Journal of Engineering and Sustainable Development
https://jeasd.uomustansiriyah.edu.iq/index.php/jeasd
https://doi.org/10.31272/jeasd.27.3.2

Vol. 27, No. 03, May 2023
ISSN 2520-0917

Original Research

RECURSIVE TERNARY-BASED ALGORITHM FOR COMPUTING PRIME IMPLICANTS OF MULTI-OUTPUT BOOLEAN FUNCTIONS

Zaid Al-Wardi[1*], Osama Al-Wardi[2]

[1] *Electrical Engineering Department, College of Engineering, Mustansiriyah University, Baghdad, Iraq*
[2] *Brandenburg University of Technology, Cottbus-Senftenberg, Germany*

Received 16/3/2023 ***Accepted in revised form 24/4/2023*** ***Published 1/5/2023***

Abstract: The problem of computing the set of prime implicants to represent a Boolean function is a classical problem that is still considered a running problem for research because all known approaches have limitations. The article reviews existing methods for computing prime implicants and highlights their limitations, particularly for multi-output functions and limited scalability due to the growth in memory required to complete the computation. Then it proposes a recursive ternary-based minimization algorithm to compute the prime implicants of multi-output Boolean functions. The algorithm is based on the concept of Programmable Logic Array (PLA) tables, which provide a structured and efficient representation of Boolean functions. The algorithm takes advantage of the ternary logic system to efficiently compute the prime implicants while maintaining scalability for large and complex functions, which has significant implications for digital circuit design and optimization.

Keywords: *Boolean Algebra; sum of products; two-level Boolean representation; Logic Circuit Synthesis*

1. Introduction

Boolean functions are mathematical expressions that take one or more binary inputs and produce a single or multiple-bit output. The output value of a Boolean function depends on the values of its input variables and the logical operations applied to them. Boolean functions are widely used in digital design, computer science, and other fields that involve binary logic [1-3].

A prime implicant of a single-output Boolean function is a product term (a logical AND of one or more input variables) that covers the function and cannot be further reduced by removing any input variables. In other words, a prime implicant is a minimal product term that is necessary to represent the function in a two-level sum-of-products SOP format [4].

Computing prime implicants is an important step in digital circuit design flow because they can be used to simplify the circuit and reduce its complexity. By identifying the prime implicants of a Boolean function, one can find the smallest set of product terms that can represent the function and use it to design simpler, faster, more efficient circuits, and lower cost [5,6].

There are several existing methods for computing prime implicants of a Boolean function. A widely used method to find all possible prime implicants is the Quine-McCluskey method, which performs pairwise comparisons of adjacent minterms in a table. The method is efficient for small functions but the complexity of this approach grows exponentially and becomes very expensive computationally for large functions [7,8].

*Corresponding Author: *alwardi@uomustansiriyah.edu.iq*

On the other hand, Petrick's method involves converting the function into a set of equations and solving them using matrix operations. The method is more efficient than the Quine-McCluskey method for large functions but can be complex and time-consuming for highly variable functions [9].

Existing exact methods can be computationally expensive for large and highly variable functions [10-12], making them impractical for use in some applications, therefore approximate and heuristic algorithms have been proposed to tackle the problem of prime implicants computation scalability. Heuristic approaches including genetic algorithms, simulated annealing, and Tabu search involve randomly generating candidate solutions and iteratively refining them based on fitness functions [13-17]. A heuristic method called the Espresso algorithm is proposed to minimize the Boolean function by iteratively selecting prime implicants that cover the most minterms. This algorithm is efficient and widely used in practice but sometimes it results in suboptimal solutions [18,19].

A mathematical approach to minimize Boolean functions using ternary representation to identify the candidate terms for minimization was proposed in [20]. The problem associated with this approach is the high memory requirement associated with the computation, due to the iterative computation style suggested in the algorithm, which limits the scalability of this approach [20]. A recursive style is considered in this article to tackle the problem.

