Factoring and Recognition of Read-Once Functions using Cographs and Normality

Martin C. Golumbic
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat Gan 69978, Israel
golumbic@cs.biu.ac.il

Aviad Mintz
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat Gan 69978, Israel
mintz@cs.biu.ac.il

Udi Rotics
School of Mathematics and Computer Science
Netanya Academic College
Netanya 42365, Israel
rotics@mars.netanya.ac.il

ABSTRACT

An approach for factoring general boolean functions was described in [5] which is based on graph partitioning algorithms. In this paper, we present a very fast algorithm for recognizing and factoring read-once functions which is needed as a dedicated factoring subroutine to handle the lower levels of that factoring process. The algorithm is based on algorithms for cograph recognition and on checking normality. Our method has been implemented in the SIS environment, and an empirical evaluation is given.

Categories and Subject Descriptors

J.6 [**Computer Application**]: Computer Aided Engineering; I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation

General Terms

Factoring Algorithms

Keywords

Read-Once Functions, Factoring, Normality, Cograph

1. INTRODUCTION

A Boolean function F is called a *read-once function* if it has a factored form in which each variable appears exactly once. For example, the function $F = \mathbf{F_1} = aq + acp + ace$ is a read-once function since it can be factored into the read-once formula $F = \mathbf{F_2} = a(q + c(p + e))$. Read-once functions were first introduced by Hayes [8] and were called *fanout-free* functions and are also known as *non-repeatable tree* functions since the parse tree of a read-once formula has no variable repeated. Read-once functions have interesting special properties [6], [9], [10] and according to [11] 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 [2].

Hayes described an algorithm for identifying and factoring fanout-free functions based on adjacency of the function variables [8], however its algorithm suffers from high complexity.

Peer and Pinter described in [11] a factoring algorithm for read-once functions. They have proved that their algorithm gets the optimal results but the main drawback of the algorithm was its non-polynomial complexity. The main reason for this non-polynomial complexity is due to the need for repeated calls to a routine that converts sum-of-products boolean function representation to a product-of-sums boolean function representation, or vice-versa.

In [5] a factoring algorithm for general Boolean functions was described which builds the factored form from top to bottom using graph partitioning, and where read-once functions are handled specially as they appear at the lower levels of the factoring process. It was noted there, that this algorithm incorporates read-once recognition and factoring with a new method having polynomial complexity. As an integral component of general factoring using graph partitioning, the polynomial algorithm for read-once functions is presented in this paper.

Algorithms based on Algebraic factoring (Quick Factor, Good Factor) [1],[13] can also be used in order to factor read-once functions. These general factoring algorithms have polynomial complexity and from our experiments they produce the read-once tree for read-once functions. No formal proof has been given (nor any counter example) showing that they correctly recognize read-once functions, but even in the case that such a proof can be given, the algorithms will spend redundant time (they will have to run to completion) before they identify that a function is not read-once. We will refer to these algorithms according to their factoring engine (ex. QF and GF) and compare their performance with our method.

2. PROPERTIES OF READ-ONCE FUNCTIONS

The underpinnings of our algorithm are based on the properties of read-once functions in [6] and on the properties of P_4-free graphs [3]. First, we will introduce several definitions in order to explain the main theorem of [6] that we use in our algorithm. Then we will describe the theorem and

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

we will discuss P_4-free graphs, also known as cographs. The graph P_4 denotes a chordless path containing only 3 edges and 4 vertices. A graph is P_4-free if it contains no copy of P_4 as an induced subgraph.

A *unate* function is a Boolean function represented by a formula such that each variable appears either in the positive or in the negative form throughout the formula. Read-once functions are unate, but $F = ab + ac + bc$ is an example of a unate function which is not a read-once function.

Let F be a boolean function. We define the *Dual function* F^* of F by the following: $F^* = F^*(a_1, ..., a_r) = F'(a'_1, ..., a'_r)$ where a' and F' denotes the negation of a and F resp. For example, let $F = ab + cd$, according to the definition of the Dual function we will have $F^* = (a'b' + c'd')'$ which is equal to $(a+b)(c+d)$. Here we note the well known property that the dual function replaces each $*$ operator with $+$ operator and vice versa.

