LogicSR: A Unified Benchmark for Logical Discovery from Data

Zimeng Zhang[1,2*†] **Xin Zheng**[1*†] **Feifei Zhang**[1†] **Yunxin Liu**[1] **Yuanchun Li**[1‡]
[1] Institute for AI Industry Research (AIR), Tsinghua University
[2] Department of CSE, The Hong Kong University of Science and Technology

Abstract

Discovering underlying logical expressions from data is a critical task for interpretable AI and scientific discovery, yet a unified evaluation remains lacking. Logical symbolic regression spans two domains: The field of Symbolic Regression (SR) primarily focuses on continuous mathematical functions, while Logic Synthesis (LS) is designed for exact, noise-free specifications, not for learning from incomplete or noisy data. This leaves a crucial gap for evaluating algorithms that can learn generalizable logical rules in realistic scenarios. To address this, we introduce LogicSR, a unified and scalable benchmark for logical symbolic regression. LogicSR is built from two sources: real-world problems from digital circuits and biological networks, and a novel synthetic data generator capable of producing a diverse set of complex logical formulas at scale. We use LogicSR to conduct a rigorous evaluation of 18 algorithms, spanning classical logic solvers, symbolic regression methods, modern machine learning models, and Large Language Models (LLMs). Our findings reveal that the logical modeling capabilities and generalization robustness of these algorithms significantly depend on task scale and logical complexity, with LLMs showing limited complex logical reasoning ability. LogicSR provides a robust foundation to benchmark progress, unify evaluation across disparate fields, and steer the future development of powerful neuro-symbolic systems. The code and data are available at: `https://github.com/deepsymbol/LogicSR`.

1 Introduction

The discovery of underlying logical expressions from data, which we refer to as *logical symbolic regression*, is a key challenge in artificial intelligence. Uncovering these expressions is highly valuable. It can provide human-interpretable rules for complex phenomena [29] and generate programs that can be directly synthesized into electronic circuits [33]. These practical applications make logical symbolic regression a crucial tool for advancing interpretable modeling and automated scientific discovery in domains governed by discrete logic, such as digital systems and gene regulatory networks.

Despite its importance, existing logical symbolic regression methods remain difficult to compare systematically. Specifically, it spans two established fields: **Symbolic Regression (SR)**, which traditionally focuses on discovering single-output continuous-valued formulas; and **Logic Synthesis (LS)**, which typically assumes exact and complete specifications with complex network structures. SR traditionally uses methods like genetic programming, and has experienced a resurgence with new machine learning-based approaches [32, 14]. However, SR research and its associated benchmarks have primarily focused on recovering continuous mathematical formulas involving operators such as addition, multiplication, and transcendental functions. Meanwhile, the expressions studied in SR are

*Work done during the internship at AIR, Tsinghua University.
†Equal contribution.
‡Yuanchun Li is the corresponding author. Email: `liyuanchun@air.tsinghua.edu.cn`

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

usually limited to small scales (e.g., fewer than 10 variables and operators). There remains a lack of standardized SR benchmarks for discovering complex Boolean expressions.

On the other hand, the field of Logic Synthesis (LS), central to Electronic Design Automation (EDA), excels at manipulating and optimizing logical expressions for hardware design [5, 11]. However, LS methods are mostly based on symbolic search techniques engineered for scenarios with exact and complete specifications (e.g., a full truth table). They are seldom designed to find approximate solutions from noisy or incomplete data, nor do they prioritize generalization to unseen inputs—properties that are paramount for machine learning applications. This leaves a critical gap for a unified benchmark designed to evaluate algorithms on their ability to learn approximate and generalizable logical rules from data.

While SR and LS share a similar objective, their respective communities have evolved in isolation. Though emerging techniques such as large language models (LLMs) exhibit a generalization ability in discovering Boolean formulas from data samples, their performance relative to traditional Logical SR or LS methods remains unknown. A unified benchmark is therefore essential to rigorously compare these disparate approaches and foster cross-domain innovation.

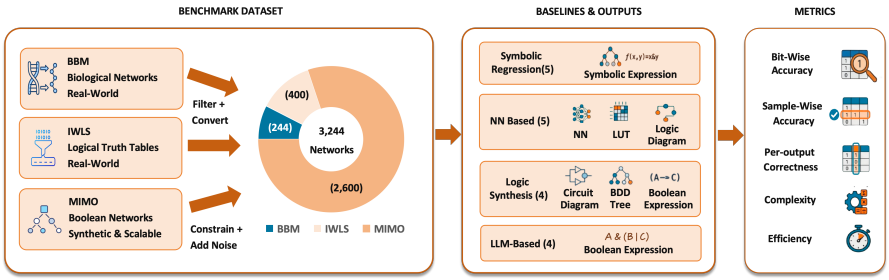

Figure 1: Main components of the LogicSR benchmark.

To fill this void, we introduce LogicSR, a unified and scalable benchmark for logical symbolic regression. Our benchmark is based on two primary sources. First, we curate real-world datasets from application domains where logical reasoning is fundamental, including data-driven logic synthesis and Boolean models of biological gene regulatory networks[21]. Second, we propose a scalable synthetic data generation algorithm. It combines truth-table analysis, structured sampling, and network-based composition to produce a diverse set of complex and non-redundant logical formulas. We further add configurable noise and partial observations to support evaluation under practical scenarios. The main components of LogicSR are shown in Figure 1.

Using our benchmark, we conduct a fine-grained evaluation of 18 representative algorithms, covering classical logic-based methods, modern machine-learning models, and large language models. Overall, the results show that existing approaches struggle with moderately scaled logical discovery tasks, either requiring substantial computational resources or delivering unsatisfactory accuracy. More specifically: (1) traditional logic synthesis performs well on small-scale problems but lacks scalability and flexibility; (2) ML and symbolic-regression models generalize better but come with significant computational overhead; and (3) current LLMs still show limited capacity for handling complex logical reasoning. These findings collectively reveal challenges in scalability, noise robustness, and operator-set compatibility across current methods.

In summary, our main contributions are as follows.

- **A unified and scalable benchmark for logical symbolic regression**, featuring a diverse set of problems derived from both real-world applications and a novel, scalable synthetic data generation process.
- **An extensive, cross-domain evaluation** of 18 diverse algorithms, evaluating performance on metrics including accuracy in distribution, generalization outside distribution, simplicity of expression and computational speed.
- **A foundational set of findings and analyses** that map the capability boundaries of current methods, and provide insights to guide further research towards addressing existing limitations in logical reasoning tasks.

We hope this study offers a solid foundation for evaluating and improving algorithms in logical discovery, enabling unified and scalable evaluation across diverse methods, problem scales, and data conditions.

2 Problem Definition

The goal of symbolic regression is to identify a function from a set of input-output examples, while logical symbolic regression frames the task to Boolean formulas and their representation spaces. An ideal benchmark must provide datasets that are formally sound while also being diverse, scalable, and representative of real-world challenges like noise and incomplete information. We formalize the associated definitions as follows.

Definition 2.1 (Logical Symbolic Regression Task)**.** Let $f : \{0,1\}^k \to \{0,1\}$ be an unknown Boolean function. Let $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$ be a dataset of N input-output pairs, where $x^{(i)} \in \{0,1\}^k$, $y^{(i)} \in \{0,1\}$, and each pair is a (possibly noisy) sample from f. The task is to find a logical expression, represented as a logic network $\mathcal{G}'$, that implements a function f' which best approximates f on both seen and unseen data.

Definition 2.2 (Logic Network)**.** A logic network is a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{I}, \mathcal{O})$ where: $\mathcal{V}$ is the set of nodes, consisting of input nodes and internal logic–gate nodes. Each internal node applies a Boolean operator from a predefined operator set $\mathcal{U}$ (e.g., $\{\texttt{AND}, \texttt{OR}\}$). $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed edges with an optional inversion; $\mathcal{I} \subseteq \mathcal{V}$ is the set of input nodes, with an in-degree of zero; $\mathcal{O} \subseteq \mathcal{V}$ is the set of nodes chosen as output, with an out-degree of zero. A logic network $\mathcal{G}$ should satisfy the following conditions: (a) **Connectivity:** Every non-input node $v \in \mathcal{V} \setminus \mathcal{I}$ must be reachable from at least one input node $v_i \in \mathcal{I}$ (i.e. not isolated). (b) **Acyclicity:** The graph $\mathcal{G}$ must be a Directed Acyclic Graph (DAG).

In essence, **Definition 2.1** specifies the learning problem and its solution space, while **Definition 2.2** describes the structural form of candidate solutions for expression-generating baselines and specifies the structure of the ground truth used in data synthesis.

3 The LogicSR Benchmark

LogicSR provides a diverse suite of problems by 1) curating real-world datasets, 2) proposing a scalable synthetic data generator, 3) incorporating a noise model with controllable noise injection, and 4) using incomplete data samples to test generalization.

Table 1 gives an overview of the composition and characteristics of the benchmark. For IWLS, it does not provide reference circuits, so Gate and I/O counts are estimated from contest submissions. For MIMO, Sample counts are reported per type (AN/ANO) and should be doubled for the total.

Table 1: Summary of LogicSR. In total, LogicSR comprises 3,244 networks and 6,132 truth tables (including noisy variants).

	BBM	IWLS (22'–25')	MIMO Small	MIMO Large
Dataset Type	Real-World	Real-World	Synthetic	Synthetic
Field	Biological Networks	Logic Circuits	Logical Networks	Logical Networks
Number of Samples	244	400	300/300	300/250/150
Input Range	0–223	5–16	5/10	20/40/80
Output Range	0–257	1–115	5/10	20/40/80
Number of Gates	4–853	Around 10–8,000	10–160	160–2,560
Noise Level	0, 0.01, 0.05	0	0, 0.01, 0.05	0

3.1 Real-world Datasets

Boolean tasks are closely related to logical circuits and biological networks. Logical circuits exhibit complex structural dependencies, while biological networks often capture temporal dynamics and nonlinear relationships. LogicSR integrates real-world data from both domains, providing a testing ground for a large set of problems.

Circuit Dataset. This dataset is derived from the International Workshop on Logic & Synthesis (IWLS) benchmarks [12]. IWLS provides participants with Boolean truth tables that represent

challenges in real-world circuit design, which aligns with our task. We selected 400 truth tables in IWLS, and converted them into a unified format for evaluation. The original format of truth tables in IWLS assumes an implicit ordering of input variables, and only provides the output combinations. To support consistent representation, we converted the truth tables to explicitly specify input assignments.

Biological Dataset. The BioDivine Boolean Models (BBM) dataset contains 244 Boolean network models of real biological systems (as of evaluation), such as gene regulatory networks [37]. Unlike IWLS, BBM describes regulatory relationships between biological variables rather than directly providing truth tables for each target function. To construct benchmark tasks, we treat each node as a distinct variable, extract its regulatory inputs and target outputs according to network dependencies, then enumerate input combinations up to a predefined threshold to simulate state correspondence. This process yields a rich collection of naturally-occurring, meaningful Boolean functions.

3.2 Synthetic Dataset Generation

While real-world datasets provide relevance, they are limited in scale and diversity. To overcome this, we developed a novel two-stage efficient and scalable algorithm that can generate large-scale, complex, and structurally diverse *ground-truth* logic networks from scratch. This approach bypasses the limitations of existing methods that rely on pre-computed libraries or computationally expensive truth-table evaluations.

3.2.1 Stage 1: Small-Scale Network Construction

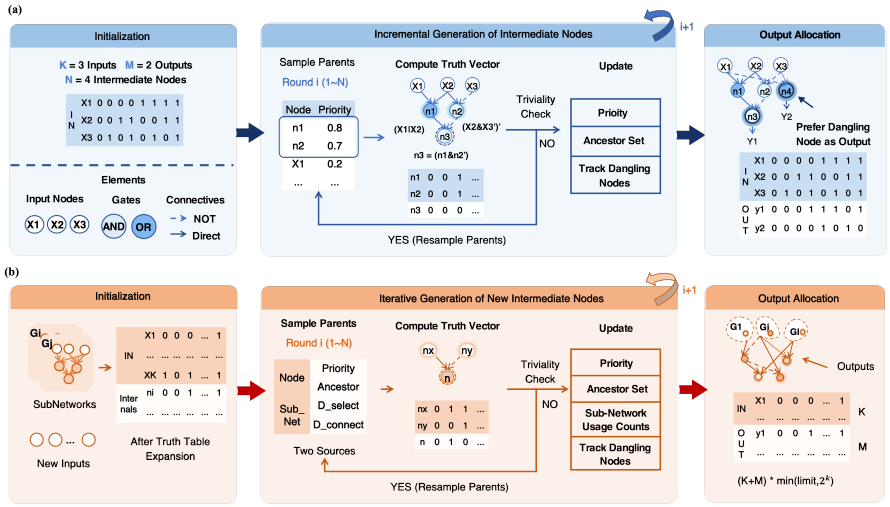

Figure 2: The procedure of logic network generation. (a) Stage 1 (small-scale generation) produces the building blocks used in large-scale synthesis, and (b) Stage 2 merges them into complex networks.

Each small network $\mathcal{G}_j(K_j, L_j, M_j)$ (K_j input, M_j intermediate nodes, and L_j nodes selected from them as outputs.) is generated through an iterative process. We first initialize K_j input nodes, and enumerate all of their 2^{K_j} truth combinations. All input nodes are initialized with priority p_0.

The iterative generation process encompasses priority updates, control of dangling nodes, and verifying triviality. Dangling nodes are those that do not exist on any end-to-end path connecting the network inputs to any designated output; while trivial nodes are characterized by outputs that convey minimal information, exemplified by truth tables that consistently yield 0 or 1.

Weighted Parent Node Selection. We only involve binary connectives, so any intermediate node has exactly two parent nodes. We first sample parent c_0 then c_1 from existing nodes or only dangling ones if their count exceeds L_j. Both are sampled by priority, but the weight of c_1 will be degraded if it is an ancestor of c_0. The new node is then connected to its parents with random edge inversions, and the truth table of v is computed. If the resulting logic is trivial, parents or edges will be resampled until nontrivial or a trial limit is reached. Newly created nodes are assigned a layer-aware priority:

$$\text{priority}(v) = p_0 + \sqrt{\text{layer}(v)/p_0} \tag{1}$$

We prioritize deeper nodes for semantically richer expressions, while parent priorities are diminished by p_0 to encourage diversity and natural complexity growth. The process loops for M_j steps, generating all M_j intermediate nodes.

Output Allocation. Ultimately, outputs are allocated by a preference sequence $\langle$ nontrivial-dangling, nontrivial-non-dangling, trivial-dangling, trivial-non-dangling $\rangle$, with the latter categories being utilized only when the former ones are inadequate. In each set, outputs are sampled using a layer-aware priority weight, and a random inversion is applied to each output. As the logic of non-dangling nodes is consistently encapsulated within dangling nodes, these strategies promote output diversity.

Key Comparison. To avoid redundancy, we adopted a key comparison strategy based on [52], where each truth vector is treated as a binary value (taking the inversion vector if its value is smaller), and the network concatenates the sorted truth vectors of all outputs as the key. If two key collide, we preserve the simpler one.

The process of Stage 1 is illustrated in Figure 2 (a), and the pseudocode is provided in Appendix A.1.1.

3.2.2 Stage 2: Large-Scale Network Synthesis

Large networks $\mathcal{G}(K, L, M)$ are constructed by merging pre-generated sub-networks $\{\mathcal{G}_1, \cdots, \mathcal{G}_\eta\}$ of arbitrary sizes and amounts, provided the sum of (K_j, L_j, M_j) does not exceed the large network's capacity respectively. In addition, the truth tables of each $\mathcal{G}_j$ are expanded to a size of $(K_j + L_j) \cdot T$ with $T < 2^K$, rather than enumerating all possible input combinations. These expanded tables are then concatenated according to their input combinations, a process we refer to as truth-table expansion, as shown in Figure 3. During merged network generation, new inputs are initialized with a priority p_0 and a partial truth vector of length T together covering T combinations.

