MinBLoG: Minimization of Boolean Logic Functions using Graph Attention Network

Prianka Sengupta
Aakash Tyagi
Jiang Hu
Texas A&M University
College Station, Texas, USA

Vivek K Rajan
Hesham Mostafa
Somdeb Majumdar
Intel Inc
USA

Abstract

The initial steps of logic synthesis of digital designs involve finding minimized representations of Boolean logic functions. Existing optimization methods rely on iterative minimization operations that can result in a rapid increase in the runtime when the number of variables and terms of the Boolean functions increase. We propose a graph attention network (GAT) based logic minimization approach called MinBLoG, to narrow down the solution search space for Boolean functions. Our approach achieves more than 96% accuracy in identifying implicants that are a part of the minimized solution and ensures functional equivalency through correctness checking procedures. Experiments show that MinBLoG delivers minimization results for a wide range of Boolean functions significantly faster than well-known existing methods.

CCS Concepts

• **Hardware → Circuit optimization**.

Keywords

Boolean Logic Minimization, Graph Attention Network, Machine Learning

ACM Reference Format:

Prianka Sengupta, Aakash Tyagi, Jiang Hu, Vivek K Rajan, Hesham Mostafa, and Somdeb Majumdar. 2024. MinBLoG: Minimization of Boolean Logic Functions using Graph Attention Network. In *2024 ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD '24), September 9–11, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3670474.3685962

1 Introduction

Optimization of combinational logic plays a crucial role in meeting timing, area and power goals in the implementation flow of digital designs. Synthesis tools perform optimization at various stages, such as pre-mapping, tech-mapping and post-mapping to meet the design constraints. Many of these optimization algorithms are computationally complex and significantly stretch the runtime of the synthesis process, especially for large and complex designs. To move past this runtime bottleneck, exploratory research is necessary to revisit and re-imagine underlying optimization algorithms that modern synthesis tools are built upon. The rapid advancement of machine learning (ML) and its application in the last decade places it as a strong tool to re-architect legacy optimization algorithms for not only better runtime, but also higher quality of result (QoR). The capabilities of state-of-the-art ML models have been demonstrated in [12], [13] by providing beneficial heuristics and accelerating implementation flows. Further integration of ML in the core layers of synthesis has the potential to leapfrog the performance of existing algorithms and aid in quicker time to convergence.

In this paper, we propose a novel and fast method for minimization of Boolean logic functions using a graph-based machine learning model (**MinBLoG**). This work aims to be a step towards redesigning underlying synthesis methods by leveraging the advances of machine learning. The main contributions of the work are as follows:

- Develop and train a graph attention network (GAT) model for minimizing Boolean logic functions that predicts minimized implicants with ~96% accuracy.
- Develop robust correctness and coverage checking procedures to ensure functional correctness of the results obtained from the graph model prediction.
- Achieve over 40-100X faster average runtime compared to Espresso [2] for a wide range of Boolean functions.
- **MinBLoG** delivers the minimization quality that is within 92% of well-known minimization tool Espresso [2] for a wide range of benchmark functions.

2 Related Research

The process of synthesizing digital designs requires both technology independent and dependent optimization steps of the combinational logic function defined in the design. Technology independent optimization attempts to minimize the number of variables and logic operations required to represent the Boolean logic functions. This minimization task deals with Boolean functions represented as either sum-of-product (SoP) or product-of-sum (PoS) and is referred to as Two-Level Logic Minimization. Throughout the years, two main categories of logic minimization algorithms have been established. Algorithms such as Quine-McCluskey method [11], Karnaugh-Map clustering method [5] perform exact two-level minimization, delivering a global minimum solution (minimum cover). Another class of heuristic algorithms, such as Espresso [2], MINI [10] and BOOM [7], aim to speed up time-to-solution by leveraging pre-defined heuristic to reduce the number of iterations needed to arrive at an acceptable solution, sacrificing the guarantee of a minimum cover. A major drawback of both exact and heuristic algorithms is their

This work is licensed under a Creative Commons Attribution-NonCommercial International 4.0 License.
MLCAD '24, September 9–11, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0699-8/24/09
https://doi.org/10.1145/3670474.3685962

worst-case computational complexity and runtime, which can grow exponentially for Boolean functions with a large number of input variables and product terms [4].

