Academic Journal of Electrical and Computer Engineering
Vol. 02, No. 01, August 2026
https://doi.org/10.31272/ajece.44
https://ajece.uomustansiriyah.edu.iq
ISSN 3079-2878
Research Article

Empirical Comparison of Multi-Output Boolean Function Minimization Algorithms using Ternary-Indexed Prime Implicants

Zaid Al-Wardi[1*]

[1]Electrical Engineering Department, Mustansiriyah University, Baghdad, Iraq
[*]Email: alwardi@uomustansiriyah.edu.iq

Received: 25/04/2026 Revised: 30/05/2026 Accepted: 03/06/2026

***Abstract*— Prime implicant computation is a fundamental step in minimizing the two-level sum-of-products representation of Boolean function – the form used in programmable logic arrays and throughout classical digital system design. Ternary indexing offers an elegant algebraic framework for representing the prime implicants of binary Boolean functions and can, in principle, improve the efficiency of the computation. It has received little empirical attention to date. This paper presents the first systematic experimental comparison of three algorithms for prime implicant computation of multi-output Boolean functions: a ternary-indexed iterative algorithm, a ternary-indexed recursive algorithm, and the classical Quine-McCluskey Phase 1 method, which serves here as a correctness reference. No prior empirical evaluation of the two ternary-indexed approaches has appeared in the literature. All three algorithms are verified correct on a suite of 16 benchmark circuits using a three-check functional coverage protocol. The experiment quantifies the memory-time trade-off between the two ternary approaches: the recursive algorithm reduces memory consumption by a factor of 704 relative to the iterative algorithm at every tested input size, at the cost of higher runtime. It also produces strictly larger sets of prime implicants, including cross-group implicants that per-output-group methods cannot reach. These results provide the first experimental foundation for the theoretical claims of both ternary approaches and clarify the conditions under which each is most advantageous.**

***Index Terms*— Simplification, recursive computation, cube indexing, prime-implicants, PLA tables.**

I. INTRODUCTION

MINIMIZING multi-output Boolean functions reduces the cost of digital circuits, which makes prime implicant computation a practical concern as well as a theoretical one. Among the standard representations, the sum-of-products (SOP) form is the most widely used – particularly in programmable logic arrays (PLAs), where product lines correspond to product terms and transistors to literals [1]. The PLA remains a relevant structure in modern design because of its regularity and predictable layout [1]. More recent work has explored PLAs built from multiple-valued logic (MVL) cells, which pack higher information density than binary cells and motivate renewed interest in efficient function representation [2] [3]. In all of these settings, the underlying minimization task reduces to finding a minimum-cost set of cubes that covers every minterm of the function – an NP-complete problem [4].

A prime-implicant (PI) is a cube whose covered minterms all belong to the on-set, and from which no literal can be removed without losing the implicant property [5]. The complete PI set is the output of Phase 1 of the Quine-McCluskey's (QMC) algorithm; Phase 2 then selects essential PIs and solves the covering problem [1] [6]. The number of PIs of a function of n variables can grow as large as $3^n/\sqrt{n}$ [5], and this growth is a primary motivation for research into memory-efficient PI computation.

Several families of algorithms have been developed to compute PIs. QMC itself merges adjacent minterms iteratively at Hamming distance 1 [5], providing an exact but memory-intensive reference. ESPRESSO avoids enumerating the full PI set though cube manipulation, trading optimality for speed [6]. Graph-based methods, most notably those built on binary decision diagrams (BDDs), bound the computation by the BDD size rather than the number of primes and so reach larger functions [7]. SAT-based and integer-programming reformulations offer a different route entirely [8]. Related mathematical formulations of Boolean minimization have been proposed outside the engineering literature, most notably in qualitative comparative analysis [9]. In parallel, work on Boolean functional synthesis has explored efficient enumeration of function properties over multi-output domains, through both parallel algorithms [10] and tractable representations [11].

Within this landscape, MVL representations have been studied as a route to more compact logic. A ternary variant of QMC has been investigated in this context [12], and a ternary indexing scheme for the cubes of n-input Boolean function was introduced in [13]: each cube is assigned a unique base-3 index in which the digit 2 denotes a don't-care, placing all 3^n cubes on a single ternary lattice and allowing cube function values to

be expressed by a simple recurrence. This idea was extended in [14], where recursive depth-first traversal of the same lattice was proposed. By jointly processing all output bits as output bitmasks in a single traversal, the recursive variant avoids the memory-intensive iteration that [13] requires.

Neither [13] nor [14] provided empirical validation. Both papers provide theoretical worst-case complexity but no runtime measurements, no memory figures, and no comparison against established methods. This leaves the practical behavior of both algorithms unknown: the actual cost on standard benchmarks, the realized memory footprint, and the nature of the trade-off between them have not been documented.

