0.05</td>
<td>Part</td>
<td>3</td>
</tr>
<tr>
<td>vg2_6</td>
<td>39</td>
<td>0.83</td>
<td>0.43</td>
<td>All</td>
<td>4</td>
</tr>
<tr>
<td>vg2_7</td>
<td>14</td>
<td>0.07</td>
<td>0.07</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>z4m1_0</td>
<td>10</td>
<td>0.05</td>
<td>0.06</td>
<td>Part</td>
<td>1</td>
</tr>
<tr>
<td>b9_a1</td>
<td>12</td>
<td>0.06</td>
<td>0.06</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>b9_i1</td>
<td>12</td>
<td>0.06</td>
<td>0.08</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>b9_z0</td>
<td>18</td>
<td>0.14</td>
<td>0.12</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>cm150a</td>
<td>47</td>
<td>0.62</td>
<td>0.59</td>
<td>Part</td>
<td>4</td>
</tr>
<tr>
<td>cm82a_g</td>
<td>18</td>
<td>0.13</td>
<td>0.14</td>
<td>All</td>
<td>1</td>
</tr>
<tr>
<td>i2</td>
<td>209</td>
<td>15.34</td>
<td>15.29</td>
<td>All</td>
<td>1</td>
</tr>
<tr>
<td>mux</td>
<td>47</td>
<td>1.94</td>
<td>0.71</td>
<td>Part</td>
<td>4</td>
</tr>
</tbody>
</table>

8. Conclusions and future work

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. Our results are competitive with the algebraic and Boolean methods. Still, more work is needed in order to find better partitioning algorithms that are good for tightly coupled graphs, and in order to produce good results on Boolean functions with XOR operations.

We also discussed how to reduce the number of P2SS2P calls and noted that more work is needed to be done to recognize when a conversion will not be worth doing. We further raise the question of whether one can use these cluster graphs not only for factoring but for other synthesis operations.

Another line of future research worthy of investigation would be to identify additional cases for which an upper bound for the performance of Xfactor can be given (besides the known case of read-once functions). We know of no other special families of Boolean functions for which a provable estimate can be given for the time-complexity of Xfactor.

References