The dual operation * has several properties including

- $F^{**} = F$
- $(F')^* = (F^*)'$
- $(F_1 + F_2)^* = {F_1}^* * {F_2}^*$
- $(F_1 * F_2)^* = {F_1}^* + {F_2}^*$

where the last two are De Morgan laws.

We define a graph $\Gamma = (V, E)$ of a unate function F which is given in sum of products form $\mathbf{F}$. The vertices in $V = \{a_1, \ldots, a_N\}$ represent the different literals in $\mathbf{F}$, so the number of vertices in the graph Γ is equal to the size of the support of $\mathbf{F}$. (We remind that the *support* of a function is the set of the function's variables). An edge (a_i, a_j) in E exists if and only if its literals a_i, a_j both appear in some product term of $\mathbf{F}$. Thus each product in $\mathbf{F}$ induces a clique in Γ.

The mapping of $\mathbf{F}$ to Γ is not one to one, there can be more than one function mapped to the same graph Γ. As an example, the functions $F_1 = abc$ and $F_2 = ab + bc + ac$ are mapped to the same graph Γ.

Similar to the mapping of $\mathbf{F}$ to Γ we define a mapping of Γ to function $\mathbf{F}_\Gamma$ as follows. For each arbitrary graph Γ let us find the set of all maximal cliques of the graph and transform each maximal clique to a cube in $\mathbf{F}_\Gamma$, thus obtaining a sum-of-products.

We will say that a function F is *normal* if mapping it to a graph Γ and vice versa will yield the original function, i.e., $\mathbf{F} = \mathbf{F}_\Gamma$. As an example the function $F_1 = abc$ is normal while the function $F_2 = ab + ac + bc$ is not.

The following is the main theorem for read-once functions stated by [6]:

THEOREM 1. *Let F be a unate function and F^* be its dual function and let $\overline{\Gamma}$ be the complement graph of Γ ($\overline{E}$ are the edges of the graph $\overline{\Gamma}$). Then all the following statements are equivalent:*

- *F (F^*) are read-once functions.*
- *F (F^*) is normal and its graph Γ (Γ^*) contains no subgraph isomorphic to P_4 (P_4-free).*
- *The graphs Γ and Γ^* have no edges in common.*
- *The union of the graphs Γ and Γ^* forms a complete graph on the support of F (F^*).*
- *$\Gamma = (V, E)$ and $\Gamma^* = \overline{\Gamma} = (V, \overline{E})$.*

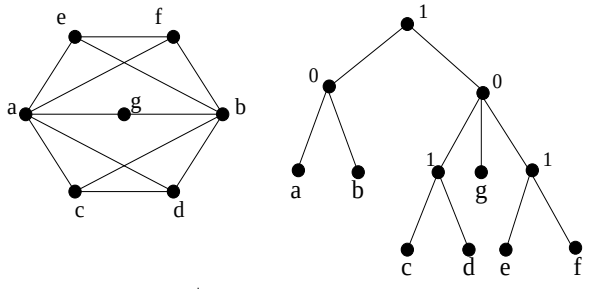

Figure 1: CoGraph and CoTree example

From this theorem one can sketch a procedure to recognize a read-once function. Given a formula, generate the graph Γ, check if the graph Γ is a P_4-free graph and then check if the function that is represented is normal.

We note here, that checking for normality is not simple. Although mapping a formula $\mathbf{F}$ to the graph Γ is obvious, the reverse mapping is not. This reverse mapping should find all maximal cliques of a graph, a problem which is NP-complete in general. We will solve this problem differently for P_4-free graphs as will be seen in section 3.1.

P_4-free graphs are known also by the name of *cographs* - complement reducible graphs and are discussed in [3]. We mention the following properties of cographs:

- Any subgraph of a cograph is also a cograph.
- The complement of a cograph is also a cograph.
- A complement of a connected cograph is disconnected.

A cograph has a unique tree representation called the *cotree*, it forms the basis of cograph recognition, and in our case for generating all maximal cliques.

The cotree is constructed as follows: The cotree leaves are the vertices of the corresponding cograph. Every internal node except possibly the root, will have two or more children; the root will have only one child exactly when the represented cograph is disconnected. Moreover, the cotree for a particular cograph is unique up to a permutation of the children of the internal nodes.