Multiple recent research contributions aim to leverage advancements of machine learning to find compact representation of Boolean logic. Boolformer [6] builds a transformer-based solution to find compact representations of unknown logic functions when provided with a truth table of the function. While this may be useful in the domain of design verification, the solution is currently limited to 10 variables and does not guarantee functionally equivalent representation for partially specified functions. Different representations of logic networks such as And-Inverter-Graphs (AIG) [15] and Binary Decision Diagram (BDD) [3] have also been explored, along with associated algorithms for achieving minimized representation. Minimization of AIG using machine learning driven heuristics is explored in the work ABC-RL [19] to decrease the node and depth metrics using ABC [15]. Another relevant work GAMORA [17] accurately identifies known logic functions in a cluster of nodes of an AIG. Although logic minimization was not explored in [17], their approach could potentially allow replacing identified functions using optimized macros to reduce the number of nodes and depth.

The literature in this domain shows a growing trend of ML heuristics being used to improve existing logic minimization algorithms. However, there is an evident lack of work that aims to replace legacy iterative algorithms and solve runtime bottlenecks by using machine learning to narrow down the solution search space. The approach in MinBLoG moves away from iterative algorithms and thus is able to overcome runtime challenges while maintaining functional correctness.

3 Background

There are different representations of Boolean logic functions, and a set of terminologies are used to describe the function properties. Some of the relevant terminologies and concepts are introduced in this section.

3.1 Terminologies

The following terminologies are widely used in Boolean logic literature and is necessary to discuss the methodology.

Literal: The inverted or non-inverted version of a variable. *Product/Cube:* A collection of literals joined by logical AND.

Sum-of-Product/SoP: A collection of cubes joined by logical OR. Boolean functions are often represented in a SoP form.

Implicant: A cube that implies the value of a Boolean function. An implicant may not contain all input variables.

Prime Implicant (PI): An implicant that is not contained by any other implicant.

Cover: A set of implicants sufficient to define a function. A function can have more than one cover.

Minimal Cover: A cover of a function with irredundant prime implicants which is not contained in any other cover.

Minimum Cover: A cover of a function with minimum number of prime implicants and literals. This is the global minimum of the function.

3.2 Boolean Functions and Minimization

Most common representations of Boolean functions are Sum-of-Product (SoP) or Product-of-Sum (PoS) forms. The cubes in a SoP function can be conveniently represented as a two-dimensional matrix using Positional Cube Notation (PCN) [10]. SoP functions encoded as PCN matrices provide a computationally efficient method for various Boolean operations. A Boolean SoP function F consisting of 3 cubes ($c = 3$) and 3 variables ($n = 3$) is shown in equation (1), along with its minimized form F_{min}.

$$F = xyz + \bar{x}yz + \bar{x}\bar{z}\ F_{min} = yz + \bar{x}\bar{z} \tag{1}$$

Each literal of an SoP function is represented using the following notation: • Non-inverted variables: 1 • Inverted variables: 0 and • Don't-Care variables: 2. Following this notation, the PCN representation of F and F_{min} will be integer matrices of size $c \times n$ and $c_m \times n$ as shown in equation (2).

$$P = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 2 & 0 \end{bmatrix}\ P_{min} = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 2 & 0 \end{bmatrix} \tag{2}$$

Here, P_{min} contains a smaller number of cubes c_m as well as literals compared to P while being functionally equivalent.

3.3 Logic Functions as Graphs

In parallel to PCN, logic functions are also often represented as graphs or trees to enable efficient processing in Boolean algorithms and logic synthesis. Some popular representations include Binary Decision Diagrams (BDD)[3], And Inverter Graphs (AIG)[18] and Majority Inverter Graphs (MIG) [1]. In this work we construct AIG graphs from the PCN representation of an SoP function. For the function in equation (1), we construct the graph shown in figure 1.

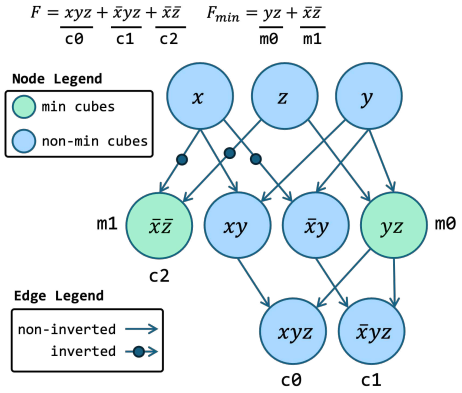

Figure 1: Graph representation of the function $F = xyz + \bar{x}yz + \bar{x}\bar{z}$ with prime implicants from F_{min} highlighted in green.

