Journal of Information & Optimization Sciences
ISSN 0252-2667 (Print), ISSN 2169-0103 (Online)
Vol. 47 (2026), No. 4, pp. 1567–1589
DOI : 10.47974/JIOS-2220

Novel Boolean logic manipulation : A symbolic computation and camouflaged graph construction

Gete Umbrey *
Department of Mathematics
Jawaharlal Nehru College
Pasighat
East Siang 791102
Arunachal Pradesh
India

Bhaba Kumar Sarma †
Department of Mathematics
Indian Institute of Technology Guwahati
Guwahati 781039
Assam
India

Bhuban Chandra Deuri §
Department of Mathematics
Jawaharlal Nehru College, Pasighat
Pasighat
East Siang 791102
Arunachal Pradesh
India

Alfonso Bustamante ‡
Department of Mathematics
University of Chile
Santiago
Santiago 8320000
Chile

* *E-mail:* gete.umbrey@rgu.ac.in (Corresponding Author)
† *E-mail:* bks@iitg.ac.in
§ *E-mail:* bhuban.math@gmail.com
‡ *E-mail:* athal.zigma@gmail.com

Botem Moyong [@]
Department of Mathematics
Dara Natung Government College
Itanagar 791111
Arunachal Pradesh
India

Abstract

This article introduces a systematic, symbolic, and pure algebraic technique called the "Extraction and Combination Method" for minimizing Boolean expressions. This method simplifies Boolean expressions through a sequence of algebraic manipulations that methodically extract factors and combine terms to reduce the complexity of redundant literal terms and operations. The algorithmic process involves the breakdown of the complex Boolean expression into distinct components, applying the natural and fundamental laws of Boolean algebra. Systematic ordering and generalization of Boolean operations/subexpressions ensure that we can invariably reach termination and achieve a unique minimized form for any given Boolean expression. The set of processes of decomposition of the expression into subexpressions, parallel minimization, and recombination is repeated iteratively until we get an optimized form. The minimization process avoids exhaustive truth table generation. We proposed to seamlessly transform the algebraic form used in our framework into a Boolean graph-a simple undirected graph in the graph-theoretic sense that allows Boolean expressions to be studied and manipulated through graph operations such as union and intersection, and establishes a natural bridge between algebraic logic and classical graph theory. The method enables tautology detection, construction of Boolean graph and camouflaged graph, identification and removal of auxiliary vertices and auxiliary subgraphs, and derivation of simplified Boolean graphs representing minimal expressions.

Subject Classification (2020): *05C25, 06E30, 94C10, 68Q25.*

Keywords: *Boolean graph, Algebraic graph theory, Tautology, Simplification, Obfuscation, Logic minimization.*

1. Introduction

Boolean minimization refers to the techniques/processes that reduce the number of gates in a logic circuit, vital for efficient hardware implementation, decreasing power consumption, lowering hardware costs, and enhancing overall performance. This optimization minimizes the heat generated by the chip, enhances computational speed, and reduces algorithmic and computational complexity. Some of the most common minimizing techniques are tabular or map methods and classical algebraic methods [26, 27]. The algebraic methods use Boolean algebraic axioms and theorems to simplify expressions step-by-step. However, this minimization procedure is not unique because it lacks specific rules to predict the succeeding step in the manipulative process. Rudeanu [27] suggests

[@] *E-mail:* botemmoyong.bm@gmail.com

that algebraic techniques should be considered powerful tools alongside traditional map methods in solving Boolean equations and optimizing logical systems. Map methods include the classical Karnaugh map (in short, K-map) method [19, 33]-initially developed as a manual and visual method, and the variable-entered Karnaugh map (VEKM) [29]. VEKM combines graphical representation with algebraic principles, significantly improving over the classical K-map method. The steps for minimization using a K-map include plotting the map, grouping cells of 1s or 0s, and forming the simplified expression based on the grouped cells. The comparative study of different Boolean minimization techniques, including algebraic methods due to Rudeanu, VEKM, Quine-MacCluskey algorithm, Petrik's method and leveraging truth table, and K-map in various combinational logic designs, circuit simplification or hardware reduction, etc. can be seen in [13, 17, 21, 26, 27, 28].

Brayton *et al.*[5] introduce the Espresso algorithm, a heuristic-based approach that combines heuristic rules, tabular methods, and cube representation to optimize Boolean expressions. Quine [25] introduces the Quine-McCluskey algorithm, a tabular method for finding prime implicants and essential prime implicants, which is a systematic way to minimize Boolean functions and simplify digital functions. The prime implicants are the simplest product terms covering minterms, and essential prime implicants are those prime implicants uniquely covering at least one minterm not covered by any other implicants (product terms that can cover minterms). The classical algebraic manipulation method grows exponentially with the increasing size of the problems. It requires a solution to NP-Complete or coNP-Complete problems [14] to determine whether there exists any assignment of input variables such that the expressions evaluate to 1 (satisfiability) or denote the same functions (equivalence). The classical K-map is a pictorial method for minimising Boolean expressions, but its effectiveness in reducing Boolean expressions decreases drastically beyond four variables.

Considering the superiority of visual notations over classical algebraic manipulations, Alharbi [3] has proposed a technique for representing and minimizing a Boolean expression by a truth graph. It converts the Boolean expression into a visual representation, however, this visual technique would be difficult in minimizing the complex Boolean expressions. It focuses on visualizing Boolean expressions with a few variables. The terminology ``truth graph'' is also used by Ma [23], who visually represents the truth table by circumference of the circles using the truth value coordinates; however, such representations are not effective in the Boolean minimization process. Based on Binary-Decision Programs [22], subsequently improved as Binary Decision Diagrams [2], a popular technique of graph-based algorithms for Boolean function manipulation was introduced by Bryant [6]. In the class of algorithms for manipulating Boolean functions introduced by Bryant, the Boolean functions are represented as directed acyclic graphs having several advantages; like all the symmetric functions are represented by graphs where the number of vertices grows at most as the square of the number of arguments, reduced graphs give a

canonical form that makes testing of equivalence and satisfiability easy. However, a particular ordering of arguments has to be chosen at the start of the algorithm, which is a complex problem. Haghparast and Mohammadi [16] present an efficient quantum-inspired genetic algorithm for Boolean function minimization, combining optimization techniques with quantum-inspired computing. Ghosh [15] uses the Boolean minimization technique to explore and optimize reversible logic circuit synthesis for FPGA (Field-Programmable Gate Array). The work of Das, and Mandal [11] discusses algorithms for minimizing Binary Decision Diagrams (BDDs), a vital data structure in Boolean function manipulation. Further readings including recent works describing Boolean minimization as a vital field, especially in circuit complexity and digital logic design can be seen in [9, 12, 18, 24, 35].

An important technique to reduce logic minimization's complexity is breaking down the expression into simpler sub-expressions. Some of the methods mentioned above have now evolved into parallel or adopted approaches from purely sequential to parallel algorithms, such as parallel Quine-MacCluskey for splitting minterms into groups and processing in parallel; Parallel Espresso for VLSI synthesis; MapReduce for extensive data (digital design verification), and more exclusive parallel processing design such as GPU-based minimizers for massive parallel speed. However, these parallel processing techniques also encounter exponential growth due to an increase in variables, load balancing in parallel (some processors finish early, while others lag), exponential growth in the truth table size, and, above all, no algorithm guarantees an optimal solution. One of the inspirations for our approach is the Shannon expansion or decomposition that allows breaking a Boolean expression into two sub-expressions based on a chosen variable, and the fullest reduction could be achieved by repeating Shannon's expansion theorem for each variable in the expression [32].