This paper addresses that gap. Its contributions are as follows:

1) The first implementation and benchmarking of both ternary-indexed algorithms on MCNC benchmarks [15] and standard test functions verified against QMC Phase 1 though a three-check functional coverage protocol.
2) A quantitative characterization of the memory-time trade-off between the recursive and the iterative approaches, including the effect of cache-free recomputation on runtime across input sizes.
3) Experimental evidence that the recursive algorithm produces cross-group prime implicants that are structurally inaccessible to any per-output-group decomposition method.

The remainder of the paper is organized as follows: Section II reviews the necessary background; Section III describes the three algorithms; Section IV details the experimental setup; Section V reports results; Section VI discusses their implications; and section VII concludes.

II. BACKGROUND

This section introduces the formal definitions and mathematical framework that underlie the three algorithms compared in this article.

A multi-output Boolean function is a mapping $\{0,1\}^n \rightarrow \{0,1\}^m$ with n inputs and m output variables [1]. Its truth table lists all 2^n input combinations together with the corresponding output vectors. The on-set of output bit b consists of those minterms at which b evaluates to 1, and the off-set of those at which it evaluates to 0. An output bitmask is an m-bit integer encoding the values of all m outputs at a single minterm; this is the data structure on which both ternary algorithms considered here operate. A more compact representation than the truth table is the PLA table, a canonical form of the multi-output SOP in which each row specifies a product term: the input plane encodes the cube, and the output plane indicates which outputs that product term covers [1] [6].

A cube is a subset of $\{0,1\}^n$ described by a string over $\{0,1,-\}$, where $'-'$ marks a don't care position. A cube, with k don't-care positions, covers 2^k minterms. An implicant of output bit b is a cube whose covered minterms all belong to the on-set of b. If no strictly larger cube -one with more don't-care positions- is also an implicant, then the cube is a prime implicant (PI): no literal can be removed from it without violating the implicant property [6]. A PI that covers at least one minterm not covered by any other PI is said to be essential and every minterm cover must contain it [1].

An output group is the set of all minterms that share the same output bitmask. Classical per-group methods -QMC and the ternary-based iterative algorithm (TBI) [13]- process each group in isolation, so a cube can be declared a PI only if every minterm lies within a single group. The ternary-based recursive algorithm (TBR) [14] removes this restriction. By evaluating primality per output bit rather than per output group, TBR can identify cubes that span several groups yet remain prime for one or more individual bits. These will be referred to throughout the paper as cross-group prime implicants.

QMC phase 1 serves as the correctness oracle: every PI set produced by TBI or TBR is compared against QMC through a three-check functional coverage protocol described in Section IV. In QMC Phase 1 itself, mintems are first partitioned by Hamming weight. Adjacent weight classes are then compared, and any two mintems that differ in exactly one bit position are merged into cube carrying a don't-care at that position. The merging repeats until no further combinations are possible; any term that was never merged is a prime implicant [5] [6]. For multi-output functions, Phase 1 is run independently on each output group, so the total number of QMC invocations equals the number of distinct non-zero bitmasks in the truth table, denotes G. The overall runtime therefore scales as $G \times O(2^n)$ [1].

The ternary indexing scheme assigns a unique integer index to every cube over n binary inputs [13]. Each cube is encoded as a ternary integer $t = \sum_{k=0}^{n-1} d_k \times 3^k$, with $d_k \in \{0,1,2\}$, where the digit value 2 encodes a don't-care at position k. Minterms are included in this encoding as a special case: they are cubes whose digits are all drawn from $\{0,1\}$ and therefore carry no 22. The set of indices $t \in \{0,1,\ldots,3^n-1\}$ covers all 3^n cubes, with minterms and cubes interleaved throughout the range. In particular, every index satisfying $t \equiv 2 (mod\ 3)$ has $d_0 = 2$ and so represents a cube with don't-care at position 0, which can never be a minterm. The universal cube, in which every position is a don't-care, occupies the highest index $t = 3^n - 1$. TABLE I illustrates this encoding for $n = 3$, with input variables x_2, x_1, x_0 placed at positions $k = 2,1,0$.

TABLE I
TERNARY-BASED CUBE INDICES

cube	$d_2\ d_1\ d_0$	Ternary index t
1 0 1 (minterm)	1 0 1	$1\times 1 + 0\times 3 + 1\times 9$ $= \mathbf{10}$
1 – 1 (one don't-care)	1 2 1	$1\times 1 + 2\times 3 + 1\times 9$ $= \mathbf{16}$
- - - (universal cube)	2 2 2	$2\times 1 + 2\times 3 + 2\times 9$ $= 3^3 - 1$ $= \mathbf{26}$

The function value $f(t)$ of any cube t is obtained from its immediate sub-cubes through the following recurrence, applied as a conjunction over all don't-care positions k [13]:

$$f(t) = \bigwedge_{\forall k:\, d_k = 2} f(t - 3^k) \wedge f(t - 2 \times 3^k) \quad (1)$$