The graph consists of the input nodes x, y and z, which are progressively merged using logical AND nodes and inverting or non-inverting edges to construct the cube expressions, denoted by $c_0 = xyz, c_1 = \bar{x}yz$ and $c_2 = \bar{x}\bar{z}$. The merging of nodes is intentionally performed in a pairwise manner, instead of hierarchical manner which would require a smaller number of nodes. A Pairwise Merged Graph (PMG) contains intermediate nodes with unique literal combinations, yielding a useful set of node expressions. In figure 1, it

can be observed that the expressions of the prime implicants (PI) from the minimized function F_{min} appears in the PMG nodes $m0$ and $m1$. To further establish this observation, we conducted an empirical study by constructing PMGs from a varying number of variables and cubes. For each graph, we check the percentage of PI expressions present in the expression of the nodes. This probability of finding PI expression in nodes is highly dependent on the pairwise selection of input nodes for merging. We explored different node-pair selection schemes: a) Random select b) Moving window of size 2 c) Odd-even index and d) Pair inverted literals with non-inverted literals, with padding if necessary. Results of our empirical study shows that option d) provides the best balance of node count and probability of finding the PI expressions in those nodes. The results in figure 2 shows the percentage of PI expressions found in the PMG nodes for functions varying in size from 8 to 40 variables. For any larger functions with more variables, the number of nodes PMG will increase, improving the probability of finding nodes with implicant expressions. Based on the observation that

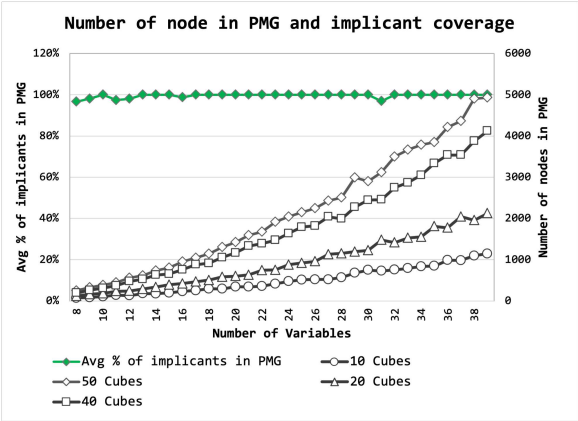

Figure 2: Empirical analysis of node count in Pairwise Merge Graph with respect to variables and cubes. It also shows average percentage of minimized implicant expressions present in node expressions of PMG.

the PMG of a function contains nodes that represent the solution of the minimization problem allows us to represent the minimization task as a graph node classification problem. Node classification is an established domain of ML, providing us with powerful learning models that are well suited for this task.

4 Methodology

Minimizing an SoP function F is the process of computing a set of implicants C_{min} that cover the set of cubes C of F. The set of literals from C_{min} and C can be denoted by the sets L_{min} and L respectively. There can be different goals for technology independent minimization: • Minimizing number of cubes: $|C_{min}| < |C|$ • Minimizing number of literals: $|L_{min}| < |L|$ • Minimizing both Cubes and Literals. An ideal minimization process should produce a minimized SoP form F_{min}, containing a set of irreduntant prime implicants necessary to cover F. If the PIs of F_{min} contain no redundant literals, then it can be considered as a minimum cover of F. Most heuristic driven minimization approaches such as Espresso constructs minimal covers that may contain redundant literals. In this work, we apply ML to identify PIs from the PMG representation of SoP functions to construct a minimal cover and minimize the number of literals required. Therefore, the quality of minimization will largely depend on the availability of irredundant PI nodes in the PMG and the accuracy of the GAT model to classify such PI nodes. For a highly accurate model, any minor lapse or conflict in functional equivalency is corrected in the post-inference stage without incurring significant computational overhead.

4.1 Logic Optimization as an ML Task

Applying graph-based ML models to solve the logic optimization problem reduces the computational complexity compared to existing iterative heuristics. An ML model yields results/predictions in polynomial and significantly shorter time. For example, the computation complexity of inference for a trained Graph Attention Network (GAT) model is $O(Lvd^2)$ [9], where v is the number of graph nodes, L is the number of convolution layers, and d is the feature vector size. Constructing the proposed approach requires realizing a training pipeline with the following components.
Training Pipeline Components:

- A synthetic SoP function generator to create training samples of PCN encoded functions of arbitrary variable and cube size.
- A method to construct Pairwise Merged Graph (PMG) from PCN encoded SoP functions and compute node features. True minimized implicants of training samples are used to label nodes with class 0: Non-implicant nodes and class 1: minimized implicant nodes.
- A graph attention network (GAT) to classify PI nodes (class 1) with an aim to minimize mis-prediction.

