HeLO: A Heterogeneous Logic Optimization Framework by Hierarchical Clustering and Graph Learning

Yuan Pu
The Chinese University of Hong Kong
Hong Kong SAR

Fangzhou Liu
The Chinese University of Hong Kong
Hong Kong SAR

Zhuolun He
The Chinese University of Hong Kong
Hong Kong SAR

Keren Zhu
Fudan University
Shanghai, China

Rongliang Fu
Chinese University of Hong Kong
Hong Kong SAR

Ziyi Wang
Chinese University of Hong Kong
Hong Kong SAR

Tsung-Yi Ho
Chinese University of Hong Kong
Hong Kong SAR

Bei Yu
Chinese University of Hong Kong
Hong Kong SAR

Abstract

Modern very large-scale integration (VLSI) designs usually consist of modules with various topological structures and functionalities. To better optimize such large and heterogeneous logic networks, it is essential to identify the structural and functional characteristics of its modules, and represent them with appropriate DAG types (such as AIG, MIG, XAG, etc.) for logic optimization. This paper proposes HeLO, a hetero-DAG logic optimization framework empowered by hierarchical clustering and graph learning. HeLO leverages a hierarchical clustering algorithm, which splits the original Boolean network into sub-circuits by considering both topological and functional characteristics. A novel graph neural network model is customized to generate the topological-functional embedding (used for distance calculation in hierarchical clustering) and predict the best-fit DAG type of each sub-circuit. Experimental results demonstrate that HeLO outperforms LSOracle, the SOTA heterogeneous logic optimization framework, in terms of node-depth product (for technology-independent logic optimization) and delay-area product (for technology mapping) by 8.7% and 6.9%, respectively.

CCS Concepts

• **Hardware → Logic Synthesis**.

Keywords

Heterogeneous Logic Synthesis, Graph Learning

ACM Reference Format:
Yuan Pu, Fangzhou Liu, Zhuolun He, Keren Zhu, Rongliang Fu, Ziyi Wang, Tsung-Yi Ho, and Bei Yu. 2025. HeLO: A Heterogeneous Logic Optimization Framework by Hierarchical Clustering and Graph Learning . In *Proceedings of the 2025 International Symposium on Physical Design (ISPD '25), March 16–19, 2025, Austin, TX, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3698364.3705354

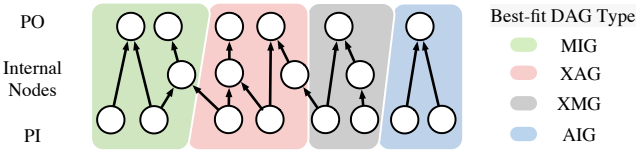

Figure 1: Illustration of heterogeneous logic optimization. Different logic components (sub-circuits) in the circuit are represented as different DAG types, on which the corresponding optimization strategies and algorithms will be applied.

1 Introduction

Logic synthesis stands as the foundational and essential step in most electronic design automation (EDA) flows, converting register-transfer-level (RTL) designs into implementations based on logic gates. Logic synthesis can be divided into two important phases: technology-independent logic optimization and technology mapping. In the phase of technology-independent logic optimization, Boolean networks are transformed into multi-level logic structures such as directed acyclic graphs (DAGs), including AND-inverter graphs (AIG) and majority-inverter graphs (MIG), etc. This transformation is followed by a series of optimization techniques aimed at reducing the total number of logic nodes and the depth of the network [1]. It has been observed that using different DAG representations for distinct portions of a circuit can enhance the quality-of-results (QoR) compared to utilizing a single DAG type across the entire network. For example, majority-inverter graph (MIG) efficiently represents carry operators, leading to more significant depth reductions in arithmetic logic functions [2]. Similarly, XOR-majority graph (XMG) excels in maintaining self-duality and reducing area [3]. With the advancing technology node and increasing complexity of very large-scale integration (VLSI), modern circuit designs are often composed of various functional modules with distinct topological structures. This complexity necessitates the adoption of heterogeneous DAG logic optimization, which maps different components in a circuit to different DAG types, and renders logic optimization independently. Figure 1 illustrates the heterogeneous logic optimization process. However, most current logic optimization strategies are homogeneous, converting the entire Boolean network into a single DAG type for optimization, which often leads to suboptimal results.

ISPD '25, Austin, TX, USA

ACM ISBN 979-8-4007-1293-7/25/03
https://doi.org/10.1145/3698364.3705354

Several academic research studies have been conducted on heterogeneous logic optimization. Amarú *et al.* developed MixSyn, which identifies AND/OR and XOR-intensive components in circuits and conducts AND/OR optimization and XOR decomposition respectively [4]. However, their methodology is customized for AND/OR-XOR dominated circuits and is hard to extend to other gate types. LSOracle [5] divides the whole Boolean network into several sub-circuits using k-way partition, the Karnaugh map of each partition is then converted into a 2-D grid image, and a deep neural network (DNN) model is leveraged to predict the best-fit DAG type for each partition (sub-circuit). LSOracle then represents and optimizes these sub-circuits using the predicted DAG types, achieving improvements in both area and delay. However, LSOracle ignores the structural and functional characteristics of the circuit during partitioning. Consequently, the sub-circuits generated may consist of logic components with varying structures and functions, which challenges the framework's effectiveness in determining the optimal DAG type for each partition. Moreover, topological information of circuits is not utilized during best-fit DAG type prediction, leading to a loss in prediction accuracy. The latest version of LSOracle [6] adopts a heuristic approach to determine the best-fit DAG type for each sub-circuit, applying different DAG logic optimizers and selecting the one that minimizes the node-depth product. However, this approach introduces redundant optimization processes and increases the computational overhead.

Furthermore, we have made two noteworthy observations regarding heterogeneous logic optimization.

Observation 1. *When two sub-circuits exhibit similarities in their structures or functionalities, they often select the same DAG type for optimal optimization results.*

This is because the efficacy of different DAG types varies with specific Boolean functions and topological structures. Previous research work provides theoretical support for this observation: Amaru *et al.* [2] showed that MIG is particularly adept at representing carry operator structures, thus enhancing its effectiveness in optimizing arithmetic designs over other DAG types. For circuits with self-duality logic functions, XMG provides more compact representation and results in more area reduction than MIG and AIG [3]. For circuits with a large amount of XOR functions, Háleček *et al.* [7] demonstrated that XAG can lead to area reduction and even delay optimization compared with other DAG types. Consequently, circuits with similar functions or topologies generally opt for the same DAG type to maximize expressive power and optimization outcomes.

