# A Comprehensive Survey on Electronic Design Automation and Graph Neural Networks: Theory and Applications

DANIELA SÁNCHEZ, LORENZO SERVADEI, and GAMZE NAZ KIPRIT, Infineon Technologies AG and Technical University of Munich, Germany
ROBERT WILLE, Technical University of Munich, Germany
WOLFGANG ECKER, Infineon Technologies AG and Technical University of Munich, Germany

Driven by Moore's law, the chip design complexity is steadily increasing. Electronic Design Automation (EDA) has been able to cope with the challenging very large-scale integration process, assuring scalability, reliability, and proper time-to-market. However, EDA approaches are time and resource demanding, and they often do not guarantee optimal solutions. To alleviate these, Machine Learning (ML) has been incorporated into many stages of the design flow, such as in placement and routing. Many solutions employ Euclidean data and ML techniques without considering that many EDA objects are represented naturally as graphs. The trending Graph Neural Networks (GNNs) are an opportunity to solve EDA problems directly using graph structures for circuits, intermediate Register Transfer Levels, and netlists. In this article, we present a comprehensive review of the existing works linking the EDA flow for chip design and GNNs. We map those works to a design pipeline by defining graphs, tasks, and model types. Furthermore, we analyze their practical implications and outcomes. We conclude by summarizing challenges faced when applying GNNs within the EDA design flow.

CCS Concepts: • **Hardware** → **Methodologies for EDA**; • **Networks** → **Network design principles**; • **General and reference** → **Surveys and overviews;**

Additional Key Words and Phrases: Electronic Design Automation, very large-scale integration, machine learning, register-transfer level, Graph Neural Networks

**ACM Reference format:**
Daniela Sánchez, Lorenzo Servadei, Gamze Naz Kiprit, Robert Wille, and Wolfgang Ecker. 2023. A Comprehensive Survey on Electronic Design Automation and Graph Neural Networks: Theory and Applications. *ACM Trans. Des. Autom. Electron. Syst.* 28, 2, Article 15 (February 2023), 27 pages.
https://doi.org/10.1145/3543853

The work described herein is partly funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) as part of the research project ProgressivKI (19A21006C).
Authors' addresses: D. Sánchez, Infineon Technologies AG, Neubiberg, Bavaria, 85579 and Technical University of Munich, Chair of Electronic Design Automation, Munich, 80333, Germany; email: Daniela.sanchez-lopera@tum.de; L. Servadei and G. N. Kiprit, Infineon Technologies AG, Neubiberg, Bavaria, 85579 and Technical University of Munich, Munich, 80333, Germany; emails: {Lorenzo.servadei, Gamzenaz.kiprit}@tum.de; R. Wille, Technical University of Munich, Munich, Germany, 80333; email: robert.wille@tum.de; W. Ecker, Infineon Technologies AG, Munich, 85579 and Technical University of Munich, Chair of Electronic Design Automation, Munich, 80333, Germany; email: Wolfgang.ecker@tum.de.

1084-4309/2023/02-ART15 $15.00
https://doi.org/10.1145/3543853

## 1 INTRODUCTION

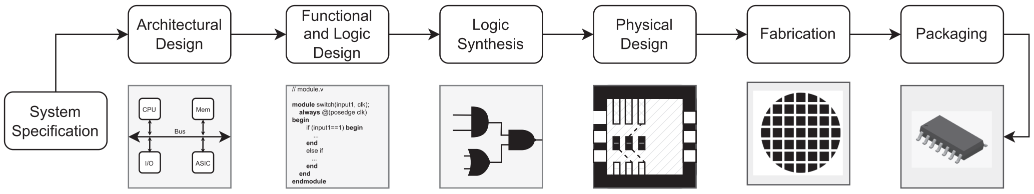
Over time, the chip design flow has incorporated multiple software tools to synthesize, simulate, test, and verify different electronic designs efficiently and reliably. The compendium of those tools is called Electroncie Design Automation (EDA). Those tools automatize the chip design flow sketched in Figure 1. Nevertheless, the flow is sequential and time demanding. Often, the design has to be verified and tested to ensure correctness, reliability, and objective closure. But only during physical verification and signoff, and testing, the quality of the design in terms of Power Performance and Area (PPA) can be measured. Corrective modifications in intermediate steps are often needed, and they result in multiple iterations of the design flow. Thus, estimations of Power, Performance, and Area (PPA) in earlier stages of the design would reduce the required iterations, increase the reliability of the design while going deeper on the flow, and, finally, improve the Quality of Results (QoR).

Over the past few years, design complexity driven by Moore's law has increased. Chip capacity has been doubling around every 2 years, which translates into increasing efforts for the design and verification of even more diversified chips. Electronic Design Automation (EDA) tools have aimed at coping with the new challenges and provided automated solutions for Very Large-scale Integration. EDA tools commonly face NP-complete problems, which Machine Learning (ML) methods could solve better and faster. Thus, Machine Learning (ML) has been integrated into EDA, especially to logic synthesis, placement, routing, testing, and verification [23]. In Reference [23], four main areas of action were recognized. First, ML is used to predict optimal configurations for traditional methods. Second, ML learns features of models and their performances to predict the behavior of unseen designs without running the costly step of synthesis. Moreover, design space exploration can be conducted by ML while optimizing PPA. Finally, Reinforcement Learning (RL) explores the design space, learns policies, and executes transformations to get optimal designs envisioning the future with an *"AI-assisted Design Flow."*

One enabling factor for using ML in EDA is the huge amount of data that is generated by the EDA tools along the design process. To apply ML over such data, these have to be pre-processed and labeled. Existing solutions use such data as Euclidean data, i.e., representing them in a two-dimensional (2D) Euclidean space, allowing the use of ML methods such as Convolutional Neural Networks (CNNs) as done in References [11, 53]. However, the trending neural network framework for graphs called Graph Neural Networks (GNNs) has shown a significant improvement in dealing with data whose structure is intuitively a graph. Even though GNNs appeared already in 2005, their recent combination with Deep Learning operations like convolution and pooling has drawn significant attention in fields such as molecular graphs [65], recommendation systems [64], and traffic prediction [66].

In EDA, the most natural representation of circuits, intermediate Register Transfer Levels (RTL), netlists, and layouts are graphs. Thus, over the past 2 years, few studies have recognized this opportunity and have incorporated the usage of GNNs to solve EDA problems.