The base case $f(\mu)$ = the truth table value of minterm μ. In words, $f(t)$ is the conjunction of the function values of the two sub-cubes obtained by replacing each don't-care in turn with 1 and then with 0. For cube "1-1" $t = 16$, the only don't-care is at position $k = 1$, giving $3^k = 3$, so $f(16) = f(16-3) \wedge f(16-6) = f(13) \wedge f(10)$, where $t = 13$ corresponds to minterm "111" and $t = 10$ to minterm "101". If both are in the on-set of some output bit b, then cube "1-1" is an implicant of function bit b; if either is in the off-set, it is not. The primality condition follows directly: cube t is a prime implicant of output bit b when $f(t)[b] = 1$, and for every concrete digit ($d_k \neq 2$), the cube τ obtained by replacing digit d_k with 2 satisfies $f(\tau)[b] = 0$, that is, no larger cube covering superset of minterms is also an implicant of b. The iterative algorithm checks this condition by a cache lookup, since $\tau > t$ and $f(\tau)$ has already been computed during the upwards scan; the recursive algorithm recomputes $f(\tau)$ on demand [13] [14].

Two alternative uses of ternary structure in this area are worth distinguishing. Ternary decision diagrams (TDDs) represent and enumerate PIs implicitly: the 1-paths of a prime TDD correspond to prime implicants [1]. This is a structurally different use of ternary representations, oriented towards storage of a PI set rather than computation of individual PI values through recurrence. Separately, PLA design based on generalized Reed-Muller expansion of MVL functions has been explored as an alternative to SOP-based two-level minimization for non-binary circuits, a line of work that reinforces the general case for efficient Boolean representations in multi-valued systems [2] [3]. What distinguishes TBI and TBR from all of these is that they operate directly on the ternary cube lattice via a scalar recurrence relation, without constructing intermediate objects such as BDDs, implicant tables, or expansion trees. Their shared contribution is the use of ternary indexing to avoid redundant cube evaluation. [13] [14]

III. Algorithm Description

This section describes the three algorithms compared in the experiment: the ternary-based iterative algorithm (TBI) [13], the ternary-based recursive algorithm (TBR) [14], and QMC Phase 1, which serves as the correctness reference. TBI and QMC each perform one independent computation per output group, while TBR performs a single depth-first traversal over all output bits simultaneously. All three produce a set of (cube, output-bitmask) pairs, which is the standard PLA table format [1].

A. Ternary-Based Iterative Algorithm (TBI)

TBI decomposes the multi-output function into output groups and processes each group independently [13]. Within a group, an iterative scan visits every ternary cube index $t \in \{0,1,\ldots,3^n - 1\}$ in ascending order, covering the complete ternary cube space of size 3^n. The total number of cube evaluations is $G \times 3^n$, where G is the number of distinct non-zero output bitmasks. When a function has many output groups this group multiplier dominates the runtime.

At each visited cube t, the function value $f(t)$ is computed through the conjunctive recurrence (1) from section II, using base case $f(\mu) = 1$ if the minterm μ belongs to the current output group and 0 otherwise. Results are stored in a memoization[1][16] cache -a dictionary of size 3^nentries- reinitialized fresh for each output group scan. Because the scan proceeds in ascending index order and every sub-cube has a smaller index than its parent, each value needed by the recurrence is already in the cache by the time requested, and no recursive call is ever issued. The cache itself is the dominant memory cost: one entry per cube, approximately 88 bytes per dictionary slot in Python, giving a peak allocation of 88×3^n bytes regardless of function density or output count [13].

Cube t is recorded as a prime implicant when $f(t) = 1$ and for every concrete digit position k (i.e. every position with $d_k \neq 2$), the parent cube $\tau = t + (2 - d_k) \times 3^k$ satisfies $f(\tau) = 0$. Equivalently, t is a prime implicant, if and only if it is an implicant but none of its immediate parents is. Since $\tau > t$, the value $f(\tau)$ has already been entered in the cache during the upward scan, so the primality test is handful of lookups with no additional computation [13]. Because TBI works on one output group at a time, it can only discover prime implicants whose covered minterms share a single bitmask; cross-group prime implicants are structurally out of reach.

QMC Phase 1 uses the same group decomposition as TBI, running once per output group [5]. As a consequence, the two algorithms operate on identical inputs and should produce identical PI sets. This identity is one of the three correctness checks defined in Section IV: QMC is treated as the reference, and TBI is considered correct whenever the two agree on every circuit. TBR is verified separately against functional coverage, since it may report additional cross-group PIs that neither TBI nor QMC can find.

B. Ternary-Based Recursive Algorithm (TBR)

TBR performs a single depth-first search (DFS) from the universal cube $t = 3^n - 1$, processing all m output bits in one traversal with no decomposition into output groups [14]. Function values are carried throughout as output bitmasks -m-bit integers in which bit b holds the function value for output b- so all outputs are updated in parallel through bitwise operations. A compact visited bitmap of size $\lceil 3^n/8 \rceil$ bytes -one bit per cube index- prevents nodes from being revisited. This visited bitmap is the only persistent data structure TBR allocates; there is no memoization cache.