Observation 2. *If two interconnected sub-circuits are functionally or structurally similar, combining them into a single circuit and performing logic optimization often yields better results than optimizing each separately.*

The reason is that most of the multi-level optimization algorithms, such as rewrite and refactor, are cut-based. By merging two circuits, new cut choices are generated at their intersection, enlarging the solution space of logic optimization.

Motivated by the two observations above, we propose HeLO, a heterogeneous DAG-type logic optimization framework. Leveraging a customized GNN model to generate the structural-functional embedding and predict the best-fit DAG type for each sub-circuit, HeLO allocates structurally/functionally similar logic components into the same sub-circuit. Each sub-circuit is represented by the predicted best-fit DAG type and optimized correspondingly. In contrast to LSOracle, which partitions circuits from top to bottom, HeLO employs an agglomerative clustering approach that works in a bottom-up manner. This process begins with the creation of initial clusters based on PO-rooted fanin cones, ensuring that each cluster retains the complete functional and structural characteristics. By bottom-up clustering, structurally/functionally similar logic components are clustered together, while the functional and structural integrity of the newly formed clusters are maintained. This bottom-up clustering allows HeLO to accurately identify and assign one optimal DAG type that is best-fit for each component within a cluster, enhancing the effectiveness of heterogeneous logic optimization.

The major contributions of this work are summarized as follows:

- This paper proposes HeLO, a novel heterogeneous DAG-type logic optimization framework leveraging graph learning and hierarchical clustering.
- We propose a novel graph neural network (GNN) model to capture the global and local structural/functional characteristics of circuits. This model generates the topological-functional embedding and predicts the best-fit DAG type for each circuit.
- We propose an agglomerative clustering algorithm guided by graph learning. This algorithm aggregates logic components with similar structures and functionalities into the same sub-circuit.
- Experimental results demonstrate that HeLO can reduce the logic optimization node-depth product (NDP) by 8.7%, and reduce the area-delay product (ADP) after technology mapping by 6.9%, compared with LSOracle.

2 Preliminaries

2.1 Basics of Logic Synthesis

In the field of logic synthesis, a Boolean network is typically represented as a directed acyclic graph (DAG), where each node in the network corresponds to a Boolean variable, and the connections between nodes represent logical operations, such as AND, OR, NOT, NAND, NOR, XOR, and XNOR. The terms Boolean network and circuit are used interchangeably. In a Boolean network, Primary inputs (PIs) are nodes that do not have incoming edges (fanins), serving as the initial points for signals entering the network. Conversely, primary outputs (POs) are nodes without outgoing edges (fanouts), representing the endpoints where signals exit the network. A cut of a node n is a set of nodes that must be traversed to reach n from PIs. A cut is K-feasible if its size does not exceed a pre-defined number K. A transitive fanin (fanout) cone of node n is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node.

Current logic optimization strategies predominantly adopt a multi-level approach. For example, SOTA logic synthesis tools such as ABC and Mockturtle offer a suite of multi-level logic optimization algorithms, including rewrite, refactor, balance, etc [8]. The primary objective of logic optimization for a Boolean network is to reduce both the node count and the logic depth, which are fundamental metrics that directly impact the efficiency and performance of the

circuit. However, optimizing these metrics often involves a trade-off. Minimizing logic depth may require additional nodes to simplify operations, while reducing nodes can increase depth due to node merging. Therefore, node-depth product (NDP) is usually adopted as the metric of technology-independent logic optimization. Moreover, NDP serves as an indicative measurement for the area-delay product (ADP) after technology mapping.

2.2 Heterogeneous Logic Optimization

For existing logic synthesis tools, Mockturtle [9] supports logic optimization for four DAG types, namely, and-inverter graph (AIG), majority-inverter graph (MIG) [2], xor-majority graph (XMG) [10] and xor-and graph (XAG) [7]. Integrating Mockturtle as the logic synthesis library, LSOracle [5] has tailored specific logic optimization scripts for the four DAG types mentioned above. In this work, we also use these DAG types for heterogeneous logic optimization, and adopt the same optimization scripts as LSOracle for each DAG type during optimization.

3 Algorithm

This section first introduces the overall algorithmic flow of HeLO. Then, DAGOpt embedding space, a latent space representing the structural/functional characteristics of circuits, is defined. Finally, three major techniques utilized by HeLO are detailed: (1) A novel GNN model that captures the structural and functional information of Boolean networks at both local view and global view. Its two key tasks involve generating the DAGOpt embedding and determining the best-fit DAG type of a circuit. (2) A customized agglomerative clustering algorithm. This algorithm uses the pre-trained GNN model to generate the DAGOpt embedding of each sub-circuit, and iteratively merges structurally/functionally-similar sub-circuits. (3) A hetero-DAG optimization approach. This approach optimizes each sub-circuit referring to the predicted best-fit DAG type, and integrates all optimized sub-circuits into one circuit.

3.1 Overall Flow

Figure 2 shows the overall flow of HeLO. Given the original circuit, the fanin cone of each PO forms the initial cluster for agglomerative clustering. During each iteration of the agglomerative clustering, the customized GNN generates the structural-functional embedding of each cluster, and the two connected clusters with the most similar embedding will be combined into a new cluster. The process of hierarchical clustering stops when the minimal distance between any pair of connected clusters exceeds a predefined threshold. By the agglomerative clustering algorithm, logic components with similar structural and functional characteristics are combined into the same sub-circuit. Sub-circuits generated by the agglomerative clustering algorithm are fed to the process of hetero-DAG logic optimization, where we employ a pre-trained GNN model to predict the best-fit DAG type for each sub-circuit. Next, each sub-circuit is represented by its predicted DAG type and optimized by the corresponding DAG optimizer. Finally, all optimized clusters are integrated into one circuit. A global logic optimization operation (rewrite) is then applied to generate the optimized circuit. Note that small connected components, such as C_4 in Figure 2, are excluded from the process of clustering. The reason is that small connected components usually