We employ a series of reduction axioms in a systematically predefined sequence to simplify Boolean expressions until a saturation point is reached. By convention, the precedence of the operations from NOT (highest) to AND, then OR (lowest) equipped with clausal subsumption naturally ensures the conversion of a given Boolean expression into *Minimal Disjunctive Normal Form (MDNF)*. *Clausal subsumption* referred to as *product term subsumption*, is treated as the highest iteration precedence in the proposed algorithm (for details on the subsumption theorems, see [8]). However, the *product term subsumption* alone cannot fully minimize a general Boolean expression. Thus, we incorporate various Boolean axioms in sequence in the algorithm, including absorption and distributivity. By iteratively breaking down into smaller sub-expressions, simplifying individually, and re-combining the sub-expressions, we ensure a simplified expression and termination of the algorithm after a finite number of iterations. Here, a simplified expression of a Boolean function means a form that cannot be further reduced, i.e., a saturation point is attained. In the following section, we define the notion of a saturation point, which guarantees the simplest form of the expression. Nevertheless, some of the existing techniques, including MinBlog [31] are reported to be fast and possess high runtime efficiency, faster than Espresso. MinBlog approach uses a Graph Attention Network (GAT) by

constructing a graph representation of Boolean functions and their subsequent simplification through training of a Machine Learning model that identifies the prime implicants. Our current approach is focused on a pure algebraic approach that will be ideal for classroom teaching, and suitable for further development of formal proofs, theoretical analysis, and delivers a deterministic, unique minimal expression.

We introduced a novel approach to transitioning from symbolic Boolean algebra to a graphical representation in the classical graph-theoretic sense in Section 5, and introduced auxiliary vertices/subgraphs and a camouflaged graph. This approach opens a new possibility of exploring graph-based cryptographic methods or secure circuit representations through obfuscated graphs. We formally introduced the concept of Boolean graph in [34]. Recent publications have inspired the idea of a transition between symbolic algebra and graph theory [7, 30], as well as the construction of a camouflaged graph from [20].

2. Preliminaries

For the ease of representation, we will represent logical symbols $\neg$ (negation) as (complement), $\vee$ (disjunction) as $+$ (addition), $\wedge$ (conjunction) as $\cdot$ (product), and $\equiv$ (equivalence). Further, the logical equivalence ($\equiv$) refers to transformed expressions while maintaining the same behaviour in truth values, while the equality ($=$) denotes exact structural equivalence. A *term* is either a variable or a product of literals, and a literal means a variable or the complement of the variable. Two literals x and $\bar{x}$ are complementary. The number of literals present in a product term is called *literal count*. We know that the *De Morgan's laws* of complement of a conjunction and complement of a disjunction are expressed as: $\bar{x} \cdot y = \bar{x} + \bar{y}$ and $\bar{x} + y = \bar{x} \cdot \bar{y}$, respectively.

- *Disjunctive Normal Form (DNF):* A Boolean expression in DNF is represented as a sum of products (logical OR of logical ANDs). Each term corresponds to a minterm of the Boolean function, evaluating it to be true if any minterm is true. Note that a minterm refers to a product term in which all the variables of a Boolean function appear either as the variable or its complement.
- *Product term:* For a Boolean function $F(x_1, x_2, \dots, x_n)$, a product term is a conjunction of Boolean variables. Examples include x_1x_2 and $\bar{x}_1$. Product terms in a Boolean expression are free from the operation $+$ and De Morgan's laws, such as $\bar{x}_1$ and x_1x_3, but not $\bar{x}_1x_2$ or $x_1 + x_2$.
- *Minimal Disjunctive Normal Form (MDNF):* A Boolean expression $F(x_1, x_2, \dots, x_n)$ is called minimal disjunctive normal form if each of its terms is a product term, separated by logical sum $+$ or $\vee$ if no product is repeated and no product subsumes another. No variable within a product term is repeated. Terms like $x_1 + x_1$ reduce to x_1, meaning $x_1 + x_1 \equiv x_1$.
- *Subsumption:* A relationship between two clauses or terms, where one term subsumes the other. For example, in $abc + abcd$, the product term abc subsumes the other product term $abcd$, hence $abc + abcd \equiv abc$.

- *Saturation Point:* In this step, we reorder and delete subsumed product terms. Arrange the pair of product terms separated by $+$ so that one product subsumes the other. Organize the product terms of a Boolean expression in pairs of the form $(X + Y)$, where X and Y are product terms, and one subsumes the other. We will call the expression $(X + Y)$ a par of product terms X and Y. Specifically, in each pair, one product term subsumes the other if all literals in one term are contained within the other. Create a sum of such pairs across the Boolean expression as $F = (X_1 + Y_1) + (X_2 + Y_2) + \cdots$. In the first pair, we select the product term with the highest literal count as Y and pair it with the product term X with the least literal count that subsumes Y. This ensures that the first pair includes the most complex term (in terms of the number of literals) and the simplest subsuming term. The pairing proceeds similarly for subsequent terms, with each pair consisting of a subsuming relationship between the two terms. Hence, the terms with the highest and most minor literal counts are selected according to the subsumption relationship in each pair. This ordering is beneficial in rapidly eliminating the complex product terms with many literals and consequently contributes to a faster simplification process. Furthermore, we avoid repeating the same product terms when pairing within a current loop.

  For instance, we extract $F_1^{(0)}$ from a Boolean expression $F = ab + ghijklm + abc + abcd + abcde$ and reorder as $F_1^{(0)} = (ab + abcde) + (abc + abcd)$, and the remaining disjoint or independent term here is $F_2^{(0)} = ghijklm$. To generalize this process, we consider the indices of the literal counts for each term within the pair as α_{ij} and β_{ij} such that $\alpha_{ij} \leq \beta_{ij}$ (see Step 0(3)).

3. Main Results

Let $F(x_1, x_2, \ldots, x_n)$ be the given Boolean expression in n variables, and for brevity, we denote it by F. In the extraction process, each loop iteratively breaks F into four distinct components, labeled F_m, G_m, H_m and I_m, where m represents the current loop (m^{th} step of algorithm). The choice or the extraction of F_m, G_m and H_m is independent, and hence they can be computationally processed and simplified in parallel symbolically without going through tabulation. The overall processes of parallel processing and recombination are iterative, finally leading to an optimized form. The initial state of the expression in the zero loops is defined as follows:

Step 0: The initialisation sets up and clarifies each term and prevents redundancy before entering the central iterative reduction.

1. *Extraction:* Define the initial segments of F as follows:
$$F_0 = 0 \equiv F_0'; \; G_0 = 0 \equiv G_0'; \; H_0 = 0 \equiv H_0'; \; I_0 = F \equiv I_0'.$$
2. *Combination:* To reconstruct the initial Form of F, the extracted components are recombined:

$$F \equiv F_0{}' + G_0{}' + H_0{}' + I_0{}'.$$

3. *Saturation Point:* Let $\alpha_{ij} \leq \beta_{ij}$, and set $F = F^{(0)}$. We extract $F_1^{(0)}$ from $F^{(0)}$ as follows:

$$F_1^{(0)} = (x_{11}x_{12}\cdots x_{1\alpha_{11}} + x_{11}x_{12}\cdots x_{1\beta_{11}}) + (x_{21}x_{22}\cdots x_{2\alpha_{12}} + x_{21}x_{22}\cdots x_{2\beta_{12}}) + \cdots + (x_{r_1}x_{r_2}\cdots x_{r\alpha_{1r}} + x_{r1}y_{r2}\cdots x_{r\beta_{1r}}).$$

Here, each x_{ij} represents the j^{th} literal in the i^{th} pair of product terms, which are either variable x_i or its negation $\bar{x}_i$. And each α_{ij} and β_{ij} represent the literal counts within the j^{th} pair of product terms in the current i^{th} loop. Let $F_2^{(0)} = F^{(0)} - F_1^{(0)}$, then the subsequent loop gives:

$$\begin{aligned} F^{(0)} &= F_1^{(0)} + F_2^{(0)} \\ &= (x_{11}x_{12}\cdots x_{1\alpha_{11}} + x_{11}x_{12}\cdots x_{1\beta_{11}}) + (x_{21}x_{22}\cdots x_{2\alpha_{12}} + x_{21}y_{22}\cdots x_{2\beta_{12}}) \\ &\quad + \cdots + (x_{r_1}x_{r_2}\cdots x_{r\alpha_{1r}} + x_{r1}y_{r2}\cdots x_{r\beta_{1r}}) + F_2^{(0)} \\ &\equiv \prod_{j=1}^{\alpha_{11}} x_{1j}\left(1 + \prod_{j=\alpha_{11}+1}^{\beta_{11}} x_{1j}\right) + \cdots + \prod_{j=1}^{\alpha_{1r}} x_{rj}\left(1 + \prod_{j=\alpha_{1r}+1}^{\beta_{1r}} x_{rj}\right) + F_2^{(0)} \\ &\equiv \prod_{j=1}^{\alpha_{11}} x_{1j} + \prod_{j=1}^{\alpha_{12}} x_{2j} + \cdots + \prod_{j=1}^{\alpha_{1r}} x_{rj} + F_2^{(0)}. \end{aligned}$$