The internal nodes in the cotree are labeled as follows: the root is labelled 1, the children of a node with label 1 are labelled 0, the children of a node labelled 0 are labelled 1. We will refer to the internal nodes of cotree as 0-nodes and 1-nodes. Two nodes x and y of the cograph are adjacent if and only if the unique path from x to the root of the tree meets the unique path from y to the root of the tree at a 1-node.

For example, Figure 1 illustrates a cograph and its related cotree of the function $F = acd + aef + ag + bcd + bef + bg = (a + b)(cd + ef + g)$. We can see that the nodes of the cograph are the literals of F and that each product in F is represented by a clique in the cograph. Furthermore, nodes a and c in the cograph are adjacent and their paths in the cotree meets at the root which is a 1-node, while nodes c and e are not adjacent in the cograph and their paths in the cotree meets at a 0-node.

An algorithm recognizing cographs and building the cotrees is based on the properties of cographs and is described in

Figure 2. We will use $\mathbf{T}$ to represent the cotree, $\{\mathbf{T}_i\}$ to represent the cotree's nodes which are labelled by 0 or 1 for any internal nodes, or labelled by the literal name for any of the tree's leaves ($\mathbf{T}_0$ represents the root of the cotree). The CoGraph_Rec algorithm is initially called with $G = \overline{\Gamma}$, $label(\mathbf{T}_0) = 1$ and $k = 0$.

```
CoGraph_Rec(G, T_i)
  if (G contains only one vertex)
    label(T_i) = literal_name
    return true
  else
    if (G is connected)
      if (T_i ≠ T_0) return false
      else
        k++
        Generate a new son to T_i named T_k
        label(T_k) = ¬ label(T_0)
        return (CoGraph_Rec(Ḡ, T_k))
    else
      foreach connected component G_j of G (j = 1, ..., l)
        k++
        Generate a new son to T_i named T_k
        label(T_k) = ¬ label(T_i)
        return (CoGraph_Rec(Ḡ_j, T_k))
```

Figure 2: CoGraph_Rec Algorithm

At each iteration of CoGraph_Rec, the algorithm first checks for a degenerate graph which includes only one vertex (a success) and ends its recursion. Otherwise it checks if the given graph is connected, this is a failure condition (indicates that the given graph is not a cograph) for all vertices except for the root where it calls itself with the complement of the graph. If the given graph is disconnected, for each connected subgraph the CoGraph_Rec algorithm calls itself using the complement of a connected subgraph as the given graph. During each iteration the cotree is built, each internal node is labelled with 1 or 0 according to the cotree properties (k is the node's index). Figure 3 shows the cotree generation process on the example from Figure 1 (which represents G_0) using the CoGraph_Rec algorithm.

Two other efficient recognition algorithms are based on the generation of a cotree and are described in [4] and [7].

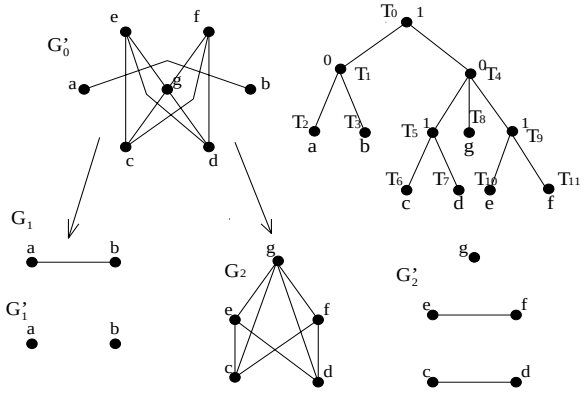

Figure 3: CoTree generation using the CoGraph_Rec algorithm

3. PROPOSED READ-ONCE ALGORITHM

Any unate function has the property that its simplified SOP or POS representations include only prime implicants which are all essential [8]. We assume that a given form is simplified and is composed of a sum of prime implicants or a product of prime implicates. A *prime implicant* (resp., *prime implicate*) is a minimal product (resp., sum) of literals whose truth (resp., falsehood) implies the truth (resp., falsehood) of the function and whose removal from the formula would change the function.

The parse tree (or computation tree) of an SOP (resp., POS) may be regarded as a three 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., $*$). In an SOP the level one nodes represent the prime implicants; in a POS they represent the prime implicates.