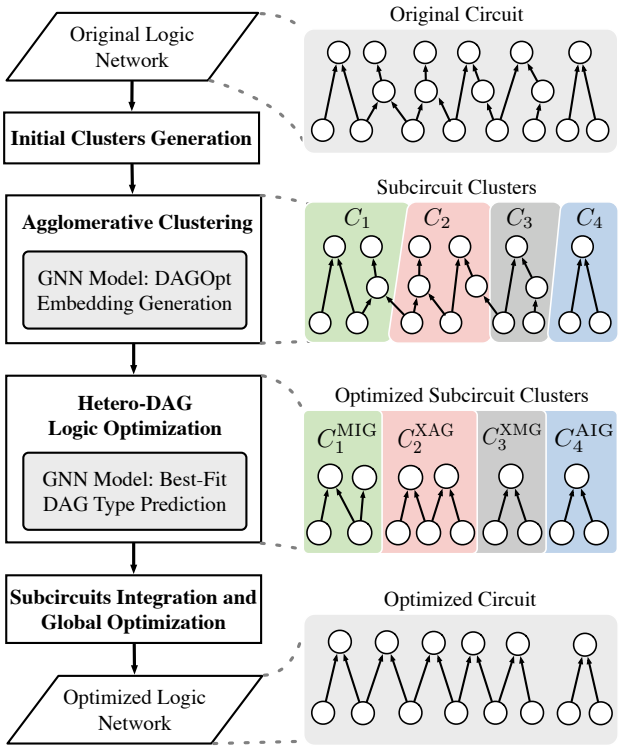

Figure 2: Overall Flow of HeLO.

possess small node numbers and simple topological structures, and have no connection with other components in the circuit. Therefore, small connected components can be optimized independently and there is no need to include them in the process of clustering.

3.2 DAGOpt Embedding Space

To quantitatively measure the structural and functional similarities across various circuits, we define the DAGOpt embedding space: For a circuit c, we represent it using four DAG types, namely, AIG, MIG, XMG and XAG. Each of these representations is optimized using the corresponding logic optimizer provided by LSOracle. After the logic optimization, the node-depth product (NDP) for each DAG representation of c is obtained. We compile these NDP values into a vector and normalize this vector by its magnitude to derive the DAGOpt embedding e^c.

$$e^c = \frac{(\text{NDP}^{\text{AIG}}, \text{NDP}^{\text{MIG}}, \text{NDP}^{\text{XMG}}, \text{NDP}^{\text{XAG}})}{\sum_{t \in \{\text{AIG, MIG, XMG, XAG}\}} \text{NDP}^t}. \tag{1}$$

According to Observation 1, circuits with similar structural or functional characteristics tend to choose similar DAG types for optimal outcomes. Therefore, the DAGOpt embeddings of structurally or functionally-similar circuits are expected to display similarity.

3.3 Customized Graph Neural Network

HeLO employs a pre-trained GNN model to predict the DAGOpt embedding of a circuit. This embedding acts as the cluster coordinate during the process of agglomerative clustering. Additionally, the GNN model enables the inference of the best-fit DAG type for a circuit based on its predicted DAGOpt embedding. To satisfy the

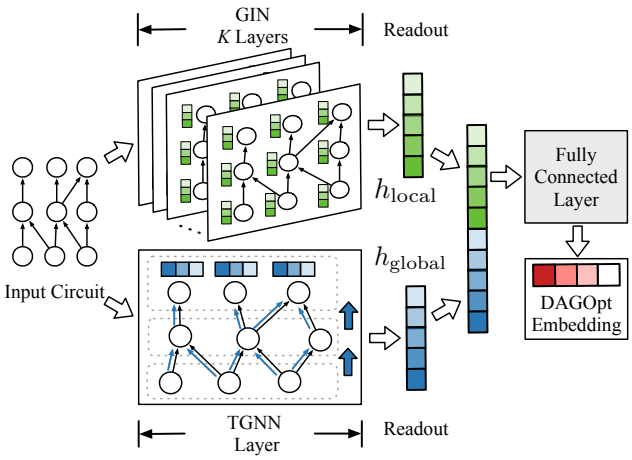

Figure 3: Overview of our Customized Graph Neural Network Model. Our model consists of two components: a graph isomorphism network (GIN) model to capture the local structural characteristics of the circuit, and a customized topological graph neural network (TGNN) to learn the structural and functional features of the circuit, at a global perspective.

above requirements, the GNN architecture is required to capture the structural and functional characteristics of the circuit. We develop a GNN architecture customized for this scenario.

Data Representation. In this work, the circuit design is initially mapped to AIG, which is then represented as a homogeneous graph. Each node in the homogeneous graph corresponds to one AND gate from the original AIG. To embed both the functional and structural information of the AIG circuit, we employ the feature engineering technique utilized in GAMORA [11]. Specifically, two features are selected for each node: node type (indicating whether the node is a Primary Input, Primary Output, or internal node) and input inversion (indicating whether the two input edges of the node are inverted or not). Additionally, we consider the number of fanouts for the node as another node feature to further explore the structural information of the circuit. By the feature of input inversion, the edge information is embedded into the node feature, making it possible to represent the circuit as a homogeneous graph and improve computational efficiency.

Graph Neural Network Model. Our GNN model, designed to capture the structural and functional characteristics of circuits from both global and local perspectives, comprises two parallel components: (1) a multi-layer graph isomorphism network (GIN) model to capture the local structural information and (2) a topological graph neural network (TGNN) to learn the global functional and structural information of a circuit through a topological messaging passing scheme. Figure 3 visualizes the architecture and workflow of our customized GNN model. The initial input, an AIG representing the circuit, is simultaneously processed by the GIN and TGNN models. These two models respectively yield local and global graph-level embeddings (denoted as h_{local} and h_{global}). These embeddings are then concatenated and fed into a Multilayer Perceptron (MLP) for generating the DAGOpt embedding. Since the DAGOpt embedding is a normalized vector summing to 1, the MLP utilizes a SoftMax

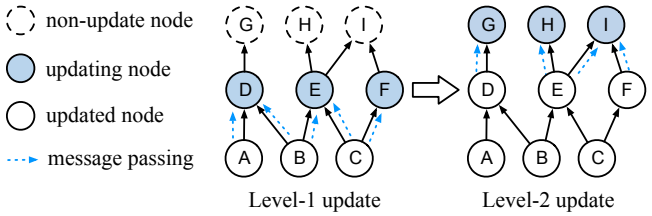

Figure 4: Illustration of the topological message passing scheme. The node embeddings of the circuit are updated in the topological order and level by level.

activation function. The generated DAGOpt embedding is used to determine the best-fit DAG type for the circuit. Specifically, the DAG type corresponding to the smallest-value entry in the embedding is selected as the predicted best-fit DAG type. Further details of each model component will be discussed in the remaining sub-section.