After this first round of the subsumption process, new pairs of terms may be generated in which one subsumes the other. Therefore, this process repeats, setting $F^{(0)} = F^{(1)}$, α_{2j} and β_{2j}; $F^{(1)} = F^{(2)}$, α_{3j} and β_{3j} and so on, similarly for subsequent loops. However, after a finite number of loops, we will reach a point where no term subsumes the other because we have only a finite number of terms. Hence, this loop will reach a saturation point as follows:

$$F^{(s_1)} = 0 + F_2^{s_1}$$
$$\Rightarrow F^{(s_1)} = F_2^{s_1}.$$

This shows that F attains its saturation point of the subsumption denoted by $F \equiv F^{(s_1)} = F_2^{(s_1)}$, where s_1 is the subsumption iteration number at 0^{th} step of the algorithm.

Step 1: Decompose the expression $F_2^{(s_1)}$ into F_1, G_1, H_1, and I_1, thereby simplifying the complexity of the extraction process. If $F_1 = 0$, $G_1 = 0$, and $H_1 = 0$, then $I_1{}' \equiv F^{(s_1)} = F_2^{(s_1)}$, indicating the loop saturation. Else, proceed with the further extraction process as follows:

1. *Extract the Boolean expression F_1 from $F_2^{(s_1)}$:* In this step, we order the pair of product terms differ by precisely one complementary variable a_i and $\bar{a}_i$, and subsequently, the complementary variables are eliminated from each pair. The literal a_i is some x_i, and p_i is the literal count of each product term such that $p_i \geq p_{i+1}$.

$$F_1 = \left(a_1 x_{11} \cdots x_{1p_1} + \bar{a}_1 x_{11} \cdots x_{1p_1}\right) + \left(a_2 x_{21} \cdots x_{2p_2} + \bar{a}_2 x_{21} \cdots x_{2p_2}\right) + \cdots + \left(a_\ell x_{\ell 1} \cdots x_{\ell p_\ell} + \bar{a}_\ell x_{\ell 1} \cdots x_{\ell p_\ell}\right).$$

We can write F_1 compactly as:

$$F_1 = \left(a_1 \prod_{j=1}^{p_1} x_{1j} + \bar{a}_1 \prod_{j=1}^{p_1} x_{1j}\right) + \left(a_2 \prod_{j=1}^{p_2} x_{2j} + \bar{a}_2 \prod_{j=1}^{p_2} x_{2j}\right) + \cdots + \left(a_\ell \prod_{j=1}^{p_\ell} x_{\ell j} + \bar{a}_\ell \prod_{j=1}^{p_\ell} x_{\ell j}\right)$$
$$\equiv \prod_{j=1}^{p_1} x_{1j}(a_1 + \overline{a_1}) + \prod_{j=1}^{p_2} x_{2j}(a_2 + \bar{a}_2) + \cdots + \prod_{j=1}^{p_\ell} x_{\ell j}(a_\ell + \bar{a}_\ell)$$
$$\equiv \prod_{j=1}^{p_1} x_{1j} + \prod_{j=1}^{p_2} x_{2j} + \cdots + \prod_{j=1}^{p_\ell} x_{\ell j} = F_1'.$$

The iteration process in this step will repeat until $\ell = 0$ such that $F_1{}' = 0$. For example, the expression $x_1 x_2 \bar{x}_3 + x_1 x_2 x_3 + x_1 \bar{x}_2$ is reduced to $x_1 x_2 + x_1 \bar{x}_2$ in the first iteration. After the second iteration, the expression is reduced to x_1.

2. *Extract the Boolean expression G_1 from $F_2^{(s_1)}$:* In this step, we order the pair of product terms differ by one complementary variable b_i and $\bar{b}_i$, and the remaining literals conjunction with b_i subsumes the remaining literals in conjunction with $\bar{b}_i$. Subsequently, $\bar{b}_i$ is eliminated from each pair.

$$G_1 = (b_1 x_{11} x_{12} \cdots x_{1\alpha_1} + \bar{b}_1 x_{11} x_{12} \cdots x_{1\beta_1}) + (b_2 x_{21} x_{22} \cdots x_{2\alpha_2} + \bar{b}_2 x_{21} x_{22} \cdots x_{2\beta_2}) + \cdots + (b_s x_{s1} x_{s2} \cdots x_{s\alpha_s} + \bar{b}_s x_{s1} x_{s2} \cdots x_{s\beta_s}),$$

where α_i and β_i are non-negative integers, and $\alpha_i < \beta_i;\ 1 \le i \le s$. The Boolean expression G_1 in a compact generalized Form is:

$$G_1 = \left(b_1 \prod_{j=1}^{\alpha_1} x_{1j} + \bar{b}_1 \prod_{j=1}^{\beta_1} x_{1j}\right) + \left(b_2 \prod_{j=1}^{\alpha_2} x_{2j} + \bar{b}_2 \prod_{j=1}^{\beta_2} x_{2j}\right) + \cdots + \left(b_s \prod_{j=1}^{\alpha_s} x_{sj} + \bar{b}_s \prod_{j=1}^{\beta_s} x_{sj}\right)$$
$$\equiv \left(b_1 \prod_{j=1}^{\alpha_1} x_{1j} + \prod_{j=1}^{\beta_1} x_{1j}\right) + \left(b_2 \prod_{j=1}^{\alpha_2} x_{2j} + \prod_{j=1}^{\beta_2} x_{2j}\right) + \cdots + \left(b_s \prod_{j=1}^{\alpha_s} x_{sj} + \prod_{j=1}^{\beta_s} x_{sj}\right) = G_1{}'.$$

The iteration process in this step will repeat until $s = 0$ such that $G_1{}' = 0$.

3. *Extract the Boolean expression H_1 from $F_2^{(s_1)}$:* In this step, we order the pair of product terms differ by one complementary variable c_i and $\bar{c}_i$, and the remaining literals conjunction with $\bar{c}_i$ subsumes the remaining literals in conjunction with c_i. Subsequently, c_i is eliminated from each pair.

$$H_1 = (\bar{c}_1 x_{11} x_{12} \cdots x_{1\gamma_1} + c_1 x_{11} x_{12} \cdots x_{1\delta_1}) + (\bar{c}_2 x_{21} x_{22} \cdots x_{2\gamma_2} + c_2 x_{21} x_{22} \cdots x_{2\delta_2}) + \cdots + (\bar{c}_t x_{t1} x_{t2} \cdots x_{t\gamma_t} + c_t x_{t1} x_{t2} \cdots x_{t\delta_t}),$$

where, γ_i and δ_i are non-negative integers, and $\gamma_i < \delta_i;\ 1 \le i \le t$. The Boolean expression H_1 in a compact generalized Form is:

$$H_1 = \left(\bar{c}_1 \prod_{j=1}^{\gamma_1} x_{1j} + c_1 \prod_{j=1}^{\delta_1} x_{1j}\right) + \left(\bar{c}_2 \prod_{j=1}^{\gamma_2} x_{2j} + c_2 \prod_{j=1}^{\delta_2} x_{2j}\right) + \cdots$$
$$+\left(\bar{c}_t \prod_{j=1}^{\gamma_t} x_{tj} + c_t \prod_{j=1}^{\delta_t} x_{tj}\right)$$
$$\equiv \left(\bar{c}_1 \prod_{j=1}^{\gamma_1} x_{1j} + \prod_{j=1}^{\delta_1} x_{1j}\right) + \left(\bar{c}_2 \prod_{j=1}^{\gamma_2} x_{2j} + \prod_{j=1}^{\delta_2} x_{2j}\right) + \cdots$$
$$+\left(\bar{c}_t \prod_{j=1}^{\gamma_t} x_{tj} + \prod_{j=1}^{\delta_t} x_{tj}\right) = H_1'.$$

The iteration process in this step will repeat until $t = 0$ such that $H_1' = 0$.

