Read-Once Functions (Revisited) and the Readability Number of a Boolean Function

Martin Charles Golumbic
Caesarea Rothschild Institute
University of Haifa

Joint work with
Aviad Mintz and Udi Rotics
Outline of Talk

- Read Once Functions
- Motivation
- CoGraphs and CoTrees
- Graph of a Formula
- Normality
- Gurvich Theorem
- Previous Work of Recognizing Read Once Functions
- Our Method for Recognition (the ROF Algorithm)
- Complexity of Checking Normality
- Results
- Advantages and Drawbacks
- Conclusions
Read-Once Functions

- A Boolean function that has a factored form in which each variable appears only once.

Example. A read-once function:

\[ F = aq + abp + abd = a(q + b(p + d)) \]
Read-Once Functions

- A Boolean function that has a factored form in which each variable appears only once.

  Example. A read-once function:

  \[ F = aq + abp + abd = a(q + b(p + d)) \]

- Example. Not read-once functions:

  \[ F' = ab + bc + cd \]

  \[ F'' = ab + bc + ac \]

- Read-once functions are assumed to be positive.
Our Motivation

- The Read-Once algorithm (so called ROF) is a dedicated factoring subroutine to handle the lower levels of the general factoring algorithm using graph partitioning.
Graph of a Boolean Function

- Each literal of function $F$ is mapped to a vertex in the graph “co-occurrence” graph $\Omega = (V,E)$.

$$V = \{a_1, \ldots, a_n\}$$

- Each prime implicant (minterm, cube) forms a clique in $\Gamma$ (not necessarily necessarily maximal).

$$E = \{(a_i, a_j) \mid a_i, a_j \in C_k \text{ for some } k\}$$

where $C_k$ is a prime implicant.
Graph of a Formula - Example

\[ F = acd + aef + ag + bcd + bef + bg \]
\[ V = \{a, b, c, d, e, f, g\} \]
CoGraphs

Recursive Definition:

- A single vertex is a cograph.
- The disjoint union of cographs is a cograph.
- The join of cographs is a cograph,
  (i.e., adding all edges between them).
CoGraphs

Recursive Definition:
- A single vertex is a cograph.
- The disjoint union of cographs is a cograph.
- The join of cographs is a cograph, (i.e., adding all edges between them).

CoTree Representation:
- Leaves are labeled by the vertices.
- Union nodes labeled by 0 -- or by +
- Join nodes labeled by 1 -- or by *.
CoTrees (Example)

cotree

cograph
CoTrees (Example)
CoGraph Properties

- The complement of a CoGraph is also a CoGraph, and
- The coTree of $G$ is obtained from the coTree of $\overline{G}$ by reversing the 0/1 labels of the internal nodes.
- The complement of a connected CoGraph is disconnected.
- Two vertices of a coGraph $G$ are adjacent iff their lowest common ancestor in the coTree is labeled 1.
- The CoTree is a unique representation of the CoGraph (up to isomorphism – blowing in the wind).

Two views: recursive construction
recursive decomposition
$P_4$-free Graphs

- $P_4$ - A chordless path containing 3 edges and 4 vertices.

- A $P_4$-free Graph does not contain any copy of $P_4$ as an induced sub-graph.

- $P_4$-free Graphs are equivalent to CoGraphs!
Theorem.
The following conditions are equivalent:
1. $G$ is a cograph *
2. $G$ has a cotree representation **
3. $G$ is $P_4$–free
4. For every subset $X$ of the vertices, $|X|>1$, either the induced subgraph $G_X$ is disconnected or its complement $\overline{G}_X$ is disconnected.

* (recursive construction)
** (recursive decomposition)
Recognizing CoGraphs

- A naïve algorithm for recognizing a given coGraph and constructing the coTree:

  1. If G is disconnected, make labeled root 0 and continue recursively on each component.

  2. If G is connected, make labeled root 1, form the complement and FAIL if it is connected; otherwise continue recursively.
Recognizing CoGraphs

- A naïve algorithm for recognizing a given coGraph and constructing the coTree:
  1. If $G$ is disconnected, make labeled root 0 and continue recursively on each component.
  2. If $G$ is connected, make labeled root 1, form the complement and FAIL if it is connected; otherwise continue recursively.