The graph isomorphism network (GIN) [12] is a type of neural network design to learn the structural information of graphs. The core principle of GIN is to enhance the representational power of graph neural networks (GNNs) to the extent of the Weisfeiler-Lehman graph isomorphism test [13], a classical algorithm for testing graph isomorphism. This is achieved by aggregating and updating node features in a manner that closely mimics this test. GIN has shown proficiency in mapping structurally similar sub-graphs to analogous hidden embeddings. This makes it apt for capturing the local structural characteristics of circuits in this work. The node embedding of node v at the k-th layer (denoted as $h_v^{(k)}$) is updated as:

$$h_v^{(k)} = \texttt{MLP}^{(k)}((1+\epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \text{Fanin}(v)} h_u^{(k-1)}), \quad (2)$$

where $\text{Fanin}(v)$ denotes the fanin nodes of node v, and $\epsilon^{(k)}$ denotes the learnable parameters of the k-th layer. By deploying a K-layer GIN model, we can obtain structural embeddings for k-hop subtrees originating from each node. These individual node embeddings are then aggregated into a single, unified structural representation, denoted as h_{local}, using a mean readout function:

$$h_{\text{local}} = \texttt{MEAN}(\{h_v : v \in V\}). \quad (3)$$

Building upon the topological message passing scheme introduced in prior graph learning research [14–18], we propose a TGNN architecture that facilitates message passing in topological order. Starting from the Primary Inputs of the graph, each node aggregates the embeddings of its fanin nodes to update its own representation. Notably, the embedding of a specific node v remains unchanged until the embeddings of all its fanin nodes have been updated. Figure 4 shows one example of the topological node embedding update: For the messaging passing at the first topological level, only the embeddings of node D, E and F are aggregated and updated. Then for the second iteration of message passing, the embeddings of G, H and I are updated.

For a node v, its aggregation at the k-th TGNN layer can be expressed as follows:

$$h_v^{(k)} = \texttt{MLP}(\texttt{Aggregate}(\{h_u^{(k)} : u \in \text{Fanins}(v)\}), h_v^{(k-1)}), \quad (4)$$

where the aggregated fanin embeddings of v are combined with the node embedding of v at $(k-1)$-th layer ($h_v^{(k-1)}$). The combined embedding is used to generate the updated node embedding $h_v^{(k)}$

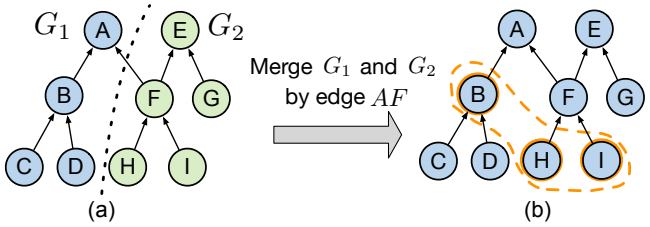

Figure 5: (a) Two independent AIGs G_1 and G_2 are connected by edge AF, the 3-feasible cuts of node A in G_1 are $\{B, F\}$ and $\{C, D, F\}$. (b). By merging G_1 and G_2 into an AIG, a new cut of node A, namely, $\{B, H, I\}$, is generated.

through an MLP layer. The topological message-passing scheme effectively emulates global-scale logic simulation, aggregating the functional and structural information of the circuit to the primary output (PO) nodes. We thus combine the embeddings of PO nodes to form the global graph embedding, denoted as h_{global}. Notably, in the training phase, the input circuit often contains small connected components, which are small in size and have no connection with other components in the circuit and are not necessary to be clustered. Including the PO node embeddings of small connected components into h_{global} could introduce irrelevant information and negatively impact the model's performance. Consequently, different from previous works which combine the node embeddings of all POs [19, 20], our approach excludes the PO embeddings of small connected components in the combination process using **mean** readout:

$$h_{\text{global}} = \texttt{Mean}(\{h_o^{(k)} : o \in \hat{\text{POs}}\}), \tag{5}$$

where $\hat{\text{POs}}$ denotes the set of POs in the circuit which excludes the PO(s) of all small connected component(s).

3.4 Agglomerative Clustering

For sub-circuit division, we propose a graph-learning-guided agglomerative clustering algorithm. This algorithm is designed to ensure that logic components sharing similar structural and functional characteristics are grouped into the same sub-circuit. The pseudo-code of the algorithm is detailed in Algorithm 1. In the initial step, named initial clusters generation (lines 2–4), the transitive fanin cone of each primary output (PO) in the Boolean network *BN* is treated as an initial cluster. This approach is based on two insights: (1) Multi-level logic optimization algorithms typically employ a cut-based method (e.g., rewrite, refactor). The fanin cone of a PO encompasses every potential cut for its nodes, thus preserving complete cut-based structural information for these nodes. (2) The logic function at each PO only depends on the logic outputs of all leave nodes in the PO fanin cone. Therefore, a PO fanin cone preserves the independent structural and functional information of the PO. Note that during the procedure of initial clusters generation, the PO-rooted fanin cones of the Boolean network are handled sequentially. If one node is already included in one initial cluster, it will be excluded from the following fanin cones. This setup guarantees that any pair of initial clusters have no overlap in their internal nodes, simplifying the later stages of hetero-DAG logic optimization and sub-circuits integration. Following initial clusters generation, each initial cluster is regarded as an independent sub-circuit with its own PIs, POs and internal nodes.

Algorithm 1 Graph-learning-guided Agglomerative Clustering

Input: Boolean network *BN*, Pre-trained GNN model $\mathcal{G}$, the threshold of maximal DAGOpt embedding distance δ
Output: Allocated sub-circuit clusters $\mathcal{S}$

1: $\mathcal{S} \leftarrow \emptyset$
2: **for** each PO-rooted transitive fanin cone $fc \subseteq BN$ **do**
3: $\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{fc\}$; ▷ initial clusters
4: **end for**
5: **for** each cluster $c \in \mathcal{S}$ **do**
6: $\quad$ Embed c into DAGOpt space by $\mathcal{G}$;
7: **end for**
8: Find nearest connected cluster pair (c_i, c_j); ▷ Defined in Eq (7)
9: $D_{\min} \leftarrow \|c_i, c_j\|$;
10: **while** $D_{\min} < \delta$ **do**
11: $\quad c' \leftarrow c_i \cup c_j$; ▷ Merge two clusters
12: $\quad$ Embed c' into DAGOpt space by $\mathcal{G}$;
13: $\quad \mathcal{S} \leftarrow (\mathcal{S} \cup c') \setminus \{c_i, c_j\}$; ▷ Update clustering
14: $\quad$ Find new nearest pair (c_i, c_j) and $D_{\min} \leftarrow \|\boldsymbol{c}_i, \boldsymbol{c}_j\|$;
15: **end while**
16: **return** $\mathcal{S}$