This survey gives a comprehensive review of some recent studies using GNNs in different stages of the EDA flow. It first provides background on both fields and, successively, a list of the seminal related works. The rest of this survey is organized as follows: In Section 2, we briefly review the EDA flow and background concepts. In Section 3, we provide a more detailed explanation of the different types of GNNs. In Section 4, we explain the GNNs pipeline from a design perspective. In Section 5, we discuss different studies applying that pipeline to EDA tasks, and in Section 6 we summarize their main outcomes. Finally, Section 7 concludes by mentioning some open challenges and future directions.



```
// module.v
module switch(input1, clk);
  always @(posedge clk)
  begin
    if (input1==1) begin
    ...
    end
    else if
    ...
    end
  end
endmodule
```

Fig. 1. Chip design flow.

## 2 BACKGROUND AND DEFINITION

In this section, we briefly review background concepts related to EDA flow, graphs, and GNNs.

### 2.1 Electronic Design Automation

The progress in EDA tools and design methods, and the use of different levels of abstractions on the design flow have improved the hardware design productivity. Figure 1 sketches the stages of a modern chip design process. The flow starts with the chip specification modeling the desired application. The architecture analysis and prototype of the design represent the design as a collection of interacting modules such as processors, memories, and buses. In the functional and logical design, the behavioral description of those modules is mapped to RTL blocks using Hardware Description Languages (HDLs) such as Verilog. Nowadays, the transition from system specification to Register Transfer Level (RTL) can be done in different ways. For instance, using High-level Synthesis (HLS), which provides an automatic conversion from C/C++/System-C specifications to Hardware Description Language (HDL), or using hardware design frameworks such as MetaRTL [51] and Chisel [5].

Logic synthesis maps the RTL blocks in HDL to a combination of gates selected from a given technology library while optimizing the design for different objectives. Normally, this optimization involves a tradeoff between timing closure, and area and power consumption.

In physical synthesis, four main steps are executed: floorplanning, placement, clock insertion, and routing. First, the main RTL blocks of the chip, macros, and ports are assigned to regions of the layout. Second, the gates of the resulting logic netlists are placed to specific locations of the chip. Finally, the wires for clock signals and for connecting the gates are added. These steps are executed targeting better area utilization, timing performance, congestion, and routability while considering design rules.

Since errors in the design cost time and resources, verification is a fundamental step and is executed after functional and physical design. After verification and sign-off, the design goes through the manufacturing flow: fabrication, packaging, and final testing. Functional errors on the design and constraint violations result in iterations of the design flow, which could take days or weeks. The digital design flow applies the "Divide and Conquer" strategy, which has produced a compendium of tools and subteams working stand-alone. Therefore, iterations of the design flow are costly. For optimizing EDA flows, a DARPA IDEA program presented the OpenROAD[1] ("Foundations and Realization of Open Accessible Design") [3] and ALIGN[2] ("Analog Layout, Intelligently Generated from Netlists") [31] projects. OpenROAD and ALIGN are free, fully autonomous, open source toolchains for layout generation for digital and analog flows, respectively.

Even though the flow from RTL to Graphic Data System II (GDSII) is highly automated, it encounters some drawbacks: (1) It relies on the hardware designer's expertise to select proper config-

[1]https://github.com/The-OpenROAD-Project.
[2]https://github.com/ALIGN-analoglayout/ALIGN-public.

urations of EDA tools; (2) design space exploration at RTL, logic synthesis, and physical synthesis is manual and, thus, limited and time demanding; (3) corrections in the design would reinitialize the flow; and (4) there is no early analysis or predictability of the results. To face some of those challenges, the EDA community and projects, such as OpenROAD and ALIGN, are incorporating ML techniques into the design flow. For instance, to optimize design space exploration for timing-driven logic synthesis [3]. Moreover, OpenROAD is developing a data collection and storage infrastructure called METRICS 2.0 [20] to enable more ML applications for tool and flow outcomes prediction. However, ALIGN already includes ML methods to create high-quality layouts that meet the expectations of expert designers.

To the best of our knowledge, neither OpenROAD nor ALIGN has started exploring the use of GNNs. However, these two open source frameworks could be enhanced to collect featured graphs, which intuitively represent better EDA objects. Overcoming the data generation challenge would promote the exploration of GNNs in EDA. This exploration is also motivated by the fact that GNNs have been proven superior to other ML models thanks to the consideration of feature and topological information.

## 2.2 Graphs

*Definition.* A graph is a data structure for representing interactions between related objects. Mathematically, it is as a tuple $G = (V, E)$, where $V$ is the set of nodes and $E$, the set of edges. The edges are defined as the connection between nodes, e.g., for the nodes $u, v \in V$, the edge is represented as $e_{u,v} \in E$. The neighborhood of a node $v$ is defined as $N(v) = \{u \in V | (u, v) \in E\}$. If the node $v$ belongs to its neighborhood, then it is a closed neighborhood. Otherwise, it is an open one. A graph can be represented then as a list of nodes and edges. But a more convenient representation is through an *adjacency matrix* $\mathbf{A}$ defined as $\mathbf{A}_{u,v} = 1$ if $e_{u,v} \in E$; otherwise, $\mathbf{A}_{u,v} = 0$. The degree of a node is the number of nodes $D$ incident to a node $u$, mathematically it can be defined as $D_u = \sum_{v \in V} \mathbf{A}_{u,v}$ [17].

A graph with $n$ nodes and $m$ edges may have node and edge features of dimension $d$ and $c$ respectively, i.e., the node features are $\mathbf{X}$, where $\mathbf{X} \in \mathbb{R}^{n \times d}$ and the edge features $\mathbf{X}^e$, where $\mathbf{X}^e \in \mathbb{R}^{m \times c}$ [17].

*Types of Graphs.* Graphs are classified into different classes, as depicted in Figure 2. If the direction of the edges matter, then the graph is *directed* and the adjacency matrix $\mathbf{A}$ will be symmetric. If the edges do not have a direction, then the graph is *undirected.* In case the edges are represented with a cost or real value, the graph is called *weighted*, and $\mathbf{A}$ will have real values as entries. *Multiplex* graphs can be decomposed into layers, and the relation between each layer and the belonging nodes are additional *intra-layer* edges. Graphs can also be *homogeneous* or *heterogeneous.* In the former, all nodes and edges have the same type. In the latter, nodes have different types and their attributes may be of distinct types, too [17]. In *dynamic* graphs, the feature and topological information vary with time. In other cases, they are referred to as *static* graphs [72]. These categories of graphs are orthogonal, i.e., they can be combined.

GNNs architectures support different categories of input graphs such as heterogeneous [68], dynamic [40], and directed [24]. However, more complex types of graphs can be encountered in real-life applications. Current research aims to expand GNNs toward more graph types such as hypergraphs [70], where hyperedges connect to more than two vertices, and bipartite graphs [33], whose vertices form two disjoint and independent sets.