The algorithm, so called Is_Read_Once_Function (Here: *IROF*) is built from several procedures; it is given a function F in its parse tree representation and produces the read-once tree if F is a read-once function, and reports failure otherwise. Figure 4 includes the major steps of the algorithm.

In the first step, Check_Unate procedure checks if F is a unate function by scanning the tree in a BFS order while checking that no literal and its negation exist simultaneously. Success in this step will yield N, the size of the support of F, while a failure will finish the algorithm.

In the next step, the Build_Graph procedure generates the graph Γ of the function F. The graph itself is implemented by an $N * N$ matrix which we call the *adjacency matrix* M. Since Γ is an undirected, unweighted graph, M is a symmetric binary matrix. $M(i, j)$ is equal to 1 iff the literals i and j share a common implicant (implicate).

Now the procedure Graph_Partition determines if the Γ graph is a cograph. Since the read-once function recognition has cograph recognition as its first stage, according to Theorem 1 the CoGraph_Rec algorithm is used (Figure 2) in order to generate the cotree, then a simple conversion transforms the cotree to the read-once parse tree. The conversion maps each 1-node in the cotree to an AND node in the read-once tree, and each 0-node to an OR node. It is obvious that a failure on the cograph recognition step will conclude the algorithm with a failure.

The final step, Check_Normality, checks if the function is normal, also as required by Theorem 1. In order to check normality the algorithm compares the read-once function represented by the read-once tree with the original input function. If the functions are logically equivalent, then the algorithm switches the original tree with the read-once tree, otherwise it fails. The efficient implementation of Check_Normality will be described in section 3.1

```
Is_Read_Once_Function( root)
  if (¬ Check_Unate( root)) return false
  Γ = Build_Graph( root)
  if (¬ Graph_Partition(Γ, new_root)) return false
  if (¬ Check_Normality( root, new_root)) return false
  Swap_Net_Subtree( root, new_root)
```

Figure 4: Is_Read_Once_Function Algorithm

The algorithm was implemented in the SIS environment, using SIS database and functions. The algorithm was also

```
Check_Normality( input_root, ro_root)
  input_root_operation = input_root's operation
   input_cube_array = Build_Cube_Array( input_root)
   ro_cube_array = Build_Cube_Array( ro_root)
  return (Is_Equal( input_cube_array, ro_cube_array))
```

Figure 5: Check_Normality Algorithm

```
Build_Cube_Array( node)
  if ( node's operation == PRIMARY_INPUT)
     val = Get_Val( node)
    Insert val to cube_array
  if ( node's operation == input_root_operation)
    foreach ( node's fanin)
      tmp_array = Build_Cube_Array( fanin)
      cube_array = Append
                        ( cube_array, tmp_array)
  else
    foreach ( node's fanin)
      tmp_array = Build_Cube_Array( fanin)
      cube_array = Vector_Multiply
                        ( cube_array, tmp_array)
```

Figure 6: Build_Cube_Array Algorithm

tested in the SIS environment, and our empirical results are reported in section 4. A full description of the SIS environment can be found at [12].

3.1 Check Normality

One of the major steps of the algorithm is checking that F, the original function, is normal. In other words, the algorithm checks that the conversion from $\mathbf{F}$ to the graph Γ and back to a function $\mathbf{F}_\Gamma$, yields the original function ($\mathbf{F} = \mathbf{F}_\Gamma$).

The comparison is between the input SOP/POS representation of the F represented by a parse tree whose root is named *input_root* and between the generated read-once tree whose root is named *ro_root*. The goal is to do the comparison with minimum complexity and without collapsing the read-once tree.

The procedure Check_Normality receives the input parse tree and the generated read-once tree. It represents each tree by an integer vector and then checks for vector equivalence. If the vectors are equal the two formulas represents the same function, thus the original formula is a read-once function. The vector representation is needed to prevent sorting literals on each of the formulas before testing for equivalence. Figure 5 describes the Check_Normality procedure.

Each value in the vector of the input tree represents each product/sum in a Sum-Of-Product/Product-Of-Sum representation of the input function. In the case of the generated read-once tree, the generated vector represents an equivalent Sum-Of-Product/Product-Of-Sum form. Both vectors are made by Build_Cube_Array procedure and the equivalence check is made by Is_Equal procedure.