Define $f\_or(t)$, as an output bitmask in which bit $b = 1$ if and only if cube t covers at least one minterm with output $b = 1$. It satisfies a disjunctive recurrence over the two sub-cubes at each don't-care position [14]:

$$f\_or(t) = \bigvee_{\forall k:\, d_k = 2} f\_or(t - 3^k) \vee f\_or(t - 2 \times 3^k) \qquad (2)$$

[1] Memoization is a computational optimization technique, used to speed up the computation of a function, in which the result of a function call is stored the first time it is computed, so that subsequent calls with the same input can retrieve the stored result directly rather than repeating the computation.

with base case $f\_or(\mu)$ = truth table output bitmask of minterm μ. When $f\_or(t) = 0$ across all output bits, no minterm below cube t lies in any on-set, and the entire subtree can be skipped. This pruning is the main mechanism by which TBR avoids irrelevant regions of the cube lattice. To keep the memory footprint far below that of TBI, f_or is recomputed from scratch at every DFS node and never stored between nodes. That recomputation is the chief reason TBR runs slower than TBI on dense circuits, where pruning rarely fires.

Define $f\_and(t)$ analogously as the output bitmask in which bit $b = 1$ if and only if every minterm covered by t has output $b = 1$:

$$f\_and(t) = \bigwedge_{\forall k:\, d_k=2} f\_and(t - 3^k) \wedge f\_and(t - 2 \times 3^k) \quad (3)$$

Cube t is an implicant for output bit b if $f\_and(t)[b] = 1$ -that is, every minterm it covers has output $b = 1$. Like f_or, f_and is recomputed from scratch at every DFS node with no caching. The base case is $f\_and(\mu)$= the truth-table output bitmask of that minterm μ.

Primality is then decided through a joint test applied to all output bits at once:

$$prime\_bits(t) = f\_and(t) \wedge \neg \bigvee_{\forall \tau} f\_and(\tau) \quad (4)$$

where τ ranges over the cubes obtained by replacing one concrete digit of t with 2, and $f\_and(\tau)$ is computed fresh for each τ [14]. Bit b of $prime\_bits(t)$ is 1 exactly when t is an implicant of output b and no cube covering a strict superset of t's mintems is also an implicant of b -the standard primality condition, evaluated for every output bit simultaneously through bitmask arithmetic. This per-bit evaluation is what makes cross-group prime implicants visible. A cross-group prime implicant is a cube whose minterms span several output groups, yet satisfies $prime\_bits(t)[b] = 1$ for at least one bit b, because primality is checked per output bit rather than per output group. No such a cube would ever be recorded by TBI or QMC.

The memory profile of TBR follows directly from the absence of memoization cache. Only three persistent structures are allocated: the visited bitmap of $\lceil 3^n/8 \rceil$ bytes, the recursion stack (bounded to depth n) and the growing list of prime implicants [14]. As a concrete point of comparison, at $n = 9$, TBI allocates 88×3^9 bytes = 1,692 KB while TBR's visited bitmap occupies $\lceil 3^9/8 \rceil$ bytes = 2.4 KB -a factor of 704 smaller at the same n. This factor is constant across all n and independent of function density or output count. The runtime trade-off goes in the opposite direction: because f_or and f_and are computed at every DFS node without caching, TBR is slower than TBI on most circuits. On functions with many output groups, however, TBR's single-pass architecture removes the G group multiplier that drives TBI's, runtime, and TBR can become competitive or even faster when G is large. This is the time-memory trade-off deliberately built into TBR, and the experiment in section IV provides the first empirical quantification of it.

IV. Experimental Setup

The benchmark suite comprises 16 circuits drawn from two sources -the MCNC benchmark collection and a group of standard textbook functions- supplemented by synthetic random circuits to give continuous coverage of input sizes from $n = 3$ to 9 [15]. The full list is shown in Table II below:

TABLE II
Benchmark Circuit Suite (16 circuits, n=3…9)

#	Circuit name	n	m	min	D%	G	
1	full-adder	3	2	7	87.5	3	T
2	rand_3in_1out	3	1	4	50.0	1	S
3	2bit_multiplier	4	4	9	56.2	6	T
4	BCD_7seg	4	7	10	62.5	10	T
5	parity_4	4	1	8	50.0	1	T
6	rand_4in_2out	4	2	7	43.8	3	S
7	majority_5	5	1	16	50.0	1	T
8	rand_5in_2out	5	2	13	40.6	3	S
9	rand_6in_3out	6	3	22	34.4	7	S
10	rand_7in_3out	7	3	47	36.7	7	S
11	rand_8in_sparse	8	2	20	7.8	3	S
12	rand_8in_4out	8	4	59	23.0	15	S
13	rand_8in_dense	8	2	186	72.7	3	S
14	ex5	8	63	256	100.0	107	M
15	rand_9in_4out	9	4	120	23.4	15	S
16	apex4	9	19	512	100.0	319	M

n: # inputs | m: # outputs | min: # minterms | G: # output groups | D% density = minterms/$2^n \times 100\%$ | **T: textbook | S: Synthetic | M: MCNC**