However, these existing methods have certain limitations that can affect their effectiveness in computing prime implicants, such as computational complexity, inefficient use of resources, limited scope, and suboptimal solutions. Overall, the existing methods for computing prime implicants have their strengths and limitations, and researchers are constantly exploring new algorithms and techniques to address these limitations and improve the efficiency and accuracy of prime implicant computation.

The research problem addressed in this work is the computation of prime implicants for multi-output Boolean functions, which are functions with multiple binary outputs. While existing methods for computing prime implicants are well-established for single-output Boolean functions, they may not be directly applicable to multi-output functions, which require a different approach.

This study aims to present a novel recursive algorithm based on ternary logic for computing prime implicants of multi-output Boolean functions. The proposed algorithm is intended to be highly efficient and precise for functions with any number of outputs. Moreover, it generates a comprehensive set of prime implicants that can be effectively utilized for circuit optimization and design purposes.

The article is organized as follows: Section 2 presents the preliminaries to ensure self-containment. Section 3, The Proposed Minimization Algorithm, illustrates the proposed method and outlines the recursive computation of prime implicants using the ternary-based minimization algorithm. In Section 4, the proposed algorithm is discussed, highlighting its distinctive characteristics compared to existing methods. Finally, Section 5 concludes the article.

2. Background

The objective of this section is to provide readers with a comprehensive understanding of Boolean function representation. Specifically, it covers the basics of two-level SOP form representation and the widely adopted PLA table representation of multi-output Boolean functions. This information is essential for readers to grasp the content of the article, ensuring that it is self-contained.

2.1. Minterms and truth-tables

A truth table of a Boolean function enumerates function values at all the points of the domain. The points of the domain are all possible permutations of binary-valued inputs. A total of $M = 2^N$ permutations, where N is the number of inputs (function's binary arguments), see e.g. Table 1. This exponential complexity doubles the size of a truth table with each new input variable. A specific permutation of inputs is called a minterm, which is a Boolean conjunction, i.e. Boolean AND operation between all input variables each with the corresponding polarity, i.e. either an inverted or non-inverted literal in the product term. A minterm can be numerically represented by the permutation that is equal to the weighted binary sum of input bit values, i.e. $m_{\boldsymbol{X}} = \sum_{i=0}^{N-1} x_i * 2^i$, where X is a specific permutation (minterm) of the N-bit input vector. For example, the minterm $m_{11} = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = m_{(1011)_2}$ represents the Boolean expression $(x_3.x_2'.x_1.x_0)$. Another example is the minterm $m_9 = m_{(1001)_2}$ is equivalent to the Boolean expression $(x_3.x_2'.x_1'.x_0)$. This exponential growth in complexity makes it practically impossible to use this representation for functions with a large number of inputs. The SOP representation of a function is the disjunction of all minterms.

Table 1. Truth-Table of Boolean function ***f(X)***

m	*x₃*	*x₂*	*x₁*	*x₀*	*f₁*	*f₀*
0	0	0	0	0	1	1
1	0	0	0	1	1	1
2	0	0	1	0	0	0
3	0	0	1	1	0	0
4	0	1	0	0	1	1
5	0	1	0	1	1	1
6	0	1	1	0	1	0
7	0	1	1	1	1	0
8	1	0	0	0	0	0
9	1	0	0	1	1	1
10	1	0	1	0	0	0
11	1	0	1	1	1	1
12	1	1	0	0	1	0
13	1	1	0	1	1	0
14	1	1	1	0	1	0
15	1	1	1	1	1	0

2.2. Cubes and PLA Tables

The minterms m_{11} and m_9 in function f demonstrated above are called adjacent minterms, which means that they have exactly one bit-difference in their expressions, i.e. (1011) and (1001). One possible Boolean expression reduction is by extracting the common part of two adjacent minterms and replacing these two minterms with one *cube*, which is only the common part of the two minterms. For example, the two adjacent minterms m_{11} and m_9 are replaced by one cube $(10-1)$. which is equivalent to the Boolean product term $(x_3.x'_2.x_0)$. without the literal x_1 because it is eliminated from the conjunction. To preserve the position of the eliminated literally in the expression, the character '-' in a cube replaces either 0 or 1. One can achieve the additional reduction by extending the same approach to adjacent pairs of cubes. A larger cube can cover more points in the domain by doing so. For example, the cube (-0-0) can be obtained by combining the adjacent cubes (00-0) and (10-0) or the cubes (-000) and

(-010). The cubes that cannot be further reduced are known as the prime implicants.