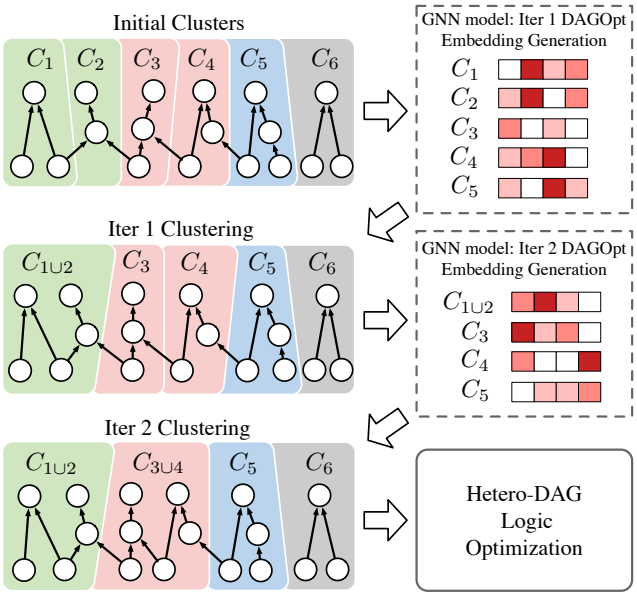

Figure 6: Illustration of the agglomerative clustering process. During each iteration of the agglomerative clustering, the customized GNN model is used to predict the DAGOpt embedding of each cluster. Then, the pair of connected clusters with the most similar DAGOpt embeddings will be combined to form a new cluster.

Referring to Observation 1, the DAGOpt embeddings of two structurally/functionally similar sub-circuits are also similar. We thus leverage the pre-trained GNN model introduced in Section 3.3 to generate the DAGOpt embedding of each cluster, and use the embedding as the cluster coordinate in the DAGOpt space (lines 5–7). After being projected to the DAGOpt space, sub-circuits with similar structural and functional characteristics are spatially closer.

Referring to Observation 2, merging two connected sub-circuits with similar structural or functional characteristics leads to larger solution space and better results for logic optimization, as exemplified in Figure 5. Therefore, during each iteration of the agglomerative clustering, we first find the pair of connected clusters c_i and c_j with the smallest distance $D_{\min}$ in the DAGOpt space (lines 8–9 and line 14). Before diving into the calculation of c_i, c_j and $D_{\min}$, we define $\hat{\mathcal{S}}$, denoting the collection of all interconnected pairs of clusters, as in Equation (6):

$$\hat{\mathcal{S}} = \{(c_a, c_b) | c_a, c_b \in \mathcal{S}, c_a \neq c_b, c_a \cap c_b \neq \emptyset, (c_b, c_a) \notin \hat{\mathcal{S}}\}, \quad (6)$$

where $\mathcal{S}$ denotes current set of clusters, and the condition $c_a \cap c_b \neq \emptyset$ ensures that each cluster pair in $\hat{\mathcal{S}}$ are interconnected. We obtain c_i and c_j and calculate $D_{\min}$ in Equation (7):

$$\begin{aligned} (c_i, c_j) &= \underset{(c_a, c_b) \in \hat{\mathcal{S}}}{\arg\min} \ \|\boldsymbol{c}_a, \boldsymbol{c}_b\|, \\ D_{\min} &= \|\boldsymbol{c}_i, \boldsymbol{c}_j\|, \end{aligned} \quad (7)$$

where $\|\boldsymbol{c}_i, \boldsymbol{c}_j\|$ denotes the distance between c_i and c_j in the DAGOpt space. Subsequently, c_i and c_j are merged into one new cluster, denoted as c' (line 12). The pre-trained GNN model $\mathcal{G}$ then generates the DAGOpt embedding of c' as its cluster coordinate (line 11). The clustering process terminates when $D_{\min}$ is less than a predefined threshold δ.

Figure 6 provides an illustrative example of agglomerative clustering. Given the initial clusters of the original circuit (6 initial clusters in total), two iterations of clustering are applied: In the first iteration, clusters C_1 and C_2, which are interconnected and have the closest embeddings in the DAGOpt space (indicating that C_1 and C_2 are structually/functionally similar), are merged into a new cluster $C_{1\cup2}$. Similarly, in the second iteration, the pair of clusters that are closest in the DAGOpt space, namely, C_3 and C_4, are merged to a new cluster $C_{3\cup4}$. Note that the small connected component in the example, namely, C_6, has a small node size and simple topological structure which can be optimized independently, and thus is excluded from clustering for computational overhead reduction.

3.5 Hetero-DAG Logic Optimization

Following the process of agglomerative clustering, the hetero-DAG logic optimization is applied to the generated sub-circuits. For each sub-circuit c, the pre-trained GNN model is employed to generate its DAGOpt embedding $e^c = \{V^{\text{AIG}}, V^{\text{MIG}}, V^{\text{XMG}}, V^{\text{XAG}}\}$. Then, the best-fit DAG type of c, denoted as t_c, is selected as the DAG type corresponding to the smallest value in e^c:

$$t_c = \underset{t \in \{\text{AIG, MIG, XMG, XAG}\}}{\arg\min} e_t^c, \quad (8)$$

where t represents a DAG type, e_t^c denotes the entry V^t in e^c, and t_c denotes the predicted best-fit DAG type for sub-circuit c. For example, if the DAGOpt embedding of a circuit c is predicted as $(V^{\text{AIG}} = 0.33, V^{\text{MIG}} = 0.12, V^{\text{XMG}} = 0.30, V^{\text{XAG}} = 0.25)$, then MIG is chosen as the best-fit DAG type of c. Next, each sub-circuit will be represented as the predicted DAG type and optimized by the corresponding optimization script. Then, all optimized sub-circuits are represented by MIG, the reason is that MIGs $\supset$ AIGs [2] ($\supset$ indicates that MIG is a superset of AIG, which implies that the capabilities and functionalities of AIG are fully encompassed within MIG), MIGs $\supset$ XMGs [10] and MIGs $\supset$ XAGs [7], and converting AIG/XMG/XAG to MIG does not modify the node count and depth. Finally, all converted sub-circuits are integrated into a single circuit. A rewrite operation is then applied to the integrated circuit to yield the globally optimized circuit.

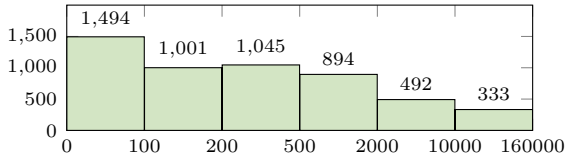

Figure 7: Distribution of node number in the collected dataset.