The Build_Cube_Array procedure builds the integer vector while scanning the tree bottom up, in a DFS order, taking care that the final form of both trees will be the same. On each leaf (representing a literal), the Build_Cube_Array procedure generates a vector containing one value given by the

```
Vector_Multiply( cube_array, tmp_array)
  if (Size( cube_array) == 0)
     new_array = tmp_array
  else
     new_array = 0, k = 0
    for(i=0; i < Size( cube_array); i++)
      for(j=0; j < Size( tmp_array); j++)
        new_array[k] =
              cube_array[i] + tmp_array[j]
        k++
```

Figure 7: Vector_Multiply Algorithm

Get_Val procedure. This value is different from leaf to leaf and is equal to 2^i, where i is the literal's index given earlier (i varies from 1 to N, where N is equal to the size of the support of the function).

On each internal node, the Build_Cube_Array procedure checks the node's operation (OR/AND). It compares the node's operation to the *input_root*'s operation (getting same format of both trees). If the operations are equal, the Build_Cube_Array procedure appends all the vectors of the node's sons to one vector which represents the current node. Otherwise, the Build_Cube_Array procedure performs a *vector multiply* operation between all the vectors of the node's sons, yielding one vector which represents the current node. The Build_Cube_Array procedure is given in Figure 6.

A *vector multiply* operation is an operation between vectors which outputs a vector with a size equal to the product of the sizes of the input vectors, and where each value is equal to a sum of one element from each vector. For example a vector multiply of $(1,2),(4,8),(16,32,64)$ will yield a vector with a size of 12 and with a value of $(21,37,69,25,41,73,22,38,70,26,42,74)$. The *vector multiply* operation is done by the Vector_Multiply procedure and is given in Figure 7.

On the last step, the Check_Normality routine has to check that the vectors are equal. This is done by first checking the sizes and only if they agree, then sorting each vector and checking element by element.

3.2 An Example

In this section we will examine an example of the algorithm. Let

$$F = ace + ade + bce + bde + f$$

be a logic function which is represented in a three level tree. First, the algorithm checks if F is a unate function and finds F as a positive unate function. During this check the algorithm sets N to 6 which is the size of the support of F. The algorithm builds the adjacency matrix according to the graph of the function (Figure 8).

After generating the adjacency matrix, the algorithm runs the cograph recognition algorithm which yields a cotree. This cotree is transformed to an AND/OR tree presented in Figure 9.

Then, the algorithm checks for normality. This procedure uses the *Build_Cube_Array* procedure to calculate the integer vectors of the original tree and of the read-once tree. Here, the original tree is given by: $\mathbf{F}_1 = ace + ade + bce + bde + f$ and the generated tree is given by $\mathbf{F}_2 = f + e(a+b)(c+d)$.

The following is the calculation of the integer vectors

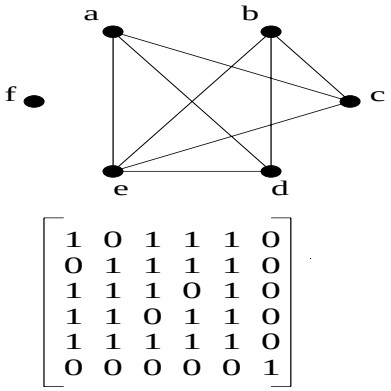

Figure 8: The graph of the function and its adjacency matrix

over the original tree: *Build_Cube_Array* starts with the *input_root* where it finds a *node* which has the same operator as original *input_root* operator (which is itself), so it calls itself with each of its sons (Here: cubes) and then appends all the son's vectors. During each recursive call (for each son) *Build_Cube_Array* performs a *vector multiply* operation on each of the *input_root*'s grandsons. But, all these nodes are primary inputs which are assigned with a vector which includes one integer. Thus, the *vector multiply* operation is reduced to a simple sum and each son is represented by a one element vector.

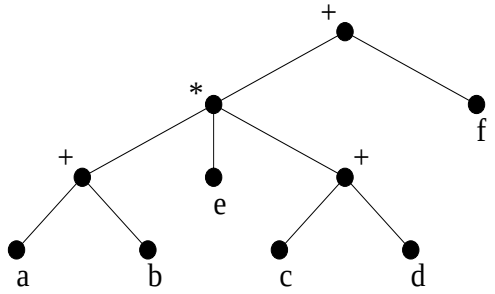

Figure 9: The Read Once tree

For an arbitrary assignment of integers (powers of 2) for each literal given by *Get_Val* procedure:

$$(a \Leftrightarrow 1, b \Leftrightarrow 2, c \Leftrightarrow 4, d \Leftrightarrow 8, e \Leftrightarrow 16, f \Leftrightarrow 32)$$

the content of *input_root* vector is: $21, 25, 22, 26, 32$.