Once a graph model is trained, a simpler inference pipeline can be constructed using the following components
Inference Pipeline Components:

- An input stage that accepts a PCN encoded SoP function and constructs a PMG.
- An inference stage where the pre-trained graph model predicts a subset of nodes as class 1 (PI nodes).
- A correctness check to filter out False-Positive (non-PI node predicted as PI node) prediction and coverage check for False-Negative (PI nodes predicted as non-PI nodes) prediction and include missing implicant/terms in the final solution.

An overview of the training and inference pipeline is shown in figure 3. The details of each step are elaborated in the following sections.

4.2 Generating Functions

A PCN encoded SoP function is represented as a $c \times n$ matrix, where c is the number of cubes and n is the number of variables. A program for generating SoP function was developed that can generate single-output functions in PCN encoded format with controllable distribution between ON-Set, OFF-Set and Don't Care-Set. The true minimized implicants of these functions were used to label prime-implicant nodes for supervised learning.

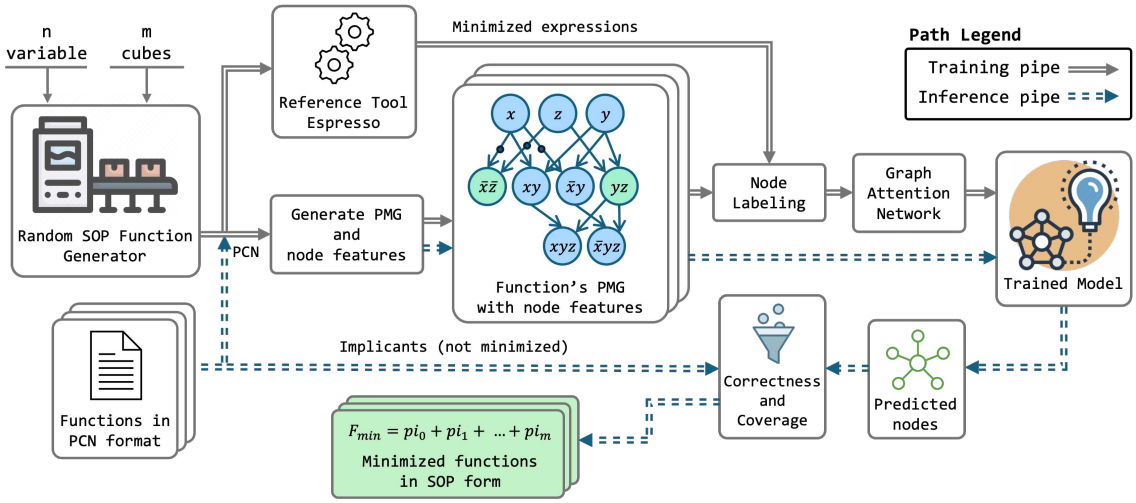

Figure 3: Overview of training and inference pipeline of MinBLoG.

4.3 Graph Representation

The maximum number of nodes required to construct a pairwise-merge graph for PCN matrix of size $c \times n$ is deterministic and can be easily calculated. Merging the n input nodes in a pairwise manner for a single cube will require $l = n - 1$ merging layers. After layer 1, each subsequent layer will contain 1 node less than the previous one. Thus, the sum of nodes across all layers can be calculated as equation (3):

$$\begin{aligned} Cube_{nodes} &= n + n - 1 + n - 2 + \ldots + n - l \\ Cube_{nodes} &= n(n+1)/2 \end{aligned} \tag{3}$$

The remaining cubes will share the input nodes and in the worst case, will require $n(n+1)/2 - n$ nodes to implement the merging. Therefore, the worst case bound for the number of nodes in a PMG can be calculated as in equation (4):

$$\begin{aligned} PMG_{nodes} &= n(n+1)/2 + (c-1)n(n+1)/2 - n \\ &\text{This can be simplified to} \\ PMG_{nodes} &= cn(n+1)/2 - n \end{aligned} \tag{4}$$

Considering the number of nodes required, the PMG construction algorithm has complexity $O(cn^2)$. However, in practice many nodes can be reused between cubes and the actual observed complexity has linear dependence on n, as seen in figure 2.

4.3.1 Node Features. To enable a graph-based ML model, the SoP graph nodes require meaningful features saved as vectors. The following node features were computed for each node.