1) **Metrics.** Four quantities recorded for each algorithm on each circuit:
   - *Wall-clock runtime*, measured using a high-resolution Python performance counter `time.perf_counter()` and reported in seconds to six decimal places.
   - *Peak memory*, measured differently for each algorithm. For TBI it is estimated analytically as 88×3^n bytes, one Python dictionary entry per cube index; because TBI always fills the complete cache regardless of function content, this estimate is exact. For TBR it is measured directly using Python `tracemalloc`, which records the peak heap allocation during the DFS; the visited bitmap size $\lceil 3^n/8 \rceil$ bytes is reported separately as the structural memory -the minimum TBR must allocate independent of the function. For QMC it is estimated analytically as 88×2^n bytes per group run.
   - *Prime implicant count*, the total number of (cube, output-bitmask) pairs produced by each algorithm.
   - *Correctness*, verified by the three-check protocol described below.
2) **Three-check correctness protocol.** Every PI set produced by TBI and TBR is subjected to the following three checks before any timing or memory reading is accepted as valid.
   - *Check 1 - Functional coverage:* for every on-set minterm m and every output bit b active at m (truth table value 1), at least one PI in the result set must cover m with bit b set in its output bitmask. A PI set that fails this check is incomplete and does not represent the function.

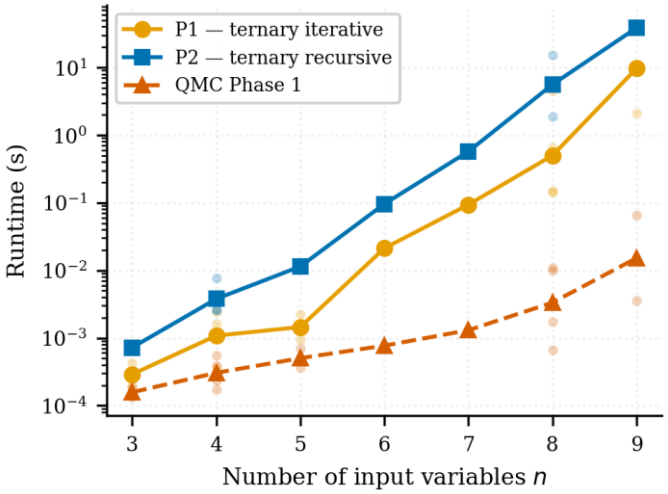

Fig. 1. Runtime vs. number of inputs.

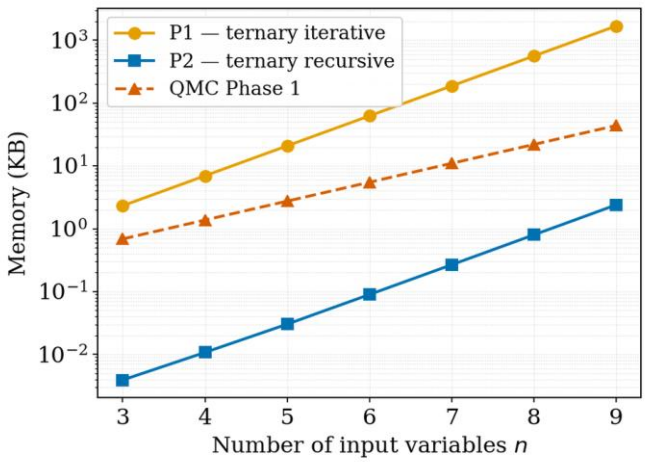

Fig. 2. Memory vs. number of inputs.

- *Check 2 – No spurious coverage:* no PI in the result set may cover an off-set minterm for any output bit whose truth-table value at that minterm is 0. A PI set that fails this check contains a cube covering minterms that it should not -it is incorrect.
- *Check 3 – Functional equivalence:* the TBI and TBR result sets must produce identical coverage for every (minterm, output-bit) pair. That is, for every minterm μ and every output bit b, the set of PIs covering (μ, b) pair must be non-empty in both result sets or empty in both. This check confirms that the two algorithms, despite potentially different PI sets, represent the same function.

QMC serves as the ground truth for Checks 1 and 2: TBI and QMC must agree on both the PI count and the set of (cube, output-bitmask) pairs on every circuit, and any disagreement flags TBI as incorrect. All 16 circuits pass all three checks for both TBI and TBR. TBR is expected to produce a larger PI set than TBI because of cross-group prime implicants, so equality of PI counts between TBI and TBR is not required -only functional coverage equivalence established by check 3.