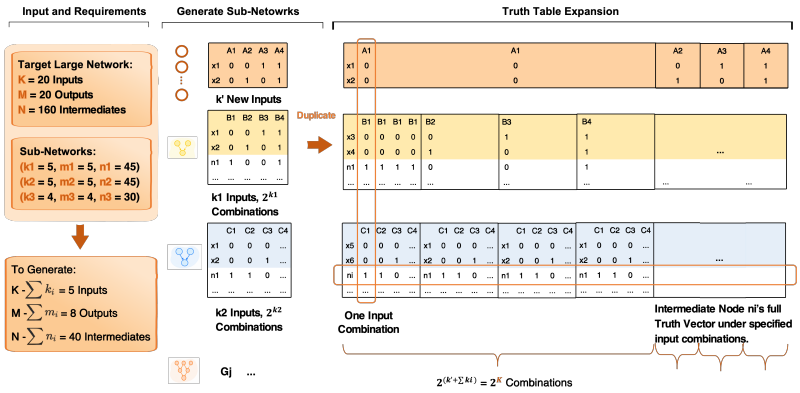

Figure 3: The procedure of truth-table expansion in Stage 2.

Weighted Parent Node Selection and Trivial Check. For each new gate, parent nodes are sampled by one of these three modes, based on sub-network usage: sampling both from new nodes, one from existing networks and one from new nodes, or both from underutilized sub-networks. Node selection is based on the weight equation:

$$w(v) = \text{trivial(v)} \cdot \left(\text{priority}(v) + \sqrt{|\text{Net_Anc}(v)|/p_0}\right)^{\alpha} \tag{2}$$

where $\alpha > 1$ is an amplification factor, and $|\text{Net_Anc}(v)|$ represents the ancestor sub-network set size of node v (i.e., number of sub-networks that contain at least one ancestor node of v). Nodes with a trivial truth table are set to zero weight, ensuring effectiveness of synthesis. This equation measures complementary dimensions: priority captures vertical symbolic complexity, while $|\text{Net_Anc}(v)|$ captures variable coverage horizontally. Since complex logic is easily canceled out by overlapping symbols, variable breadth compensates for the risk from growing depth.

Finally, after iterative generation of new nodes, outputs are assigned using the same strategy in Stage 1. This process of large-scale network synthesis is illustrated in Figure 2 (b), and the pseudocode is provided in Appendix A.1.2. As synthetic data mimics real-world distributions while overcoming real-world scalability limitations, we discuss their relationships in Appendix A.2.3.

3.3 Data Imperfection Modeling

Noise Model. Bit-flip perturbations [3] can simulate the systematic errors observed in biological measurements and logic circuits. In LogicSR, we use the fixed-budget noise model [34], which allows the noise magnitude at each preset level to be controlled precisely and reproducibly. Specifically, for each Boolean function $y_j : \{0,1\}^k \to \{0,1\}$ with k input variables, we randomly select exactly $\lfloor \eta \cdot 2^k \rfloor$ input combinations in its truth table and flip the corresponding output values, where η denotes the noise ratio. Let S_j be the selected subset of inputs (with $|S_j| = \lfloor \eta \cdot 2^k \rfloor$). The noisy output function $\tilde{y}_j(x)$ is then defined as:

$$\tilde{y}_j(x) = \begin{cases} y_j(x) \oplus 1, & \text{if } x \in S_j, \\ y_j(x), & \text{otherwise.} \end{cases} \tag{3}$$

Incomplete Data Sampling. To emulate realistic settings where only partial observations are available [10], LogicSR incorporates incomplete truth-table sampling for BBM and MIMO. For the biological dataset (BBM), if a gene's regulatory function contains no more than 12 inputs, we construct its complete truth table; otherwise, we uniformly sample a subset of input-output combinations from the full input space $\{0,1\}^k$, to mimic the sparse subset of discrete states in biological networks. For large-scale synthetic Boolean functions (MIMO), the number of input combinations grows exponentially with input dimensions, making the full enumeration of 2^k combinations computationally infeasible for large k. To avoid this combinatorial explosion, we set a threshold L. When $k \leq L$, we generate complete truth tables; otherwise, we uniformly sample 2^L input combinations from the full input space $\{0,1\}^k$ to obtain a partially observed truth table.

3.4 LogicSR-Core: A Lightweight Evaluation Set

Running all evaluation settings is computationally expensive; without a standard lightweight evaluation set or recommended protocol, comparisons could be less consistent. We address this by providing a fixed lightweight set and a standardized protocol. LogicSR-Core (v1) is a dedicated subset that can be evaluated through a single predefined setting. Its membership is fixed by one scale criterion (at most 10 inputs and 10 outputs), ensuring consistent comparison across suites. It contains 370 instances, $\sim$6% of the 6,132 truth tables. Inputs 20/40/80, the intermediate noise level, and the hard tiers (excluded from Core by construction) remain the settings for scalability and robustness. The composition, manifest and evaluation protocol are detailed in Appendix A.3.

4 Experiments

We conduct all experiments on a server equipped with 8 NVIDIA A800-SXM4-80GB and Intel Xeon Platinum 8358 (2.60 GHz, 64 cores, 128 threads). Released instances are for testing only: each method fits the observed truth-table entries and is evaluated on held-out ones (3:1 split when the input count exceeds 7; otherwise the full table is used for both to preserve information integrity). Methods needing training data should generate it independently, e.g., with our generator (Boolformer uses its released pretrained weights without fine-tuning). For benchmark construction, we set $L = 15$ and the noise levels to 0%, 1%, and 5%. All input nodes are initialized with the same priority value $p_0 = 3$, and the priority decay coefficient μ is also set to p_0. We choose 18 representative algorithms from 4 categories, which are listed below:

- **Logic Synthesis Methods:** ABC [5], BDD [25], Espresso [42] and Quine-McCluskey [40, 30];
- **Symbolic Regression Methods:** PySR [13], Operon [7], DEAP [17], GPlearn [44] and TenGP [26];
- **Neural Network-Based Methods:** DiffLogic [39], AI4EDA_TNet [51], NeuraLUT [2], SATNet [50] and Boolformer [14];
- **LLM-Based Methods:** Feeding the input–output samples to large language models (GPT-4o [35] and GPT-o3 [36]) and prompting the model to generate expressions that fit the samples. We also incorporate the Program-of-Thought (PoT) reasoning paradigm [38] as additional LLM baselines.

We evaluate the performance of these baselines using the LogicSR benchmark, which includes the aforementioned real-world datasets (BBM and IWLS) and synthetic datasets. The synthetic datasets consist of two expression types: ANO (expressions constructed with `AND`, `NOT` and `OR` gates) and AN (expressions constructed with `AND` and `NOT` gates).

Fairness. Both axes of fairness are controlled by design and verified empirically: the operator set is fixed identically across methods per suite, and the metric is blind to output representation (Appendix A.4.3).

We have designed the following unified evaluation metrics (see Appendix A.6.1 $\sim$ A.6.2 for details):

- **Complexity of Generated Network:** This metric is evaluated by counting the total number of logical operators (`AND`, `NOT` and `OR` if applied) in the logic network generated for each dataset, divided by the number of outputs.
- **In-Distribution / Out-of-Distribution Accuracy:** Accuracy is evaluated at three granularities: bit-wise accuracy, sample-wise accuracy (a sample is correct only if all of its outputs match), and per-output correctness (POC; an output is correct only if all of its bits match).
- **Efficiency:** This metric reflects the total processing time required by an algorithm to produce the final expression or output predictions based on a given dataset. For neural network-based methods, this metric is mainly about the model training time.

Note that this section reports the performance of baselines on LogicSR, while an analysis of network synthesis algorithms is deferred to Appendix A.2.

4.1 Quantitative Results Across Datasets and Scales

Table 2 summarizes key observations on the adaptability and scalability of baselines across synthetic and real-world datasets, emphasizing exact fitting for each expression output.

Traditional logic synthesis methods maintain 100% training accuracy for small-to-medium scales (≤ 10 input variables). However, methods such as ABC, Quine-McCluskey, and BDD rely heavily on exact fitting to training data, resulting in poor generalization and explosive computational costs at larger scales due to redundant rules. Thus, these methods are best suited for small-scale (≤ 5) discrete logical tasks. Espresso, however, surpasses this limitation, demonstrating excellent accuracy and generalization capability at input scales up to 20; nevertheless, it remains constrained by the representational bottleneck of its original symbolic output.

Symbolic regression and neural network-based methods leverage formulaic representations and model training, respectively, thereby possessing generalization advantages and, for some of them, compatibility with larger-scale truth tables. In detail, PySR, Operon, SATNet and Boolformer achieve higher accuracy on small-scale datasets. On in10_out10, PySR and Operon clearly exceed the classical GP baselines. Both DiffLogic and TenGP exhibit low accuracy, which makes them unsuitable for logical symbolic regression tasks.

LLM-based methods are severely limited by context length constraints, rendering them incapable of handling large-scale data. However, at small scale, GPT-o3 demonstrates strong performance, generating precise expressions with low redundancy. GPT-4o, in contrast, lags significantly behind other baselines, revealing fundamental weaknesses in logical reasoning capabilities. PoT multi-step reasoning improves the accuracy of GPT-4o and GPT-o3, but at the cost of producing significantly more complex expressions.

4.2 Comprehensive Evaluation of Baseline Performance

This part presents a comprehensive evaluation of algorithmic accuracy, robustness, scalability, and efficiency. Figure 4 investigates the scalability limits of different methods, together with their accuracy and robustness under reasonable input scales, while Figure 5 focuses on time efficiency and analyzes how overall accuracy changes with increasing scale.

Under noise-free (noise=0) conditions illustrated by Figure 4(a), the relationship between input scale and test sample-wise accuracy reveals the logical complexity boundaries for various modeling paradigms. Among traditional logic synthesis methods, surprisingly, Espresso maintains high accuracy for input scales ranging from 5 to 20. In contrast, ABC, Quine-McCluskey and BDD are

Table 2: Methods' performance on datasets (without noise). The top row in each cell is the train and test accuracy (no test accuracy for in5_out5 because small-scale networks do not have a test split). The bottom value in parentheses is the complexity. "–" means that the method cannot handle this scale, because of the time limit or model constraints, while "N/A" means the method was not tested due to operator set incompatibility.

		Synthetic ANO			BBM	Synthetic AN			IWLS
		in5_out5	in10_out10	in20_out20	(ANO)	in5_out5	in10_out10	in20_out20	(AN)
SR	DEAP	51.1 (7.5)	5.9 7.1 (9.2)	–	18.4 18.3 (6.5)	39.0 (5.8)	4.4 5.6 (7.8)	–	5.5 5.5 (17.1)
	GPlearn	36.7 (3.8)	5.7 6.9 (3.2)	4.2 4.3 (1.1)	15.5 15.3 (3.1)	28.5 (3.3)	3.7 4.8 (3.0)	3.3 3.3 (0.9)	12.5 12.6 (5.3)
	TenGP	0.1 (11.7)	0.0 0.0 (8.8)	–	0.0 0.0 (8.2)	0.1 (9.8)	0.0 0.0 (8.3)	–	0.1 0.3 (8.2)
	PySR	99.3 (3.1)	63.6 65.1 (6.6)	–	20.3 20.3 (1.8)	95.7 (3.1)	45.0 46.5 (5.9)	–	24.2 24.3 (6.6)
	Operon	96.9 (10.5)	58.3 59.0 (21.0)	–	25.2 25.2 (5.1)	90.4 (12.4)	42.4 42.9 (22.0)	–	26.7 26.6 (22.7)
NN	DiffLogic	18.2 (110.4)	1.1 2.2 (206.4)	2.3 2.4 (1821.4)	7.4 7.5 (256.2)	15.7 (110.4)	0.7 1.9 (206.4)	2.6 2.6 (1821.4)	6.3 6.3 (614.4)
	AI4EDA_TNet	26.0 (186.4)	2.0 2.8 (192.4)	–	8.7 8.7 (141.2)	28.3 (186.4)	0.3 1.6 (192.4)	–	6.1 6.2 (499.6)
	NeuraLUT	63.7 (6.4)	79.0 70.5 (102.4)	–	28.0 27.1 (51.2)	64.3 (6.4)	82.1 71.7 (102.4)	–	27.2 26.4 (409.6)
	Boolformer	100.0 (5.0)	83.1 80.1 (20.4)	–	18.2 18.2 (3.5)	N/A	N/A	N/A	N/A
	SATNet	100.0 (40.0)	78.1 72.7 (80.0)	72.7 73.9 (160.0)	41.3 40.2 (102.4)	99.8 (40.0)	71.5 66.9 (80.0)	73.2 74.4 (160.0)	32.7 31.8 (121.6)
Logic	ABC	N/A	N/A	N/A	N/A	100.0 (5.8)	100.0 0.8 (501.2)	–	43.3 7.4 (717.6)
	BDD	100.0 (56.8)	100.0 0.5 (5718.9)	–	21.5 10.9 (2416.9)	N/A	N/A	N/A	N/A
	Espresso	100.0 (6.2)	100.0 84.6 (22.4)	100.0 93.7 (66.3)	48.4 48.4 (10.8)	N/A	N/A	N/A	N/A
	Quine-McCluskey	100.0 (6.5)	100.0 0.5 (912.7)	–	21.5 10.8 (363.7)	N/A	N/A	N/A	N/A
LLM	GPT-o3	90.1 (6.9)	–	–	34.1 34.0 (2.8)	88.0 (9.5)	–	–	21.7 21.7 (89.9)
	GPT-4o	1.6 (6.4)	–	–	0.0 0.0 (5.9)	2.1 (5.5)	–	–	0.3 0.3 (23.3)
	GPT-o3 (PoT)	97.1 (41.0)	–	–	39.5 39.4 (30.2)	95.0 (44.8)	–	–	29.2 29.3 (102.6)
	GPT-4o (PoT)	16.8 (21.7)	–	–	5.1 5.1 (10.2)	17.4 (21.5)	–	–	5.4 5.3 (67.1)

suitable only for very small input sizes (≤ 5), with their accuracies sharply declining to nearly zero as the input scale increases to 10 (in5_out5 networks do not have a test split), indicating severe limitations in generalization. Most symbolic regression and neural network methods experience accuracy decline as scale increases, with DiffLogic and GPlearn capable of handling larger datasets.

Based on the comparison of bit-wise accuracy and sample-wise accuracy shown in Figure 4(b), with an input size of 10, output size of 10, and under noise-free conditions, we observe that bit-level accuracy tends to be higher due to the tolerance inherent in bit-wise local matching, whereas sample-wise accuracy decreases because of the stringent requirement that all outputs of a sample match. This reveals a common issue among methods, wherein a few bit-wise errors frequently make entire samples incorrect. Notably, Espresso, SATNet, NeuraLUT, PySR and Operon maintain good accuracy and generalization performance at both granularity levels.