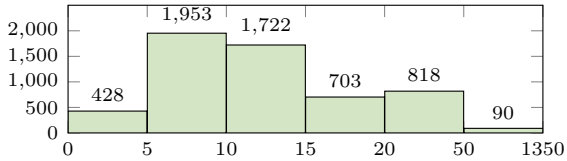

Figure 8: Distribution of depth for the circuits in the collected dataset.

4 Experimental Results

We develop the GNN model and the agglomerative clustering algorithm in Python with DGL and pytorch. The step of hetero-DAG logic optimization is implemented in C++ with Mockturtle [9], a C++-17 logic network library providing logic network implementations for AIG, MIG, etc. The whole flow is evaluated on a Linux machine with 16 Intel Xeon Gold 6226R cores (2.90GHz) and one NVIDIA A100 GPU with 40 GB of main memory.

In the remaining part of this section, we will first introduce our dataset preparation process. Then, the training setting and the model evaluation result will be demonstrated. Next, we will introduce the experimental setting for the heterogeneous logic synthesis flow, including the detailed settings and implementation details of HeLO and other baselines. Moreover, the experiments for logic optimization and technology mapping will be conducted. Finally, we render runtime analysis.

Dataset Preparation. We collected a dataset comprising 5714 sub-circuits from EPFL combinational arithmetic [21] and ISCAS'89 [22] benchmark suites, and OpenCores [23]. Each sub-circuit in the dataset is considered as an individual instance. For data labelling of an instance c, we employed four DAG-type logic optimizers (AIG, MIG, XMG, XAG) provided by LSOracle to optimize each instance, and obtained the resulting node-depth product (NDP) for each DAG type. These NDP values were then integrated to generate the DAGOpt embedding e^c, which was used as the ground truth label for each instance. The best-fit DAG type for a circuit c was determined as the one with the smallest value in its DAGOpt embedding e^c. In our dataset, the distributions of node numbers and depths of the sub-circuits are shown in Figure 7 and Figure 8 respectively, and the distribution of sub-circuits favoring AIG, MIG, XMG, and XAG as the best-fit DAG type is depicted in Figure 9.

Training and Evaluation. For the hyper-parameters of our GNN model, we use 3 GIN (Graph Isomorphism Network) layers and 1

Table 1: Technology-independent logic toptimization result. NDP denotes the product of node count and depth.

Circuit	Original			ABC (30*resyn)			Flowtune			Mockturtle			LSOracle			HeLO (ours)		
	#nodes	depth	NDP	#nodes	depth	NDP	#nodes	depth	NDP	#nodes	depth	NDP	#nodes	depth	NDP	#nodes	depth	NDP
pico-rv	18139	31	562309	15775	30	473250	14641	52	761332	20036	21	420756	18838	21	395598	19268	18	**346824**
chip_bridge	58789	31	1822459	57733	26	1501058	56377	23	1296671	59237	19	1125503	59538	19	1131222	58317	19	**1108023**
s38417	8568	28	239904	7842	24	188208	7730	22	170060	9016	18	162288	9028	18	162504	9522	16	**152352**
fpu	66522	33	2195226	58477	31	1812787	56731	33	1872123	66889	23	1538447	67248	22	1401752	68099	20	**1361980**
aes_core	21522	26	559572	19822	20	396440	19302	22	424644	20825	21	437325	21561	27	582147	21867	18	**393606**
des_perf	72720	16	1163520	72394	16	1158304	56593	16	**905488**	70176	15	1052640	70176	15	1052640	70176	15	1052640
ethernet	69763	41	2860283	66443	34	2259062	65684	31	2036204	70226	25	1755650	68482	24	1643568	71896	20	**1437920**
dyn_node	3926	27	106002	3620	24	86880	3596	22	79112	3979	19	**75601**	4191	21	88011	4034	18	72612
DMA	4295	20	85900	3450	17	58650	3301	19	**62719**	4348	15	65220	4208	17	71536	4342	15	65130
vga_lcd	105828	22	2328216	103583	21	2175243	103191	21	2167011	107657	16	**1722512**	108465	16	1735440	101534	17	1726078
fpga_bridge	318195	42	13364190	315998	37	11691926	301337	36	10848132	340217	26	8845642	325698	27	8793846	324356	24	**7784544**
i2c	1342	20	26840	1047	14	14658	1009	11	11099	1417	9	12753	1387	11	15257	1385	8	**11080**
mem_ctrl	46836	114	5339304	43608	104	4535232	36366	81	**2945646**	51762	69	3571578	52123	68	3544364	56592	61	3452112
normalize	1.000	1.000	1.000	0.967	0.882	0.860	0.911	0.863	0.769	1.037	0.656	0.678	1.018	0.678	0.673	1.019	0.596	**0.619**

Table 2: ASIC technology mapping result using the ASAP7 PDK. Area is in um^2 and delay is in ps. ADP denotes the product of delay and area.

Circuit	Original			ABC (30*resyn)			Flowtune			Mockturtle			LSOracle			HeLO (ours)		
	area	delay	ADP	area	delay	ADP	area	delay	ADP	area	delay	ADP	area	delay	ADP	area	delay	ADP
pico-rv	775.5	439.1	340492.0	779.2	434.5	338563.3	764.3	680.6	520193.6	841.5	312.9	263316.0	778.2	312.9	243503.0	831.6	290.0	**241153.0**
chip_bridge	3016.0	310.1	935394.0	3097.2	325.8	1008896.6	3010.2	308.3	928117.2	2988.2	294.2	879253.0	3038.9	294.2	894162.0	3028.2	263.9	**799010.0**
s38417	418.6	280.3	117319.0	415.4	304.4	126459.9	415.8	302.0	125572.7	415.7	283.0	117648.0	432.3	267.8	115776.0	416.2	266.7	**111016.0**
fpu	3115.4	466.5	1453351.0	3134.1	467.5	1465111.8	3101.4	523.3	1622869.6	3111.5	455.9	1418551.0	3062.2	452.2	1384682.0	3127.9	324.8	**1016029.0**
aes_core	1032.6	280.0	289164.0	1019.9	293.6	299428.1	941.5	320.3	301562.5	1033.1	277.5	286670.0	1001.5	291.2	291683.0	1061.0	251.0	**266276.0**
des_perf	3325.8	242.2	805408.0	3647.0	265.4	968020.6	3114.1	267.8	833899.1	3457.8	232.8	**804807.0**	3457.8	232.8	**804807.0**	3457.8	232.8	**804807.0**
ethernet	3476.7	384.6	1337324.0	3424.2	470.4	1610860.5	3495.8	465.4	1627101.5	3486.5	309.6	1079382.0	3366.8	306.5	1031939.0	3407.0	289.4	**985948.0**
dyn_node	204.4	293.2	59925.0	198.7	317.8	63156.4	203.9	286.7	58450.0	201.7	266.3	53703.0	212.0	251.5	53321.0	205.5	231.7	**47610.0**
DMA	182.6	196.9	35963.6	178.7	198.1	**35394.3**	179.1	212.4	38041.2	185.3	206.9	38350.3	186.1	196.7	36611.9	185.8	196.7	36556.3
vga_lcd	6125.5	300.0	1837402.0	5374.8	262.4	1410242.6	5491.7	306.7	1684097.0	5896.3	237.8	1401913.8	5751.7	237.8	**1367531.3**	5627.5	259.5	1460459.2
fpga_bridge	17049.5	584.6	9967166.9	16578.4	499.6	8281754.7	15978.7	580.2	9271167.1	17053.6	331.0	5644403.8	16760.3	356.9	5982606.9	16385.7	340.9	**5585048.8**
i2c	57.7	247.3	14278.5	50.4	230.0	11597.4	50.9	161.1	8201.7	59.5	131.7	**7840.7**	59.7	133.0	7940.7	60.2	131.7	7931.6
mem_ctrl	2282.5	1559.8	3560243.5	2165.5	1496.9	3241553.5	1850.2	1191.1	**2203815.5**	2333.1	1086.0	2533863.3	2340.3	1086.0	2541704.5	2395.2	1021.3	2446214.2
normalize	1.000	1.000	1.000	0.976	0.997	0.909	0.940	1.004	0.926	1.000	0.792	0.700	0.985	0.791	0.711	0.979	0.734	**0.665**