- Complexity of naïve is $O(n^3)$, but there are

- Linear time algorithms: Corneil,....Habib ....
Recognizing CoGraphs

- A naïve algorithm for recognizing a given coGraph and constructing the coTree:

  1. If G is disconnected, make labeled root 0 and continue recursively on each component.

  2. If G is connected, make labeled root 1, form the complement and FAIL if it is connected; otherwise continue recursively.

- Complexity of naïve is $O(n^3)$, but there are

- Linear time algorithms: Corneil,….Habib ....

- The term “cograph” stands for “complement reducible graph” since it can be decomposed using the naïve algorithm.
CoGraph Generation - Example
CoGraph Generation – Example
(Cont. 1)
CoGraph Generation – Example (Cont. 2)
CoGraph Generation – Example (Cont. 3)
Recall: Co-occurrence Graph $\Gamma$ of a Boolean Function

- Each literal of the function $F$ is mapped to a vertex in the graph “co-occurrence” graph $\Omega = (V,E)$.

$$V = \{a_1, \ldots, a_n\}$$

- Each prime implicant (minterm, cube) forms a clique in $\Gamma$ (not necessarily necessarily maximal).

$$E = \{(a_i, a_j) \mid a_i, a_j \in C_k \text{ for some } k\}$$

where $C_k$ is a prime implicant.
Graph of a Formula - Example

\[ F = acd + aef + ag + bcd + bef + bg \]

\[ V = \{a, b, c, d, e, f, g\} \]
Extracting a Function from a Graph

**Before:** function to a graph.  **Now:** graph to a function.

- For an arbitrary graph $\Gamma$, we transform each maximal clique into a prime implicant (minterm) in $F_\Gamma$

$$F_\Gamma = abc + ad + df + cef$$

- Generally, constructing $F_\Gamma$ is an NP-Complete problem, since we must find all maximal cliques.
Normality

- $F$ is **normal** if mapping it to graph $\Gamma$ and back to a function $F_{\Gamma}$ will yield the original function, i.e. is $F = F_{\Gamma}$?

**Example:**

$F_1 = abc$

$F_2 = ab + ac + bc$

That is, $F$ is **normal** iff the cliques of $\Gamma$ are precisely the prime implicants of $F$. 
Gurvich Theorem

Let $\Gamma$ be a positive function and let $\Gamma$ be the graph of $F$. Then the following statements are equivalent:

- $F$ is a read-once function.
- $F$ is normal and its graph $\Gamma$ contains no subgraph isomorphic to $P_4$ (CoGraph).
- The graphs $\Gamma$ and $\Gamma^d$ of the function $F$ and the dual function $F^d$ are complementary.
- For all prime implicants $P$ and dual prime implicants $D$, we have $|P \cap D| = 1$. 
Example

\[ F = x_1 x_2 + x_2 x_3 + x_3 x_4 + x_4 x_5 + x_5 x_1 \]
co-occurrence graph is the chordless 5-cycle \( C_5 \).
- \( F \) is normal, but \( \Gamma_F \) is not a cograph (has a \( P_4 \))

\[ F^d = x_1 x_2 x_4 + x_2 x_3 x_5 + x_3 x_4 x_1 + x_4 x_5 x_2 + x_5 x_1 x_3 \]
co-occurrence graph is the complete graph \( K_5 \).
- \( F^d \) is not normal, but \( \Gamma_{F^d} \) is a cograph (has no \( P_4 \))
Proposed Method for Recognition

Function $f$

Boolean Formula (SOP, POS)

Build Graph

Initial Graph

CoGraph Recognition

CoGraph

Normality Checking

Read-Once

Graph $\Gamma_f$

CoTree and function $F_{\Gamma_f}$
Checking normality in polynomial complexity.

Result:
The proposed algorithm has polynomial complexity of $O(L \times N)$ where $L$ is the formula’s size and $N$ is the number of function’s variables (the support).
Previous Work on Read Once Recognition

- Peer and Pinter [1995] – Recognition and factoring of Non-Repeatable functions based on Non-Repeatable Trees. *

* Both algorithms have Non-Polynomial complexity.