4. *Let I_1 be the remaining portion of $F(x_1, x_2, \ldots, x_n)$ after F_1, G_1, and H_1 have been extracted:*

$$I_1 = F_2^{(s_1)} - (F_1 + G_1 + H_1) \equiv I_1'.$$

5. *Continue the extraction and combination process:*

$$F^{(1)} = F_1^{(1)} + F_2^{(1)}. \text{Considering } \gamma_{\text{ij}} \leq \delta_{\text{ij}},$$
$$F^{(1)} = (v_{11}v_{12}\cdots v_{1\gamma_{11}} + v_{11}v_{12}\cdots v_{1\delta_{11}}) +$$
$$\cdots + (v_{\theta 1}v_{\theta 2}\cdots v_{\theta\gamma_{1\theta}} + v_{\theta 1}v_{\theta 2}\cdots v_{\theta\delta_{1\theta}}) + F_2^{(1)}$$
$$F^{(1)} \equiv \prod_{j=1}^{\gamma_{11}} v_{1j}(1 + \prod_{j=\gamma_{11}+1}^{\delta_{11}} v_{1j}) + \cdots + \prod_{j=1}^{\gamma_{1\theta}} v_{\theta j}(1 + \prod_{j=\gamma_{1\theta}+1}^{\delta_{1\theta}} v_{\theta j}) + F_2^{(1)}$$
$$F^{(1)} \equiv \prod_{j=1}^{\gamma_{11}} v_{1j} + \cdots + \prod_{j=1}^{\gamma_{1\theta}} v_{\theta j} + F_2^{(1)}.$$

By similar extraction, and setting $F^{(1)} = F^{(2)}$, we have:

$$F^{(2)} = F_1^{(2)} + F_2^{(2)}$$
$$F^{(2)} = (w_{11}w_{12}\cdots w_{1\gamma_{21}} + w_{11}w_{12}\cdots w_{1\delta_{21}}) +$$
$$\cdots + (w_{\psi 1}w_{\psi 2}\cdots w_{\psi\gamma_{2\psi}} + w_{\psi 1}w_{\psi 2}\cdots w_{\psi\delta_{2\psi}}) + F_2^{(2)}$$
$$F^{(2)} \equiv \prod_{j=1}^{\gamma_{21}} w_{1j}(1 + \prod_{j=\gamma_{21}+1}^{\delta_{21}} w_{1j}) + \cdots + \prod_{j=1}^{\gamma_{2\psi}} w_{\psi j}(1 + \prod_{j=\gamma_{2\psi}+1}^{\delta_{2\psi}} w_{\psi j}) + F_2^{(2)}$$
$$F^{(2)} \equiv \prod_{j=1}^{\gamma_{21}} w_{1j} + \cdots + \prod_{j=1}^{\gamma_{2\psi}} w_{\psi j} + F_2^{(2)}.$$

This iterative process continues until:

$$F^{(s_k)} = 0 + F_2^{s_k}$$
$$F^{(s_k)} = F_2^{s_k}.$$

This shows that F attains its saturation point of the subsumption denoted by $F \equiv F^{(s_k)} = F_2^{(s_k)}$.

If $F_2 = 0$, $G_2 = 0$, and $H_2 = 0$, then $I_2' \equiv F^{(s_k)} = F_2^{(s_k)}$, showing the overall saturation of the loop, which is the required simplest form. Else, we proceed with the subsequent loops similarly until $I_m' \equiv F^{(s)} = F_2^{(s)}$, where s is the number of the subsumption iteration at m^{th} step of the algorithm.

The above algorithm is summarized as follows:

4. Algorithm

1. *Check for Minimal Disjunctive Normal Form (MDNF) :* If $F(x_1, x_2, \dots, x_n)$ is not in MNDF, ensure that each term is a product and apply subsumption iterations at the 0^{th} outer loop to obtain $F \equiv F^{(s_1)} = F_2^{(s_1)}$. Otherwise, proceed to the next step (Step 1) to obtain $F \equiv F^{(s_k)} = F_2^{(s_k)}$.
2. If $F_2 = 0$, $G_2 = 0$, and $H_2 = 0$, then $I_2{}' \equiv F^{(s_k)} = F_2^{(s_k)}$, showing the overall saturation of the loop, which is the required simplest form. Else, we proceed with the subsequent loops similarly until $I_m{}' \equiv F^{(s)} = F_2^{(s)}$, which is the saturation of overall iterations at s^{th} iteration and m^{th} step of the algorithm.

Remark 4.1: In the above algorithms, we have not mentioned the Boolean function F containing the expressions like $x_1x_2 + \bar{x}_1x_3$, where the pair of terms contain the complementary variables and the remaining literals or products are disjoint. However, this case is covered by the subsumption iterations as the subsumption iterations search for all the possible pairs based on subsumption relations. We will pair the term x_1x_2 with a term subsumed by it, and similarly for $\bar{x}_1x_3$. If we have an expression $x_1x_2 + \bar{x}_1x_3 + x_2x_3$, how will subsumption iteration work? Here, the term x_2x_3 can be made to be subsumed by the terms x_1x_2 and $\bar{x}_1x_3$ by the Consensus theorem or Redundancy theorem [1, 4] as follows: $x_1x_2 + \bar{x}_1x_3 + x_2x_3 = x_1x_2 + \bar{x}_1x_3 + x_2x_3(x_1 + \bar{x}_1) = (x_1x_2 + x_1x_2x_3) + (\bar{x}_1x_3 + \bar{x}_1x_2x_3) = x_1x_2 + \bar{x}_1x_3$

Definition 4.1: A Boolean function F said to be in its simplest form if:

1. F is in minimal disjunctive normal form.
2. There is no pair of product terms that differ by exactly one complementary variable a_i and $\bar{a}_i$.
3. There is no pair of product terms differ by complementary variables b_i and $\bar{b}_i$ such that the remaining literals either conjunction with b_i subsumes the remaining literals conjunction with $\bar{b}_i$ or conjunction with $\bar{b}_i$ subsumes the remaining literals conjunction with b_i.

Theorem 4.1: *If $I_{m'} = F_2^{(s)}$, where s is the number of iterations of the subsumption process and m is the number of steps of extraction and combination algorithm, then the Boolean function F on n variables is in its simplest form.*

Proof: Setting $F \equiv F^{(s)}$ at m^{th} step of the algorithm, we have

$$F^{(s)} = F_1^{(s)} + F_2^{(s)}, \tag{1}$$

Where $F_2^{(s)} = F_{m'} + G_{m'} + H_{m'} + I_{m'}$ and $F_1^{(s)}$ is the sub-expression of $F^{(s)}$ containing the subsuming pairs. But $I_{m'} = F_2^{(s)}$ implies that the value of

$F_2^{(s)}$ is solely decided by the value of $I_{m'}$, therefore, the expressions $F_{m'}, G_{m'}$, and $H_{m'}$ are redundant. Thus $F_{m'}, G_{m'}$ and $H_{m'}$ must all be false (0) whenever $F_2^{(s)}$ is true (1) i.e., $F_{m'} = 0, G_{m'} = 0$ and $H_{m'} = 0$. Next, we have the following observations:

1. $F_{m'} = 0$ shows that after the m^{th} step of the iteration, no pair of product terms differ by exactly one complementary variable a_i and $\bar{a}_i$.
2. $G_{m'} = 0$ and $H_{m'} = 0$ shows that there is no pair of product terms differ by complementary variables b_i and $\bar{b}_i$ such that the remaining literals either conjunction with b_i subsumes the remaining literals conjunction with $\bar{b}_i$ or conjunction with $\bar{b}_i$ subsumes the remaining literals conjunction with b_i.
3. We have $F^{(s)} = F_1^{(s)} + F_2^{(s)}$. We wish to show that the value of $F^{(s)}$ is entirely decided by the value of $F_2^{(s)}$ at s, a finite number of subsumption iteration processes. The potential subsuming terms will be decided by the presence of common literals, leading to a finite number of pairings. Thus, the finite number of product terms in F directly implies that there are only a finite number of subsuming pairs of product terms in any Boolean expression with n variables. Thus, the process will reach a saturation point at a finite s^{th} subsumption iteration i.e., $F_1^{(s)} = 0$. Consequently, F attains the minimal disjunctive normal form.