- Node Type Encoding (INPUT, AND).
- Number of outputs from the node.
- Distance from the input nodes.
- Number of variables, inverted variables, non-inverted variables and don't care variables in native node expression (N_{expr}).
- Result of checking if N_{expr} implies the original function.

The first three features are necessary to provide the GAT model spatial knowledge of each node in a PMG. The 4th feature provides informs the model regarding the characteristics of the node expression and last feature acts as a preliminary indication of non-implying nodes that should not be considered as a PI node.

4.3.2 Node Labels. For specifying node labels, the N_{expr} is compared against the true minimized-implicants to determine between: • Class 1/PI nodes: Node expression contains a PI • Class 0/Non-PI nodes: Node expression does not contain a PI. The GAT model was trained to perform binary classification to identify PI nodes.

4.4 GAT Model Training

A Graph Attention Network (GAT) [16] model was configured to learn the binary node classification task. The GAT model consisted of 5 GAT convolution layers, 4 attention heads, 64 hidden layers and target-to-source message propagation. The model was trained with 5000 synthetically generated SoP functions and evaluated with 1000 unseen SoP functions.

4.5 Correctness and Coverage Check

SoP minimization is a deterministic task, and a correct solution must ensure complete coverage of the ON-Set and avoid overlap from the OFF-Set of the function defined in the n dimensional Boolean space by the original function F. A solution can be correct yet sub-optimal if the minimized SoP contains redundant literals or cubes. In the context of the GAT model prediction, the following unwanted outcomes can be expected:

- *False Positive(FP): True=0, Pred=1*
  Wrongly predicts a non PI-node as a PI-node. These results must be excluded from the solution.
- *False Negative(FN): True=1, Pred=0*
  Wrongly predicts PI-node as non PI-node. The lapse in function's coverage due to FN predictions must be amended.

Given the statistical nature of the GAT prediction process, False-Negatives and False-Positives will occur during every inference. The logical sequence of ensuring correctness is shown in figure 4.

4.5.1 Filtering False-Positives. Wrongly predicted PI-node can be detected by computing the intersection of the predicted implicant

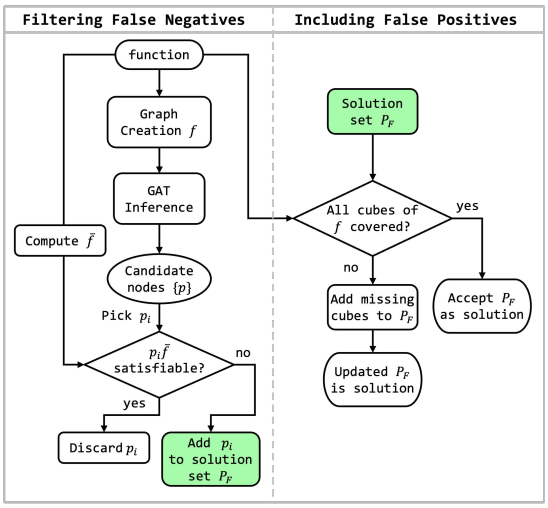

Figure 4: Logical steps of correctness checking and coverage amendments for a prediction set of implicants

against the original functions OFF-Set. If the intersection is not empty, then the predicted implicant does not imply the Function. In formal definition:

A predicted implicant p is valid iff $p\bar{F}$ is un-satisfiable

To perform this check, the input function must be inverted. Inversion of a Boolean function is necessary when the Off-set is not provided in the function's PCN specification. The Off-set can be achieved efficiently applying the unate recursive complement algorithm [14] on the union of a function On-set and DC-set (if-any). We implement a fast vectorized version of Boolean function inversion and compute the result of $p\bar{F}$ to filter out wrongly predicted nodes. Although the worst-case runtime complexity of inversion is $O(2^n)$, the real-world complexity of our implemented function was linear with n, resulting from efficient tautology checks and algebraic minimization heuristics used for early convergence.

4.5.2 Including False-Negative Coverage. When PI-nodes are predicted as non PI-nodes, the minimization solution will fail to cover the ON-Set of the function. Fortunately, this lack of coverage can be checked by inspecting how many of input cubes are covered by the correctly predicted implicants. Any uncovered cubes can be included as a part of the solution to ensure the coverage of the solution. The computational complexity of checking coverage is $O(cn)$ where c is the number of cubes and n is the number of variables. Although this approach degrades the minimization quality of solution, the effect was not significant in our experiments.