A Programmable Logic Array (PLA) is a digital circuit design that allows the creation of complex logic functions by combining simpler logic functions. A PLA table is a compact representation of a Boolean function using a truth table that includes the input and output values for the function. The PLA table is structured as a two-dimensional array with a set of input columns, a set of output columns, and a set of product terms that map the inputs to the outputs.

In a PLA table, each row corresponds to a unique combination of input values, and each column corresponds to a Boolean function that maps the inputs to a single output. The product terms represent the logic operations that are performed on the input variables to generate the output variables. The product terms are typically represented as a string of 0s, 1s, and don't care symbols, where each symbol corresponds to the state of a particular input variable [21].

```
.i 4
.o 2
.p 5
.ilb x3 x2 x1 x0
.ob f1 f2
-1--  10
1-11  10
-001  11
10-1  01
0-0-  11
.e
```

Figure 1. A simple example of a PLA Table file

The sum-of-products SOP representation of a Boolean function is the disjunction of its prime implicants such that it covers all points in the domain. PLA table representation of a multiple-output Boolean function is based on this minimized two-level SOP representation of Boolean functions, see e.g. Figure 1. The entities of this table are the prime implicants of the two-output function f. In the left column array in the table, we find the list of the cubes that form the function, while the right column contains the function value in response to each cube. This representation is only exponential in the worst case and hence it is suitable to represent larger functions.

The computation of the entities of a PLA table, i.e. the prime implicants, is not an intuitive problem. Quine-McCuskey's method, for instance, requires an exhaustive search to detect adjacent minterms [7,8]. Most approaches are exponential computational problems both time and space-wise.

3. The Proposed Minimization Algorithm

This section presents the main contribution of this article, which is the explanation of a new algorithm for computing prime implicants. The proposed algorithm addresses two challenges that have not been adequately handled in existing approaches: handling multiple output functions, and dealing with the memory size limit of available computer systems when using exact approaches.

To address these challenges, the proposed algorithm is based on a ternary approach that computes the prime implicants for all output bits in a single computation. The algorithm uses recursion to reduce memory requirements and improve scalability. This approach is more efficient and accurate than existing methods for computing prime implicants.

3.1. Ternary Coding of Cubes

Unlike minterms that can be numerically represented, cubes contain the non-numeric symbol '-' to indicate an eliminated literal in the conjunction term. To this end, we can observe that we have a representation with three different symbols {0, 1, -}.

Ternary-based algorithms are a type of algorithm that uses a three-valued logic system (i.e., 0, 1, and don't care) to compute the prime implicants of a Boolean function. Unlike binary-based algorithms that use only 0 and 1 values, ternary-based algorithms can also take advantage of the don't care values, which can greatly reduce the computational complexity and memory requirements of the algorithm [20-25]. The advantages of ternary-based algorithms include:

- Improved Efficiency: Ternary-based algorithms can be more efficient than binary-based algorithms, especially for large and complex functions. By taking advantage of the don't care values, ternary-based algorithms can reduce the number of minterms that need to be compared and merged, which can significantly reduce the computational complexity and memory requirements of the algorithm.
- Scalability: Ternary-based algorithms can be easily scaled to handle functions with any number of inputs and outputs, which makes them well-suited for modern digital circuit design.

Overall, ternary-based algorithms offer several advantages over binary-based algorithms for computing prime implicants of Boolean functions and have become an increasingly popular approach in digital circuit design and optimization.