In the noise scenario depicted in Figure 4(c), symbolic regression methods exhibit comparatively stronger robustness under noise on larger inputs, partly because their search focuses on discovering compact structural patterns rather than exactly fitting every individual sample. As a result, small perturbations do not necessarily alter the preferred structural solutions. In contrast, logic synthesis methods are built to ensure exact consistency with the input truth table: every provided input–output assignment must be satisfied. When the truth table contains noise, these corrupted entries act as

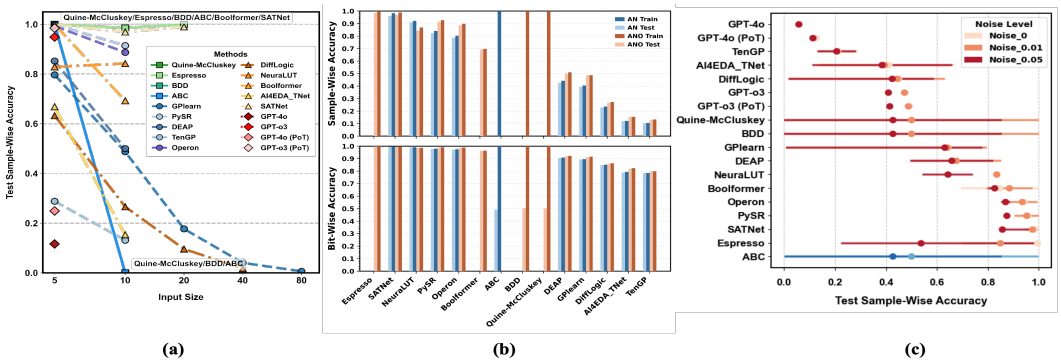

Figure 4: The correlation between the accuracy and different factors for each tested method. (a) Accuracy on logic networks with varying input sizes (without noise); (b) Sample-wise and bit-wise accuracy on AN and ANO datasets (input = 10); (c) Accuracy on logic networks with varying noise levels (averaged on input=5/10). In (a) and (c), the accuracy is measured on the ANO dataset for all methods except ABC, which is measured on the AN dataset due to its limited applicability.

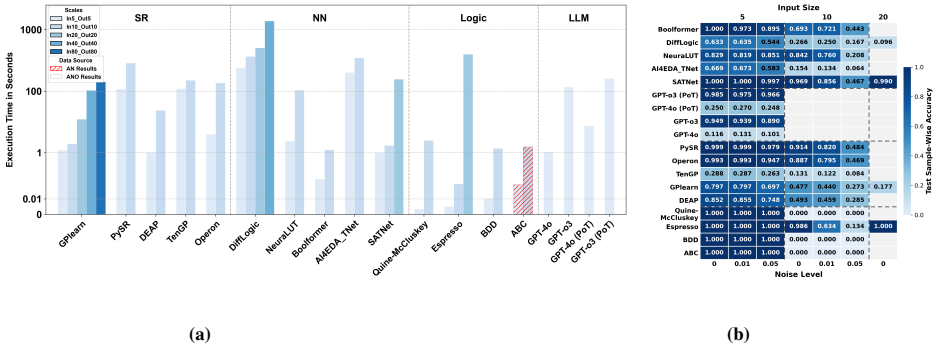

Figure 5: Comparison of baseline methods. (a) Computational efficiency under different scales; (b) Sample-wise fidelity under varying input sizes and noise levels. In (a) and (b), the ABC* method only supports synthetic AN data, while the other 17 methods are evaluated on synthetic ANO.

constraints, causing the synthesized expressions to deviate from the ground-truth function as the noise level increases.

Figure 5(a) evaluates the time efficiency of the baseline methods on the synthetic dataset. From this figure, SR methods show a substantial increase in runtime (from 10^0 to 10^3 seconds), indicating high sensitivity to the input scale. This is because SR methods, run here in per-output mode, perform a separate iterative search for each output, and the search space of each search expands rapidly as the input scale increases; running PySR in its multi-output mode does not reduce this cost (Appendix A.4.8). Traditional logic synthesis methods, leveraging rule-based derivation mechanisms, simultaneously generate multiple outputs in a single run or in parallel, thus exhibiting the shortest and most stable processing time at small scales (e.g., in5_out5).

Figure 5(b) reports sample-wise fidelity, defined as accuracy measured against the provided (potentially noisy) truth table, reflecting consistency with observed data rather than recovery of the original clean expressions. We observe that classical GP baselines and several neural network-based methods are strongly affected by increasing input scale, Quine-McCluskey, BDD and ABC degrade primarily due to their limited generalization capability, and Espresso is mainly affected by noise.

5 Related Work

The present research lies at the intersection of symbolic regression, logic synthesis, and AI benchmarking.

Symbolic Regression. Symbolic Regression (SR) involves identifying a mathematical expression in symbolic form that best fits a given dataset [27]. The search space for SR encompasses all possible expressions, which is infinite and complex. Genetic Programming (GP) has been the predominant method for SR since its popularization by Koza [23]. In GP, a population of candidate expressions is evolved over generations using operations like selection, crossover, and mutation to enhance their fit to the data.

Nevertheless, the SR community has traditionally focused on finding functions in continuous domains, using a basis set of arithmetic operators ($+, -, \times, \div$), analytic functions ($\sin, \cos, \exp, \log$), and real-valued constants. Consequently, widely used SR benchmarks, such as Koza's original SR test suite [23] and the SRBench suite [9], primarily feature regression tasks over real numbers. The evaluation of SR for discrete or Boolean domains remains underexplored, and a standardized, large-scale benchmark for logical discovery has been a notable omission in the field. Our work directly addresses this gap.

Logic Synthesis. Logic Synthesis is a core component of Electronic Design Automation (EDA) that transforms an abstract specification of a digital circuit's behavior, typically a Register-Transfer Level (RTL) description, into an optimized logic gate implementation [24, 41]. The primary goals of logic synthesis are to minimize metrics such as circuit area, power consumption, and signal delay, while strictly preserving the functional correctness specified by a Boolean function or network.

Well-known tools in this area, such as ABC [5], are highly effective for optimization but operate under the assumption of a complete and exact logical specification. The benchmarks used in this field, like the EPFL combinational circuits [1] and IWLS [12] contest designs, consist of established digital circuit designs. The evaluation paradigm is not concerned with learning from incomplete data samples or generalizing to unseen inputs, which are central objectives in machine learning. Our benchmark, in contrast, is specifically designed to evaluate these learning-based criteria, making it suitable for assessing modern approximate and data-driven methods that are not the focus of traditional LS.

Neuro-Symbolic Methods and Benchmarking. The integration of neural networks with symbolic reasoning, known as neuro-symbolic AI, is a rapidly growing field [48, 15]. These methods aim to combine the robust learning and pattern recognition capabilities of neural networks with the interpretability and reasoning power of symbolic systems. Many neuro-symbolic models are designed to address problems that involve discovering underlying rules or programs from data.

Progress in AI has often been driven by the introduction of high-quality, large-scale benchmarks [16, 49]. The benchmarks offer a shared basis for comparing various approaches, monitoring progress, and revealing the weaknesses of current methods. Our work advances this tradition by offering a unified and scalable benchmark for logical symbolic regression, which we believe will be crucial in developing and assessing the next generation of neuro-symbolic systems designed for logical discovery.

6 Conclusion

In this paper, we introduced LogicSR, a unified and scalable benchmark dedicated to logical symbolic regression. LogicSR addresses a critical gap between the fields of continuous symbolic regression and exact logic synthesis by creating a diverse suite of benchmark tasks. Using LogicSR, we conducted an extensive evaluation of 18 algorithms, spanning classical solvers, modern machine learning models, and large language models. Our analysis provides a clear map of the current state of the art, revealing that existing methods face significant challenges with scalability, noise robustness, and logical operator compatibility. We hope to offer a robust foundation to benchmark progress, foster innovation across disparate research communities, and steer the future development of powerful and interpretable neuro-symbolic systems.

Acknowledgments and Disclosure of Funding

This research was supported in part by the National Natural Science Foundation of China under Grant No. 62272261 and Wuxi Research Institute of Applied Technologies, Tsinghua University under Grant 20242001120.

References

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, pages 1–6, 2015.

[2] Marta Andronic and George A. Constantinides. NeuraLUT-Assemble: Hardware-aware assembling of sub-neural networks for efficient LUT inference. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 208–216. IEEE, 2025. doi: 10.1109/FCCM62733.2025.00077.

[3] Robert C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, 2005. doi: 10.1109/TDMR.2005.853449.

[4] Armin Biere, Nils Froleyks, and Mathias Preiner. Hardware model checking competition (hwmcc'24). `https://hwmcc.github.io/2024/`, 2024. Accessed: 2025-09-25.

[5] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40, 2010. doi: 10.1007/978-3-642-14295-6_5.

[6] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 1929–1934, 1989. doi: 10.1109/ISCAS.1989.100747.

[7] Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. Operon c++: An efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, page 1562–1570, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371278. doi: 10.1145/3377929.3398099.

[8] Oliver Cassidy, Marta Andronic, Samuel Coward, and George A. Constantinides. Reducedlut: Table decomposition with "don't care" conditions. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 36–42, 2025. doi: 10.1145/3706628.3708823.

[9] William La Cava, Bogdan Burlacu, Marco Virgolin, Michael Kommenda, Patryk Orzechowski, Fabrício Olivetti de França, Ying Jin, and Jason H. Moore. Contemporary symbolic regression methods and their relative performance. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, volume 1, pages 1–16, 2021.

[10] Stéphanie Chevalier, Julia Becker, Yujuan Gui, Vincent Noël, Cui Su, Sascha Jung, Laurence Calzone, et al. Data-driven inference of boolean networks from transcriptomes to predict cellular differentiation and reprogramming. *npj Systems Biology and Applications*, 11(1):105, 2025. doi: 10.1038/s41540-025-00569-z.

[11] Animesh Basak Chowdhury, Marco Romanelli, Benjamin Tan, Ramesh Karri, and Siddharth Garg. INVICTUS: optimizing boolean logic circuit synthesis via synergistic learning and search. *arXiv preprint arXiv:2305.13164*, 2023.

[12] IWLS Contest. International workshop on logic & synthesis programming contest. `https://www.iwls.org/contest/`. Accessed: 2025-12-27.

[13] Miles Cranmer. Interpretable machine learning for science with PySR and symbolicregression.jl. *arXiv preprint arXiv:2305.01582*, 2023.

[14] Stéphane d'Ascoli, Arthur Renard, Emmanuel Abbe, Vassilis Papadopoulos, Samy Bengio, and Joshua M. Susskind. Boolformer: Symbolic regression of logic functions with transformers. In *ICML Workshop*, 2025.

[15] Lauren Nicole DeLong, Ramon Fernández Mir, and Jacques D. Fleuriot. Neuro-symbolic ai for reasoning over knowledge graphs: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 36(5):7822–7842, 2025. doi: 10.1109/TNNLS.2024.3420218.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[17] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012. doi: 10.5555/2503308.2503311.

[18] Konstantinos Iordanou, Timothy Atkinson, Emre Ozer, Jedrzej Kufel, Grace Aligada, John Biggs, Gavin Brown, and Mikel Luján. Low-cost and efficient prediction hardware for tabular data using tiny classifier circuits. *Nature Electronics*, 7(5):405–413, 2024. doi: 10.1038/s41928-024-01157-5.

[19] Jiaqing Jiang, Xiaoming Sun, Yuan Sun, Kewen Wu, and Zhiyu Xia. Structured decomposition for reversible boolean functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2410–2421, 2020. doi: 10.1109/TCAD.2019.2928974.

[20] Claus Kadelka and Kishore Hari. Critical assessment of the ability of boolean threshold models to describe gene regulatory network dynamics. *PNAS Nexus*, 4(8):pgaf228, 07 2025. ISSN 2752-6542. doi: 10.1093/pnasnexus/pgaf228.

[21] Claus Kadelka, Taras-Michael Butrie, Evan Hilton, Jack Kinseth, Addison Schmidt, and Haris Serdarevic. A meta-analysis of Boolean network models reveals design principles of gene regulatory networks. *Science Advances*, 10(2):eadj0822, 2024. doi: 10.1126/sciadv.adj0822.

[22] Roman Kalkreuth, Zdenek Vasícek, Jakub Husa, Diederick Vermetten, Furong Ye, and Thomas Bäck. General boolean function benchmark suite. In *Proceedings of the 17th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*, pages 84–95. ACM, 2023. doi: 10.1145/3594805.3607131.

[23] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. ISBN 0-262-11170-5.

[24] Pran Kurup and Taher Abbasi. *Logic Synthesis Using Synopsys*. Springer, 1997.

[25] C.Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959. doi: 10.1002/j.1538-7305.1959.tb01585.x.

[26] Jaroslav Loebl. Tengp - cartesian genetic programming with numpy (or pytorch). `https://jarino.github.io/tengp/`, 2018. Accessed: 2026-11-06.

[27] Nour Makke and Sanjay Chawla. Interpretable scientific discovery with symbolic regression: A review. *Artificial Intelligence Review*, 57(1):2, 2024. doi: 10.1007/s10462-023-10622-0.

[28] Seyed Amir Malekpour, Amir Reza Alizad-Rahvar, and Mehdi Sadeghi. Logicnet: Probabilistic continuous logics in reconstructing gene regulatory networks. *BMC Bioinformatics*, 21(1):318, 2020. doi: 10.1186/s12859-020-03651-x.

[29] Joao Marques-Silva and Alexey Ignatiev. Delivering trustworthy ai through formal xai. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 12342–12350, 2022. doi: 10.1609/aaai.v36i11.21499.

[30] E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956. doi: 10.1002/j.1538-7305.1956.tb03835.x.

[31] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, volume 2, pages 1135–1142, 1999. doi: 10.5555/2934046.2934074.

[32] T. Nathan Mundhenk, Mikel Landajuela, Ruben Glatt, Cláudio P. Santiago, Daniel M. Faissol, and Brenden K. Petersen. Symbolic regression via neural-guided genetic programming population seeding, 2021.

[33] Răzvan Nane, Vlad Mihai Sima, C. Michael Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Dean Brown, Fabrizio Ferrandi, Jason Helge Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. doi: 10.1109/TCAD.2015.2513673.

[34] Nagarajan Natarajan, Inderjit S. Dhillon, Pradeep Ravikumar, and Ambuj Tewari. Learning with noisy labels. In *Advances in Neural Information Processing Systems*, volume 26, pages 1196–1204, 2013.

[35] OpenAI. Gpt-4 technical report, 2024.

[36] OpenAI. Introducing OpenAI o3 and o4-mini. `https://openai.com/index/introducing-o3-and-o4-mini/`, 2025. Accessed: 2025-12-27.

[37] Samuel Pastva, David Šafránek, Nikola Beneš, Luboš Brim, and Thomas Henzinger. Repository of logically consistent real-world Boolean network models. *bioRxiv*, 2023. doi: 10.1101/2023.06.12.544361.

[38] Patomporn Payoungkhamdee, Pume Tuchinda, Jinheon Baek, Samuel Cahyawijaya, Can Udomcharoenchaikit, Potsawee Manakul, Peerat Limkonchotiwat, Ekapol Chuangsuwanich, and Sarana Nutanong. Towards better understanding of program-of-thought reasoning in cross-lingual and multilingual environments. In *Annual Meeting of the Association for Computational Linguistics*, pages 15810–15828, 2025. doi: 10.18653/v1/2025.findings-acl.817.

[39] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Deep differentiable logic gate networks. In *Advances in Neural Information Processing Systems*, volume 35, pages 2006–2018, 2022.

[40] Willard Van Orman Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952. doi: 10.2307/2308219.

[41] Robert Bryan Reese and Mitchell Aaron Thornton. *Introduction to Logic Synthesis Using Verilog HDL*. Morgan & Claypool Publishers, 2006.

[42] Richard L. Rudell and Alberto Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987. doi: 10.1109/TCAD.1987.1270318.

[43] Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D. Doan, and Chandan K. Reddy. LLM-SRBench: A new benchmark for scientific equation discovery with large language models. In *Forty-second International Conference on Machine Learning (ICML)*, 2025. `https://openreview.net/forum?id=SyQPiZJVWY`.

[44] Trevor Stephens et al. Genetic programming in python, with a scikit-learn inspired api. `https://github.com/trevorstephens/gplearn`, 2015. Accessed: 2026-11-06.

[45] Ajay Subbaroyan, Olivier C. Martin, and Areejit Samal. Minimum complexity drives regulatory logic in boolean models of living systems. *PNAS Nexus*, 1(1):pgac017, 2022. doi: 10.1093/pnasnexus/pgac017.

[46] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. Smapproxlib: Library of FPGA-based approximate multipliers. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, pages 1–6, 2018. doi: 10.1109/DAC.2018.8465845.