Additional theoretical work has been done by:
- Gurvich [1991]
- Karchmer, Linial, Newman, Saks and Wigderson [1993]
- Goldman, Kearns and Schapire [1993]
- Bshouty, Hancock and Hellerstein [1995]
- Others…
Read-once Recognition Algorithm
Golumbic, Mintz and Rotics, 2004-08

- **Step 0:** Check that the function $f$ is positive (unate).
- **Step 1:** Build the co-occurrence graph $\Gamma_f$ of $f$.
- **Step 2:** Test whether $f$ is $P_4$-free, and construct the cotree $T$ for $f$. Otherwise, exit with “failure”.
- **Step 3:** Test whether $f$ is a normal function, and output $T$ as the read-once expression. Otherwise, exit with “failure”.

How do we do it efficiently?
CoGraph and CoTree Generation

Use one of the fast algorithms by Corneil, et al. or use the Naive algorithm:

- If the graph has only one vertex
  - The algorithm ends the recursion with a success.

- If the graph is connected and it is the initial graph
  - The algorithm calls itself with the complement of the graph.

- If the graph is connected and is not the initial graph
  - The algorithm ends the recursion with a failure.

- If the graph is disconnected
  - For each connected sub-graph, the algorithm calls itself with the complement of that sub-graph.
Checking Normality: Is $F = F_{\Gamma}$?

Compare the input SOP/POS representation of $F$ and the generated function $F_{\Gamma}$ (both represented by parse trees).

- **Build for each of $P$:** the prime implicants of $F$ and $C$: the cliques of $\Gamma$, a **bit matrix**, where each row is the characteristic vector of the subset.

- **Check for their equivalence** (Sort Lexicographically).

![Input Tree](vector-1)

![Generated Tree](vector-2)
The matrices $P$ and $C$: of $F$ and $F_\Gamma$.

- $P$ represents the prime implicants of $F$.
- $C$ represents the prime implicants of $F_\Gamma$.
- Variable $a_i$ is denoted by the bit $i = 1$.
- Each row of the matrix is an $n$-bit number encoding a set of variables.
Checking Normality: Using the cotree $T$

Clique Calculations – bottom up

- For a node $a$ of $T$, we denote by $T_a$ the subtree of $T$ rooted at $a$.

- $T_a$ is also the cotree representing the subgraph of $\Gamma_f$ induced by the set of labels of the leaves of $T_a$.

- Construct the set $C(T)$ of maximal cliques of $\Gamma_f$, recursively, traversing the cotree $T$ from bottom to top.

- More precisely, we construct the set of maximal subcliques $C(T_a)$ for each internal node $a$, combining them as we move up the tree, using two operations, set union $\cup$ and set join $\otimes$, and the following lemma:
Lemma 1. Let $h$ be an internal node of the cotree $T$, and let $h_1, \ldots, h_r$ be the children of $h$ in $T$.

(i) If $h$ is labeled with 0 in the cotree, then
\[
C(T_h) = C(T_{h_1}) \cup \cdots \cup C(T_{h_r}).
\]

(ii) If $h$ is labeled with 1 in the cotree, then
\[
C(T_h) = C(T_{h_1}) \otimes \cdots \otimes C(T_{h_r}).
\]

For example, \(\{a_1\}, \{a_2\} \otimes \{a_3, a_4\}, \{a_5, a_6\}, \{a_7\} = \{a_1, a_3, a_4\}, \{a_1, a_5, a_6\}, \{a_1, a_7\}, \{a_2, a_3, a_4\}, \{a_2, a_5, a_6\}, \{a_2, a_7\}\)

or using bit vectors, \((1,2)*(12, 48, 64) = (13, 49, 65, 14, 50, 66)\).
Checking Normality Example – The Input Tree

\[ F = acd + aef + ag + bcd + bef + bg \]

Append \((13, 49, 65, 14, 50, 66)\)

"Multiply"
Checking Normality Example – The Generated Tree

\[ F = (a + b) \times (cd + ef + g) \]
Comparing the Vectors

- For $F$: the vectors represent the prime implicants of $F$ by definition.
- For $F_{\Gamma}$: the vectors represent the maximal cliques of the cograph, and hence the prime implicants of $F_{\Gamma}$.
- By sorting the vectors, they can easily be compared.

**Important Complexity Issue** (Heuristic Speed-up):
- Pre-compute the number of cliques (and compare).
- Pre-compute the size of $F_{\Gamma}$ (and compare).
Complexity

To bound the complexity, the algorithm also computes a parameter:

\[ s(T_a) : \text{the number of the cliques in } C(T_a). \]

By comparing \( s(T_a) \) with the number of prime implicants \( k \) at each step, a speed-up mechanism is obtained -- insuring that the number of maximal cliques never exceeds the number of prime implicants (a necessary condition for normality).