5 Experiment Results

5.1 Accuracy of GAT Model

The training of the GAT model was conducted using 5000 SoP functions and their corresponding graphs. The node classes in the training samples were imbalanced as there is large number of non PI-nodes in each graph compared to PI-Nodes. This imbalance was compensated with a dynamic class bias being applied while computing the loss of each batch. The binary classification metrics for the GAT classifier is shown in figure 5.

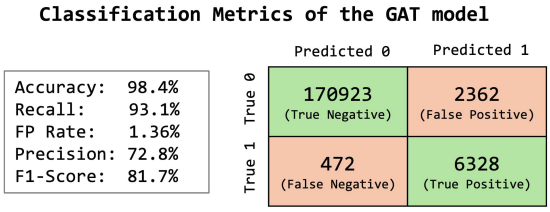

Figure 5: The classification metrics of the GAT classifier for classifying the nodes of PMGs from 1000 SoP functions

The success rate of the model in identifying minimized implicants for large functions were also analyzed and can be seen in figure 6. For a set of function ranging from 50 to 200 variables and 200 cubes, the model achieves >92% success rate of detecting minimized-implicants.

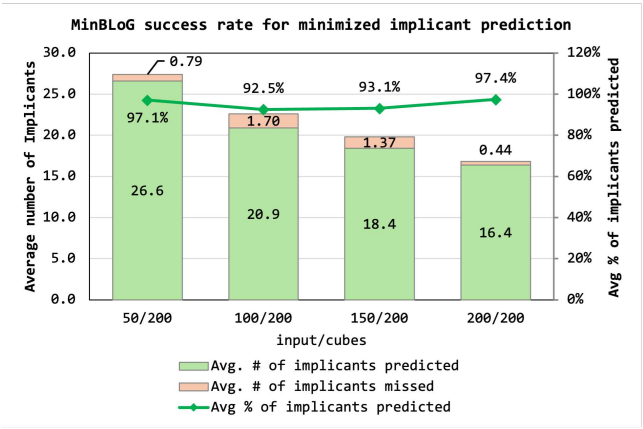

Figure 6: The probability of the GAT model to correctly identify PIs in a function's PMG. Higher identification rate reduces time spent in subsequent correctness steps.

5.2 Runtime and QoR Results

The reduction of computational complexity and speeding up the time to solution was one of the primary driving factors of this research. Figure 7 shows the growth in runtime for the main components of MinBLoG with respect to the number of literals of a function, highlighting the graph building stage to be the largest contributor to the overall runtime of MinBLoG.

The runtime and quality of minimization from MinBLoG was compared against legacy logic minimization algorithm Espresso [2] (Heuristic Mode), ABC[15], TTMin [8] and BOOM-II [7] as well as recent works from [6], [19] using synthetic benchmarks of different function sizes. The type of the function was also varied between • FR type: On-Set and Off-set are defined, • FD type: On-Set and DC-set are defined and • F type: Only On-Set is defined. We analyzed Espresso in both Exact and Heuristic mode and chose to exlclude exact mode from the results as it required significantly longer runtime for most testcases and provided inconsistent literal reduction compared to the heuristic mode. We also evaluated the recent work Boolformer [6] and found its capability limited to only 10 variables. Moreover, the results from Boolformer for the functions were not functionally equivalent on many occurrences. For each tool included in the results, we report a quality of results

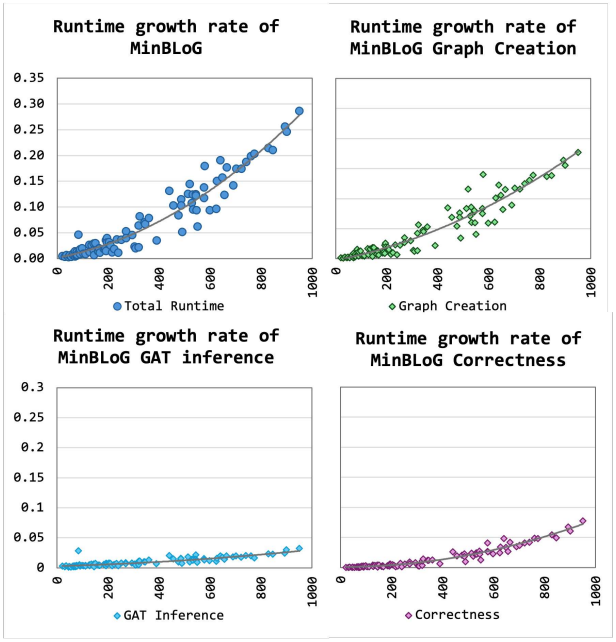

Figure 7: The growth of runtime for main components of MinBLoG with respect to the number of literals ($cube \times variables$).