2) **Implementation.** All three algorithms are implemented in Python 3.13.12. Runtime is measured with `time.perf_counter()`; peak memory for TBR is recorded using `tracemalloc`. The implementations follow the description in Section III exactly, with no algorithmic optimization beyond what appears in the original publications [13] [14], so the comparison reflects the algorithms as published rather than choices made during coding. A timeout of 120 seconds is applied to TBR; circuits exceeding it are recorded as TIMEOUT. Their visited bitmap memory remains reported, since it is fixed by n alone, but they are excluded from runtime figure. In the current benchmark set this applies to apex4 and ex5, both 100% dense circuits in which TBR's pruning mechanism never fires. The benchmarks are read from standard PLA-format files [15], parsed into truth table, and passed identically to all three algorithms. All experiments run on a single machine -an Intel® Core™ i5-5200U CPU, operating at 2.2 GHz, with RAM of 8GB.

V. Results

Results are reported for all 16 circuits across three subsections covering runtime, memory, and prime implicant sets. All 16 circuits passed all three correctness checks for both TBI and TBR.

A. Runtime

Fig.1 shows the wall-clock runtime of the three algorithms as a function of the input size n, which ranges from 3 to 9. The y-axis is log-scaled, and each point is the geometric mean over all circuits at that n. QMC is the fastest method at every n, TBI is intermediate, and TBR is the slowest across most of the range.

TBI runtime grows as $G \times 3^n$, where G is the number of output groups. On apex4, for example, has $G = 319$ at $n = 9$, giving $319 \times 3^9 \approx 6.3$ million cube evaluations and a measured runtime of 44.5 s. For a single-output circuit ($G = 1$) the runtime collapses to a single 3^n scan, the theoretical minimum for the iterative approach. At $n = 8$, TBI completes in 0.11 s on rand_8in_sparse and 4.5 s on ex5 -a factor of forty between the fastest and slowest circuits at the same size, driven entirely by differences in G.

TBR performs a single DFS pass regardless of G, so its runtime carries no group multiplier. On rand_9in_4out with $G = 15$, TBR takes 34.2 s against TBI's 1.95 s -slower despite the absence of the multiplier, because f_or and f_and are recomputed from scratch at every DFS node. Two circuits apex4 and ex5, exceeded the 120 s timeout. Both are 100% dense: every minterm is in the on-set, so $f\_or(t) = 1$ for every cube t and the pruning rule never eliminates a subtree. In these cases, TBR must visit all 3^n nodes and recompute f_or and f_and at each one with no benefit from pruning. At the opposite extreme, rand_8in_sparse (density 7.8%) is the kind of circuits where pruning should help TBR the most, yet TBR still takes 13.6 s against TBI's 0.11 s: even when many subtrees are skipped, the recomputation cost of the surviving nodes -each requiring a descent to the minterms- dominates the total runtime.

B. Memory

Fig. 2 shows the peak memory of the three algorithms against n. The central result is unconditional: TBR's visited bitmap occupies $\lceil 3^n/8 \rceil$ bytes, and TBI's cache occupies 88×3^n. Both quantities are determined by n alone -TBR because its bitmap reserves one bit per cube index regardless of what the function does, and TBI because its scan always fills the complete 3^n cache regardless of density or output count. The ratio between the two is therefore constant at 704 for every circuit in the benchmark suite, not an average or favorable case. At $n = 9$ the absolute figures are 1,692 KB for TBI and 2.4 KB for TBR, with QMC sitting between them at $88 \times 2^n = 44$KB per group run.

C. Prime Implicant Sets

On all 16 circuits QMC Phase 1 and TBI produce identical PI sets -exactly the same (cube, output-bitmask) pairs- confirming that the two per-group methods are equivalent in practice. TBR produces a larger set on every multi-output circuit: on rand_8in_4out ($G = 15$) TBR produces 73 PIs against 58 for TBI and QMC; on rand_8in_dense ($G = 3$) TBR finds 219 against 156. On single-output circuit ($G = 1$) all three algorithms produce identical sets, since no cross-group implicants can exist when there is only one group. Fig. 3 contrasts the TBI/QMC and TBR prime implicant counts on these two representative multi-output circuits.

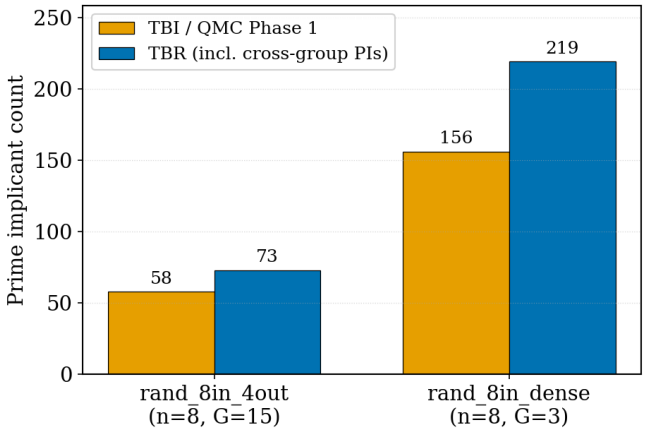

Fig. 3. Prime implicant counts produced by TBI/QMC and by TBR on two representative multi-output circuits.