[47] Marco Venere, Alessandro Barenghi, and Gerardo Pelosi. A quantum method to match vector boolean functions using simon's solver. In *Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 542–549, 2024. doi: 10.1109/ICCD63220.2024.00089.

[48] Zishen Wan, Che-Kai Liu, Hanchen Yang, Ritik Raj, Chaojian Li, Haoran You, Yonggan Fu, Cheng Wan, Ananda Samajdar, Yingyan Celine Lin, Tushar Krishna, and Arijit Raychowdhury. Towards cognitive ai systems: Workload and characterization of neuro-symbolic ai. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 268–279, 2024. doi: 10.1109/ISPASS61541.2024.00033.

[49] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, 2018. doi: 10.18653/v1/W18-5446.

[50] Po-Wei Wang, Priya Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97, pages 6545–6554, 2019.

[51] Zhihai Wang, Jie Wang, Qingyue Yang, Yinqi Bai, Xing Li, Lei Chen, Jianye Hao, Mingxuan Yuan, Bin Li, Yongdong Zhang, and Feng Wu. Towards next-generation logic synthesis: A scalable neural circuit generation framework. In *Advances in Neural Information Processing Systems*, volume 37, pages 99202–99231, 2024. doi: 10.52202/079017-3147.

[52] Xuliang Zhu, Ruofei Tang, Lei Chen, Xing Li, Xin Huang, Mingxuan Yuan, Weihua Sheng, and Jianliang Xu. A database dependent framework for k-input maximum fanout-free window rewriting. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023. doi: 10.1109/DAC56929.2023.10247727.

A Appendix

Appendix Outline

- **A.1. Pseudocode for Logical Network Synthesis**
  - A.1.1. Generating a Single Small-Scale Logic Network
  - A.1.2. Merging Pre-Generated Logic Networks
- **A.2. Evaluation of Synthetic Datasets**
  - A.2.1. Analysis of Benchmark Complexity
  - A.2.2. Parameter Analysis
  - A.2.3. Real-World Relevance
- **A.3. LogicSR-Core**
  - A.3.1. LogicSR-Core Characterization
  - A.3.2. How Many Instances Are Needed for a Reliable Comparison?
- **A.4. Considerations for Baseline-Related Experiments**
  - A.4.1. Additional Experimental Settings
  - A.4.2. Criteria for Baseline Selection
  - A.4.3. Fairness: Operator Sets and Output Formats
  - A.4.4. Discussion of Experimental Results
  - A.4.5. PySR Configuration, Inputs and Outputs
  - A.4.6. Sensitivity of PySR to maxsize
  - A.4.7. Re-evaluation of PySR Search Hyperparameters
  - A.4.8. PySR in Multi-Output Mode
- **A.5. Detailed Descriptions of Baselines**
- **A.6. Illustration of Evaluation Metrics**
  - A.6.1. Calculation of Complexity
  - A.6.2. Calculation of Accuracy
  - A.6.3. Explanation of Time Efficiency
- **A.7. Analysis of Evaluation Approaches for Baselines**
- **A.8. Discussion of Downstream Applications**
- **A.9. Extending LogicSR Beyond Boolean Functions Using Basic Logic Gates**
  - A.9.1. Multivalued and Fuzzy Logic
  - A.9.2. Implication Operators

- **A.10. Comparison with Other Benchmarks**
- **A.11. Limitation and Future Work**
- **A.12. Dataset Licenses**

A.1 Pseudocode for Logical Network Synthesis

A.1.1 Generating a Single Small-Scale Logic Network

We formally describe the ground-truth small-scale logic network generation process in Algorithm 1. Before network construction, the algorithm specifies the network specifications and initializes the input nodes (lines 1-4). It then iteratively generates M_j gates (lines 5–29), and finally ensures O_j outputs through a designated output selection process (line 30).

Each iteration consists of two stages : *generating and connecting a new node* and *computing truth vectors*.

Generating and Connecting a New Node. In the connection node generation stage, a new node v_i with gate type g_i is created (line 6). The candidate pool $\mathcal{C}$ is set to the dangling nodes if dangling number suffices for the required number of outputs; otherwise, we initialize all nodes as candidate (line 7). Inversion (inv_0, inv_1) to v_i's parenting nodes (c_0, c_1) are initialized (line 8). Then, the first parent c_0 of v_i is sampled from the pool weighted by priority, and removed from the (copy of the) pool (line 9).

After that, the algorithm enters the stage for selection of the second parent c_1 (lines 10-19), and weights are computed for every candidate node $v_x \in \mathcal{C}$. There are three cases: 1) If v_x is not an ancestor of c_0, its weight is directly set as the priority (lines 11-12); 2) If it appears only in one polarity on c_0's ancestor path, alignment of v_x with its previous polarity is enforced, and weight is set by a decayed factor $\text{priority}(v_x)/p_0$ (lines 13-15); 3) Otherwise, the priority is set as zero for the node, which must be trivial under this case (lines 16-18).

Computing Truth Vectors. After sampling both of the parents, the truth vector of v_i is computed by $(\text{truth}(c_0)^{inv_0}, \text{truth}(c_1)^{inv_1})$. If the truth vector is trivial, resampling is repeated until a nontrivial configuration is found or the maximum number of trials is reached (lines 21-25). Once condition met, v_i is added to $\mathcal{G}_j$ by connecting to c_0, c_1, and ancestor set $\text{Anc}(v_i)$ of v_i is set as the union $\text{Anc}(c_0) \cup \text{Anc}(c_1) \cup \{v_i\}$ (line 26). $\{c_0, c_1\}$ is removed from the Dangling set $\mathcal{H}_j \leftarrow \mathcal{H}_j \cup \{v_i\} \setminus \{c_0, c_1\}$, and added to $\mathcal{S}_j \leftarrow \mathcal{S}_j \cup \{v_i\}$. The layer of v_i is set to be greater than that of its parent nodes:

$$\text{layer}(v_i) = \max(\text{layer}(c_0), \text{layer}(c_1)) + 1. \tag{4}$$

Priority of v_i is set by a layer-aware rule:

$$\text{priority}(v_i) = p_0 + \sqrt{\frac{\text{layer}(v_i)}{p_0}}, \tag{5}$$

while the priorities of its parents c_0 and c_1 are decayed by p_0 to avoid logical redundancy (line 27). This scheme promotes depth growth and structural diversity. Finally, the new node is added to the network, and the maximum layer depth of the network is updated (line 28).

After M_j iterations, the structure of the network is complete. The output allocation step is then performed by Algorithm 2, which selects an exact number of L_j outputs (line 30).

Network Output Allocation. The goal of *Network Output Allocation* is to assign exactly L_{req} outputs to a given network, in a way that ensures rationality and diversity. The procedure begins by classifying nodes into four categories: nontrivial $\mathcal{N}$, which can be further classfied into dangling nontrivial $\mathcal{D}$ or non-dangling non-trivial; or (non-)dangling trivial $\mathcal{T}$ (line 1).

If the size of $\mathcal{D}$ is already larger than the required outputs L_{req}, outputs are directly sampled from $\mathcal{D}$ by Equation 6 (lines 3-4):

$$\text{output_weight} = \max\left(\text{priority}(v) + \min\left(\frac{\text{layer}(v) - depth/\beta_1}{\beta_2}, 0\right), \epsilon\right) \tag{6}$$

where $depth$ is the maximum layer depth in this network, and $\beta_1, \beta_2 > 1$ are decaying factors and ϵ is a minor constant ($\beta_1 = 6, \beta_2 = 2, \epsilon = 10^{-6}$ in our experiments).

Algorithm 1 Small-Scale Logic Network Generation.

Require: number of inputs K_j, outputs L_j, steps M_j, maximum loop repeats MAX_R
Ensure: constructed network $\mathcal{G}_j$

Initialize K_j input nodes $\mathcal{I}_j$ with priority p_0; set network depth $depth \leftarrow 1$
Initialize set of all nodes $\mathcal{S}_j$ and dangling nodes set $\mathcal{H}_j$ with $\mathcal{I}_j$
Assign each new input in $\mathcal{I}_j$ a full truth vector of length 2^{K_j}, covering all input combinations
Sample gate operations sequence $\{g_1, \ldots, g_{M_j}\}$ uniformly
for $i = 1$ to M_j **do**
  Create node v_i with uniformly sampled gate type $g_i \sim$ Connectives
  Set candidate pool $\mathcal{C} \leftarrow \mathcal{H}_j$ if $|\mathcal{H}_j| \geq L_j - 1$, else $\mathcal{S}_j$
  Initialize inversion flags $(inv_0, inv_1) \sim \{0, 1\}^2$ at random
  Sample parent $c_0 \in \mathcal{C}$ by priority, update $\mathcal{C} \leftarrow \mathcal{C} \setminus \{c_0\}$
  for all $v_x \in \mathcal{C}$ **do** {Assigning weights to candidates.}
    if $v_x \notin$ Anc(c_0) **then**
      $w_{\mathcal{C}}(v_x) \leftarrow$ priority(v_x)
    else if v_x appears only as non-inverted **or** inverted ancestor of c_0 in Anc(c_0) **then**
      $inv_{v_x} \leftarrow$ True if v_x is a inverted ancestor of c_0, else False
      $w_{\mathcal{C}}(v_x) \leftarrow$ priority$(v_x)/p_0$
    else
      $w_{\mathcal{C}}(v_x) \leftarrow 0$
    end if
  end for
  Sample parent $c_1 \in \mathcal{C}$ by $w_{\mathcal{C}}$; set attempts $\leftarrow 0$ {Sample the second parent node based on candidate weights, and initialize the parameter for triviality check.}
  while trivial(truth(v_i)) **and** attempts $<$ MAX_R **do** {Trivial check.}
    Resample (c_0, c_1) from $\mathcal{C}$ by priority
    Compute truth(v_i) with (inv_0, inv_1) reinitialized
    attempts $\leftarrow$ attempts+1
  end while
  Connect v_i to $({c_0}^{inv_0}, {c_1}^{inv_1})$, add to $\mathcal{G}_j$; update Anc(v_i), $\mathcal{H}_j$, $\mathcal{S}_j$
  Update layer(v_i), priority(v_i); decay priority(c_0), priority(c_1)
  $depth \leftarrow \max($layer$(v_i), depth)$ {Update network depth.}
end for
Allocate outputs by Algorithm 2

Otherwise, if $\mathcal{D}$ is insufficient for output allocation, outputs are selected following a preference order $\langle$ nontrivial-dangling, nontrivial-non-dangling, trivial-dangling, trivial-non-dangling $\rangle$ (lines 5-9). Specifically, the algorithm first selects all nodes in $\mathcal{D}$ as output (line 6), then fills any remaining slots from the other categories, and the number required is (line 2):

$$r = L_{req} - |\mathcal{D}|. \tag{7}$$

The algorithm selects outputs first from non-dangling nontrivial nodes (line 7) and trivial nodes as the last choice (line 8). This procedure ensures exactly L_j outputs are allocated, prioritizing structurally meaningful and non-redundant outputs while maintaining robustness in case of insufficient candidates.

Finally, each selected node is marked as an output, with a random inversion inv applied to increase the diversity of logical forms (lines 10-12).

A.1.2 Merging Pre-Generated Logic Networks

Algorithm 3 outlines the process of large-scale network generation based on merging smaller networks. Algorithm 3 takes a sequence of sub-networks $\mathcal{A}$ as input, whose sizes are manually designed, as long as their total sizes (K_j, L_j, M_j) does not exceed the target network's corresponding sizes (K, L, M) (the required number of inputs, outputs, internal gates). The small-scale networks are initialized with all states (i.e., priorities and truth vectors) during their generation preserved. $T < 2^K$ is the length of the partially preserved truth vector for each node.

Initialization. The algorithm first computes the number of additional input nodes required (K_{new}), and initializes them with base priority p_0 and truth vectors of length T (lines 1–3). It then initializes

Algorithm 2 Network Output Allocation.

Require: network $\mathcal{G}$, number of outputs required L_{req}
Ensure: network $\mathcal{G}$ with exactly L_{req} outputs allocated
1: Identify set of nontrivial nodes $\mathcal{N}$; dangling nontrivial nodes $\mathcal{D} \subseteq \mathcal{N}$; trivial nodes $\mathcal{T}$
2: $r \leftarrow L_{req} - |\mathcal{D}|$ {Compute how many additional outputs are required after exhausting all dangling nontrivial nodes.}
3: **if** $r < 0$ **then** {The case where the number of dangling nontrivial nodes is sufficient.}
4: Select L_{req} nodes from $\mathcal{D}$ by weighted sampling
5: **else** {Fill output nodes in order of priority.}
6: Assign all nodes in $\mathcal{D}$ as outputs
7: Select $\min(r, |\mathcal{N} \setminus \mathcal{D}|)$ nodes from $\mathcal{N} \setminus \mathcal{D}$ by weighted sampling
8: Select $\max(r - |\mathcal{N} \setminus \mathcal{D}|, 0)$ nodes from $\mathcal{T}$, prioritizing dangling ones
9: **end if**
10: **for** each selected node v **do**
11: Mark v as output with random inversion $inv \sim \{0, 1\}$
12: **end for**

the sub-network usage counters D_s and D_c, and sets node sets S and H to the new input nodes (lines 4–5).

Iterative Gate Generation. The main loop (lines 6–19) iteratively generates M gates. In each iteration, the algorithm selects parent nodes for the new node, connects the new node to its parent nodes, and updates the corresponding statistics. For each newly generated gate node v_i with gate type g_i (line 7), the candidate pool is initialized by dangling nodes set $\mathcal{H}$ if number of dangling nodes meets or exceeds the number of expected outputs L, or set of all nodes S otherwise (line 8).

After initializing control variables including selection probabilities and a connectivity indicator u_n (line 9), the algorithm then enters the parent nodes sampling stage (lines 11-15) by Algorithm 4. In Algorithm 4, weight $w(v)$ follows a power-law distribution, incorporating selection of less-used and structurally complex nodes:

$$w(v) = \text{nontrivial}(v) \cdot \left(\text{priority}(v) + \sqrt{|\text{Net_Anc}(v)|/p_0}\right)^{\alpha} \tag{8}$$

where $|\text{Net_Anc}(v)|$ denotes the size of node v's ancestor sub-network set, and $\alpha > 1$ is an amplification factor. Trivial nodes are specially set with zero weight (i.e. $w(v) = 0$). The function `Sample_Net`(candidates,weights,prioritize,sample_size) then works by prioritizing unconnected sub-networks (i.e., $\mathbb{1}_{D_c(\mathcal{G}_x)=0} = 1$), and sample remaining ones by a weight inversely proportional to the selection count.

After selecting parenting sub-networks and nodes by Algorithm 4, truth vector of v_i is computed by truth of its parents (line 17). If truth(v_i) is trivial, resampling continues until either a valid nontrivial truth table is obtained or the retry limit is exceeded (line 11). In the latter case, v_i is marked as trivial and will not contribute to any other nodes' formation. The subroutine finally returns v_i with its truth vector computed as well as its parents and related sub-networks for post-processing (line 20). Getting node v_i, Algorithm 3 loops until truth(v_i) is nontrivial or it reaches a maximum trial time.

Integration of New Node and Statistics Update. The post-processing stage then connects v_i to the network, decays parent node priorities by p_0, updates ancestor sets of v_i, and sets the priority for the new node using the layer-aware strategy in Equation 5(lines 16-17). After that, dangling and entire node set are also updated the same way described in small-scale logic network generation, and essential counters of sub-networks are updated (line 18).

If the second subnetwork $\mathcal{G}_k$ returned by Algorithm 4 is not null, which means both the parents are selected from subnetworks, then D_c will not be increased for both $\mathcal{G}_j$ and $\mathcal{G}_k$. If only $\mathcal{G}_j$ is not null, we update $D_c(\mathcal{G}_j) \leftarrow D_c(\mathcal{G}_j) + 1$ as it is connected to the main structure. Finally, v_i is added to $\mathcal{G}$.