   Hence, the Conditions $1, 2$ and 3 ensure that F is in its simplest form.

Proposition 4.2: For any Boolean expression F, the simplified form $F^{(s)}$ obtained through the extraction and combination method is unique.

The proof is simple as the extraction and combination method involves a systematic sequence of steps where the finite number of literals or product terms are extracted, subsumed, or substantially reduced repeatedly until the saturation point, and all the operations are deterministic. The same initial expression and sequence of steps will always result in the same simplified form.

Theorem 4.3: *Let $\mathcal{R}$ be the set of algebraic minimization rewrite rules for the extraction and combination algorithm, then, for every Boolean expression F, rewriting under $\mathcal{R}$ has the following properties:*

1. *Every transformation is logically equivalent.*
2. *The process terminates in a finite number of steps.*
3. *The irreducible form is unique irrespective of reduction order.*
4. *The irreducible form is a minimal in sum-of-products.*

Proof:

1. Each rewrite rule in $\mathcal{R}$ is derived from a Boolean tautology in step *0(3)*, where the matching pattern of the form $X + Y$, where X and Y are product terms, and one of them subsumes the other. Here, the tautology of the form $\left(1 + \prod_{j=\alpha_{11}+1}^{\beta_{11}} x_{1j}\right), \cdots, \left(1 + \prod_{j=\alpha_{1r}+1}^{\beta_{1r}} x_{rj}\right)$ is iteratively removed.

Further, the extraction of the Boolean expression F_1 from $F_2^{(s_1)}$ in *step 1(1)* removes the contradiction using the matching pattern of the form $a_i + \bar{a}_i$, where a_i and $\bar{a}_i$ are complement to each other. Here, contradiction of the type: $v_1 = (a_1 + \bar{a}_1), v_2 = (a_2 + \bar{a}_2), \cdots, v_l = (a_l + \bar{a}_l)$; are iteratively removed till $l = 0$. Overall steps $0 - 4$ of the algorithm cover idempotence, consensus, and absorption of the types $X + XY = X, X + \bar{X}Y = X + Y$ and $XY + \bar{X}\bar{Y} = Y + \bar{X}$. Hence, every step preserves the logical equivalence of the original expression.

2. Let $M(F)$ be a measure of the total number of literals in all product terms. Each reduction strictly decreases $M(F)$, since the rules either remove redundant terms or simplify products. Since $M(F)$ is a nonnegative integer, infinite descent is impossible. This guarantees termination.

3. If $F \to F_1$ and $F \to F_2$ are two different reductions. Because the two rules preserve equivalence and are directed toward irreducibility, the two resultant expressions are logically equivalent. By Church-Rosser theorem [10], there is a common expression F^* derivable from both F_1 and F_2. All reduction paths therefore point to a unique irreducible normal form.
4. Upon termination, each remaining product term is a prime implicant: no term can be further reduced, subsumed, or eliminated. Further, all redundant prime implicants have been eliminated, and thus the final expression is an irredundant prime cover. The normal form is therefore a minimal sum-of-products representation.

With all these properties, the algorithm follows the rewrite system $\mathcal{R}$ proving the correctness of the Boolean minimization procedure.

While it may be too soon to say this method is better than current state of art methods, it presents a new and easy approach. It provides new ways for visualization, algebraic reasoning, and logical obscuring, rather than quick efficiency improvements. Therefore, the proposed comparison of our method with existing methods in Table 1 should be viewed as intuitive and exploratory. It demonstrates the new interpretive and structural aspects of the algebraic-graph method, creating a new transitional avenue in symbolic logic and classical graph theory.

Table 1

A brief exploratory comparison of the proposed method with some classical methods

Method	Approach	Strengths	Limitations
Karnaugh Map (K-Map)	Graphical representation using grids of 1s and 0s to group adjacent minterms.	Visual and effectively fast for up to $4 - 6$ variables, and guarantees minimal SOP.	Impractical beyond 6 variables due to exponential cell growth, and manual grouping not automatable for large maps.

Quine–McCluskey (Q–M)	Tabular method of finding all prime implicants by group minterms, constructing prime implicant table or chart to select minimal cover.	Systematic, algorithmic and exact-guarantying minimal form; improvement over K-map in handling more variables effectively.	High computational complexity-grows exponentially with increase in variable; infeasible for large variable counts ($n12$).
Espresso (Heuristic Minimizer)	Iterative heuristic two-level cover minimization-avoiding exhaustive search over assuring absolute optimality; takes expansion, reduction, irredundant stages	Produces a minimal or near minimal set of product terms; efficiently handles large-scale circuits; compatible with don't-cares; extensively used in CAD/EDA tools; scalable to hundreds of variables.	Greedy approach to find good solutions quickly that may not guarantee absolute minimum; depends on initialization and heuristics; some sequential dependencies limit parallelism.
Shannon Expansion	Recursive decomposition using identity: $f = xf_{x=1} + \bar{x}f_{x=0}$, where the original function f is represented as combination of simpler functions $xf_{x=1}$ and $\bar{x}f_{x=0}$.	Foundation for recursive algorithms, basis for BDDs (Binary Decision Diagrams), and SAT solving; decomposes problem systematically and well suited for parallel processing.	Exponential recursion in the worst case; efficiency relies on variable ordering; not directly a minimization algorithm, but a decomposition technique, hence no guarantee of minimum circuit implementation, and not suitable for logic design problems.
Proposed Symbolic Algebraic Method (this work)	Truth-table-free rewrite system using Boolean axioms with parallel simplification of extracted subexpressions. Pure algebraic manipulation and visual representation of Boolean expression in classical graph theoretic setting.	Deterministic and unique minimal SOP; inherently parallelizable; avoids exponential truth-table enumeration; creates new avenue for seamless transition between symbolic Boolean algebra and classical graph theory; creating camouflaged or obfuscated graphs-setting foundation for developing graph-based cryptographic methods or secure circuit representations.	Requires experimental validation against the state of art methods; worst-case combinatorial blow-up possible if not efficiently implemented.

5. Construction of Boolean Graph

By the *Boolean graph*, we mean a simple graph representing a Boolean expression. In our work on Boolean graphs, we utilize concepts similar to those implemented in computational tools like Wolfram Research's *BooleanGraph* function, which allows for the representation and manipulation of Boolean expressions as graph structures [36].

5.1 *Determination and Removal of Auxiliary Vertices in Boolean Graph Induced by OR* $(+)$

The auxiliary vertex v is an additional vertex to represent Boolean operations in graph form. A vertex in a Boolean graph may be a single *literal* or a *term* in a Boolean expression. This auxiliary vertex plays a crucial role in distinguishing between the OR $(+)$ and AND $(\cdot)$ operations in visual form. This vertex allows for a clear visual and structural differentiation between the AND and OR operations.

- *AND Operation $(\cdot)$:* Direct edge between vertices indicates both conditions must be met.
- *OR Operation $(+)$:* Auxiliary vertex v connects to either of the vertices, indicating at least one condition must be met.

The auxiliary vertex ensures that the graph representation of Boolean expressions is unambiguous. It helps to visually differentiate between the strict condition of AND and the loose condition of OR. Consider the Boolean expressions $x \cdot y$ and $x + y$ and their graph representations as follows.