The radix-3 ternary numbering system can be used to represent any cube numerically. In this approach, the don't care bits '-' in the cubes are replaced with a 2, following the suggestion in [9, 20]. In this case, there will be a unique positive integer representation for each cube that is computed from the ternary weight sum of the bits, i.e. $c_X = \sum_{i=0}^{N-1} x_i * 3^i$ where c_X is the cube representation of the N-bit ternary vector X. Example: In the Boolean function $f(a.b.c.d)$. The product term $(a\,.\,b'.\,d)$ has an inverted literal b', with two non-inverted literals a and d, while the literal c is missed, indicating an eliminated literal. The PLA table coding of this product term would be a cube $(10-1)$. In the suggested ternary coding of this cube '-' symbol has to be replaced by a 2, resulting in the ternary-coded integer $(1021)_3 = 34$, which is calculated from the ternary-weighted digits' sum of $34 = 1*3^3 + 0*3^2 + 2*3^1 + 1*3^0$. The ternary coding of any possible cube in a Boolean function can be exploited in the binary version of the radix-generic minimization algorithm of multiple-valued logic functions, as proposed in [25], as in (1) below:

$$f(t) = \bigvee_{\forall\, t_k=2} f(t-3^k) \wedge f(t-2*3^k) \quad (1)$$

where t_k refers to the kth bit position with a value equal to 2, which indicates a don't care, i.e. an eliminated literal in the kth bit position of cube $t = c_X$. The proposed algorithm scans the array of all possible cubes to compute the cubes that cover the function. This approach suffers from the serious drawback of exponential memory growth that practically limits the scalability of the algorithm. It is important to mention that the function f is not limited to single output Boolean function, but rather this relation is applicable to multi-output Boolean functions. In this case, the conjunction and disjunction operators are bit-wise AND and OR operators respectively.

The computation is recursive as shown in relation (1) above. The recursive computation begins with the function call and the argument $(t = 3^N - 1)$, where N represents the number of input bits. The recursion process terminates when the minterm t does not contain any don't care bit.

3.2. Recursive Computation of Prime Implicants

To reduce the exponential memory space requirement, an equivalent recursive computation of the same formula (1) can be proposed, without the need to store the values in an array of all possible cubes. Instead, it stores only those cubes that are prime implicants and hence candidate entities in the PLA table.

To test whether a specific cube *t* of function *f* is a prime implicant, the function is tested among any possible cube (τ) that may cover the cube *t*, where $\tau_k = 2 \,|\forall\, t_k \neq 2$, i.e. replacing the *k*th position in the cube *t* with 2 for each non '-' bit in *t*. The cube is prime if the following inequality is correct:

$$0 \neq \bigvee_{\forall\, t_k \neq 2} f(t) \wedge f'(\tau) \qquad (2)$$

The result of the disjunction of the function value in response to the cube *t* with all other adjacent cubes identifies whether *t* has to be stored with the set of prime implicants in the PLA table.

This recursive computation continues until the base case of a minterm value is passed as an argument to the function call. In this case, the function returns the function value at the specified point in the domain instead of another recursion. The value $t = 3^k - 1$ is used for the main function call, i.e. the top-level recursion.

4. Discussion

The proposed recursive ternary-based algorithm offers several advantages over existing approaches.

Firstly, the algorithm is designed to efficiently handle multiple output functions, which is a significant improvement over existing algorithms that were originally designed for single output functions. This ensures that the algorithm can provide comprehensive sets of prime implicants that can be used for circuit design and optimization.

Secondly, the algorithm reduces the memory requirement of the computation by employing a recursive approach. Compared to existing approaches, which encounter exponential growth in computation time and memory space, this makes the algorithm more scalable.

Additionally, the ternary-based approach used in the algorithm allows for the determination of prime implicants of all output bits within the same computation. This makes the algorithm more efficient and accurate compared to existing approaches that require separate computations for each output bit.

Overall, the proposed algorithm provides an efficient and exact approach to computing prime implicants of multi-output Boolean functions. The recursive and ternary-based approach used in the algorithm ensures that it is scalable and can handle large multi-output functions without compromising its accuracy and efficiency.