After M iterations, the merged structure is complete. Outputs are allocated following Algorithm 2 (line 20).

Algorithm 3 Large-Scale Logic Network Construction.

Require: $\mathcal{A} = \{\mathcal{G}_j(K_j, L_j, M_j)\}$; target scale K, L, M; truth vector reserved length $T < 2^K$; maximum repeat times $R_{\text{inner}}, R_{\text{outer}}$
Ensure: Merged network $\mathcal{G}(K, L, M)$

$K_{\text{new}} \leftarrow K - \sum K_j$
Initialize K_{new} inputs I, add to $\mathcal{G}$ with priority p_0
Assign each new input a truth vector of length T, covering T input combinations
Initialize selection count D_s and connection count D_c as null mappings
Initialize dangling nodes set $\mathcal{H}$ and set of all nodes $\mathcal{S}$ with $\mathcal{I}$
for $i = 1$ to M **do** {Node generation and connection.}
  Create v_i with uniformly sampled gate type $g_i \sim$ Connectives
  Set candidate pool $\mathcal{C} \leftarrow \mathcal{H}$ if $|\mathcal{H}| \geq L$, else $\mathcal{S}$
  $p_{\text{pool}} \leftarrow \frac{|D_s|}{|\mathcal{A}|}$, $p_{\text{net}} \leftarrow \max\left(\epsilon, \sqrt{\frac{|\mathcal{I}|}{K_{\text{new}}}}\right)$, $u_n \leftarrow (|\mathcal{A}| - |D_c|) \geq (M - i - 1)$
  attempts $\leftarrow 0$, initialize truth$(v_i) \leftarrow \{0\}^T$
  while trivial(truth(v_i)) and attempts $< R_{\text{outer}}$ **do** {Trivial check.}
    Initialize random inversions $(inv_0, inv_1) \sim \{0, 1\}^2$
    Derive $(v_i, c_0, c_1, \mathcal{G}_j, \mathcal{G}_k)$ by Algorithm 4
    attempts $\leftarrow$ attempts $+ 1$
  end while
  Update Anc(v_i), layer(v_i), priority(v_i)
  Decay priorities of c_0, c_1 by p_0
  Update $D_s, D_c, I, \mathcal{H}, \mathcal{S}$; add v_i to $\mathcal{G}$
end for
Allocate outputs by Algorithm 2

Algorithm 4 Parent Node Selection based on Trivial Check.

Require: $v_i, \mathcal{C}, \mathcal{A}, p_{\text{pool}}, p_{\text{net}}, u_n, D_s, D_c, inv_0, inv_1, R_{\text{inner}}$
Ensure: Parenting node pair (c_0, c_1)

attempts $\leftarrow 0$
while attempts $< R_{\text{inner}}$ and trivial(truth(v_i)) **do**
  Initialize candidate sub-networks $\mathcal{G}_j, \mathcal{G}_k$ as null
  if $((\text{rand}() < p_{\text{pool}}) \wedge (\mathcal{C} \neq \emptyset)) \vee u_n$ **then**
    $c_0 \leftarrow$ Sample$(\mathcal{C}, w(v))$ {Sample from $\mathcal{C}$ weighted by w(v).}
    if (rand$() > p_{\text{net}}) \wedge \neg u_n$ **then**
      $c_1 \leftarrow$ Sample$(\mathcal{C},$ priority$(v))$
    else
      $\mathcal{G}_j \leftarrow$ Sample_Net$(\mathcal{A}, \frac{1}{1+D_s(\mathcal{G}_x)}, \nVdash_{D_c(\mathcal{G}_x)=0}, 1)$ {Sample 1 $\mathcal{G}_x$ from $\mathcal{A}$ with weight $\frac{1}{1+D_s(\mathcal{G}_x)}$, prioritizing unconnected ones with $D_c(\mathcal{G}_x) = 0$.}
      $c_1 \leftarrow$ Sample$(\mathcal{G}_j, w(v))$
    end if
  else
    $\mathcal{G}_j, \mathcal{G}_k \leftarrow$ Sample_Net$(\mathcal{A}, \frac{1}{1+D_s(\mathcal{G}_x)}, \nVdash_{D_c(\mathcal{G}_x)=0}, 2)$
    $c_0 \leftarrow$ Sample$(\mathcal{G}_j, w(v))$
    $c_1 \leftarrow$ Sample$(\mathcal{G}_k, w(v))$
  end if
  truth$(v_i) \leftarrow$ truth$(c_0)^{inv_0} \odot_{g_i}$ truth$(c_1)^{inv_1}$
  attempts $\leftarrow$ attempts $+ 1$
end while
return $v_i, c_0, c_1, \mathcal{G}_j, \mathcal{G}_k$

A.2 Evaluation of Synthetic Datasets

A.2.1 Analysis of Benchmark Complexity

we introduce several commonly used evaluation metrics in logic-synthesis and graph-based benchmarks to evaluate our benchmark complexity: 1) Fanout Layer Variance (FLV): Variance in fanout counts across layers, capturing the structural dispersion of signals in the network. 2) Max_Depth: Length of the longest path from primary inputs to outputs, reflecting overall complexity. 3) Normalized Depth: Max_Depth normalized by circuit size, computed as max_depth/ $\log_2$ Size, enabling cross-scale comparison. 4) Uniqueness Ratio: The proportion of distinct truth vectors (i.e., if two vectors differ in any bit). Measured within each gate level (*Average Uniqueness*) or across the entire network (*Global Uniqueness*). 5) Max_Repeat: The proportion of all generated functions occupied by the most frequently occurring one. Table 3 provides a representative overview of the generator's output space by averaging over samples generated under MIMO's scale configurations. The table demonstrates the generator's ability to maintain structural diversity and low redundancy, and the metrics' comprehensiveness in presenting data features.

Table 4 summarizes the detailed composition of synthetic datasets along with their generation parameters. Each tuple in the Source Sub-networks column specifies the number of inputs and outputs of each sub-network, the number of sub-networks used to assemble a larger network, and the number of gate nodes per sub-network. The corresponding uniqueness distribution is shown in Figure 6, where most configurations achieve a uniqueness greater than 0.8 averaged across noise levels. Trivial vectors are included in the denominator, but excluded from valid unique patterns.

Table 3: Statistics of generated boolean networks under AN and ANO operator set configurations.

Operator Set	In/Out Scale	FLV	Max_Depth (Norm.)	Avg_Uniqueness	Global_Uniqueness	Max_Repeat
AN	5	0.3060	10.4233 (2.1449)	0.8553	0.7433	0.0047
	10	0.6226	19.5233 (2.8973)	0.9900	0.9883	0.0010
	20	0.4719	72.9933 (8.3278)	0.9252	0.9112	0.0030
	40	0.5063	72.1233 (7.4624)	0.8898	0.8592	0.0014
	80	0.5936	75.1733 (7.0502)	0.8898	0.8811	0.0021
ANO	5	0.3173	10.2933 (2.1113)	0.8520	0.7507	0.0053
	10	0.6298	19.3500 (2.8705)	0.9930	0.9900	0.0013
	20	0.4704	73.4233 (8.3735)	0.9212	0.9048	0.0028
	40	0.4989	72.1067 (7.4691)	0.8947	0.8568	0.0013
	80	0.5926	75.5600 (7.0814)	0.8894	0.8815	0.0000

Table 4: Synthetic dataset composition and source sub-networks across input-output scales.

Category	Scale	Gate Counts	Number of Networks	Total Networks	Source Sub-networks
Small Scale	in5_out5	10	100	300	–
		20	100		–
		40	100		–
	in10_out10	40	100	300	–
		80	100		–
		160	100		–
Large Scale	in20_out20	160	100	300	(5,5,3,30)
		320	100		(5,5,3,80)
		640	100		(8,8,2,170)
	in40_out40	320	100	250	(6,6,6,40)
		640	100		(6,6,6,90)
		1280	50		(5,5,7,150)
	in80_out80	640	50	150	(8,8,8,60)
		1280	50		(9,9,8,130)
		2560	50		(10,10,7,310)

A.2.2 Parameter Analysis

in this part, we analyze how hyperparameter settings affect the generator's performance, demonstrating its flexibility and boundaries.

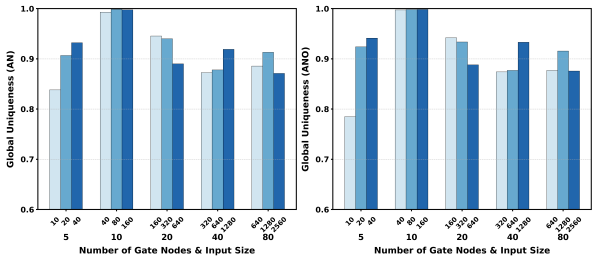

Figure 6: Global uniqueness across different AN/ANO configurations.

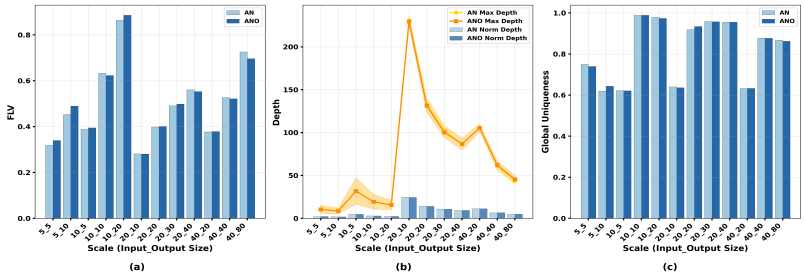

Figure 7: Comparative analysis of FLV (a), depth (b), and uniqueness metrics (c) for AN and ANO across asymmetric configurations.

Input-Output Scale. For synthetic datasets, we use symmetric input–output scales (equal numbers of inputs and outputs), to ensure consistent complexity scaling across tasks, fair comparison among multi-output settings, and controlled environment for evaluating generalization. This design follows common practices in prior work [47, 19] and complements industrial benchmarks, which often lack symmetric structures [1, 6].

Our experiments further validate that symmetric input–output configurations produce more stable and predictable circuit structures, whereas asymmetric settings (in $\neq$ out) lead to lower truth-table uniqueness and greater depth variance (Figure 7). The decrease in global uniqueness under input-heavy configurations (inputs $>$ outputs) is expected, as fewer outputs may limit many internal distinct truth vectors not expressible at the output layer. Conversely, configurations with more outputs maintain high uniqueness, reflecting the generator's inherent logical diversity and scalability.

Hyperparameter Sensitivity Analysis. We conducted analysis on 3 key hyperparameters that influence the generator's behavior. 1) The amplification factor α in Equation 8, is evaluated on networks with 20 inputs, 20 outputs, and 640 logic gates. 2) The priority decay coefficient μ, which controls the decay rate of parent node priority during expansion, is analyzed on networks with 10 inputs and 10 outputs. 3) The remaining steps, defined as the difference between the target gate count M and the total number of gate nodes already provided by the merged subnetworks $\sum_j M_j$, is studied on networks with 40 inputs, 40 outputs, and 640 gates.

Table 5: Comparison of structure metrics for different parameter settings.

Hyperparameter	Settings (Param. Value)	FLV	Max_Depth	Normalized_Depth	Global_Uniqueness
α	Linear (1)	0.4885	122.54	13.08	0.901
	Default (1.5)	0.4000	132.80	14.18	0.928
	Strong Amplify (3)	0.4838	123.82	13.22	0.912
μ	No Decay (1)	0.7482	17.97	2.660	0.986
	Default (3)	0.6214	19.51	2.894	0.990
	Stronger Decay (9)	0.6271	21.75	3.236	0.989
Remaining Steps	Large (480)	0.4210	90.12	9.58	0.982
	Medium (300)	0.5236	73.18	7.78	0.978
	Small (100)	0.6562	51.48	5.47	0.869

The observations are as follows: 1) The default amplification configuration ($\alpha = 1.5$) yields the highest uniqueness and depth, with lowest FLV. This indicates that using an exponential weighting

($\alpha > 1$) is essential for emphasizing deeper nodes, thereby promoting structural complexity and diversity. 2) A stronger decay μ produces deeper networks but may reduce uniqueness, while the default setting achieves the best balance between the two. 3) More remaining steps yields lower FLV, greater depth, and higher uniqueness. This indicates that combining partial sub-network reuse with a substantial amount of newly generated gates reduces structural repetition and maintains sufficient depth.

Other hyperparameters, including the initial value of p_0 (in range $[1, 12]$), have little impact on the metrics within a specific range, and the analysis are therefore omitted.

A.2.3 Real-World Relevance

Structural Correspondence. To examine how well our synthetic data reflects real-world Boolean networks, we compare their structural characteristics with those of biological networks (BBM) and logic circuits (IWLS 2023), as shown in Table 6. In principle, gate-to-input ratios can vary widely, like EPFL circuits may contain tens of thousands of gates but few inputs [1]). But we follow ISCAS-85 circuit structures, which features moderate sizes and balanced input-output distributions, ensuring representative and hierarchically structured designs at a reasonable scale.

Table 6: Structural Comparison between Real-World and Synthetic Data

Metric (mean±std [global min,max])	BBM	IWLS 2023 Submissions	Synthetic
Number of Nodes	63.99 ± 87.25 $[4, 853]$	351.86 ± 803.12 $[16, 7264]$	$--$ $[10, 2560]$
Number of Inputs	13.21 ± 21.71 $[0, 223]$	12.87 ± 3.24 $[6, 16]$	$--$ $[5, 80]$
Number of Outputs	7.23 ± 19.63 $[0, 257]$	11.00 ± 14.75 $[1, 103]$	$--$ $[5, 80]$
Network Average Fanout	2.61 ± 1.40 $[0, 13]$	1.54 ± 0.27 $[0, 653]$	1.72 ± 0.21 $[0, 11]$
Cyclicity	Cyclic	Acyclic	Acyclic (Configurable)
Longest Path (Depth)	6.52 ± 7.07 $[0, 99]$	3.21 ± 1.71 $[0, 10]$	19.98 ± 15.11 $[0, 61]$

The comparison shows that synthetic data captures key characteristics of both domains while spanning a wider difficulty spectrum. Specifically, BBM features strong feedback and long paths, while IWLS features low fanout, acyclicity, and shallow depth. Synthetic data covers and extends these ranges through configurable network size, fanout, and depth, enabling evaluation of scalability and robustness beyond current limits.

In real-world applications, most logic exists in the form of networks rather than isolated formulas. Moreover, many mainstream logic synthesis methods (such as ABC) can only process inputs in the form of logic networks (AIGs). Therefore, we generate our logic datasets based on network structures rather than individual formulas, which also enables better-controlled complexity levels.

Task Difficulty and Practical Significance. Small-scale tasks use complete truth tables for exact logic recovery, while large-scale tasks introduce bit-flip noise and incomplete observations to evaluate generalization under practical constraints. This reveals method strengths and limitations across scales and observation conditions, and reflects practical challenges of structural recovery from incomplete data. Unlike real applications targeting circuit optimization or biological relationship modeling, LogicSR formulates these problems as a general logic discovery task, evaluating the recovery of logical dependencies in terms of accuracy and optimality.

A.3 LogicSR-Core

A.3.1 LogicSR-Core Characterization

LogicSR-Core (v1): a lightweight fixed evaluation set. We provide a dedicated subset that can be evaluated through a single predefined setting. Its membership is fixed by one scale criterion (at most 10 inputs and 10 outputs), ensuring consistent comparison across suites without arbitrary instance selection (Table 7).

The manifest also records the operator vocabulary of each suite (AN, IWLS: `AND/NOT`; ANO, BBM: `AND/OR/NOT`) and a version-tagged instance list carrying a SHA-256 of its contents; a SHA-256 checksum is also released for every instance file. This ensures that the evaluation setting is fixed and reproducible. The released manifest can be directly verified against the data, so reported numbers remain checkable.

Table 7: Composition of LogicSR-Core (v1).