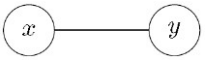


Figure 1
A simple graph representing the Boolean expression $x \cdot y$

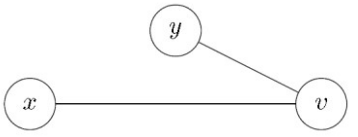


Figure 2
A simple graph representing the Boolean expression $x + y$

The Boolean expression $(x \cdot y)$ is an edge with end vertices x and y, and the expression $x + y$ represents the presence of at least one edge involving x or y as shown in Figures 1 and 2, respectively.

Consider another Boolean expression $\bar{x}_1 \cdot (x_2 + x_3)$ and $\bar{x}_1 \cdot x_2 + x_3$ and their graph representation as follows.

In Figure 4, x_3 is connected to $\bar{x}_1$ and x_2 through an auxiliary vertex v. In general, the auxiliary vertex v induced by OR $(+)$ will always be a tautology or evaluate to true (1), which does not change the logical value of the original expression when combined with the expression by the AND operation $(\cdot)$. For example, in the Figure 4, by considering the auxiliary vertex v as a symbolic representation of tautology, we have the following equivalence expression:

$$\begin{aligned}\bar{x}_1 \cdot x_2 + x_3 &= v \cdot \bar{x}_1 \cdot v \cdot x_2 + v \cdot x_3 \\ &= v \cdot (\bar{x}_1 \cdot x_2) + (v \cdot x_3) \\ &= v \cdot (\bar{x}_1 \cdot x_2 + x_3).\end{aligned}$$

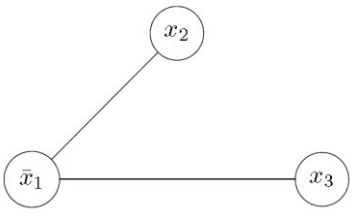


Figure 3
A simple graph of $\bar{x}_1 \cdot (x_2 + x_3)$

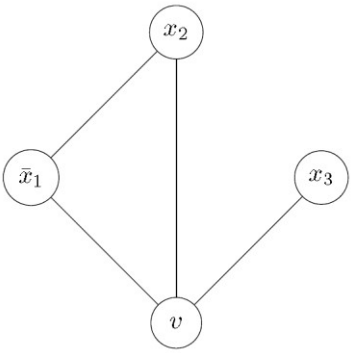


Figure 4
A simple graph of $\bar{x}_1 \cdot x_2 + x_3$.

An auxiliary vertex is a literal or a term in the basic atomic canonical form as follows:

A: $1 + Z$, where Z is a sub-expression of the subsumed term X or Y of $X + Y$. For example, the expression $abc + abcde$ can be written as $abc + abcde = abc(1 + de)$. Here, the auxiliary vertex is $1 + de$ and $Z = de$ from the subsumed term $abcde$.

B: $a + \bar{a}$, where a and $\bar{a}$ are literal singular symbols that are complement of each other.

Since the auxiliary vertices are tautologies, all its other forms eventually collapse to either form A or form B. A fixed constant truth value 1 is called a trivial auxiliary vertex. Therefore, a Boolean graph representing a minimal disjunctive normal form having no pair of product terms differing by precisely one complementary variable a_i and $\bar{a}_i$ has no auxiliary vertices (except, trivially 1), and each minterm of such a Boolean expression will be a **clique** (the proof is trivial due to commutative property of AND (·)). Therefore, the process of identification and removal of auxiliary vertices leads to the determination of all possible cliques in a Boolean graph. Suppose G_1 and G_2 are two simplified Boolean graphs, then the product $G_1 \cdot G_2$ gives the sum of common cliques of G_1 and G_2, representing the common minterms of the Boolean expression corresponding to G_1 and G_2. Furthermore, a Boolean graph representing a simplified or minimized Boolean expression will contain only trivial auxiliary vertices, i.e., 1 or 0, hence in Figure 4, the auxiliary vertex $v \equiv 1$. Similarly, Figure 3 is a representation of a simplified Boolean expression; we have just omitted the trivial auxiliary vertex 1 connecting x_1, x_2 and x_3 as $1 \cdot \{\bar{x}_1 \cdot (x_2 + x_3)\} \equiv \bar{x}_1 \cdot (x_2 + x_3)$.

1. *Saturation Point* in step *0(3)* is a process of determining and removing the auxiliary vertices using the matching pattern of the form $X + Y$, where X and Y are product terms, and one of them subsumes the other. Here, the auxiliary vertices are of the form:
   $\left(1 + \prod_{j=\alpha_{11}+1}^{\beta_{11}} x_{1j}\right), \cdots, \left(1 + \prod_{j=\alpha_{1r}+1}^{\beta_{1r}} x_{rj}\right)$. The Boolean graph corresponding to the saturated form of the expression will be free from the auxiliary vertices of this form.
2. *Extracting the Boolean expression* F_1 *from* $F_2^{(s_1)}$ *in step 1(1)* is a technique to determine and remove the auxiliary vertices using the matching pattern of the form $a_i + \bar{a}_i$, where a_i and $\bar{a}_i$ are complement to each other. Here, the auxiliary vertices are of the form: $v_1 = (a_1 + \bar{a}_1), v_2 = (a_2 + \bar{a}_2), \cdots, v_l = (a_l + \bar{a}_l)$; if $l = 0$, then there is no auxiliary vertices of this form.

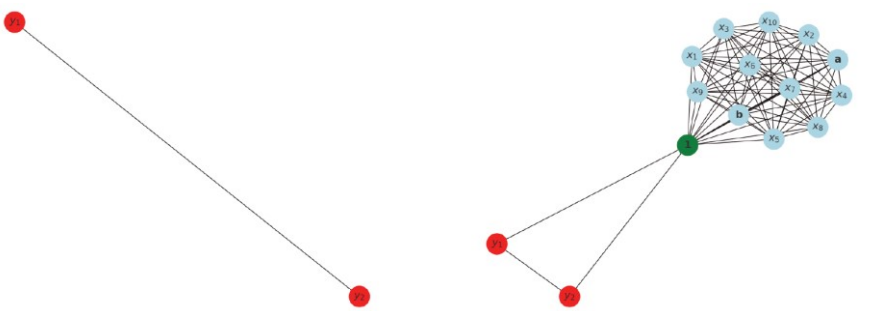
The Boolean graph corresponding to the simplified Boolean expression is called the simplified Boolean graph, denoted by G_B. Hence, G_B does not contain any non-trivial auxiliary vertices.

5.2 *Auxiliary Vertices Induced by AND* $(\cdot)$

The auxiliary vertices induced by AND $(\cdot)$ in a Boolean graph are contradictions of type $v \cdot \bar{v}$. Since adding a literal $v \cdot \bar{v}$ to any term in a Boolean expression will not change its logical value, the auxiliary vertices induced by $(\cdot)$ can be added to any vertex in a Boolean graph. The symbol 0 is a trivial auxiliary vertex in this case.