## 2.3 Shallow Embeddings Methods

The traditional approach for processing graph-structured data is to use shallow embedding methods. These aim to decompose the node information into low-dimensional embedding vectors that

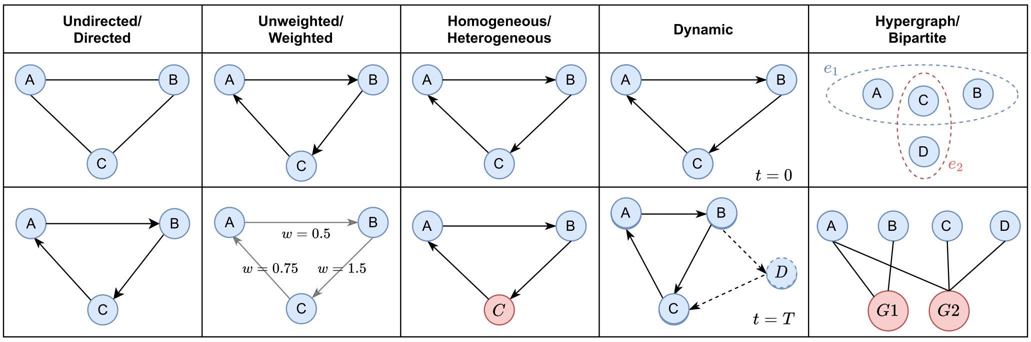


Fig. 2. Overview of different topologies of graphs. The colors of the nodes represent their type, i.e., the red and blue nodes are of different type. Note that all graphs in the figure are static, except for the dynamic graph in the fourth column. In the hypergraph example, $e_1$ and $e_2$ represent hyperedges.

consider the position of the nodes in the graph and the structure of the neighborhood [17]. One of the best-known graph embedding techniques is Random Walk [36]. In this technique, given a starting point within a graph, a random neighbor point is selected. As a second step, a neighbor of the randomly selected point is chosen again. This is done in a recursive fashion. This generates a random sequence of points, namely the random walk. DeepWalk [45] and Node2vec [16] are well-known graph embedding methods that are based on random walks. Although these methods have achieved groundbreaking success, they have some limitations [17]:

- Their encoders, which map important information from the graph to an embedding space, optimize a unique embedding vector per node. This can be computationally/statistically expensive in large graphs.
- They are *transductive*, i.e., they can generate embeddings only for nodes that were seen during training. This is a major drawback, as the model cannot generalize to unseen nodes.
- They do not consider node features that could provide precious information during the encoding

Similarly, other Neural Networks (NNs), such as Convolutional Neural Networks (CNNs), cannot operate directly on graph data without mapping their structure to a fixed-size vector format due to two main reasons. First, graphs do not have a fixed notion of locality or sliding window to perform the convolution operation. Second, a graph does not have a fixed order of nodes. Operations acting over graphs should be invariant to the node ordering or permutation.

## 2.4 Graph Neural Networks

To overcome the limitations of the shallow methods and deep Neural Networks (NNs), a novel NN called GNN was introduced in Reference [14].

GNNs are a framework for NNs that operate directly on graphs. In algebra, a permutation is an operation that changes the ordering of elements. Since graph-structured data are not assumed to be in any particular order, a network that depends on the order of nodes would give different results for two isomorphic graphs. Thus, GNNs consist of permutation-invariant but also permutation-equivariant functions, so it is possible to operate at node granularity too. Finally, GNNs can process the graph-structured data without losing topological and feature information [14]. Originally, GNNs were formulated as a type of Recurrent Neural Networks (RNNs) trained by a version of backpropagation through time [26].

Having an input graph, a GNN aims to learn the embedding vectors per node, defined as $\mathbf{h}_u$ $\forall u \in V$, which encodes the neighborhood information of each node [60]. GNNs use a message-passing strategy, as the node embedding or messages are passed through the neighbors. In this strategy, node pairs exchange vector-based messages, and a non-linear differentiable function is applied to update their node embeddings $\mathbf{h}_u, \forall u \in V$. The graph structure influences the message-passing updates, as each node aggregates messages from its local neighborhood.

The message-passing strategy has been inherited by novel architectures of GNN, which are described in the following section.

## 3 CLASSIFICATION OF GRAPH NEURAL NETWORKS

GNNs are divided into four types [60]. In this section, we review them and describe their core ideas.

### 3.1 Recurrent Graph Neural Networks

Recurrent Graph Neural Networks (RecGNNs) [60] process the node information recurrently by assuming that the nodes exchange information with their neighbors until a stable point is reached. Recurrent Graph Neural Networks (RecGNNs) define the node aggregation function as in Equation (1),

$$\mathbf{h}_u^t = \sum_{v \in N(u)} \phi\left(\mathbf{x}_u, \mathbf{x}^e_{(u,v)}, \mathbf{x}_v, \mathbf{h}_v^{(t-1)}\right), \tag{1}$$

where $\phi(\cdot)$ is a non-linear differentiable recurrent function and $\mathbf{h}_u^0$ is initialized randomly. Different from other GNN types, RecGNNs consider timesteps denoted as $t$. Thus, to obtain the hidden vector $\mathbf{h}_u^t$ of the node $u$ in the current timestep $t$, the feature vector of the node itself $\mathbf{x}_u$, the feature vector of its neighbors $\mathbf{x}_v$, and the feature vector of the linked edges $\mathbf{x}^e_{(u,v)}$ are aggregated with the hidden feature vector of the neighboring nodes in the previous timestep $t-1$, i.e., $\mathbf{h}_v^{(t-1)}$. This is recurrently done until a convergence criterion is satisfied. Note that $N(u)$ is an open neighborhood. Hence, RecGNNs do not consider the embedding vector of the node itself for the calculation of $h_u$ across the timesteps.

In Reference [50], the proposed architecture is an Recurrent Neural Network (RNN) where the connections between the neurons are classified into internal and external connections. While the former refers to the internal connections within units of the network, the external ones refer to the edges of the processed graph.

### 3.2 Convolutional Graph Neural Networks

Convolutional Graph Neural Networks (ConvGNNs) aim to learn embedding vectors by stacking multiple graph convolutional layers. Unlike RecGNNs where weights are shared, Convolutional Graph Neural Networks (ConvGNNs) use different weights per layer. ConvGNNs are a generalization of CNNs to graph data and are divided into spectral, and spatial approaches [60].

*Spectral Approaches.* In this case, graphs are assumed to be undirected. Since graph domains have no common coordinate system and no shift-invariance, convolution on graphs is defined using the convolution theorem from signal processing, which states that the Fourier transform diagonalizes the convolution operator, i.e., the convolution of two signals in the time domain is equal to the multiplication of these two signals in the frequency domain. Thus, spectral approaches are based on the Laplacian's eigen basis, which differs depending on the graph structure.