To this we add an additional pre-computed parameter:

\[ L(T_a): \text{the “total length” of the list of cliques at } T_a, \text{ namely, } L(T_a) = \sum \{ |C| \mid C \in C(T_a) \}. \]

In this way, at the root of the cotree, we can also check the necessary (but not sufficient) condition that the length \( L(T) \) must equal \( |F| \), again bounding the complexity, before actually computing the clique set \( C(T) \).
Complexity

- Building the co-occurrence graph $G$ is $O(\sum n_i^2)$ for $n_i$ variables in implicant $P_i$.
- Testing whether $G$ is a coGraph and building the coTree is $O(n^2)$.
- Checking normality is $O(L*n)$.

Result:
The proposed algorithm has polynomial complexity of $O(L*n)$ where $L$ is the formula’s size and $n$ is the number of variables.
Empirical Results

- Comparing ROF with the traditional factoring methods Good Factor (GF) and Quick Factor (QF) of SIS and PPF of Peer-Pinter.

- All the different algorithms yield the read-once factorization. The next table compares the algorithms CPU time (in seconds).
## Results (Cont.)

<table>
<thead>
<tr>
<th>Formula</th>
<th>SOP</th>
<th>Literals</th>
<th>PPF</th>
<th>GF</th>
<th>QF</th>
<th>IROF</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2_b10</td>
<td>10240</td>
<td>20</td>
<td>44.67</td>
<td>Fail</td>
<td>0.11</td>
<td>0.3</td>
</tr>
<tr>
<td>L4_b3</td>
<td>3072</td>
<td>24</td>
<td>8.43</td>
<td>4.67</td>
<td>0.11</td>
<td>0.1</td>
</tr>
<tr>
<td>L4_b6</td>
<td>7290</td>
<td>24</td>
<td>25.1</td>
<td>147.41</td>
<td>0.19</td>
<td>0.21</td>
</tr>
<tr>
<td>L6_b4</td>
<td>672</td>
<td>20</td>
<td>0.41</td>
<td>0.3</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>L6_b8a</td>
<td>132</td>
<td>52</td>
<td>0.48</td>
<td>0.06</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td>L6_b8b</td>
<td>24192</td>
<td>52</td>
<td>345.04</td>
<td>Fail</td>
<td>1.52</td>
<td>0.74</td>
</tr>
<tr>
<td>L8_b5</td>
<td>3380</td>
<td>29</td>
<td>6.16</td>
<td>5.88</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>L10_b3</td>
<td>2160</td>
<td>30</td>
<td>2.67</td>
<td>2.33</td>
<td>0.11</td>
<td>0.07</td>
</tr>
<tr>
<td>L14_b3</td>
<td>6720</td>
<td>42</td>
<td>22.45</td>
<td>20.96</td>
<td>0.44</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Advantages and Drawbacks

**Advantages**

- Fast method to **recognize** read-once functions.
- Fast method to **factor** read-once functions.
- Low complexity – $O(L*n)$

**Drawbacks**

- Factors only read-once functions.
Conclusions

- Very fast algorithm for recognizing and factoring read-once function is presented.
- The algorithm appears in the general factoring algorithm of [GM 99].
- It is based on CoGraph recognition and on Normality checking.
- Comparison to other methods is shown.
Learning a Read-Once Function

- **Input:** A oracle (black box) to evaluate $F$ at any given point, where $F$ is not known, but is known to be read-once.

- **Output:** A read-once expression for $F$.

Example:

Is $(1,1,0,0)$ true?

YES: $(1,1,1,0)$ $(1,1,0,1)$ $(1,1,1,1)$ are all true.

NO: $(1,0,0,0)$ $(0,1,0,0)$ $(0,0,0,0)$ are all false.
Learning a Read-Once Function

Angluin, Hellerstein, Karpinski (1993):

**Theorem.** The Read-Once Learning problem can be solved using $O(n^3)$ time and $O(n^2)$ queries.

They showed how to build the co-occurrence graph $\Gamma$ using an oracle.
Open Questions #1

- What can be said about RECOGNITION if $F$ is given in some other type of representation?