Suppose literals a and b complement each other such that $a \cdot b = 0$, then a simple expression $y_1 \cdot y_2$ representing an edge with vertices y_1 and y_2 can be camouflaged as complicated by the addition of auxiliary vertices such as $a \cdot b \cdot (x_1 \cdot x_2 \cdots x_n)$. The following is an illustration with $n = 10$.

$$\begin{aligned} y_1 \cdot y_2 &\equiv y_1 \cdot y_2 + 0 \equiv y_1 \cdot y_2 + a \cdot b \equiv y_1 \cdot y_2 + a \cdot b \cdot (x_1 \cdot x_2 \cdots x_{10}) \\ &\equiv 1 \cdot (y_1 \cdot y_2) + 1 \cdot \{(a \cdot b) \cdot (x_1 \cdot x_2 \cdots x_{10})\} \\ &\equiv 1 \cdot \{y_1 \cdot y_2 + a \cdot b \cdot (x_1 \cdot x_2 \cdots x_{10})\} \end{aligned}$$

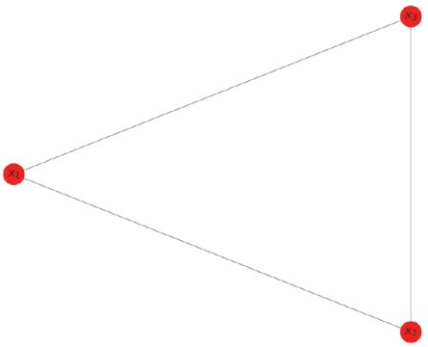


(a) Graph of $y_1 \cdot y_2$ (b) Graph of $1 \cdot \{y_1 \cdot y_2 + a \cdot b \cdot (x_1 \cdot x_2 \cdots x_{10})\}$

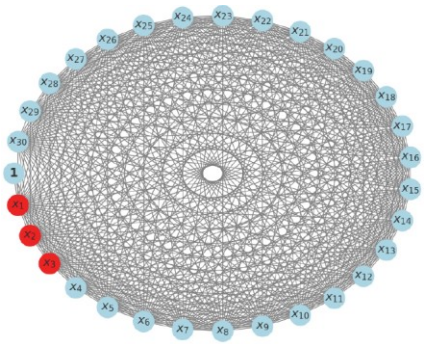
Figure 5
Graphs $y_1 \cdot y_2$ and $1 \cdot \{y_1 \cdot y_2 + a \cdot b \cdot (x_1 \cdot x_2 \cdots x_{10})\}$ are logically equivalent

In a complete Boolean graph, if a literal a and its complement b are its vertices, then the complete graph is equivalent to $1 \cdot 0$-an edge with trivial auxiliary vertices 1 and 0.

Definition 5.1: In a Boolean graph $G(F)$ of a Boolean function F, if an auxiliary vertex v is a vertex of a clique C, then C will be called an auxiliary clique (AC) of $G(F)$. The largest possible clique with v as one of its vertices will be called the maximal auxiliary clique (MAC) of $G(F)$. Any other subgraphs representing a tautology or a contradiction will be called auxiliary subgraphs (AS). For example, the graph $1 \cdot 0$ with distinct vertices 1 and 0 is a trivial auxiliary subgraph of every Boolean graph. A Boolean graph containing a non-trivial auxiliary vertex or an auxiliary subgraph will be called **camouflaged graph**.



(a) Graph of $x_1 \cdot x_2 \cdot x_3$



(b) Graph of $x_1 \cdot x_2 \cdot x_3 \cdot (1 + x_4 \cdot x_5 \cdots x_{30})$

Figure 6
Graphs $x_1 \cdot x_2 \cdot x_3 \cdot (1 + x_4 \cdot x_5 \cdots x_{30})$ is a camouflaged graph of $x_1 \cdot x_2 \cdot x_3$

The graph in Figure 6b is a camouflaged graph of the graph in Figure 6a, they are logically equivalent graphs. Similarly, the graph in Figure 5b is a camouflaged graph of the graph in Figure 5a. For instance, $\prod_{j=1}^{\alpha_{11}} x_{1j}\left(1 + \prod_{j=\alpha_{11}+1}^{\beta_{11}} x_{1j}\right) + \cdots + \prod_{j=1}^{\alpha_{1r}} x_{rj}\left(1 + \prod_{j=\alpha_{1r}+1}^{\beta_{1r}} x_{rj}\right) + F_2^{(0)}$ can be a very

complex as a camouflaged graph of its simpler form $\prod_{j=1}^{\alpha_{11}} x_{1j} + \prod_{j=1}^{\alpha_{12}} x_{2j} + \cdots + \prod_{j=1}^{\alpha_{1r}} x_{rj} + F_2^{(0)}$.

5.3 *Complement of a simplified Boolean Graph*

Is the complement of a simplified Boolean graph equivalent to the usual graph complement? In this section, we will define the simplified Boolean graph complement with an illustrative example, and see that the simplified Boolean complement and the usual graph complements are the same. However, the hidden technique for obtaining a simplified Boolean graph complement differs from determining the usual graph complements. Consider a simplified Boolean graph $G(F_{\min})$ with vertices x_1, x_2, x_3, and x_4 and edges representing the simplified Boolean expression $I = x_1 + (x_2 \cdot x_3) + (x_2 \cdot x_4)$. Figure 7 represents $I = x_1 + (x_2 \cdot x_3) + (x_2 \cdot x_4)$ and Figure 8 is the usual graph complement of the Boolean graph in Figure 7.

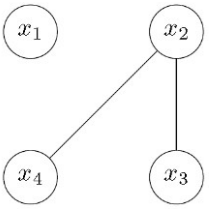


Figure 7
Boolean Graph *I*

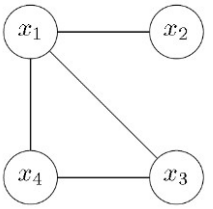


Figure 8
Usual Graph Complement $\bar{I}$

The Boolean complement of this graph expression is:

$$\bar{I} = \overline{x_1 + (x_2 \cdot x_3) + (x_2 \cdot x_4)}$$

Using De Morgan's laws, we get:

$$\bar{I} = \bar{x}_1 \cdot (\bar{x}_2 + \bar{x}_3) \cdot (\bar{x}_2 + \bar{x}_4)$$

Breaking it down further:

$$\bar{I} = (\bar{x}_1 \cdot \bar{x}_2) + (\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_4) + (\bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_2) + (\bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_4)$$

The corresponding complement graph is drawn in Figure 9 as follows:

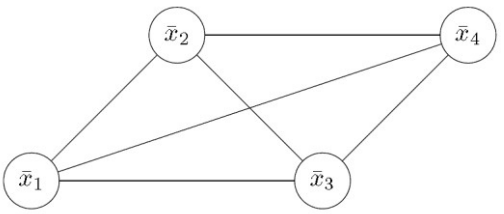


Figure 9
Complement Graph $\bar{I}$

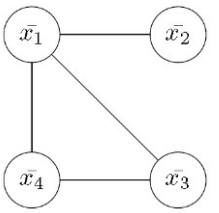


Figure 10
Simplified $\bar{I}$

We observe that the complement of a simplified Boolean graph differs from the conventional graph complement. However, this graph is just an intermediate graph as the expression $\bar{I} = (\bar{x}_1 \cdot \bar{x}_2) + (\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_4) + (\bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_2) + (\bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_4)$ is not in its simplest form. The simplest form of $\bar{I}$ is $\bar{x}_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_4$. Therefore, the corresponding Boolean complement graph in simplified form is given by Figure 10.

Replacing each Boolean variable in the simplified $\bar{I}$ by its complement, we get the Boolean complement graph of I, which is the same as the usual graph complement $\bar{I}$. By Boolean graph complement, we mean the complement of the graph without non-trivial auxiliary vertices or auxiliary subgraphs. In other words, we discuss only the complements of a simplified Boolean graph. Enumerating the complement of a camouflaged graph is computationally an unnecessary burden; instead, the camouflaged graph must first be reduced to its simplified form by removing the auxiliary vertices or auxiliary subgraphs before applying the complement rule.

Theorem 5.1: *If F is a Boolean expression in variables $x_1, \ldots, x_n$ and $G(F)$ is its Boolean graph, then the graph complement $\bar{G}(F)$ is isomorphic to $G(\bar{F})$-the Boolean graph of the complement expression $\bar{F}$, up to replacing every vertex by its complement (i.e. $x \mapsto \bar{x}$ and $\bar{x} \mapsto x$).*

Proof: Consider a vertex x representing a variable in $G(F)$. In the Boolean complement $\bar{F}$, the role of x is taken by $\bar{x}$, so under the mapping $x \mapsto \bar{x}$ (and $\bar{x} \mapsto x$), every vertex is replaced by its complement.

In $G(F)$, an edge between any pair of literals x_1 and x_2 means $x_1 \cdot x_2$ appears in F. In $\bar{G}(F)$, two vertices are connected when they were not connected in $G(F)$. But this is exactly what happens in $\bar{F}$: a conjunction that does not satisfy F will satisfy $\bar{F}$.