(QoR) metric by calculating the ratio between original number of literals and the number of literals in the minimized form. A higher QoR number indicates a better literal reduction ratio. From the results listed in table 1, it can be observed that the runtime of Espresso increases rapidly for an increase in size of FR type functions. Although this slowdown doesn't occur for FD and F type functions, the achieved QoR is subpar compared to MinBLoG. Logic minimization in ABC [15] and ABC-RL [19] is treated as an AIG node minimization problem and iterative re-structuring of the AIG is performed to minimize the function. However, this approach does not yield significant literal reduction as seen in the table 1. Boom-II delivers results only for FR type functions and fails to reach QoR results compared to Espresso or MinBLoG. MinBLoG is the only tool in the comparison that is able to provide consistent literal reduction across all function types (FR, FD and F) while being functionally correct. For large FR functions with 400 and 800 cubes, MinBLoG delivers results more than 40X and 100X faster respectively. Therefore, it can be concluded that MinBLoG outperforms the existing methods in runtime and yields better QoR with functionally correct minimized results.

6 Conclusion

In this work we presented a Boolean logic minimization method that relies on a graph attention network to identify minimized implicants of Boolean logic functions. Using a graph representation constructed from cubes of SoP functions, the trained GAT model is able to identify minimal cover implicants for a function, while ensuring functional correctness and coverage. The proposed method delivers significantly faster runtime over Espresso and more recent methods while maintaining near ideal quality of result.

Table 1: Runtime and QoR Comparison of MinBLoG against existing solutions for a wide range of synthetic functions.

Testcases		**Espresso(H)**[2]		**ABC**[15]		**ABC-RL**[19][1]		**BOOM-II**[7][2]		**MinBLoG [This Work]**	
Func. Type	Var/Cube	Runtime(s)	QoR	Runtime(s)	QoR	Runtime(s)	QoR	Runtime(s)	QoR	Runtime(s)	QoR
FR Type On-Set & Off-Set	100/400	548.6	100.7	16.5	2.0	1143.4	2.0	0.1	53.4	13.5	94.6
	100/800	3625.4	91.4	31.4	2.0	1861.6	2.0	0.6	47.2	31.0	88.6
	200/400	2520.0	283.0	48.8	2.0	3927.0	2.0	0.3	158.9	95.5	271.6
	200/800	16651.8	278.4	74.5	2.0	6008.1	2.0	3.1	156.3	219.6	247.0
	400/400	11574.5	704.2	142.8	2.0	Indefinite		0.3	395.4	779.3	664.2
	400/800	Indefinite long runtime		276.7	2.0	long runtime		2.3	386.8	1792.4	595.7
	Average	6984.1	291.5	98.5	2.0	3233.0	2.0	1.1	199.7	533.9	327.0
FD Type On-Set & DC-Set	100/400	0.4	1.7	19.5	1.7	489.7	1.6	FD Type Functions are not supported in BOOM-II		14.5	85.1
	100/800	7.5	1.6	30.4	1.6	2392.0	1.5			33.4	75.0
	200/400	0.9	1.6	60.6	1.6	Indefinite long runtime				98.4	230.5
	200/800	41.5	1.8	120.7	1.8					226.2	211.0
	400/400	2.9	1.74	162.2	1.7					794.9	528.7
	400/800	1058.8	1.6	440.8	1.6					1828.3	487.9
	Average	185.3	1.7	139.0	1.7	1440.8	1.2			549.1	269.7
F Type On-Set	100/400	1.5	1.0	31.5	1.0	1483.6	1.0	F Type Functions are not supported in BOOM-II		14.1	82.5
	100/800	67.5	1.0	70.5	1.0	4830.6	1.0			32.2	73.5
	200/400	3.2	1.0	95.3	1.0	Indefinite long runtime				94.5	225.9
	200/800	1120.8	1.0	335.3	1.0					217.5	215.3
	400/400	569.3	1.0	319.0	1.0					872.8	523.4
	400/800	Indefinite long runtime		726.3	1.0					2007.5	492.7
	Average	352.4	1.0	263.0	1.0	3157.1	1.0			588.2	268.9

* All runtime measured on system with AMD-5600G with 16GB RAM.
[1] ABC-RL runtime are for multiple iterations till the literal reduction metric stops improving.
[2] BOOM-II runs are conducted with literal reduction goals set to meet results achieved from Espresso.