*Spatial Approaches.* These approaches are based on the spatial structure of the graph, i.e., they work on spatially close neighbors. CNNs are an example of such approaches. Considering the pixels of an image as nodes of a graph, then the CNN performs a weighted average of the pixel values of the selected central node and its neighbor nodes are calculated within each channel. Note that, unlike images, graphs have no regular neighbors nor fixed size.

In Reference [27], a spectral approach for performing the convolution operations on graph structures has been presented. This is named Graph Convolutional Networks (GCNs). The main principle of this approach is to extract the high-level embeddings of the nodes by aggregating the features of the central node and the neighbor nodes. Mathematically, this is expressed as in Equation (2),

$$\mathbf{h}_u^{(l+1)} = \phi\left(\left[\mathbf{W}^{(l)} \sum_{v \in N(u)} \frac{1}{c_{(u,v)}} \mathbf{h}_v^{(l)} + \mathbf{B}^{(\mathrm{l})} \mathbf{h}_u^{(l)}\right]\right), \tag{2}$$

where $c_{(u,v)} = \sqrt{|N(u)|}\sqrt{|N(v)|}$ is the normalization constant of the edge $e_{u,v}$, whereas $|N(u)|$ and $|N(v)|$ are the degree of the node $u$ and $v$, respectively, $\mathbf{W}^{(l)}$ and $\mathbf{B}^{(l)}$ are the trainable matrices that learn features of the neighbors and the node itself, respectively, $l \in \{1, \ldots, L\}$ with $L$ being the number of layers, and $\phi$ is the non-linear activation function [60]. In Equation (2), $N(u)$ is an open neighborhood, but Graph Convolutional Networks (GCNs) can also be implemented to consider a closed neighborhood. In that case, the second term of the summation involving the matrix **B** is not needed. Even though GCN are very powerful in generating low-dimensional embeddings of large graphs, they are *transductive*, i.e., all possible types of nodes have to be present during training. To alleviate this, GraphSAGE [18] proposes an inductive framework that generalizes to unseen nodes. It utilizes an update rule that is similar to Equation (2), by defining $c_{(u,v)} = |N(u)|$. GraphSAGE, instead of learning the node embeddings itself, learns the aggregation function $\sum_{v \in N(u)} \frac{\mathbf{h}_v^{(l)}}{|N(u)|}$ in Equation (3), which produces the node embeddings at the $l$th layer by concatenating the current node with the neighbor's embeddings. Here, || is the featurewise concatenation,

$$\mathbf{h}_u^{(l+1)} = \phi\left(\left[\mathbf{W}^{(l)} \sum_{v \in N(u)} \frac{\mathbf{h}_v^{(l)}}{|N(u)|} \;||\; \mathbf{B}^{(l)} \mathbf{h}_u^{(l)}\right]\right). \tag{3}$$

Especially, GraphSAGE is classified as GCN, as it uses a convolutional mean aggregator. In addition, GraphSAGE concatenates previous embeddings with neighbor embeddings, while GCN aggregates them. GraphSAGE uses a fixed-size neighborhood of the nodes, which can limit the network performance during inference. To solve this, Graph Attention Networks (GATs) [57] were introduced. A Graph Attention Network (GAT) computes each node embedding by going through all the neighbors using the self-attention mechanism presented in Reference [34]. Mathematically, this node embedding process can be expressed as in Equation (4),

$$\mathbf{h}_u^{(l+1)} = \phi\left(\sum_{v \in N(u)} \alpha_{(u,v)}^{(l)} \mathbf{z}_v^{(l)}\right), \tag{4}$$

where $\mathbf{z}_v^{(l)} = \mathbf{W}^{(l)} \mathbf{h}_v$ and $\alpha_{(u,v)}^{(l)}$ is the normalized attention score of the node $v$ to $u$ calculated by the $l$th attention mechanism. In Equation (4), $N(u)$ is considered as a closed neighborhood by default, but it can also be open. The main benefit of GAT is not only the consideration of the entire neighborhood of each node but also the increasing model's *expressiveness* that comes with the specification of different relevance scores of each edge for a given node. This attention score is computed per every edge. Thus, GAT could be computationally expensive. However, the self-attention layer for each edge can be embarrassingly parallelizable.

## 3.3 Graph Autoencoders

Graph Autoencoders (GAEs) [60] belong to the family of unsupervised frameworks, as training data have no ground-truth labels. Thus, the computed loss is dependent on the topological information of the entire graph, including node and edge features [58].

Graph Autoencoders (GAEs) are employed in two types of tasks: graph-based representation learning and graph generation. In both tasks, an encoder is first employed to calculate the corresponding embedding vectors $h_u$ for every node $u \in V$ by using ConvGNN or RecGNN layers. Note that the graph embeddings extracted by the encoder are low-dimensional vectors that hold node features yet retain the graph's topology information [60]. The embeddings provided by the encoder are used as inputs to the decoder. The decoder aims to reconstruct the adjacency matrix $\hat{\mathbf{A}}$ so that the distance to the original matrix $\mathbf{A}$ is minimized, as shown in Equation (5), where $\sigma(\cdot)$ is the logistic sigmoid function [60],

$$\hat{\mathbf{A}}_{u,v} = decoder(\mathbf{h}_u, \mathbf{h}_v) = \sigma\left(\mathbf{h}_u^T \mathbf{h}_v\right). \tag{5}$$

For representation learning tasks, the graph structural information is reconstructed by building a new adjacency matrix. In the case of graph generation, the process might involve a stepwise generation of the nodes and edges or output the entire graph at once.

In one of the most commonly used GAEs proposed in Reference [28], GCNs are used as encoders to extract graph embeddings. The decoder network aims to learn similarities between the original and the reconstructed adjacency matrix by applying a simple inner product. Moreover, they introduced the variational version of the GAE to learn the generative distribution of the data. This tackles the overfitting problem that can be caused due to the high model capacity.

## 3.4 Spatial-Temporal Graph Neural Networks

In many real-world applications, the structure of the input graph changes over time, i.e., the matrixes $\mathbf{A}$ and $\mathbf{X}$ change along the time axis. One example using temporal-spatial graphs is the action recognition problem [63] from the computer vision field. In this problem, the human joints, i.e., the inherent intra-body connections, represent the spatial graph connections. The edges linking the human joints across successive frames, i.e., the inter-frame edges, represent the temporal graph connections.