5. Conclusions

In conclusion, most of the known methods to compute prime implicants are designed for single-output Boolean functions. The exponential complexity of this problem limits the scalability of most exact methodologies. This is tackled with either heuristic approaches which result in semi-optimum or approximate results, or with extensive usage of memory resources and also intensive computations to obtain exact solutions. In this article, we proposed a recursive ternary-based algorithm for computing prime implicants capable of efficiently handling multi-output Boolean functions. The algorithm provides an exact set of prime implicants. As compared to existing approaches, the proposed algorithm reduces the memory size required to run the computation and increases scalability because of its recursive nature. This reduction in memory requirement is traded-off with extra computational complexity. Therefore, further research efforts may be invested in reducing computational complexity with respect to time, and in generalizing this approach in minimizing Boolean systems that are combined from several interconnected sub-functions.

Acknowledgments

The authors would like to acknowledge Mustansiriyah University, Baghdad, Iraq, (www.uomustansiriyah.edu.iq), for their support and resources during this research. The facilities provided by the university were instrumental in the successful completion of this study.

Abbreviations

X	The vector of binary-valued inputs
N	Number of bits in X
$f(X)$	A Boolean function of X
x_i	The ith rightmost-bit in the vector X
m_x	The minterm x binary-coded
c_x	The cube x ternary-coded
t	Ternary-coded integer
τ	Adjacent ternary-coded integer

Conflict of interest

The authors declare that there are no conflicts of interest regarding the publication of this manuscript.

Author Contribution Statement

Author Zaid Al-Wardi: proposed the algorithm.

Author Osama Al-Wardi: analyzed the complexity of the proposed algorithm and developed the software.

Both authors discussed the results and contributed to the final manuscript.

References

1. Prasad, V. C. (2018). Novel method to simplify Boolean functions. Automatyka/Automatics, 22(2), 29. https://doi.org/10.7494/automat.2018.22.2.29

2. Rai, S., Neto, W. L., Miyasaka, Y., Zhang, X., Yu, M., Yi, Q., Fujita, M., Manske, G. B., Pontes, M. F., da Rosa, L. S., de Aguiar, M. S., Butzen, P. F., Chien, P.-C., Huang, Y.-S., Wang, H.-R., Jiang, J.-H. R., Gu, J., Zhao, Z., Jiang, Z., … Chatterjee, S. (2021). Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization. 2021 Design, Automation &amp; Test in Europe Conference &amp; Exhibition (DATE). https://doi.org/10.23919/date51398.2021.9473972

3. Fujita, M. (2019). Basic and Advanced Researches in Logic Synthesis and their Industrial Contributions. Proceedings of the 2019 International Symposium on Physical Design. https://doi.org/10.1145/3299902.3311069

4. Salhi, Y. (2018). Approaches for Enumerating All the Essential Prime Implicants. Lecture Notes in Computer Science, 228–239. https://doi.org/10.1007/978-3-319-99344-7_21

5. Mados, B., Bilanova, Z., Chovancova, E., and Adam, N. (2019). Field Programmable Gate Array Hardware Accelerator of Prime Implicants Generation for Single-Output Boolean Functions Minimization. 2019 17th International Conference on Emerging ELearning Technologies and Applications (ICETA). https://doi.org/10.1109/iceta48886.2019.9040020

6. Kubica, M., Opara, A., and Kania, D. (2020). Methods for Representing Boolean Functions—Basic Definitions. Technology Mapping for LUT-Based FPGA, 15–24. https://doi.org/10.1007/978-3-030-60488-2_2

7. Quine, W. V. (1952). The Problem of Simplifying Truth Functions. The American Mathematical Monthly, 59(8), 521–531. https://doi.org/10.1080/00029890.1952.11988183

8. McCluskey, E. J. (1956). Minimization of Boolean Functions*. Bell System Technical Journal, 35(6), 1417–1444.

https://doi.org/10.1002/j.1538-7305.1956.tb03835.x

9. Petrick, S. R., and Sethares, G. C. (1968). On the Determination of Complete Sets of Logical Functions. IEEE Transactions on Computers, C-17(3), 273–273. https://doi.org/10.1126/science.162.3858.1109

10. Seda, P., Seda, M., Hosek, J., Dvorak, J., and Sedova, J. (2019). The Improvement of Quine-McCluskey Method Using Set Covering Problem for Safety Systems. 2019 4th International Conference on Intelligent Green Building and Smart Grid (IGBSG). https://doi.org/10.1109/igbsg.2019.8886174