Acknowledgments

This work is partially supported by NSF (CCF-2106725 and CCF-2212346) and SRC (GRC-CADT-3013.001/3014.001).

References

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2016. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 5 (2016), 806–819. https://doi.org/10.1109/TCAD.2015.2488484

[2] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, USA.

[3] Randal E Bryant. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)* 24, 3 (1992), 293–318.

[4] Ashok K. Chandra and George Markowsky. 1978. On the number of prime implicants. *Discrete Mathematics* 24, 1 (1978), 7–11. https://doi.org/10.1016/0012-365X(78)90168-1

[5] Olivier Coudert. 1994. Two-level logic minimization: an overview. *Integration* 17, 2 (1994), 97–140.

[6] Stéphane d'Ascoli, Samy Bengio, Josh Susskind, and Emmanuel Abbé. 2023. Boolformer: Symbolic Regression of Logic Functions with Transformers. *arXiv preprint arXiv:2309.12207* (2023).

[7] Petr Fiser and Hana Kubátová. 2006. Flexible two-level Boolean minimizer BOOM-II and its applications. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*. IEEE, 369–376.

[8] Petr Fiser and David Toman. 2009. A Fast SOP Minimizer for Logic Funcions Described by Many Product Terms. In *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. IEEE, 757–764.

[9] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. arXiv:2004.11198 [cs.LG]

[10] S. J. Hong, R. G. Cain, and D. L. Ostapko. 1974. MINI: A Heuristic Approach for Logic Minimization. *IBM Journal of Research and Development* 18, 5 (1974), 443–458. https://doi.org/10.1147/rd.185.0443

[11] Jiangbo Huang. 2014. Programing implementation of the Quine-McCluskey method for minimization of Boolean expression. arXiv:1410.1059 [cs.OH]

[12] Min Li, Sadaf Khan, Zhengyuan Shi, Naixing Wang, Huang Yu, and Qiang Xu. 2022. DeepGate: Learning Neural Representations of Logic Gates. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 667–672. https://doi.org/10.1145/3489517.3530497

[13] Yingjie Li, Mingju Liu, Alan Mishchenko, and Cunxi Yu. 2023. Invited Paper: Verilog-to-PyG - A Framework for Graph Learning and Augmentation on RTL Designs. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–4. https://doi.org/10.1109/ICCAD57390.2023.10323741

[14] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits* (1st ed.). McGraw-Hill Higher Education.

[15] Alan Mishchenko et al. 2007. ABC: A system for sequential synthesis and verification. *URL http://www. eecs. berkeley. edu/alanmi/abc* 17 (2007).

[16] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

[17] Nan Wu, Yingjie Li, Cong Hao, Steve Dai, Cunxi Yu, and Yuan Xie. 2023. Gamora: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks. *arXiv preprint arXiv:2303.08256* (2023).

[18] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. 2017. Fast algebraic rewriting based on and-inverter graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2017), 1907–1911.

[19] Keren Zhu, Mingjie Liu, Hao Chen, Zheng Zhao, and David Z Pan. 2020. Exploring logic optimizations with reinforcement learning and graph convolutional network. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 145–150.

A Artifact Appendix

A.1 Abstract

The synthetically generated boolean functions used in this paper are available online for use in any future research in this domain.

A.2 Description

A.2.1 How to access. The boolean function dataset cant be accessed at the following Zenodo link:
https://zenodo.org/doi/10.5281/zenodo.13307943
Additionally, a GitHub repository is made available for MinBLoG:
https://github.com/puprianka/minblog.git

A.3 Details of Dataset

The dataset contains synthetically generated boolean functions in PLA format that can be used for testing runtime and QoR of logic minimization approaches (e.g Espresso, BOOM, MinBLoG etc.) The size of the functions ranges from 100 to 400 variables and 200 to 1000 cubes. The dataset is divided into three different types of boolean functions:

F-Type: Functions specified using their ON-Set.

FR-Type: Functions specified using their ON-Set and OFF-Set.

FD-Type: Functions specified using their ON-Set and Don't-Care-Set (DC-Set).

The filenames of the individual PLA files indicate the size of the function it contain. For example: fd_200_400-0.pla will contains an FD type function with 200 variables and 400 cubes. The suffix 0 at the end indicates it is the first of the 5 random functions generated with 200 variables and 400 cubes.