Thus, after relabeling each vertex by its complement, the edges of $\bar{G}(F)$ match exactly the edges of $G(\bar{F})$. Therefore, $\bar{G}(F) \cong G(\bar{F})$.

6. Conclusion and Future Direction

Extraction and Combination is a systematic and parallel approach to minimize Boolean expressions through algebraic manipulations. The method is designed to simplify complex Boolean expressions by breaking into four sub-expressions of predefined rules so that those sub-expressions can be easily searched and simplified individually and recombined. The overall iteration repeats until the extraction process exhausts, creating a unique, simplified form. This systematic and predictable approach is easy to use, and visualizes the Boolean algebraic simplification process. It is potentially an excellent educational tool for teaching digital logic minimization, providing practical knowledge of logic reduction techniques, and visualizing the sequence of logical operations. On further refinement, this method can also be highly beneficial in AI for optimizing logical decision processes that rely on complex Boolean logic and rule-based systems.

Our proposed symbolic approach provides an additional conceptual advantage beyond conventional Boolean minimization and such a transformation also opens the possibility of constructing camouflaged or obfuscated graphs, which can conceal logical structure while preserving functional equivalence. These features open a potential basis for developing graph-based cryptographic methods or secure circuit representations, especially where symbolic expressions and their corresponding graphs exist as two forms of the same logical system. Particularly, we will present a conceptual framework for a *Camouflaged-Graph Cryptosystem (CGC)*, building on the symbolic Boolean to graph representation created in this work. Confluent rewrite rules of Theorem 4.3/algebraic minimization or similar algorithms may be used during the decryption process to restore the graph's canonical form after a cryptographic obfuscation or a camouflaged Boolean graph has been created during encryption.

Acknowledgment

This research was supported by the Science and Engineering Research Board (SERB) under TARE Project File No. TAR/2022/000333. It was also partially supported under the Strengthening Component of the Star College Programme to Jawaharlal Nehru College, Pasighat, East Siang District, Arunachal Pradesh, Administrative Order No. HRD-11011/17/2023-HRD-DBT.

7. Conflict of Interest

The authors declare no conflict of interest.

8. Author's Contributions

The authors declare that they have all made equal contributions.

9. Availability of data and materials

There are no data and materials associated with this manuscript.

References

[1] V. D. Agrawal, ELEC 2200-002 Digital Logic Circuits – Logic Minimization (Chapter 3), Lecture Slides, Auburn University (2014).

[2] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516 (1978).

[3] E. Alharbi, "Truth graph: A novel method for minimizing Boolean algebra expressions by using graphs," in Diagrammatic Representation and Inference, Lecture Notes in Computer Science, vol. 12169, A. V. Pietarinen et al., Eds. Springer, pp. 8–36 (2020).

[4] A. Blake, Canonical Expressions in Boolean Algebra. Chicago, USA: University of Chicago Press (1937).

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, "Logic minimization algorithms for VLSI synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 3, no. 1, pp. 2–13 (1984).

[6] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691 (1986).

[7] A. Bustamante, G. Umbrey, B. Moyong, and B. K. Sarma, "A Zykov algebra approach to clique propagation: Classifying clique complexes in graphs," *Discrete Mathematics, Algorithms and Applications*, vol. 17, no. 6, pp. 1–32 (2025).

[8] S. H. N. Cheng and R. de Wolf, "The subsumption theorem in inductive logic programming: Facts and fallacies," in Proc. Int. Conf. on Inductive Logic Programming (ILP), Leuven, pp. 147–160 (1995).

[9] A. B. Chowdhury, T. Benjamin, M. Romanelli, R. Karri, and S. Garg, "Retrieval-guided reinforcement learning for Boolean circuit minimization," in Proc. Int. Conf. on Learning Representations (ICLR) (2024).

[10] A. Church and J. B. Rosser, "Some properties of conversion," *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482 (1936).

[11] D. Das and S. Mandal, "An efficient algorithm for the minimization of large binary decision diagrams," *Computers & Mathematics with Applications*, vol. 66, no. 5, pp. 779–789 (2013).

[12] C. Eduardo and J. Levy, "General Boolean formula minimization with QBF solvers," in Artificial Intelligence Research and Development. IOS Press, pp. 347–358 (2023).

[13] W. I. Fletcher, An Engineering Approach to Digital Design. Englewood Cliffs, NJ, USA: Prentice Hall (1980).

[14] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, USA: Freeman (1979).

[15] A. Ghosh, "Multi-objective genetic algorithm for efficient FPGA implementation of reversible circuits," *Integration, the VLSI Journal*, vol. 75, pp. 61–70 (2020).

[16] M. Haghparast and S. Mohammadi, "An efficient quantum-inspired genetic algorithm for Boolean function minimization," *Computers & Electrical Engineering*, vol. 59, pp. 1–14 (2017).

[17] S. L. Harris and D. Harris, "Combinational logic design," in Digital Design and Computer Architecture. *Morgan Kaufmann*, pp. 52–104 (2022).

[18] J. P. Hayes, Introduction to Digital Logic Design. Boca Raton, FL, USA: CRC Press (2020).

[19] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers*, Part I, vol. 72, no. 5, pp. 593–599 (1953).

[20] P. Kadam, "Financial fraud detection using jump-attentive graph neural networks," in Proc. Int. Conf. on Machine Learning and Applications (ICMLA), *IEEE*, pp. 628–635 (2024).

[21] A. A. Kadhim, "A Boolean approach to the study of system reliability," *AIP Conference Proceedings,* vol. 2457, no. 1, Art. no. 020002 (2023).

[22] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal,* vol. 38, pp. 985–999 (1959).

[23] L. Ma, "Truth graph method: A handy method different from that of Lesniewski's," *Studies in Logic, Grammar and Rhetoric*, vol. 42, no. 55, pp. 101–115 (2015).

[24] M. Manna and S. Roy, "A novel heuristic algorithm for efficient minimization of large Boolean functions," *Applied Intelligence*, vol. 49, no. 7, pp. 2355–2373 (2019).

[25] W. V. O. Quine, "Simplification of Boolean functions," in Digital Computers: Advanced Coding Techniques (1952).

[26] S. Rudeanu, Boolean Functions and Equations. Amsterdam, Netherlands: North-Holland (1974).

[27] S. Rudeanu, "Algebraic methods versus map methods of solving Boolean equations," *International Journal of Computer Mathematics*, vol. 80, no. 7, pp. 815–817 (2003).

[28] A. M. Rushdi, "A comparison of algebraic and map methods for solving general Boolean equations," *Journal of Qassim University: Engineering and Computer Sciences*, vol. 5, no. 2, pp. 147–173 (2012).

[29] A. M. Rushdi, "Improved variable-entered Karnaugh-map procedures," *Computers & Electrical Engineering*, vol. 13, no. 1, pp. 41–52 (1987).

[30] B. K. Sarma, G. Umbrey, and M. K. Enduri, "The algebra of simple graphs and maximal cliques," *Mathematical Forum*, vol. 33 (2025).

[31] P. Sengupta, A. Tyagi, J. Hu, V. K. Rajan, H. Mostafa, and S. Majumdar, "MinBLoG: Minimization of Boolean logic functions using graph attention network," in Proc. ACM/IEEE Int. Symp. on Machine Learning for CAD (MLCAD) (2024).

[32] C. Senthilpari, K. Diwakar, K. Munusamy, and J. S. Francisca, "Layout parameter analysis in Shannon expansion theorem based on 32-bit adder circuit," *Engineering Science and Technology, an International Journal*, vol. 20, no. 1, pp. 35–40 (2017).

[33] A. S. Singh, "Minimization of Boolean function using K-map," Electrically4U (2021).

[34] G. Umbrey, B. K. Sarma, S. Rahman and A. Bustamante, "Boolean algebra and lattice of graphs with their applications in cryptography," *Discrete Mathematics, Algorithms and Applications*, (in press) (2026).

[35] V. M. E. Van Valkenburg, Logic and Computer Design Fundamentals. Pearson (2014).

[36] Wolfram Research, "BooleanGraph," Wolfram Language Documentation (2010).

Received October, 2025