D. Cross-Group Prime Implicants

The additional implicants found by TBR are cross-group prime implicants as defined in Section II: cubes whose covered minterms span two or more output groups, yet remain prime for one or more individual output bits under the per-output-bit primality criterion. Because all three correctness checks pass for TBI and TBR on every circuit, the two PI sets are functionally correct representations of the same function despite differing in size and composition. The gap between the two counts widens with G: circuits with more output groups show a larger TBI-TBR difference, consistent with the observation that more groups create more opportunities for cubes to cross between them.

VI. Discussion

The results exhibit a clear time-space trade-off, where TBR eliminates the memoization cache entirely, cutting the memory requirement to a small fraction of TBI's, and pays for this with the recomputation of f_or and f_and from scratch at every DFS node. This is the deliberate architectural choice of [14], and the experiments confirm that the trade-off behaves as the theoretical prediction suggests. Memory is reduced by a factor of 704 at every circuit, while runtime is higher than TBI's on most circuits. The memory advantage is unconditional -it holds irrespective of density, output count or function structure- whereas the runtime penalty varies with these properties, as Section V-A shows.

TBR's single-pass architecture becomes advantageous when the number of output groups G is large. TBI's runtime grows proportionally with G, since the algorithm performs one independent 3^n scan per group; TBR, by contrast, always performs exactly one DFS pass. On apex4 with $G = 319$, TBI must carry out 319 full scans of the ternary cube space -a workload TBR avoids entirely, paying instead for recomputation within each DFS node. Conversely, TBI is the faster algorithm on circuits with low G and high density: its cache guarantees that each $f(t)$ is computed exactly once, while TBR, without cache, repeatedly recomputes the same sub-cube values along different DFS paths. The pathological case is 100% density, where TBR's pruning mechanism never fires and both apex4 and ex5 time out. QMC remains the fastest of the three throughout, because it operates on binary cube space size 2^n, rather than the ternary space size 3^n, and applies its group decomposition through a lightweight iterative merging procedure with negligible per-group overhead.

In practical terms, the choice between TBI and TBR depends on the memory budget available and on the value of G for the target function. TBR is preferable when memory is the binding constraint; TBI is preferable when runtime dominates the requirements and sufficient memory is available.

A separate result of the experiment concerns the structure of the PI sets themselves. TBR produces a strictly larger set than TBI on every multi-output circuit, because it finds cross-group prime implicants that no per-output-group method can reach. Both sets pass all three correctness checks, so the difference is not an error: the two algorithms implement two different but equally valid definitions of primality -per-group for TBI and per-output-bit for TBR. A larger and more complete PI set may in turn enable a more compact cover in Phase 2, the essential PI selection and covering problem [1], since cross-group implicants can contribute to covering minterms from several groups at once and potentially reduce the number of product terms in the final SOP expression. Investigating this is a natural next step and lies outside the scope of the present work. Consistent with its origin in the multi-output setting, the PI-count gap grows with G and vanishes for single-output circuits ($G = 1$), where the three algorithms produce identical sets.

Several limitations should be noted. The three algorithms are implemented in Python, which introduces per-function-call overhead that a compiled implementation would not incur; the

absolute runtime figures should therefore not be extrapolated to C or hardware implementations. The asymptotic structure of each algorithm is language-independent, but the constants differ significantly. The input range is capped at $n \leq 9$ in this study; the memory curves in Fig. 2 already show a clean and constant separation between TBI and TBR, so extending to larger n would confirm the scaling trend without changing the qualitative conclusions. The TBR timeout on fully dense circuits such as apex4 and ex5 is a structural consequence of the cache-free design -when pruning never fires, the recomputation cost grows unbounded relative to TBI's cached scan. Finally Phase 2 of the minimization problem -essential PI selection and covering [1]- is outside the scope of this paper, so the relative quality of the covers that would be produced from TBI's and TBR's PI sets is not evaluated here.