Based on the layers used to capture spatial and temporal relations, respectively, different Spatial-Temporal Graph Neural Networks (STGNNs) architectures are presented in the state of the art. In Reference [60], two main architectures are highlighted: CNN based and RNN based. The former uses a ConvGNN to capture the spatial relation and a 1D-CNN layer to capture the temporal dependency. Thus, the ConvGNN operates over $\mathbf{A}^t$ and $\mathbf{X}^t$, and the 1D convolutional kernels are sliced over $\mathbf{X}$ along the time axis. RNN-based Spatial-Temporal Graph Neural Networks (STGNNs) use two RecGNN layers to capture spatial and temporal relations, respectively. One RecGNN layer is applied to $\mathbf{X}^t$ and the other to $\mathbf{H}^{t-1}$. The outputs of both layers are aggregated to build the final embedding matrix $\mathbf{H}^t$.

STGNNs address a wide range of problems such as pandemic forecasting, activity recognition, and other time-series forecasting. For instance, a novel STGNN model is proposed in Reference [66] to solve the traffic forecasting problem. In this model, the spatial relation is captured by using graph convolutions, and the temporal relation is modeled by employing convolutions on the time axis.

## 4 GRAPH NEURAL NETWORKS PIPELINE

In Reference [72], the design pipeline for GNNs was introduced. They split the design flow into four main steps: input graph construction, graph type and scale definition, loss function selection,

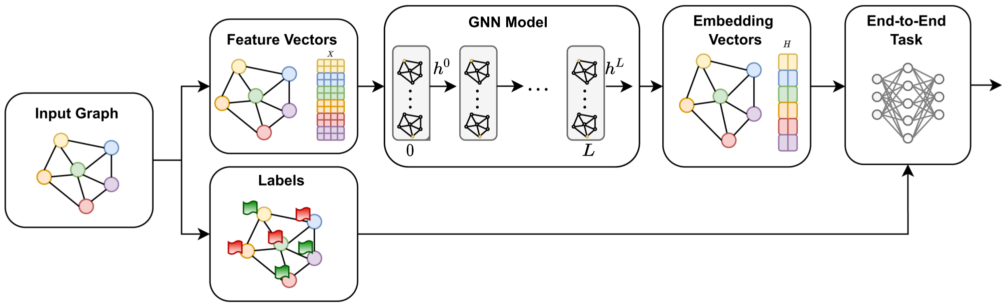


Fig. 3. Design pipeline using GNNs. The input graph definition includes high-dimensional feature vectors $X$, and labels, only in supervised and semi-supervised learning settings. The GNN model is built stacking $L$ layers of computational modules. The embedding vectors $H$ are used to solve an end task usually using NNs.

and GNN model construction. The graph embeddings learned by the GNN can be used as inputs to other ML models building an end-to-end framework shown in Figure 3. We summarize this flow in three main building blocks depicted in Figure 4.

## 4.1 Graph Definition

This step concerns how to map a task to a graph structure and to which graph category from the ones discussed in Section 2.2. In Reference [72], two types of task scenarios are defined: *structural* and *non-structural*. The latter refers to a task where data are not explicitly graphs and should be converted first. In this step, feature vectors per node or edges should be defined.

## 4.2 Task Definition

Based on the problem to solve, the granularity and supervision setting of the task is defined. There are three levels of tasks for a GNN: node, edge, and graph level. In the node-level tasks, the embedding vector of each node is used to predict features of individual nodes, classify or cluster them. At the edge level, the embedding vectors of pairs of nodes are compared to classify the node's connections or predict whether a link exists. Finally, in graph-level tasks, all node embedding vectors are aggregated to draw conclusions of the whole graph.

Based on the availability of the target labels in the end-to-end framework, we train the models in different manners: supervised, if all nodes are labeled; semi-supervised, if only some labels are known; and unsupervised, when no labels are available [60]. The target task and supervision setting together determine the loss function to be used during training. For instance, a node-level-supervised regression task requires the Mean Squared Error function as a loss function for all the nodes in the training set. In the case of a node-level semi-supervised classification task, the cross-entropy loss could be used for the few provided labeled nodes.

Moreover, GNNs can be *transductive* or *inductive*. Transductive GNNs learn the embedding vectors per node in the training graphs. Thus, during inference, they cannot generalize to unseen nodes. On the contrary, an inductive GNN learns the aggregation function that combines the node's neighborhood features rather than the embeddings themselves [60].

## 4.3 Model Definition

In Reference [72], a GNN model is defined as a compendium of stacked computation modules. They proposed three different types of modules to build GNNs models that can deal with large-scale graphs: propagation, sampling, and pooling.

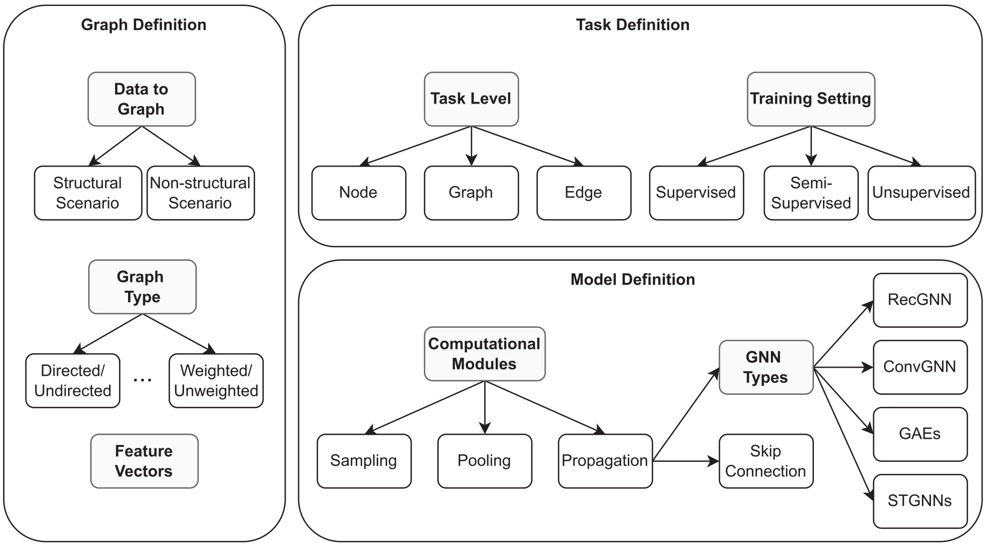


Fig. 4. Building blocks of applications using GNNs.

The propagation modules propagate and aggregate information between nodes preserving feature and topological information. The different types of GNNs layers presented in Section 3 are propagation modules. The *skip connection* mechanism motivated by Residual Neural Networks [21] is considered a propagation module as well. The skip connection module exploits the node embeddings of previous layers to alleviate the vanishing gradient problem caused by the propagation of information through deep GNNs. Examples of skip connection patterns are connections to all previous layers or a connection of all layers going only to the final layer [7].