11. Joshi, M., Sunori, S. K., Tewari, N., Maurya, S., Joshi, M., and Juneja, P. K. (2021). Formulation of C++ program for Quine–McCluskey Method of Boolean Function Minimization. Machine Learning, Advances in Computing, Renewable Energy and Communication, 341–346. https://doi.org/10.1007/978-981-16-2354-7_31

12. Vu, H.-G., Bui, N.-D., Nguyen, A.-T., and ThanhBangLe. (2021). Performance Evaluation of Quine-McCluskey Method on Multi-core CPU. 2021 8th NAFOSTED Conference on Information and Computer Science (NICS). https://doi.org/10.1109/nics54270.2021.9701506

13. Balasubramanian, P., Bernasconi, A., Ciriani, V., and Villa, T. (2021). A Boolean Heuristic for Disjoint SOP Synthesis. 2021 24th Euromicro Conference on Digital System Design (DSD). https://doi.org/10.1109/dsd53832.2021.00019

14. Su, S., Zou, C., Kong, W., Han, J., and Qian, W. (2020). A Novel Heuristic Search Method for Two-Level Approximate Logic Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(3), 654–669. https://doi.org/10.1109/tcad.2018.2890532

15. Miao, J., Gerstlauer, A., & Orshansky, M. (2013). Approximate logic synthesis under general error magnitude and frequency constraints. 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). https://doi.org/10.1109/iccad.2013.6691202

16. Ammes, G., Neto, W. L., Butzen, P., Gaillardon, P.-E., & Ribas, R. P. (2022). A Two-Level Approximate Logic Synthesis Combining Cube Insertion and Removal. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(11), 5126–5130. https://doi.org/10.1109/tcad.2022.3143489

17. Ammes, G., Butzen, P. F., Reis, A. I., & Ribas, R. (2022). Two-Level and Multilevel Approximate Logic Synthesis. Journal of Integrated Circuits and Systems, 17(3), 1–14. https://doi.org/10.29292/jics.v17i3.661

18. Kanakia, H., Nazemi, M., Fayyazi, A., & Pedram, M. (2021). ESPRESSO-GPU: Blazingly Fast Two-Level Logic Minimization. 2021 Design, Automation &amp; Test in Europe Conference &amp; Exhibition (DATE). https://doi.org/10.23919/date51398.2021.9473961

19. Potvin, N., Bersini, H., and Milojevic, D. (2022). Espresso to the rescue of genetic programming facing exponential complexity. Proceedings of the Genetic and Evolutionary Computation Conference

Companion. https://doi.org/10.1145/3520304.3529005

20. Duşa, A. (2008). A mathematical approach to the boolean minimization problem. Quality &amp; Quantity, 44(1), 99–113. https://doi.org/10.1007/s11135-008-9183-x

21. Rudell, R. L., and Sangiovanni-Vincentelli, A. (1987). Multiple-Valued Minimization for PLA Optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 6(5), 727–750. https://doi.org/10.1109/tcad.1987.1270318

22. Fiser, P., Rucky, P., and Vanova, I. (2008). Fast Boolean Minimizer for Completely Specified Functions. 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems. https://doi.org/10.1109/ddecs.2008.4538768

23. Lee, S.-Y., Kim, S., and Kang, S. (2019). Ternary Logic Synthesis with Modified Quine-McCluskey Algorithm. 2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL). https://doi.org/10.1109/ismvl.2019.00035

24. Stanković, R. S., Astola, J. T., and Moraga, C. (2012). Representation of Multiple-Valued Logic Functions. Synthesis Lectures on Digital Circuits & Systems, Morgan & Claypool Publishers. ISBN: 978-3-031-79852-8

25. Al-Wardi, Z. (2021). Radix-p Multiple Valued Logic Function Simplification using Higher Radix Representation. Journal of Physics: Conference Series, 1804(1), 012016. https://doi.org/10.1088/1742-6596/1804/1/012016