Suite	In-band instances	
AN & ANO	in5_out5 / in10_out10, noise 0/0.05, first 10 per gate level.	240
BBM & IWLS	Every in-band instance.	130
Total	$\sim$6% of all truth tables	370

Evaluation Protocols. We fix the evaluation settings as default: a 0.75 train/test split with a fixed seed, and three accuracy granularities (bit/sample-wise, per-output) reported with gates/output and wall-clock time. Because Core is cheap, the recommended protocol repeats each run over three seeds, with tooling provided — `logicsr_core.aggregate --by-cell` reports the mean and across-seed spread with per-method coverage, making error bars affordable to produce.

Full scalable suite. Inputs 20/40/80, the intermediate noise level, and the hard tiers (excluded from Core by construction) remain the settings for scalability and robustness.

A.3.2 How Many Instances Are Needed for a Reliable Comparison?

To check whether a small subset can reproduce the comparison obtained on the full set, we repeatedly draw random subsets of k instances (2,000 draws without replacement for each k) from a fixed cell and compare the subset results with the full-cell results. We use two noise-free ANO cells: in5_out5 (14 methods on the 299 instances they all cover, evaluated on the full truth table since this scale has no test split) and in10_out10 (11 methods on 150 instances, evaluated on the test split). Methods covering less than 95% of a cell are excluded. All methods are scored by per-output correctness (POC, Appendix A.6.2); the LLM baselines in this analysis are the PoT variants. For each draw we record (i) the absolute error of each method's mean POC relative to its full-cell value, averaged over methods; (ii) the Kendall-τ between the subset ranking and the full-cell ranking of methods; and (iii) whether the best method on the full cell is also the best on the subset. Table 8 reports the averages over draws.

Table 8: Precision of subset-based comparison versus subset size k (noise-free ANO, 2,000 draws per k). Error: mean absolute error of per-method POC (percentage points); τ: Kendall-τ against the full-cell ranking; Top-1: percentage of draws in which the full-cell best method remains best.

	in5_out5 (14 methods, N=299)			in10_out10 (11 methods, N=150)		
k	Error (pp)	τ	Top-1 (%)	Error (pp)	τ	Top-1 (%)
10	2.77	0.941	100.0	2.92	0.928	83.0
20	1.95	0.957	100.0	2.06	0.957	92.7
30	1.54	0.965	100.0	1.60	0.972	96.7
50	1.15	0.970	100.0	1.12	0.986	99.7
100	0.73	0.975	100.0	0.57	0.994	100.0

With only $k = 30$ instances, the per-method error is about 1.5–1.6 percentage points and the method ranking is nearly unchanged ($\tau \geq 0.96$). On in10_out10, the best method is preserved in 96.7% of draws at $k = 30$ and in 99.7% at $k = 50$; on in5_out5, five methods reach 100% POC, so Top-1 retention is trivially satisfied there. The discriminative power of the benchmark therefore saturates far below its full size. LogicSR-Core follows this finding: it contains 30 synthetic instances per (suite, scale, noise) cell (10 for each of the three gate counts).

A.4 Considerations for Baseline-Related Experiments

A.4.1 Additional Experimental Settings

In the experiments, for small-scale logic networks with 5 or 10 inputs, the truth vectors of the output variables were perturbed under three noise levels: 0.0 (noise-free), 0.01, and 0.05. These noisy variants are stored as separate instances. Our experimental protocol applied a per-instance timeout of 1200 seconds. The numbers of MIMO instances used in the experiments are shown in Table 9, while the full IWLS and BBM benchmark suites were used without modification.

Table 9: Statistics of the experimental subset for both AN and ANO.

Scale	Noise Level	Number of Networks
in5_out5	0	300
	0.01	300
	0.05	300
in10_out10	0	150
	0.01	150
	0.05	150
in20_out20	0	300
in40_out40	0	150
in80_out80	0	60

A.4.2 Criteria for Baseline Selection

In the experiments, different sets of baselines are selected for distinct datasets. Specifically, for datasets ANO and BBM, which include the complete operator set {`AND`, `OR`, `NOT`}, we used 17 baseline algorithms. Conversely, for datasets AN and IWLS, which contain only the operator set {`AND`, `NOT`}, 14 baseline algorithms were selected.

Exclusion of Certain Algorithms on AN and IWLS Datasets. We test only 14 baselines on the AN and IWLS datasets, excluding the algorithms *Quine-McCluskey*, *BDD*, *Espresso*, and *Boolformer*. The native minimal outputs of the *Quine-McCluskey* and *BDD* algorithms inherently include `OR` gates, being represented as Sum-of-Products and cube lists respectively. Similarly, *Espresso* and *Boolformer* algorithms explicitly include `OR` gates in their original expressions. Therefore, these four algorithms are more suitable for scenarios where the ground-truth is expressed using the {`AND`, `OR`, `NOT`} operator set. Forcing their outputs to map onto a pure {`AND`, `NOT`} operator set would require additional De Morgan transformations or logic synthesis, which could distort the original forms of the algorithm outputs.

Rationale for ABC Evaluation Only on {AND, NOT} Operator Set. The *ABC* algorithm is not tested on the ANO and BBM datasets, as it is only evaluated on datasets whose ground-truth expressions are represented using the {`AND`, `NOT`} operator set. Specifically, the internal structure of *ABC* is an And-Inverter Graph (AIG), which consists solely of two-input `AND` gates, with inversion operations implemented through edge labeling. Thus, the {`AND`, `NOT`} operator set is sufficient to encompass all functionalities handled by *ABC* without impacting its generality. Moreover, this matches the AIG representation used by related benchmarks, such as the bit-level AIGER track of HWMCC [4].

A.4.3 Fairness: Operator Sets and Output Formats

Since different methods support different operator sets and output formats, we address both axes with controlled experiments rather than assurances.

Operator set — a controlled variable, quantified. Every method on a suite gets the vocabulary matching that suite's construction, identical across methods (AN, IWLS: `AND`/`NOT`; ANO, BBM: `AND`/`OR`/`NOT`). To show this is not a hidden confound, we ran a paired ablation on three operator-configurable methods, scoring the same instances under matched vs. mismatched vocabularies (3 seeds, 60 instances/suite). Δ = effect of the operator change vs. the matched baseline (Table 10).

Table 10: Operator-set ablation. Δ = effect of the operator change vs. the matched baseline. Notes: paired Wilcoxon signed-rank; *** $p < 10^{-3}$, ** $p < 0.01$, * $p < 0.05$; across-seed spread ≤ 0.004; effect holds cell-by-cell (per scale and gate count).

Method	Operator change	Δ bit-acc	Δ sample-acc
GPlearn	ANO: remove `OR`	-0.013^{***}	-0.042^{***}
Operon	ANO: remove `OR`	-0.006^{***}	-0.041^{***}
DiffLogic	ANO: remove `OR`	-0.011^{***}	-0.030^{***}
GPlearn	AN: add `OR`	$+0.011^{***}$	$+0.030^{***}$
Operon	AN: add `OR`	$+0.008^{***}$	$+0.054^{***}$
DiffLogic	AN: add `OR`	$+0.007^{**}$	$+0.023^{*}$

Every entry is significant with a consistent sign — removing a needed operator (OR from ANO) lowers accuracy, while adding a superfluous one (OR to AN) raises it. The operator set is never neutral, so we fix it: every method uses the same generation-matched vocabulary per suite — a controlled condition, not a per-method artifact. This applies to operator-based methods; others are unaffected. Since {AND, NOT} is functionally complete, restricting operators changes inductive bias, not expressiveness. OR-on-AN gains likely reflect a richer primitive set expressing the same functions more compactly, not a different target class.

Output format cannot advantage or penalize a method. No method is scored on its representation. Every output — symbolic expression, ABC/BLIF netlist, BDD, QM sum-of-products, or a neural prediction array — is executed on the same truth-table rows and compared to the target. We verify this by encoding identical functions in all five formats and confirming byte-identical scores (Table 11).

Table 11: Output-format invariance: identical functions encoded in all five formats.

Target function	Rows	Preds identical across 5 formats	bit / sample / output-acc
majority-3	8	✓	1.000 / 1.000 / 1.000
parity-3	8	✓	1.000 / 1.000 / 1.000
mux (ignores x_4)	16	✓	1.000 / 1.000 / 1.000
and-or-4	16	✓	1.000 / 1.000 / 1.000
2-output (aggregate)	–	✓	– / 1.000 / 1.000
equally-wrong answer	8	✓ (symbolic vs BLIF)	penalized to 0.875 in both

All five reduce to the same per-row behaviour: a correct function scores 1.000 in every format, and an equivalently-wrong one is penalized identically (0.875). Scoring is therefore representation-invariant — format changes how a prediction is produced, not how it is scored.

Both axes are thus controlled by design and verified empirically: the operator set is fixed identically across methods per suite, and the metric is blind to output representation.

A.4.4 Discussion of Experimental Results

Performance varies across all methods on real-world and synthetic datasets, revealing the adaptability gap between structured benchmark tasks and complex logical interactions in practical scenarios. The synthetic data in LogicSR serve as controlled benchmarks, characterizing performance boundaries under clean logical conditions. In contrast, real-world datasets may contain more complex characteristics, including coupled multiple logical operators and implicit noise, imposing higher demands on the generalization and robustness of the evaluated methods.

A.4.5 PySR Configuration, Inputs and Outputs

Implementation. We call SymbolicRegression.jl v1.11.3, the release series used by PySR 1.5.9, directly via PyJulia, not the Python PySRRegressor wrapper. The reported PySR results use the main configuration based on the search defaults of PySR 1.5.9 (100 iterations, `maxsize` 30), but five adaptations remain: (i) Primitives are restricted to the operator set of each suite. (ii) Constants are disabled, since the domain is constant-free: `should_optimize_constants` is false, `complexity_of_constants` is 1000, and new leaves are always variables. The last change is needed because new leaves are otherwise drawn half from constants, and a tree holding a constant exceeds `maxsize` under this complexity and can hardly mutate. (iii) Each run uses a single thread, with a wall-clock limit of 1200 s per instance. (iv) Computation uses 64-bit floats rather than PySR's default of 32-bit. (v) The returned expression is the lowest-loss candidate, equivalent to PySR's `model_selection="accuracy"`, rather than the default `"best"`. For comparison, we also conduct experiments using a reference configuration based on v0.24.5 defaults (Appendix A.4.7). Table 12 lists both configurations; their mutation weights follow the same respective defaults.

Inputs. Each instance is a k-input / m-output Boolean function; we take its binary truth table (X, Y), split 75/25 into train/test with a fixed seed (truth tables of at most 2^7 rows are used in full for both), and fit each output by an independent search (default L2 loss over all training samples, no batching). Primitives: AND $= \min(a, b)$, OR $= \max(a, b)$, NOT $= 1 - a$ (AN: AND/NOT; ANO: AND/OR/NOT).

Outputs. For each output we keep the lowest-loss Hall-of-Fame member and map its tree to a logic expression; an instance yields m expressions, evaluated as Boolean logic (exact 0/1) and scored by exact per-output recovery, plus bit- and sample-accuracy.

Table 12: PySR search configuration on SymbolicRegression.jl v1.11.3. The main configuration uses the PySR 1.5.9 defaults (`maxdepth` defaults to `maxsize`), except for the entries marked †, while the reference configuration is based on the v0.24.5 defaults.

Parameter	Main	Reference	Parameter	Main	Reference
`maxsize`	30	20	`parsimony`	0	0.1
`maxdepth`	30	10	`adaptive_parsimony_scaling`	1040	20
`populations`	31	100	`crossover_probability`	0.0259	0.066
`population_size`	27	33	`fraction_replaced`	0.00036	0.00036
`ncycles_per_iteration`	380	1000	`fraction_replaced_hof`	0.0614	0.035
`niterations`	100	20	`warmup_maxsize_by`	0	0
`tournament_selection_n`	15	12	`should_optimize_constants`	false†	false
`tournament_selection_p`	0.982	0.86	`complexity_of_constants`	1000†	1000
`annealing`	false	false	`batching`	false	false
`elementwise_loss`	L2	L2	`turbo`	false	false

Other baselines. The remaining methods use the fixed configurations in our released code, based on their libraries or reference implementations, with task-specific adaptations (restricting the primitive set and disabling real-valued constant handling).

A.4.6 Sensitivity of PySR to maxsize

We vary only `maxsize` $\in \{10, 20, 30, 40, 50\}$ and keep every other setting in the main configuration of Table 12 fixed; `maxdepth` follows `maxsize`. The study covers all noise-free in10_out10 instances (150 AN and 150 ANO) and runs single-threaded under the 1200 s per-instance limit; no run times out. The number of evolution steps does not depend on `maxsize`, so the five settings share the same nominal search budget. We use in10_out10 because it is the largest synthetic scale that PySR completes: all 30 in20_out20 instances we tried exceed the limit already at `maxsize`=30. We report per-output correctness (POC) and sample accuracy on the test split, the gate count per output, the mean expression size (nodes), and the median runtime per instance.

Table 13: PySR sensitivity to `maxsize` (in10_out10, noise-free, 150 instances per suite). POC and Sample are test accuracies (%); Gates is the complexity per output; Size is the mean number of nodes per expression; Time is the median runtime per instance (s).

	AN					ANO				
`maxsize`	POC	Sample	Gates	Size	Time	POC	Sample	Gates	Size	Time
10	11.1	57.0	2.2	7.8	231	18.9	67.8	3.1	8.2	237
20	34.3	75.2	4.7	14.9	332	53.1	86.1	5.6	14.0	359
30†	46.5	82.4	5.9	18.7	455	65.1	91.4	6.6	16.5	510
40	54.7	86.5	6.8	21.4	575	71.9	93.3	7.2	18.1	651
50	58.8	87.9	7.3	23.1	700	75.6	94.9	7.6	19.3	790

† Setting used in the main experiments.

Accuracy rises at every step and has not saturated at `maxsize`=50. From 40 to 50, POC still gains 4.1 points on AN and 3.7 on ANO, and each step is significant (paired Wilcoxon signed-rank test on POC, $p < 10^{-4}$). The search does use the larger limit: mean expression size grows from about 8 to 23 nodes on AN and to 19 on ANO, and the largest expressions reach each limit. The share of expressions at the limit falls from 32% (AN) and 43% (ANO) at `maxsize`=10 to below 1% at 50. The gain has a cost: median runtime about doubles from 20 to 50, though the longest run (923 s) stays within the limit.

The setting changes the ranking on ANO. At `maxsize`=50, PySR (75.6) passes SATNet (72.7) and NeuraLUT (70.5) but stays below Espresso (84.6) and Boolformer (80.1) on in10_out10 (Table 2). On AN it (58.8) stays below NeuraLUT (71.7) and SATNet (66.9). We keep the PySR 1.5.9 default of 30 in the main experiments, so Table 2 does not reflect the best `maxsize` for PySR.

A.4.7 Re-evaluation of PySR Search Hyperparameters

Two aspects of the PySR configuration require clarification: whether populations and iterations were balanced, and whether newer library defaults would perform better. We address both on the noise-free in10_out10 instances. All settings share the same per-instance compute budget (one thread, 1200 s),

and every run in Table 14 finishes within it. Each setting runs once per instance, and differences are tested with a paired Wilcoxon signed-rank test over the 150 instances of a suite.

Since the settings also differ in cycles per iteration and tournament size, we also report their nominal search budget in evolution steps:

$$N_{\mathrm{pop}} \times \lceil S_{\mathrm{pop}}/S_{\mathrm{tour}} \rceil \times N_{\mathrm{cyc}} \times (N_{\mathrm{iter}} + 1),$$

where N_{pop} is the number of populations, S_{pop} the population size, S_{tour} the tournament size, N_{cyc} the cycles per iteration, and N_{iter} the iterations; the extra iteration is the initial one before the main loop. For example, the reference configuration takes $100 \times 3 \times 1000 \times 21 = 6.30$M steps, and the main configuration takes $31 \times 2 \times 380 \times 101 = 2.38$M. The budget is nominal because the loss evaluations actually performed also depend on the search, as candidates that violate the constraints are rejected without evaluation.