In GNNs, the information of a node is aggregated to the information of its neighbors in the previous layer. Thus, deep GNNs result in an exponential growth of neighbors to be considered and aggregated. Sampling modules alleviate the neighbor explosion. The sampling granularity can be at the node, layer, and sub-sampling levels. At the node level, the neighborhood size is limited by considering a fixed number of neighbors per node. At the layer level, only a fixed number of nodes per layer is considered for aggregation. Finally, sub-graph level sampling limits the neighborhood search to the sampled sub-graphs [72].

Pooling modules, motivated by pooling layers in CNNs, reduce the size of a graph while getting more general features. There are two main classes of pooling modules in the literature: Directed pooling, also called readout or global pooling, which applies nodewise operations on node features to get a graph-level representation, and hierarchical pooling, which follows a hierarchical pattern and learns graph representations by layer [72].

## 5 PIPELINE APPLICATION TO EDA

These two seminal papers [25, 41] highlight the important link between EDA tasks and GNNs.

The study in Reference [41] is the first to recognize the high potential of GNNs in EDA. They stated that graph structures are the most intuitive way to represent Boolean functions, netlists, and layouts, which are the main focus of the EDA flow. They see GNNs as an opportunity for EDA to improve the Quality of Resultss (QoRs) and to replace the used traditional shallow methods or mathematical optimization techniques. The article lists related studies that have been applying analytical and heuristic approaches and shallow methods to EDA. Finally, they introduced

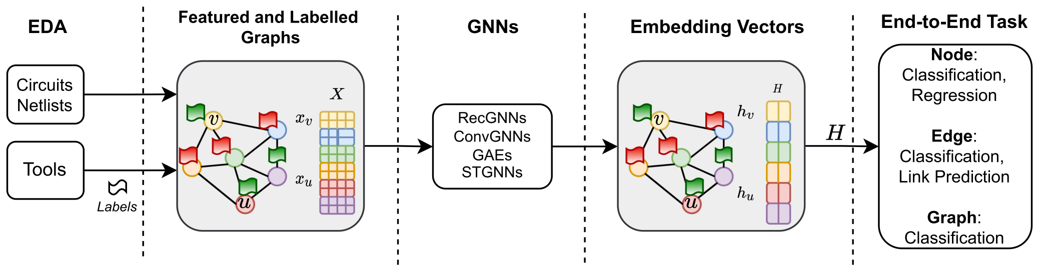


Fig. 5. End-to-End flow using EDA objects as graphs. Feature nodes $x$ and ground-truth labels (e.g., red and green flags) for supervised/semi-supervised training are collected using EDA tools. One of the GNNs flavors extracts node embeddings $h$, which are the inputs to other ML methods for classification or regression at the node, graph, or edge level.

*spectral-based* and *spatial-based* GNNs and presented only two cases of study: Test point insertion and timing model selection.

In Reference [25], a review of CNNs, as well as GNNs used in the EDA flow, was presented. They stated that ML could improve the QoRs during the chip design flow by predicting important metrics in different phases such as design space exploration, power analysis, physical design, and analog design. Similarly to Reference [41], they envisioned the use of Deep Reinforcement Learning (DRL) to solve combinatorial optimization problems in EDA, close to what is done in Reference [52].

In Reference [41] and Reference [25], the motivation for incorporating the trending GNNs into the EDA flow is clear. However, they did not exclusively center on both areas. Reference [41] focuses also on existing applications using traditional shallow methods. However, Reference [25] reviews the use of CNNs and GNNs. Moreover, Reference [25] compares the applications from an EDA perspective, without revealing the details of the GNN concepts behind.

Considering the drawbacks of the above-mentioned work, this survey gives a background and a review of recent important studies applying GNNs to the EDA field. To clarify the link between both areas, the review of these studies is organized according to their corresponding stages in the digital design flow. Table 1 lists the studies considered in this review. Extending the discussion in Reference [54], this survey considers more applications and analyzes their outcomes more in-depth.

The EDA pipeline is depicted in Figure 5. In general, EDA applications are a *structural* scenario where the data are intuitively graphs and no extra mapping is required. However, the definition of graph and feature vectors is a crucial step that requires knowledge expertise about the end-to-end task to be solved. However, feature vectors should contain features that relate to our end-to-end goal and that can be collected for all nodes from EDA tools and flow outcomes. The same applies to node labels in a supervised or semi-supervised setting.

We map the design pipeline described in Section 3 to EDA applications by giving details about graph types, feature vectors, and GNN architectures used.

## 5.1 Logic Synthesis

During logic synthesis, the RTL blocks describing the hardware design are mapped to logic cells from a technology library. This mapping must meet the timing constraints to operate at the desired clock rate while considering area and power. Therefore, synthesis is a complex optimization problem where ML can be applied. For instance, providing earlier QoR predictions to avoid multiple runs of the time-demanding synthesis step.

Table 1. Summary of GNNs for the Digital EDA Flow

| Section | GNN Model | Task Level | End-to-End Task | Reference |
|---|---|---|---|---|
| **Logic Synthesis** | GraphSAGE | Node Classification | Learning mapping patterns from HLS to FPGA blocks | D-SAGE [55] |
| **Verification and Signoff** | GCN | Node Regression | Vector-based average power estimation | GRANNITE [71] |
| **Floorplanning** | GCN | GNN-RL | Optimization as RL task | Edge-GNN [44] |
| **Placement** | GAT | Node Regression | Net length estimation | Net$^2$ [61] |
| | GraphSAGE | Node Clustering | Optimization of placement groups as guidance for tool | PL-GNN [38, 39] |
| | GraphSAGE | GNN-RL | Generalization PPA optimization as RL task | — [2] |
| | GraphSAGE | GNN-RL | Parameters optimization | — [1] |
| **Routing** | GAT | Node Regression | Routing congestion estimation | CongestionNet [29] |
| | GraphSAGE | Graph Regression | Post-route TNS prediction | — [37] |
| | GCN | GNN-RL | Joint placement and routing optimization | DeepPR [9] |
| **Testing** | GCN | Node Classification | Observation point candidates prediction | — [42] |
| **Reverse Engineering** | GAT | Node Classification | Extraction and labeling of components in flattened netlists | GNN-RE [4] |
| | ABGNN | Node Classification | Classification of IO boundaries of arithmetic blocks | ABGNN [22] |