In the case of the read-once tree, the vector calculation process is represented in Figure 10.

After calculating of both vectors, the *Is_Equal* procedure checks for equivalence by first comparing vectors sizes (Here: 5) and then sorting each vector and comparing them.

3.3 Complexity Analysis

In order to compute the complexity of the algorithm, we will need several notations. Let L be the size of the input length of **F** (in SOP or POS representation). Let **F** be composed of K cubes/sums, where each of the cubes/sums

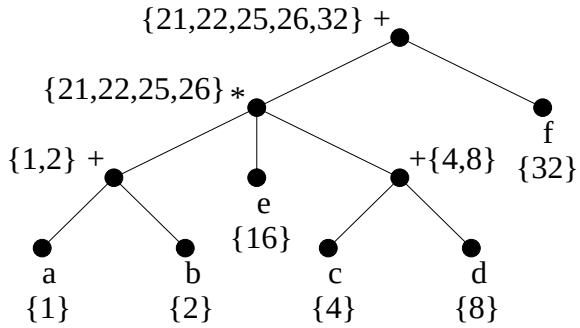

Figure 10: Vector Calculation Process

includes at most C literals, thus $L \leq K \times C$. Let N be the number of different literals in the read-once function. Note that $N > C$. For the worst case we can assume $N \approx C$ (thus $L \approx K \times N$).

The algorithm is built from a number of different procedures, thus the one that has the maximum complexity will dominate the algorithm complexity.

- *Check_Unate* - the procedure scans the input tree, thus its complexity is $O(L)$.
- *Build_Graph* - the procedure runs over the input tree and for each cube/sum it generates a clique, thus the complexity is equal to K times the complexity of building a clique which is C^2. Thus, the complexity is $O(K \times C^2)$, taking $C \approx N$ yields $O(L \times N)$.
- *Graph_Partition* - this procedure has a known linear implementation [4],[7] in $O(E+N)$, where E represents the number of edges in graph Γ. Thus the complexity is $O(E + N)$ which is bounded by $O(N^2)$.
- *Check_Normality* - this procedure builds two vectors of integers, where each integer is represented by N bits. It first counts each appearance of each literal in the input tree and in the read-once tree, then it sorts the two vectors and compares them. The complexity of building the vector corresponding to the original function is $O(L \times N)$. Building the vector corresponding to the read-once tree can be implemented in $O(L \times N)$ as follows. The Build_Cube_Array procedure will keep a counter for the sizes of the intermediate vectors it generates while scanning the read-once tree. The procedure will perform an append/multiply operation on two vectors only if the size of the resulted vector will not exceed K. Otherwise, the procedure will stop and claim that the function is not normal. This test guarantees that the size of the vector resulted from the read-once parse tree will not exceed K. If the read-once tree is binary, then each append/multiply operation takes at most $O(K \times N)$ time and since there are at most $O(N)$ operations, the total complexity is $O(K \times N^2) = O(L \times N)$. If the read-once tree is not binary, it can be transformed to a binary tree by replacing each non-binary append/multiply operation by a sequence of binary append/multiply operations, such that the total number of operations in

the resulted binary tree is at most $O(N)$. The complexity of sorting the two K length vectors is $O(N \times K \log K)$, since the operation of comparing two elements of these vectors takes $O(N)$ time. Thus the total complexity of Check_Normality function is the maximum between $O(L \times N)$ and $O(N \times K \log K)$. But $O(L \times N)$ and $O(N \times K \log K)$ are equal, since $O(L) = O(N \times K)$ and $\log K < N$. Thus the complexity of the Check_Normality procedure is given by $O(L \times N)$.

- *Swap_Net_Subtree* - this procedure just swaps pointers, and thus has a complexity of $O(1)$.

Gathering all this data, we can see that the most critical procedures are *Build_Graph* and *Check_Normality* with complexity of $O(L \times N)$ time. Thus the complexity of the whole algorithm is given by $O(L \times N)$.