Table 14: PySR search settings (in10_out10, noise-free, 150 instances per suite). Budget: evolution steps per search, with the share of the reference configuration in parentheses. POC and Sample are test accuracies (%); Time is the median runtime per instance (s). Differences in the text are computed before rounding.

					AN			ANO		
Setting	Pop. × size	Iter.	`maxsize`	Budget	POC	Sample	Time	POC	Sample	Time
Reference (v0.24.5)	100×33	20	20	6.30M (100%)	30.5	73.5	690	47.0	83.3	712
Re-balanced	15×33	66	20	3.02M (48%)	27.9	71.6	354	42.9	81.8	359
Re-balanced	15×33	133	20	6.03M (96%)	30.1	72.6	710	45.9	84.0	736
Main (PySR 1.5.9 default)	31×27	100	20	2.38M (38%)	34.3	75.2	332	53.1	86.1	359
Main (PySR 1.5.9 default) †	31×27	100	30	2.38M (38%)	46.5	82.4	455	65.1	91.4	510

† Setting used in the main experiments. The others use the same engine and constant handling.

Populations versus iterations. The re-balanced settings keep all other parameters fixed and vary only the number of populations and iterations. We consider two iteration counts: 66 iterations to provide a more balanced search, and 133 iterations for comparison. At 96% of the reference budget, re-balancing does not significantly change POC (AN 30.1 vs. 30.5, $p = 0.34$; ANO 45.9 vs. 47.0, $p = 0.067$). Sample accuracy differs by less than 1 point, though in opposite directions (AN -0.9, $p = 0.002$; ANO $+0.6$, $p = 0.015$). At 48% of the budget, POC is lower by 2.6 and 4.1 points ($p < 10^{-5}$), and doubling the iterations to 133 recovers 2.3 and 3.0 points ($p < 10^{-3}$). The search therefore benefits from more steps, while at a fixed budget the balance between populations and iterations has little effect in this domain.

PySR 1.5.9 defaults. With only 38% of the steps, the defaults exceed the reference configuration by 16.0 POC points on AN and 18.1 on ANO, and by 9.0 and 8.1 points of sample accuracy ($p < 10^{-20}$ in all four cases). They also run faster, with a median of 455 s and 510 s against 690 s and 712 s, but return larger expressions (5.9 and 6.6 gates per output against 4.5 and 5.4). At the same `maxsize` of 20, the POC gain is 3.9 and 6.1 points ($p < 10^{-7}$); raising `maxsize` to the default of 30 adds another 12 points on each suite, so most of the gain comes from `maxsize`. We therefore use these defaults in the main experiments, which raises PySR's test POC on in10_out10 from 30.5 (AN) and 47.0 (ANO) in the reference to 46.5 and 65.1 (Table 2).

Equal step budget. We cannot report the defaults at the reference configuration's step budget: at 255 iterations (6.03M steps), nearly all instances we ran (159 of 164) exceed the 1200 s limit. Because the reported defaults use fewer steps than the reference configuration, this omission does not give them an advantage in the comparison above.

A.4.8 PySR in Multi-Output Mode

PySR (SymbolicRegression.jl) can also handle all outputs of a truth table in a single search call, in which each output keeps its own populations and Hall of Fame. In the main experiments, we run it in per-output mode, i.e., a separate call for each output. To verify that this choice does not affect our conclusions, we re-ran PySR in multi-output mode on the noise-free synthetic AN and ANO datasets (in5_out5 and in10_out10) under identical settings and with the same evaluation code; 300 instances are used for in5 and 150 for in10, pooled over the three complexity levels. Table 15 reports the results.

Table 15: PySR in per-output vs. multi-output mode (noise-free). Accuracies are test accuracies in %; #Gates is the average complexity per output; Time is the average runtime per instance.

Dataset	Mode	Bit Acc.	Sample Acc.	Output Acc.	#Gates	Time (s)
AN in5	per-output	99.78	98.99	95.73	3.07	131
	multi-output	99.75	98.79	94.80	3.03	128
AN in10	per-output	97.73	82.43	46.47	5.95	463
	multi-output	97.80	83.06	46.87	5.97	464
ANO in5	per-output	99.98	99.89	99.33	3.08	135
	multi-output	99.96	99.82	99.00	3.07	130
ANO in10	per-output	98.93	91.45	65.07	6.62	512
	multi-output	98.91	91.25	65.07	6.56	504

The two modes are essentially equivalent: bit-wise, sample-wise, and per-output accuracies differ by at most 0.1, 0.7, and 1.0 percentage points, respectively; the number of gates differs by at most 0.06; and mean runtime differs by less than 4%. In a paired Wilcoxon signed-rank test, the only significant differences are on AN in5, where multi-output mode is lower by 0.9 points in per-output accuracy ($p = 0.016$) and 0.2 in sample accuracy ($p = 0.026$). Hence, the choice of output mode does not affect our conclusions.

A.5 Detailed Descriptions of Baselines

Traditional Logic Synthesis. We select four representative methods from traditional logic synthesis: 1) *ABC* [5], a logic network-based synthesis and optimization tool; 2) *BDD* [25], a Binary Decision Diagram (BDD)-based simplification approach; 3) *Espresso* [42], a heuristic two-level logic minimizer, used through the pyeda package; 4) *Quine-McCluskey* [40, 30], a logic minimization method provided by the quine_mccluskey Python module.

Symbolic Regression. Within the symbolic regression category, we select five representative methods, all used through Python: 1) *PySR* [13], an open-source library for practical symbolic regression; 2) *Operon* [7], a modern genetic programming framework for symbolic regression, used through its Python bindings (PyOperon); 3) *DEAP* [17], an evolutionary algorithm library supporting tree-based genetic programming; 4) *GPlearn* [44], a scikit-learn-compatible genetic programming library for symbolic regression; 5) *TenGP* [26], a Cartesian genetic programming library [31] with optional tensor-based computation.

Neural Network-Based Methods. The five neural network-based methods we select are as follows: 1) *DiffLogic* [39], which performs gradient-based optimization on differentiable logical functions; 2) *AI4EDA_TNet* [51], a neural network-based method for circuit generation in electronic design automation (EDA) applications; 3) *NeuraLUT* [2], which uses supervised learning to train logic look-up tables (LUTs); 4) *Boolformer* [14], a transformer-based architecture specifically designed for Boolean logic reasoning; 5) *SATNet* [50], which integrates a differentiable SAT solver into neural networks to enable gradient-based logical reasoning.

LLM-Based Methods. Considering the significant potential demonstrated by large language models (LLMs) in reasoning tasks, we select two state-of-the-art LLMs: *GPT-4o* [35], developed by OpenAI, and *GPT-o3* [36], a reasoning model from OpenAI, to perform prompt-based symbolic reasoning.

1) LLM Few-Shot Prompt: Based on few-shot learning, the LLM is prompted to act as a symbolic regression problem solver, aiming to automatically derive the simplest Boolean expression from a given truth table. The prompt comprises: (1) a role definition, (2) a task description, (3) dynamically configured operators based on AN or ANO, (4) illustrative truth-table examples as few-shot teaching sequences, and (5) loaded complete truth table data ready for analysis.

Few-Shot Prompt Example

System Prompt
You are performing a logic symbolic regression task. Based on the truth table's input-output relationships, find the simplest boolean expressions.

Task Description

1. You are given a complete truth table with inputs and outputs.
2. For each output y_K, produce a correct and as-simplified-as-possible boolean expression.
3. Use variables x_1, ..., $x_{n_{\text{inputs}}}$ for inputs. Do not use variables outside this range.
4. Return exactly one line per output in the form `y_K = <expression>`.
5. Use explicit parentheses to indicate grouping.
6. If an output y_K is a constant function, you may use the constant 0 or 1 for its expression.
7. No extra commentary besides the lines `y_K = <expression>`.
8. Optimization Goal (in order of priority):
   (a) Exact correctness on the provided truth table.
   (b) Minimal expression size (fewest gates / shortest simplified form).

Operator Configuration

1. Available Operators: `{allowed_operators}` (dynamically configured)
2. Usage Rules:
   (a) Use lowercase operator names only
   (b) Strictly use only the available operators listed above
3. Configuration Note: The operator set is dynamically controlled by the system configuration. Only the operators marked as available in the current configuration may be used in your expressions.

Few-shot Examples: Teaching Sequences

Example 1 (2-input AND):

x1	x2	y1
0	0	0
0	1	0
1	0	0
1	1	1

Answer: y1 = x1 and x2

Example 2 (2-input XOR):

x1	x2	y1
0	0	0
0	1	1
1	0	1
1	1	0

Answer: y1 = (x1 and not x2) or (not x1 and x2)

Data Preparation: Complete Truth Table

1. Data Loading: The complete truth table data has been prepared and loaded for analysis.
2. Table Structure: `{n_inputs}` input variable(s) ($x_1..x_{\{n_inputs\}}$) and `{m_outputs}` output variable(s) ($y_1..y_{\{m_outputs\}}$).

Return results now

2) LLM Multi-Step Program-of-Thought Reasoning Prompt: The PoT prompt uses a two-step procedure. First, the model performs a chain-of-thought analysis of the truth table, identifying minterms and outlining simplification logic without producing final expressions. Second, the model converts this reasoning into executable Python code that derives and verifies the Boolean expressions using only the allowed operators. All reasoning appears as code comments, and only executable PoT code is returned.

Multi-Step PoT Reasoning Prompt Example

Step 1: Chain-of-Thought Reasoning (Natural-Language Analysis)

System Prompt: You are performing a logic symbolic regression task. Based on the truth table's input-output relationships, analyze and find the simplest boolean expressions.

Task Description:

1. You are given a complete truth table with {n_inputs} inputs (x_1, x_2, ..., $x_{\{n_inputs\}}$) and {n_outputs} outputs (y_1, y_2, ..., $y_{\{n_outputs\}}$).
2. Each row represents one input combination and its corresponding output values.
3. Your goal is to find the simplest boolean expression for each output y_K that matches the truth table exactly.

Chain-of-Thought Analysis Requirements:
1. Identifies minterms / maxterms for each output y_K (which input combinations produce output 1).
2. Explains grouping / simplification ideas (how to combine minterms into simpler expressions).
3. Mentions any sanity checks you would perform (e.g., verifying all rows match).

Important: Do NOT output final expressions or code yet. Just provide your reasoning. End with a short "Reasoning Summary" describing the best candidate expression for each output.

Truth Table: {table_md} |

Chain-of-thought:

Step 2: Program-of-Thought Code Generation (Executable Python)

System Prompt: You MUST perform Program-of-Thought reasoning and output ONLY executable Python code.

Task: Implement a `solve()` function that finds boolean expressions for {n_outputs} outputs based on the truth table.

Data Format (provided in runtime context):
- `X_data`: numpy array of shape ({len(X)}, {n_inputs}) - each row is one input combination [x_1, ..., $x_{\{n_inputs\}}$]
- `Y_data`: numpy array of shape ({len(X)}, {n_outputs}) - each row is corresponding output values [y_1, ..., $y_{\{n_outputs\}}$]
- Use the provided `X_data` and `Y_data` arrays (do NOT reconstruct them).

Code Requirements:
1. For each output y_K (K = 1 to {n_outputs}):
   (a) Identify minterms (rows where Y_data[:, K-1] == 1)
   (b) Derive the simplest boolean expression
   (c) Verify the expression matches ALL rows in the truth table
2. Use only allowed operators: `{allowed_ops}` (e.g., 'and', 'or', 'not').
3. Print exactly {n_outputs} lines in the format y_K = <expression>.
4. All reasoning must appear inside Python comments + executable logic.
5. No natural-language text outside the code block.

Chain-of-Thought Analysis Provided:
{cot_text}

Instruction: Follow the Chain-of-Thought above and now produce the Program-of-Thought code.

Few-shot Example (2-input AND):

```
def solve():
    # Step 1: Identify minterms for y1 (rows where y1=1)
    minterms = []
    for i in range(len(X_data)):
        if Y_data[i, 0] == 1:
            minterms.append(tuple(X_data[i]))
    # Result: minterms = [(1, 1)]

    # Step 2: Derive expression from minterms
    # For AND: y1 = x1 and x2
    def f(x1, x2):
        return int(x1 and x2)

    # Step 3: Verify against all rows
    for i in range(len(X_data)):
        x1, x2 = X_data[i]
        assert f(x1, x2) == Y_data[i, 0]

    # Step 4: Print final expression
```

```
    print("y1 = (x1 and x2)")
```

Output: ONLY the final Python code implementing `solve()`, nothing else.

A.6 Illustration of Evaluation Metrics

A.6.1 Calculation of Complexity

For methods that explicitly generate logical expressions, we measure expression complexity by averaging the number of Boolean operators `AND`, `NOT`, and `OR` in the output expressions when testing each baseline algorithm on a truth table.

The formula for calculating the number of operators is given as:

$$\frac{1}{N}\sum_{i=1}^{N}\text{ops}(E_i) \tag{9}$$

where $\text{ops}(E_i)$ represents the total operator count of the ith expression, defined as:

$$\text{ops}(E_i) = \texttt{NOT}_{num}(E_i) + \texttt{AND}_{num}(E_i) + \texttt{OR}_{num}(E_i) \tag{10}$$

In practical implementation, the *Quine-McCluskey* and *BDD* algorithms do not directly provide gate-level representations. Therefore, we calculate the total number of `AND`, `NOT`, and `OR` gates based on the *Quine-McCluskey* algorithm's Sum-of-Products form and the cube list generated by the *BDD* algorithm.

For the *Quine-McCluskey* algorithm, the gate count of the ith expression is given by:

$$\text{ops}(E_i) = \sum_{j=1}^{m_i} z_{i,j} + \sum_{j=1}^{m_i}\max(k_{i,j}-1,0) + \max(m_i-1,0) \tag{11}$$

where m_i is the number of terms in the ith *Quine-McCluskey* output, $k_{i,j}$ denotes the total number of literals in the jth term of the ith expression, and $z_{i,j}$ is the number of negations required in the jth term of the ith expression.

For the *BDD* algorithm, the gate count of the ith expression can be represented using the same formula (Equation 11), with differences only in the definitions of individual symbols: m_i represents the number of paths in the ith *BDD*, $k_{i,j}$ denotes the number of variables in the jth path of the ith expression, and $z_{i,j}$ represents the number of negations required in the jth path of the ith expression.

For neural network-based approaches, we consider five models in total, among which four (*DiffLogic*, *NeuraLUT*, *AI4EDA_TNet* and *SATNet*) only output predicted truth tables rather than explicit symbolic expressions. Consequently, their Boolean expression complexity cannot be derived directly. However, because their architectures bear a structural correspondence to certain Boolean operator compositions or LUT-like realizations, we can estimate their *equivalent Boolean circuit complexity* from the underlying architectural semantics. The details are as follows:

1) DiffLogic. *DiffLogic* stacks $L(k)$ layers of differentiable Boolean units, each containing $h(k)$ soft AND/OR operators. The first $L(k)-1$ layers provide intermediate conjunctive/disjunctive compositions, while the final layer collapses to a LUT-like enumeration of all output configurations. Thus, the equivalent Boolean complexity is:

$$G_{\text{DiffLogic}}(k) = \big(L(k)-1\big)\cdot h(k) + 2^k. \tag{12}$$

2) NeuraLUT. *NeuraLUT* parameterizes a lookup table with 2^k entries. This corresponds to a canonical realization with 2^k minterms in the underlying Boolean function. Therefore, its equivalent Boolean complexity is:

$$G_{\text{NeuraLUT}}(k) = 2^k. \tag{13}$$

3) AI4EDA_TNet. *AI4EDA_TNet* augments a LUT-like output layer with h hidden gated units, each corresponding to a logical operation or transformation. Its overall Boolean-equivalent complexity is estimated as:

$$G_{\text{AI4EDA_TNet}}(k) = h + 2^k. \tag{14}$$

4) SATNet. *SATNet* represents logical constraints using a set of learnable soft clauses. Its representational capacity is determined by the number of learnable clauses, which we take as its Boolean-equivalent complexity:

$$\chi \cdot (n_{\text{in}} + n_{\text{out}}) \tag{15}$$

We use the official *SATNet* implementation* and set $\chi = 4$. Since its clauses jointly constrain all variables, we report the total clause count instead of a per-output value.