To predict a more accurate delay for Field Programmable Gate Array (FPGA) blocks, Reference [55] proposes to learn the mapping and clustering patterns of arithmetic operations in Field Programmable Gate Arrays (FPGAs), especially Digital Signal Processor (DSP) and carry blocks. They recognized that current High-level Synthesis (HLS) solutions only sum up the individual delays of each component along the paths. This does not consider the underlying optimizations done during the synthesis. As a solution, they proposed a novel architecture D-SAGE [55], a GNN to predict the complex technology mapping done by logic synthesis. In Reference [55], designs are mapped to Data Flow Graphs (DFGs). The nodes are the set of operations (i.e., additions or multiplications), and the edges are the data dependencies between the nodes. Node types and bit widths of the data are considered as node attributes. Nodes and edges are labeled according to the end-to-end task. For instance, if the nodes are mapped to Digital Signal Processor (DSP) blocks or Lookup Tables, then they are labeled to one or zero, respectively. Similarly, edges are marked with one if their connected nodes are mapped to the same device. D-SAGE leverages GraphSAGE to support directed graphs and distinguish between successors $SU(u)$ and predecessors $PR(u)$ of a node $u$. To that end, Equation (2) is split over $SU(u)$ and $PR(u)$, and, finally, the successors and predecessors embeddings are concatenated as in Equation (6),

$$\mathbf{h}_u^{(l+1)} = \phi\left(\mathbf{h}_{u,PR}^{(l+1)},\ \mathbf{h}_{u,SU}^{(l+1)}\right). \tag{6}$$

Using the graph embeddings, D-SAGE solves two end-to-end tasks: Binary node classification, to predict which nodes are mapped to which device, and binary edge classification, to cluster nodes mapped into the same device. D-SAGE outperforms HLS tools in node classification, edge prediction, and also in operation delay estimation across all data paths.

## 5.2 Verification and Signoff

Verification is done to check the functionality of the design after functional, logic, and physical design. Especially before fabrication, the correctness of the design has to be assured. In the signoff step, a set of verification steps is executed to formally verify the functionality but also design closure, signal integrity, lifetime checks, and so on. The design closure is determined in terms of PPA. Negative results in this step translate into going backward on the flow and increasing the time-to-market of the chip. Therefore, early, accurate and fast estimations of those constraints could accelerate the design process.

For instance, the power integrity signoff requires power analysis of the design. For this, vector-based methods are preferred because of their accuracy, but they require gate-level simulations that are time demanding and replaced in practice by Switching Activity Estimators (SAEs). Switching Activity Estimators (SAEs) are fast but not accurate. In Reference [71], an alternative method based on GNN is proposed using toggle rates as inputs and improving the prediction accuracy. To that end, they built a graph based on the netlist, where single-output components are the nodes. The edges are defined as the connection between gates. From the RTL simulations, input and register toggle rates are taken as feature nodes, which are encoded in a 4D vector. Finally, the predicted toggle rates per gate are evaluated against the ground-truth labels obtained by the gate-level simulation. Intuitively, the toggle rates are expected to propagate from one level to the next one. Therefore, they proposed GRANNITE [71], a sequential and inductive version of a GCN, in which the node embeddings are not calculated in parallel but sequentially. Using the node embeddings, GRANNITE predicts average toggle rates from RTL simulations in a few seconds with more accuracy than classical SAEs.

## 5.3 Floorplanning

In chip floorplanning, the main and larger blocks of a netlist are placed onto 2D grids aiming for optimal PPA, while obeying design rules. This can be represented as a Markov process, which can be solved using RL.

The most significant work in this area is presented in Reference [44]. Google demonstrates the success of chip macro placement using a Deep Reinforcement Learning (DRL) framework for the floorplanning of Tensor Processing Unit accelerators. In Reference [44], a GNN is incorporated to the RL framework to encode the different states of the process, predict the reward labels for congestion, density, and wire length, and generalize to unseen netlist. The proposed architecture is called Edge-Based Graph Neural Network (Edge-GNN) [44], which calculates the node and edge embeddings for the whole netlist. This RL agent gives comparable or better results than a human designer but takes hours instead of months.

## 5.4 Placement

During placement, the design gates are mapped to the exact locations of the chip layout. The larger the design, the more complex this process is. A poor decision during placement can increase the chip area but also worsen the chip performance and even make it unsuitable for manufacturing if the wire length is higher than the available routing resources. Usually, multiple placement iterations are required, which is time-consuming and computationally inefficient. Therefore, placement is seen as a constrained optimization problem. ML and especially GNNs are being explored to ease this step [1, 2, 38, 39, 61].

A GAT called Net$^2$ is used to provide pre-placement net and path length estimations in Reference [61]. To that end, they converted the netlists to directed graphs, where nets represent nodes, and edges connect nets in both directions. The number of cells, fan-in, fan-out sizes, and areas are used as feature nodes. The edge features are defined using clustering and partitioning results. The ground-truth label for the nodes is the net length obtained as the half-perimeter wire length of the bounding box after placement. During inference, Net$^2$ predicts the net length per node, outperforming existing solutions. For instance, Net$^{2a}$, a version targeting accuracy, is 15% more accurate in identifying long nets and paths, and Net$^{2f}$ targeting runtime, is 1,000× faster.

In References [38, 39], GraphSAGE is leveraged to build PL-GNN, a framework helping placer tools to make beneficial decisions to accelerate and optimize the placement. PL-GNN considers netlists as directed hypergraphs and then transforms them to undirected clique-based graphs, where nodes and edges features are based on the hierarchy information and the affinity of the net with memory blocks. Hierarchical features are considered, as components in the same hierarchy have more connections among each other than with elements in other hierarchies, while the number of logic levels to memory macros provides an intuition about critical paths. A GNN is used to learn the node embeddings, which are clustered by the weighted $k$-means algorithm [10]. The resulting clusters are the optimal placement groups. Moreover, PL-GNN generalizes to any netlist as the node embeddings are learned by training an unsupervised loss function. The clustered placement groups can be used as placement guidance for any placer tool. This placement guidance leads to an improvement of 3.9% wire length, 2.8% power, and 85.7% performance compared to the default placement of a commercial tool.

A proof of concept framework mapping the PPA optimization task in EDA to an RL problem is presented in Reference [2]. This RL framework uses GraphSAGE with unsupervised training to learn node and edges embeddings that can generalize to unseen netlist. The GCN is a key component, because it extracts local and global information, which is needed by the RL agent. Wire length optimization during 2D placement is analyzed as a case of study.

In Reference [1], an autonomous RL agent finds optimal placement parameters in an inductive manner. Netlists are mapped as directed graphs, and nodes and edges features are handcrafted placement-related attributes. GraphSAGE learns the netlist embeddings and helps to generalize to new designs.