Comment: If $F$ is some other formula (not a DNF or CNF) or if $F$ is represented as a BDD, or whatever, we might have to pay a high price to convert it into a DNF and use the GMR method.
Lisa replies:

- If F is a non-monotone DNF formula, for example, the complement of the recognition problem (i.e. does F not represent a monotone read-once function?) is NP-complete.

Thank you
Related Topics and Applications

Readability of a Boolean Function

Factoring General Boolean Functions
The Readability Number

- What if $F$ is not read-once?
- Could it be read-twice?
- Readability Number $k$:
  
  The smallest value of $k$ such that $F$ can be factored so that each variable appears at most $k$ times.
The Readability Number

**Theorem:** If \( F \) is a normal function and \( \Gamma_F \) is a partial \( k \)-tree, then \( F \) has readability number at most \( 2^k \).

Moreover, a formula can be constructed in \( O(n^{k+1}) \) time.

**Corollary:**

If \( \Gamma_F \) is a tree, then \( F \) is read-twice.
Example: A 2-tree

Figure 2: A 2-tree $G$.

$$f(G) = 1(2(3 + 4 + 5) + 3(9 + 10) + 4 \times 11) + 2(3 \times 7 + 5 \times 6) + 3 \times 7 \times 8$$
Open Questions #2

- What other graphs give read-twice functions?
  - Grids give read-twice functions. Others?
- Is our $2^k$ upper bound tight for partial $k$-trees?
- Find other families for which the readability can be determined.

Remark. Our $2^k$ method also holds for normal functions for which $\Gamma_F$ has an ordering of its vertices $v_1, v_2, \ldots, v_n$ such that $v_i$ is adjacent to at most $k$ of $v_1, v_2, \ldots, v_{i-1}$. 
Factoring General Boolean Functions Using Graph Partitioning

Aviad Mintz
Martin C. Golumbic

University of Haifa
**Cluster Intersection Graph**

- **Definition:** $G(C, E)$ is an undirected weighted graph where:
  - $C$ - clusters of the function
  - $E = \{(i, j) \mid C_i \cap C_j \neq \emptyset\}$
  - $W_{(i,j)}$ - the number of common literals

**Example:**

$$F = abd + abg + acd + cg$$
Separability Function

- Measure separability of the cluster intersection graph by a graph partitioning algorithm.
- Try to partition the graph evenly (to minimize the number of stages).
- Choose the minimal separability and return its value $\sigma$ and partitions ($A, B$ - sets of vertices).
Form Conversions – P2S and S2P Functions

- P2S - convert sum of product to product of sums
- S2P - convert product of sums to sum of products
- P2SS2P – a shortcut for both
Factoring Algorithm Using Graph Partitioning - Xfactor

Input: Parse tree T of the Boolean function being factored.

Xfactor(T).

- If T is a Read Once function - Construct the Read Once tree and exit.
- Otherwise, Construct the Cluster Intersection Graph G corresponding to T and Calculate the Separability.
- If the Separability of G is not 0 - Build $T^*$ by P2SS2P, Construct the Cluster Intersection Graph $G^*$ and Calculate the Separability.
- If the Separability of G is lower than the Separability of $G^*$ - Partition T accordingly and Call Xfactor with each part.
- Otherwise, Partition $T^*$ and Call Xfactor with each part.
Factoring Algorithm - Example

\[ F = abd + abg + acd + cg \]
Factoring Algorithm - Example (Cont. 1)

\[ F^* = (a + c)(a + g)(b + c)(d + g) \]

\[ G^*(C^*,E^*): \]

\[ Xfactor \text{ chooses the second option and partitions } G^* \text{ evenly.} \]

\[ F_1 = (a + g)(d + g) \]

\[ F_2 = (a + c)(b + c) \]
Factoring Algorithm - Example (Cont. 2)

\(F_1\) and \(F_2\) are Read Once functions.

\[
F_1 = (a + g)(d + g) = ad + g
\]

\[
F_2 = (a + c)(b + c) = ab + c
\]

Thus,

\[
F = (ad + g)(ab + c)
\]

Algebraic factoring (GF) produces worse result:

\[
F = (a(b(d + g) + cd) + cg
\]

Boolean factoring produces same result as \(Xfactor\).