VII. Conclusion

This paper has presented the first systematic experimental comparison of the ternary-based iterative algorithm (TBI), the ternary-based recursive algorithm (TBR), and QMC phase 1 for prime implicant computation of multi-output Boolean functions. On all 16 benchmark circuits, both TBI and TBR pass the three-check functional coverage protocol, establishing for the first time that the theoretical claims of [13] and [14] hold in practice. TBR's visited bitmap occupies $\lceil 3^n/8 \rceil$ bytes -a factor of 704 smaller than TBI's memoization cache of 88×3^n bytes- at every tested n, and this advantage is unconditional: it does not depend on function density, output count, or circuit structure. TBI is the faster of the two ternary algorithms on most circuits because its cache eliminates redundant sub-cube evaluations, while TBR's cache-free design forces repeated recomputation at every DFS node and runs up against the 120 s timeout on fully dense circuits where pruning cannot fire. On every multi-output circuit, TBR produces a strictly larger PI set than TBI, corresponding to cross-group prime implicants that per-output-group methods cannot find; both sets are functionally correct, and TBR's set represents the more complete enumeration under the per-output-bit primality criterion.

Future extensions from this work may include: a compiler implementation of TBR in C would decouple Python's per-call overhead from the algorithm's intrinsic recomputation cost, allowing the runtime behavior to be assessed at $n = 16$ or beyond, and on circuits where the group multiplier of TBI becomes dominant. A second direction is to apply Phase 2 of the minimization procedure to TBR's larger PI set, to determine whether cross-group prime implicants yield a measurably more compact SOP cover than the set produced by TBI or QMC. A third is the extension of the recursive approach to multiple-valued logic: the radix-p generalization proposed in [13] lifts the ternary cube lattice to arbitrary radix p, and the recursive architecture may carry a comparable memory advantage.

Acknowledgment

This work is part of the scientific plan of the Electrical Engineering Department, in the College of Engineering, Mustansiriyah University, (www.uomustansiriyah.edu.iq).

References

[1] O. Coudert and T. Sasao, "Two-level logic minimization," in *Logic Synthesis and Verification*, Springer, 2002, pp. 1--28.

[2] E. Zaitseva, V. Levashenko, I. Lukyanchuk, M. Kvassay, J. Rabcan and P. Rusnak, "Application of Generalised Reed-Muller Expansion in Development of Programmable Logic Array," in *IDAACS*, 2019.

[3] E. Zaitseva, V. Levashenko, I. Lukyanchuk, J. Rabcan, M. Kvassay and P. Rusnak, "Application of Generalized Reed–Muller Expression for Development of Non-Binary Circuits," *Electronics,,* vol. 9, no. 1, p. 12, 2020.

[4] C. Umans, T. Villa and A. Sangiovanni-Vincentelli, "Complexity of Two-Level Logic Minimization," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 7, p. 1230–1246, 2006.

[5] E. McCluskey, "Minimization of Boolean Functions," *Bell System Technical Journal,* vol. 35, no. 6, p. 1417–1444, 1956.

[6] R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, Logic Minimization Algorithms for VLSI Synthesis, vol. 2, Springer Science & Bussiness Media, 1984.

[7] O. Coudert and J. Madre, A New Implicit Graph-Based Prime and Essential Prime Computation Technique, Kluwer Academic Publishers,, 1992.

[8] V. M. Manquinho, P. F. Flores, J. P. M. Silva and A. L. Oliveira, "Prime Implicant Computation using Satisfiability Algorithms," in *Proceedings Ninth IEEE International Conference on Tools with Artificial Intelligence*, 1997.

[9] A. Dușa, "A mathematical approach to the boolean minimization problem," *Quality & Quantity,* vol. 44, pp. 99-113, 2010.

[10] S. Akshay, S. Chakraborty, A. K. John and S. Shah, "Towards parallel Boolean functional synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Uppsa;a, Sweden, 2017.

[11] S. Akshay, S. Chakraborty and S. Shah, "Tractable representations for Boolean functional synthesis," *Annals of Mathematics and Artificial Intelligence,* vol. 92, no. 5, pp. 1051-1096, 2024.

[12] S.-Y. Lee, S. Kim and S. Kang, "Ternary logic synthesis with modified Quine-McCluskey algorithm," in *IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*, 2019.

[13] Z. Al-Wardi, "Radix-p MVL Function Simplification using Higher Radix Representation," in *Journal of Physics: Conference Series*, 2021.

[14] Z. Al-Wardi and O. Al-Wardi, "RECURSIVE TERNARY-BASED ALGORITHM FOR COMPUTING PRIME IMPLICANTS OF MULTI-OUTPUT BOOLEAN FUNCTIONS," *Journal of Engineering and Sustainable Development,* vol. 27, no. 3, pp. 308-316, 2023.

[15] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," Microelectronics Center of North Carolina (MCNC), 1991.

[16] D. S. Warren, "Memoing for logic programs," *Commun. ACM,* vol. 35, no. 3, p. 93–111, 1992.

Zaid Al-Wardi Born in Baghdad-Iraq 1971. Earned a B.Sc. in Electrical Engineering, from the University of Baghdad, in 1992. Then earned an M.Sc. in Computer and Control Engineering also from the University of Baghdad, 1996. In 2018 he earned the Ph.D. in Computer Engineering (Dr.-Ing.) from the University of Bremen, Bremen, Germany.