A.6.2 Calculation of Accuracy

To ensure a unified evaluation process across all methods, we first convert all outputs into a standard format of 0/1 truth table predictions. Specifically, for methods that inherently generate Boolean expressions, we run these expressions on the evaluated input combinations to obtain 0/1 predictions. For methods that directly output 0/1 sequences or predictions, we use their predictions directly.

Each method produces a predicted truth table $\hat{Y}$. We denote the number of evaluated input combinations as S, and the number of output variables as M. Under this notation, both the predicted and ground-truth truth tables are represented as:

$$\hat{Y}, Y \in \{0,1\}^{S \times M} \tag{16}$$

where the s-th row corresponds to the s-th input combination, and the m-th column corresponds to the m-th output variable.

Based on the unified representation above, we define three evaluation metrics:

1) Bit-Wise Accuracy:

$$\text{bit_accuracy} = \frac{1}{S \times M} \sum_{s=1}^{S} \sum_{m=1}^{M} \mathbb{I}(\hat{y}_{s,m} = y_{s,m}) \tag{17}$$

This metric calculates the proportion of correctly predicted bits over the evaluated truth table (total $M \times S$ bits).

2) Sample-Wise Accuracy:

$$\text{sample_accuracy} = \frac{1}{S} \sum_{s=1}^{S} \mathbb{I}(\hat{\mathbf{y}}_s = \mathbf{y}_s) \tag{18}$$

This metric counts a prediction as correct only if the entire output vector for a given input sample matches the ground-truth output exactly. It measures the proportion of input samples for which all output bits are predicted correctly.

3) Per-Output Correctness (POC):

$$\text{POC} = \frac{1}{M} \sum_{m=1}^{M} \prod_{s=1}^{S} \mathbb{I}(\hat{y}_{s,m} = y_{s,m}) \tag{19}$$

Following the single-output evaluation protocol commonly used in symbolic regression, and the exact-match criterion adopted in logic synthesis, we consider an output correct only if its predicted expression exactly matches the ground-truth function across all evaluated input combinations. Extending this definition to the multi-output setting, the POC metric reports the proportion of outputs whose expressions satisfy this exact-match requirement.

*https://github.com/locuslab/SATNet

A.6.3 Explanation of Time Efficiency

Across all categories of methods, efficiency refers to the wall-clock time required to produce the final expressions (or predictions) for a given dataset instance. This definition is implementation-agnostic and applies uniformly to logic-synthesis, symbolic, neural, and LLM-based approaches.

Logic-synthesis / symbolic-regression (non-learning) methods. The reported cost corresponds to the single-pass solving time, i.e., loading the dataset, generating expressions from the training set, and producing predictions on the test set using the generated expressions.

Machine-learning-based methods. For *DiffLogic*, *AI4EDA_TNet*, *NeuraLUT* and *SATNet*, the reported cost is the end-to-end time, including both training and inference. This consists of training the neural network on the training set and then predicting on the test set using the trained model. For *Boolformer*, since a pretrained model is used, we report only the time to generate expressions from the pretrained model and the inference time on the test set.

LLMs. No fine-tuning is performed. The total time includes the API calls used to generate expressions from the training set.

A.7 Analysis of Evaluation Approaches for Baselines

To comprehensively position our proposed benchmark, we examined the evaluation methodologies for baseline methods.

Among mature logic synthesis and symbolic regression methods, *ABC* [5] demonstrates its advantages using clear performance metrics such as runtime, node count, and resource consumption. The *DEAP* framework [17], on the other hand, evaluates its strengths in rapid prototyping and developing custom evolutionary algorithms by comparing code length, example richness, and clarity of practical implementation.

Among neural network-based methods, *NeuraLUT* [2] evaluates accuracy against hardware cost (LUTs, F_{max}, latency, and area-delay product) on MNIST, jet substructure, and network intrusion detection tasks. *DiffLogic* [39] utilizes multiple classic datasets, including the logical reasoning classification MONK series (MONK-1, MONK-2, MONK-3), the Adult Census dataset, the medical dataset of Breast Cancer, and the image datasets MNIST and CIFAR-10. Its evaluation metrics encompass classification accuracy, model parameter count, inference speed (CPU/GPU execution time), memory usage, and computational complexity (OPs/FLOPs). *ReducedLUT* [8] is evaluated on the Jet Substructure (JSC) dataset from particle physics classification tasks and the MNIST dataset, using metrics including physical lookup table (P-LUT) utilization, maximum circuit operating frequency (F_{max}), and neural network test accuracy. *Boolformer* [14] performs evaluations on synthetic Boolean function datasets, real-world binary classification datasets from the PMLB collection, and Gene Regulatory Network (GRN) inference benchmarks. Its evaluation metrics include fitting accuracy and perfect recovery rate on synthetic datasets, F1 scores for real-world classification tasks, and dynamic accuracy, as well as structure-prediction-related binary classification metrics like F1 score and Youden's J in gene network tasks.

Key Differentiators: Compared with these evaluation approaches, LogicSR maintains diverse data sources while introducing noise and constructing partially observable datasets, thus enabling a more effective assessment of algorithm robustness. Furthermore, our evaluation metrics are more refined, involving accuracy analysis at different granularities (bit-wise, sample-wise, and per-output), along with the simplicity of expressions and time efficiency to assess the outputs. This multi-perspective approach provides a unified evaluation framework with broader applicability.

A.8 Discussion of Downstream Applications

The task domains involved in LogicSR (such as gene regulation logic and circuit Boolean functions) can all be regarded as related application scenarios:

In biological Boolean networks (such as BBM), the regulatory relationships of genes are usually modeled using Boolean update rules to describe which regulatory factors activate or inhibit target

genes under which combination of conditions. Real biological data are affected by sequencing errors, cellular heterogeneity, and experimental limitations, and thus capture only sparse, noisy, and discrete gene states. The significance of LogicSR lies in its ability to evaluate the restoration of logical dependencies under noise and partial observation. This ability can help propose more interpretable and analyzable regulatory hypotheses. Related works such as LogicNet [28] model gene expression as a probability distribution and jointly identify network topology and underlying logical structure; Research [20] systematically studied the accuracy of Threshold rules in Simple Boolean Threshold Networks relative to real biological logic.

In the EDA field, under the condition that a certain error is allowed, the Boolean functions recovered through learning can be used as interpretable approximations of circuit modules. For example, the average relative error of the FPGA approximate multiplier of SMApproxLib [46] is within 0.126, and the complexity is approximately within 64 LUTs; Research [18] proposed a method for automatically generating a table data classification predictor circuit, with the number of logic gates controlled within 300 and an accuracy of 40-100% on 33 tasks.

In addition, EDA pipelines typically span multiple stages such as logic synthesis, optimization, and mapping. The ability to directly restore Boolean functionality from data means that some costly steps can be skipped, thereby enhancing design efficiency.

A.9 Extending LogicSR Beyond Boolean Functions Using Basic Logic Gates

Our algorithms are flexible and can be extended to larger operator sets in multiple ways, as shown in Table 16, with detailed extensions provided below.

Table 16: Logic types and their extension schemes.

Logic Type	Extension Scheme	Example Operations
Multi-valued (ternary)	Co-domain: $\{0,1\}^m \to \{0,1,2\}^m$	Post algebra: MIN, MAX, Cycle
Fuzzy logic	Co-domain: $\{0,1\}^m \to [0,1]^m$	t-norms/s-norms (Gödel, product)
Probabilistic Boolean	Mapping: $\{0,1\}^m \to \Delta(\{0,1\}^m)$	Belief functions, Noisy-AND/OR

A.9.1 Multivalued and Fuzzy Logic

LogicSR can be extended to non-Boolean functions by the following measures:

1) Extended Operator Set: Generalize binary operator sets to k-ary operators (e.g., ternary MIN/-MAX) and continuous-value operators (e.g., t-norms for fuzzy logic). During the second-stage merging of the algorithm, parent node selection extends to sequential sampling without replacement of nodes, with other strategies preserved or adaptively optimized for specific settings.

2) Generalized Data Representation: Extend the binary value set $\{0,1\}$ to discrete sets $\{0,\cdots,U-1\}$, expanding node representation vectors from 2^n to U^n for multi-valued logic. For fuzzy or probabilistic logic, continuous sampling replaces enumeration with t-norm/t-conorm computation.

3) Evaluation Metrics: The current bit-wise and sample-wise accuracy metrics can be replaced by multi-class accuracy for multi-valued logic, or to continuous similarity measures for fuzzy logic.

A.9.2 Implication Operators

Scaling to more complex reasoning tasks with implications requires handling of relationships between logical rules, for example, $A \to B$ or chained rules $A \to B \to C$. Although these tasks fall under the category of rule-based reasoning, they can ultimately be encapsulated in a Boolean function: The current operator set is completely adequate in terms of expressive power, since $A \to B$ can be expanded using `AND/NOT` ($\neg A \vee B$). The following complementary methods could make LogicSR naturally extend to such tasks:

1) Extending the Operator Set to Include Implication: Implication ($\neg A \vee B$) could be directly added to the operator sets of LogicSR without any modifications, as it is a 2-ary operator, and every node could be regarded as an implication relation between its parenting nodes.

2) Modeling with a Higher-Depth Network: As multi-step implication could be represented by network structures with a higher depth, and LogicSR's two-stage synthesis framework could increase

Table 17: Comparison of existing logic and symbolic regression benchmarks.

Benchmark / Feature	Domain	Scalability	#Samples	Noise / Sampling	MIMO	Boolean	Metrics
EPFL [1]	Circuit	Fixed	23	×	×	✓	Size / Depth
ISCAS'85/'89 [6]	Circuit	Fixed	85: 10, 89: 31	Partial	✓	✓	Size / Depth
IWLS'25 [12]	Circuit	Fixed	100	Partial	×	×	Size / Depth
FOGA 2023 [22]	Logic synthesis	Fixed	30+	×	✓	✓	Hamming distance, Cryptographic property sum
LLM-SRBench [43]	Real-valued	Scalable	239	✓	×	×	NMSE, numeric precision, etc
SRBench [9]	Real-valued	Fixed	252	✓	×	✓	Complexity, R^2
BBM Boolean Models [37]	Biological networks	Fixed	244	Partial	✓	✓	Function-level metrics
MCBF [45]	Biological networks	Fixed	2687	✓	✓	✓	Boolean complexity, sensitivity, enrichment p-value
LogicSR (ours)	**Boolean networks**	**Scalable (5–80+)**	**1000+**	✓	✓	✓	**Bit/Sample accuracy, complexity**

the network depth by including more gating nodes and input variables in the network structures, without introducing any new mechanisms.

A.10 Comparison with Other Benchmarks

As shown in Table 17, existing benchmarks each exhibit clear limitations. In this section, we systematically compared LogicSR with the current three major types of related benchmarks (Table 17) - the logic synthesis benchmark (EPFL, ISCAS, IWLS) [1, 6, 12], and the continuous symbolic regression benchmark (LLM-SRBench, SRBench) The Benchmark [43, 9] and the Boolean network benchmark (BBM, MCBF) in systems biology [37, 45] are summarized as follows:

The logic synthesis benchmark (EPFL [1], ISCAS [6], IWLS [12]) only supports exact synthesis. The circuit structure is fixed and unadjustable, does not contain noise or some observable data, and cannot be used to evaluate generalization ability. Its evaluation metrics are limited to size/depth and do not involve expression recovery or learning performance.

Continuous symbolic regression benchmarks (LLM-SRBench [43], SRBench [9]) operate in the real number field with continuous operators $(+, -, \times, \sin, \text{etc.})$, which are completely incompatible with Boolean logic semantics; Its accuracy index relies on continuous residuals (such as NMSE, R^2), and cannot be applied to the recovery task of scatter truth tables. Furthermore, the combinatorial explosion of Boolean logic expressions makes them significantly superior to continuous SR in structural scale and solution difficulty.

Although some benchmarks (BBM [37], MCBF [45]) also contain Boolean structures, they usually have a fixed scale and focus a single domain. Moreover, they lack adjustable operators and complex network structures that enables scaling of tests.

Key Differentiators: Unlike the above three types of benchmarks, LogicSR offers for the first time an extensible input scale (5-80+), controllable network depth, structural complexity, and node sharing mode. Supports noise, partial truth table sampling, multi-output (MIMO) dependency structure, uniformly evaluates multiple types of methods, and cross-domain tasks. It exceeds the existing benchmarks in terms of scalability and complexity.

A.11 Limitation and Future Work

LogicSR still exhibits several limitations that provide opportunities for future investigation:

1) Structural Constraints: The binary fan-in design ensures validity but limits expressiveness for highly interconnected circuits, while the simplified operator set may not fully capture optimization challenges in industrial gate libraries (e.g., AOI, MUX). Extending to multi-fanin nodes and technology-mapped operators would better reflect industrial complexity.

2) Scalability Constraints: Evaluation remains constrained to practical-scale problems. Distributed frameworks and hierarchical generation could enable benchmarking of industrial ultra-large-scale systems (e.g., ASICs).

3) Domain Applications: While the benchmark focuses on general Boolean regression, the synthetic data generation currently lacks domain-specific inductive biases; in this work, we compensate for this by incorporating real-world datasets, but future extensions may further explore domain-knowledge-driven synthetic logic generation and richer forms of logic found in real systems, such as temporal logic in control and scheduling scenarios.

A.12 Dataset Licenses

We preserve the original licensing terms of BBM and IWLS. Specifically, the BBM dataset retains the original copyrights of the respective model authors and/or publishers, while IWLS is released under the MIT License. The synthetic MIMO dataset is generated entirely by our framework and is released under the MIT License.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state that this work proposes a benchmark for symbolic regression based on Boolean networks, with claims supported by experiments.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The work discussed limitations in the last part of appendix.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: This paper only provides a new algorithm and dataset along with the experiment results on various methods, but not any theoretical studies.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides all experiment settings, details and result analysis in the main paper and appendix. Related code and datasets are also provided in supplementary materials and GitHub repository.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The paper provides open access to both dataset and code in supplementary materials and GitHub repository.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: The paper specifies all training and testing details mainly in appendix (e.g. hyperparameters, evaluation metrics, data splits, etc.) for a full understanding of the results.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: Due to the substantial computational cost of experiments on large-scale datasets, the main comparison runs each method once per instance and reports no error bars. Controlled studies in the appendix report paired Wilcoxon signed-rank tests, and LogicSR-Core recommends three seeds so that users can obtain error bars at low cost.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper reports the computer resources in Section 4, provides runtime comparisons in Figure 5(a), and includes the implementation code to facilitate reproducibility.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The paper conforms with all Code of Ethics of the conference.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: Though the benchmark we provide in this paper has applications in fields like natural science, it does not have any large societal influences. It currently serves only to facilitate scientific discoveries and research.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release a model, and datasets do not have safety risks as they are either provided in official repositories or automatically generated.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: This paper cites related data and original works properly, with the details provided in supplementary materials and GitHub repository. Dataset licenses are detailed in appendix A.12, and the original licenses are preserved for datasets derived or obtained from third-party works.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We introduced new datasets in this paper. They are provided in supplementary materials and GitHub repository, with features documented in appendix.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects, it is only related to logical discovery mainly in the field of science.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper evaluates some LLM methods and the use and results are described in paper.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.