4. EMPIRICAL RESULTS

We compared our IROF algorithm to the PPF algorithm [11] and to the QF and GF algorithms from the SIS package by running them on various read-once functions. All the algorithms generate the read-once factorization in all the examples and the results given in Table 1 compare the CPU time (given in seconds) running each algorithm on each example. **SOP** indicates the number of literals in the Sum Of Product form and N denotes the number of variables.

We named each formula we tested by a sequence of characters containing two numbers and two letters l and b. The number after the letter l represents the number of levels in the read-once tree and the number after the letter b represents the number of branches in the first level. Results are given textually in Table 1.

Table 1: Read Once results (CPU time)

	SOP	N	QF	GF	PPF	IROF
l2_b10	10240	20	0.11	fail	44.67	0.3
l4_b3	3072	24	0.11	4.67	8.43	0.1
l4_b6	7290	24	0.19	147.41	25.1	0.21
l6_b4	672	20	0.03	0.3	0.41	0.02
l6_b8a	132	52	0.04	0.06	0.48	0.02
l6_b8b	24192	52	1.52	fail	345.04	0.74
l8_b5	3380	29	0.09	5.88	6.16	0.09
l10_b3	2160	30	0.11	2.33	2.67	0.07
l14_b3	6720	42	0.44	20.96	22.45	0.2

From the results given in Table 1 we can see that both PPF and GF algorithms are much slower than QF and our IROF algorithms. The PPF algorithm suffers from Sum of Products to Product of Sums and Product of Sums to Sum of Products conversions which consumes much CPU time. The GF algorithm suffers from finding all the kernels of each example which is a very difficult process.

The difference in run time between the QF and IROF results are quite small, from the table we can see that except for the first example (l2_b10), the IROF algorithm consumes less or equal CPU time as QF. We also observe that QF is faster on read-once functions with few levels while IROF is faster on read-once functions with many levels.

However, another major difference between our IROF algorithm and algorithms based on Algebraic division is when they are executed with a non-read-once function. In that case, IROF finds that the function is not read-once during its Graph_Partition and Check_Normality routines (see Figure 4) while the algorithms based on Algebraic division need to complete their Algebraic factoring and only then check that there are literals that appear more than once.

5. CONCLUSIONS

We have presented a very fast algorithm for recognizing and factoring read-once functions. This algorithm appears in the general factoring algorithm [5] as an essential subroutine for factoring the read-once functions which appear at the lower level of the general factoring process. Our method is based on cograph recognition and on normality checking. We discussed the algorithm complexity and we showed empirical results running the algorithm in the SIS environment in comparison to other methods.

6. REFERENCES

[1] R. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems, (Rome, May 1982)*, pages 49–54, 1982.

[2] N. Bshouty, T.R.Hancock, and L. Hellerstein. Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates. *J. Comput. System Sci.*, 50:521–542, 1995.

[3] D. Corneil, H. Lerchs, and L. Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3:163–174, 1981.

[4] D. Corneil, Y. Perl, and L. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 14:926–934, November 1985.

[5] M. Golumbic and A. Mintz. Factoring logic functions using graph partitioning. In *Proc. IEEE/ACM Int. Conf. Computer Aided Design, November 1999*, pages 195–198, 1999.

[6] V. Gurvich. Criteria for repetition-freeness of functions in the algebra of logic. *Soviet Math. Dokl.*, 43(3):721–726, 1991.

[7] M. Habib, C. Paul, and L. Viennot. Partition refinement techniques: An interesting algorithmic tool kit. *International Journal of Foundations of Computer Science (IJFCS)*, 10:147–170, 1999.

[8] J. P. Hayes. The fanout structure of switching functions. *J. of the ACM*, 22:551–571, 1975.

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

[10] I. Newman. On read-once boolean functions. In *Boolean Function Complexity: Selected Papers from LMS Symp. Durham, July 1990*, pages 24–34, 1990.

[11] J. Peer and R. Pinter. Minimal decomposition of boolean functions using non-repeating literal trees. In *IFIP, November 1995*, 1995.

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

[13] A. R. R. Wang. *Algorithms for Multilevel Logic Optimization.* Ph.D. Thesis, University of California, Berkeley, CA, 1989.