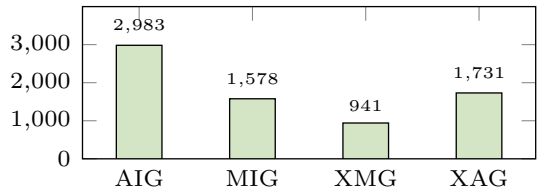

Figure 9: Distribution of best-fit DAG type selections for the collected dataset. Note that some circuits may have more than one best-fit DAG types when different logic optimizers yield identical optimization outcomes, thus the sum of the four columns is larger than the size of the dataset (5714).

TCNN (topologial graph neural network) layer. The hidden dimension of all the GNN layers and MLP is set to be 1024. The network architecture is shown in Figure 3. The dataset is divided into training and testing sets in an 85:15 ratio. We trained the model for 200 epochs with Adam optimizer, the batch size and learning rate are set to 10 and 1e-4, respectively. Regarding model evaluation, we evaluate the accuracy of best-fit DAG type prediction: After inferring the DAGOpt embedding e^c of a circuit c by pre-trained model, the DAG type corresponding to the smallest value in e^c is chosen as the best-fit DAG type of circuit c. For the test case with more than one best-fit DAG type, our prediction is considered accurate if it matches one of those best-fit DAG types. The overall accuracy of best-fit DAG type prediction by our trained model is 79.99%.

During the implementation of our proposed heterogeneous logic optimization framework, HeLO, the size threshold of small connected component is set to 300: any connected component within the tested circuit with fewer than 300 nodes bypasses the hierarchical clustering stage and is directly optimized using the predicted best-fit DAG type. For the evaluation benchmark selection of HeLO, we use the same benchmarks from the original version and latest version of LSOracle[1] [5, 6]. Additionally, we expand our benchmark set to include designs from the EPFL combinational random/control benchmark suite with node sizes exceeding 1000. This expansion, specifically including the *i2c* and *mem_ctrl* circuits, is based on the observation that there is no significant performance difference between homogeneous and heterogeneous logic optimization for circuits with small size. Consequently, we set the benchmark size threshold at 1000. Overall, our evaluation benchmark comprises 13

[1]The source of one design (*oc_aquarius*) is not available online, so we ignore this case.

circuits. The detailed design information of these designs are listed in the "Original" column of Table 1.

Experimental Setting. For baseline comparison, we compare HeLO against the latest version of LSOracle, two state-of-the-art (SOTA) logic synthesis tools (ABC [24] and Mockturtle [21]) and Flowtune [25], one automatic logic optimization exploration framework. For ABC, we conduct the **resyn2** script sequentially for 30 times for AIG optimization. To ensure a fair comparison, both HeLO and Mockturtle employ identical optimization scripts provided by LSOracle for each DAG type. The optimization strategies provided by LSOracle include the combinations of balance, rewrite, refactor, fraig, re-substitution, and so on. The details of the optimization scripts are available in the source code of LSOracle [2]. During the experiments, for Mockturtle, we apply the logic optimizer of each DAG type to the circuit separately, and select the lowest node-depth product as the optimal result. For LSOracle, which requires manual setting of the parameter **number of partitions**, we experiment with a range of partition numbers (from 2 to 10). Similarly, the minimal node-depth product is selected as the final result. For Flowtune, the hyper-parameters of repeat/iteration/sample/stage are set to 1, 3, 5 and 2, respectively. Since there is no option for node-depth co-optimization in Flowtune, we set the target metric to 0, which corresponds to the AIG node minimization mode.

Evaluation of Logic Optimization. Table 1 shows the technology-independent logic optimization result. Note that the output networks of Mockturtle, LSOracle and HeLO are MIGs, while ABC and Flowtune outputs AIGs. Given that MIGs $\supset$ AIGs (AND node can be converted to majority-of-three node by setting one input to be constant), we follow the experimental setting of LSOracle, where a direct comparison of node count and depth is conducted across ABC, Flowtune, Mockturtle, LSOracle and HeLO. As is shown in Table 1, compared with other logic synthesis tools, HeLO achieves the largest depth reduction. The resulted node-depth product (NDP) of HeLO is reduced by 38.9%, 24.3%, 9.6% and 8.7%, compared with the result of ABC (30*resyn), Flowtune, Mockturtle and LSOracle.

Evaluation of Technology Mapping. Although AIG node can be converted to MIG node by setting one input to be constant, the direct comparisons of node count and circuit depths between MIGs and AIGs may still be biased due to the unoptimized nature of the AIG-MIG conversion. To ensure a fair and consistent comparison between HeLO and the baselines, and to further evaluate the practicality of HeLO, we conduct ASIC technology mapping on logic-optimized networks in Table 1, using ASAP 7nm standard cell library [26]. For the detailed implementation, we first convert each logic-optimized network into the Verilog format, and then leverage ABC to employ technology mapping on them, using the same script across all cases. Table 2 shows the result after technology mapping: among all logic synthesis tools, HeLO achieves the smallest delay. In addition, the area-delay product (ADP) of HeLO is reduced by 36.6%/39.2%/5.2%/6.9% compared with ABC (30*resyn)/Flowtune/Mockturtle/LSOracle.

Runtime Analysis. The runtime of logic optimization for each logic synthesis tool and HeLO is listed in Table 3. ABC, which only provides AIG optimization, shows the least runtime but also the least effective optimization results. Compared with ABC, Flowtune conducts multi-armed bandit exploration for the combinations and orders of ABC operators, thus is much slower. Compared with the heterogeneous logic synthesis, ABC and Flowtune only conduct AIG optimization, and is faster than HeLO. With the increase of the circuit size, the runtime cost of Mockturtle for logic optimization increases non-linearly. Mockturtle is quite slow when applied on large Boolean networks. For example, it takes Mockturtle 72461 seconds to finish AIG-oriented optimization on the test case *fpga_bridge*, a large circuit with 318195 nodes. By dividing the whole network into sub-circuits for optimization separately, HeLO is 1.467× faster than Mockturtle. Moreover, HeLO is 1.735× faster than LSOracle. The reason is that LSOracle of the latest version employs the logic optimizers of 4 DAG types on each partition to determine the best-fit DAG type; Meanwhile, HeLO predicts the best-fit DAG type by the pre-trained model, and employs only one logic optimizer on each sub-circuit, thus significantly increasing its speed.

Table 3: Runtime analysis of ABC, Flowtune, Mockturtle, LSOracle and HeLO for logic optimization. The unit of the runtime is second (s).

Circuit	ABC (30*resyn)	Flowtune	Mockturtle	LSOracle	HeLO (ours)
pico-rv	15	324	29	113	57
chip_bridge	62	541	792	1616	262
s38417	7	81	3	56	30
fpu	86	665	28	633	480
aes_core	23	166	41	120	63
des_perf	121	530	93	680	70
ethernet	70	1007	1276	5106	803
dyn_node	3	20	23	18	12
DMA	3	24	2	28	59
vga_lcd	128	9174	3002	12460	3254
fpga_bridge	3032	9160	72461	70789	47611
i2c	1	12	7	11	21
mem_ctrl	56	876	61	414	337
Normalize.	0.068	0.426	1.467	1.735	1.000

5 Conclusion

This paper proposes HeLO, a heterogeneous DAG-type logic optimization framework leveraging hierarchical clustering and graph learning. Utilizing an agglomerative clustering algorithm, HeLO partitions Boolean networks into sub-circuits based on topological and functional characteristics. Then, a specialized GNN model is employed to predict the best-fit DAG types for sub-circuits. Each sub-circuit is represented in the predicted DAG type and optimized by the corresponding logic optimizer. Experimental results demonstrate that HeLO achieves improvements over LSOracle, reducing Node-Depth Product and Delay-Area Product (after technology mapping) by 9.0% and 12.8%, respectively.

Acknowledgment

This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14210723 and No. CUHK14211824), and the MIND project (MINDXZ202404).

[2] https://github.com/lnis-uofu/LSOracle

References

[1] L. Lavagno, I. L. Markov, G. Martin, and L. K. Scheffer, *Electronic design automation for IC implementation, circuit design, and process technology: circuit design, and process technology*, 2016.

[2] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 5, pp. 806–819, 2015.

[3] S. Rai, A. T. Calvino, H. Riener, G. De Micheli, and A. Kumar, "Utilizing xmg-based synthesis to preserve self-duality for rfet-based circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 3, pp. 914–927, 2022.

[4] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "MIXSyn: An efficient logic synthesis methodology for mixed XOR-AND/OR dominated circuits," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2013, pp. 133–138.

[5] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "LSOracle: A logic synthesis framework driven by artificial intelligence," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–6.

[6] M. Austin, S. Temple, W. L. Neto, L. Amarù, X. Tang, and P.-E. Gaillardon, "A scalable mixed synthesis framework for heterogeneous networks," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2020, pp. 670–673.

[7] I. Háleček, P. Fišer, and J. Schmidt, "Are xors in logic synthesis really necessary?" in *Proc. DDECS*, 2017, pp. 134–139.

[8] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2006, pp. 532–535.

[9] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage *et al.*, "The EPFL logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.

[10] W. Haaswijk, M. Soeken, L. Amarú, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic rewriting," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2017, pp. 151–156.

[11] N. Wu, Y. Li, C. Hao, S. Dai, C. Yu, and Y. Xie, "Gamora: Graph learning based symbolic reasoning for large-scale boolean networks," *arXiv preprint arXiv:2303.08256*, 2023.

[12] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[13] A. Leman and B. Weisfeiler, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsiya*, vol. 2, no. 9, pp. 12–16, 1968.

[14] M. Zhang, S. Jiang, Z. Cui, R. Garnett, and Y. Chen, "D-vae: A variational autoencoder for directed acyclic graphs," *Annual Conference on Neural Information Processing Systems (NIPS)*, vol. 32, 2019.

[15] V. Thost and J. Chen, "Directed acyclic graph neural networks," *arXiv preprint arXiv:2101.07965*, 2021.

[16] M. Li, S. Khan, Z. Shi, N. Wang, H. Yu, and Q. Xu, "Deepgate: Learning neural representations of logic gates," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 667–672.

[17] Z. Shi, H. Pan, S. Khan, M. Li, Y. Liu, J. Huang, H.-L. Zhen, M. Yuan, Z. Chu, and Q. Xu, "Deepgate2: Functionality-aware circuit representation learning," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[18] M. Li, Z. Shi, Q. Lai, S. Khan, S. Cai, and Q. Xu, "Deepsat: An eda-driven learning framework for sat," *arXiv preprint*, 2022.

[19] Z. Wang, Z. He, C. Bai, H. Yang, and B. Yu, "Efficient arithmetic block identification with graph learning and network-flow," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[20] Z. Wang, C. Bai, Z. He, G. Zhang, Q. Xu, T.-Y. Ho, B. Yu, and Y. Huang, "Functionality matters in netlist representation learning," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 61–66.

[21] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IEEE/ACM International Workshop on Logic Synthesis*, 2015.

[22] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1989, pp. 1929–1934.

[23] "Opencores," https://opencores.org/.

[24] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer-Aided Verification (CAV)*, 2010, pp. 24–40.

[25] W. L. Neto, Y. Li, P.-E. Gaillardon, and C. Yu, "Flowtune: End-to-end automatic logic optimization exploration via domain-specific multi-armed bandit," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[26] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.