## 5.5 Routing

In this step, the placed components, gates, and clock signals are wired while following design rules (e.g., type of permitted angles). These rules determine the complexity of the routing, which is mostly an NP-hard or NP-complete problem. Thus, routing tools are mostly based on heuristics, and they do not aim to find an optimal solution. ML methods could enhance the routing process by providing earlier estimations, which can be used to adjust the placement accordingly and avoid high area and long wires.

In Reference [29], a GAT is used to predict routing congestion values using only a technology-specific gate-level netlist obtained after physical design. To that end, the netlists are built as undirected graphs, where each gate is a node, and the edges are defined as the connections between gates that are connected by a net. The feature nodes are 50D vectors containing information about cell types, sizes, pin counts, and logic descriptions. To get the ground-truth labels for the nodes, congestion maps are split into grids, and the congestion value of each grid is taken as a label for the cells that were placed into that grid. The architecture presented in Reference [29] called CongestionNet is not more than an eight-layer GAT following Equation (4). The node embeddings are used to predict local congestion values. Using GATs improves the quality of the predictions, and the inference time is around 19 seconds for circuits with more than one million cells.

In Reference [37], an end-to-end framework is proposed using GNN-based Long Short-Term Memory (LSTM) architectures to predict the port-route TNS. Their final goal is to predict whether a design will meet timing sign-off constraints but from earlier stages of the design flow. The resulted netlists from three stages: detailed placement, optimized placement, and Clock Tree Synthesis, are mapped to featured graphs. The features per node are task related such as the worst slack of a cell, worst output and input slews, and switching power of the driving net. These are collected from the technology library and generated reports. The netlists of each stage is encoded by a global GNN. In each of the three stages, the use of the graph embeddings is twofold: They are the inputs to a prediction model and to an Long Short-Term Memory (LSTM), where the graph embeddings are used as time series. The single prediction model predicts the TNS per stage, while the LSTM maps the synthesis flow to a sequential flow. The per-stage prediction achieves on average less than 12.6% normalized root-mean-squared error, while the GNN-based LSTM prediction achieves less than 5.2% over two test circuits.

In Reference [9], two RL frameworks are proposed to optimize only placement (DeepPlace) and placement and routing together (DeepPR). The gate-level netlist is mapped to a hypergraph, where the nodes are the gate cells, and the hyperedges are the nets connecting components. The policy network for both RL frameworks uses a global embedding from a CNN and a detailed node embedding coming from a GCN. The former uses the chip canvas as input mapped to a binary image. The latter uses the netlists as hypergraphs. Both global and detailed embeddings are concatenated to form the policy. DeepPlace aims to learn optimal placement of macros and standard cells, while DeepPR learns macro placement and routing.

## 5.6 Testing

Testing takes place only after the packaging of the design. The larger the design, the higher the complexity and the execution time of the testing tools. Moreover, testing should guarantee a high coverage, avoiding redundant test cases. Testing is not scalable, and it strongly relies on human expertise. To overcome those challenges, ML is being incorporated into the testing phase.

For instance, to reduce test complexity by providing optimal test points in a design. Reference [42] proposes a GCN to insert fewer optimal test points on the design while maximizing the fault coverage. To that end, a directed graph is built using the components of the netlist and primary ports as the nodes, and the wires between them as edges. The node features are 4D vectors containing information about the logic level of each gate, controllability, and observability attributes. Using Design-for-Test tools, the ground-truth labels are collected, and the nodes are labeled as “easy-to-observe” or “difficult-to-observe.” The proposed GCN model generates the node embeddings using Equation (2), replacing the aggregation function with a weighted sum to distinguish between predecessor and successor nodes.

Having the node embeddings, a binary node classification is performed. During inference, the nodes of new netlists are classified into “difficult” or “easy-to-observe.” This information is then used by testing tools to reduce the test complexity. Compared with commercial test tools, Reference [42] reduces the observation points by around 11% while keeping the same coverage.

## 5.7 Reverse Engineering

In EDA, Reverse Engineering (RE) is the process of getting the Intellectual Property or Integrated Circuit (IC) design, technology, or functionality by analyzing each layer of a chip. The outcomes of such analysis can be employed for verification, safety, and security purposes [43]. The *physical* Reverse Engineering (RE) flow obtains the gate-level netlist from the Integrated Circuit (IC) by de-packaging first or from the Graphic Data System II (GDSII) file by employing delayering, imaging, segmentation, and feature extraction techniques. However, the *functional* RE flow obtains the gate-level netlist directly from the GDSII file. Then, from the reversed-engineered netlist, high-level components are extracted and labeled [4].

GNN-RE [4] is a GNN applied to flattened netlists for functional RE. Specifically, the end-to-end tasks are sub-circuit extraction and classification. To that end, the reversed-engineered gate-level netlists are mapped to undirected graphs, where the nodes correspond to the gates and edges to the wires connecting them. The feature vectors per gate include the number of connections to primary input and output ports, the number of two-hops-away-connected gates by type, and input and output degrees. This extensive work compares different GNN types and computational modules such as GraphSAGE, GAT, sampling, and skip connection. The best results were achieved by using GAT with a sampling module called *GraphSAINT* [67]. The gate embeddings learned by GNN-RE are passed to a Fully-connected (FC) layer with a SoftMax function, which classifies each gate into a sub-circuit type with an average accuracy of 98.82%.

Similarly to GNN-RE, the Asynchronous Bidirectional GNN (ABGNN) [22] is applied to flattened gate-level netlists for sub-circuit classification but with a focus on arithmetic blocks. The netlists are mapped to Data Acyclic Graphs (DAGs), where the nodes are the circuit components and the edges, the interconnecting wires. The end-to-end tasks are two independent binary classifications for input and output boundaries of adder blocks, respectively. To identify the role of a gate, the nodes representing the inputs (fan-in cones) and the outputs (fan-out cones) of a gate are considered. Hence, the incoming edges to a node are not just directed but also bidirectional. Inspired by heterogeneous GNNs, ABGNN uses two GNNs to aggregate information for the fan-in and fan-out cones, respectively. The node embedding combination distinguishes between predecessors and successors, similar to Equation (6) for D-SAGE [55]. Finally, motivated by the acyclic property of netlists, ABGNN asynchronously propagates the node embeddings. The resulting node embeddings are the input to a Multi-layer Perceptron (MLP) with a binary cross-entropy loss.

## 5.8 Analog Design

The high complexity of the analog design flow is due to the large design space and the signal susceptibility w.r.t. noise. Thus, the analog flow could strongly benefit from modern approaches like ML to modernize the methods and improve the QoRs. Especially, GNNs are being applied to this field. Table 2 lists the studies briefly considered in this review.

In Reference [46], a GNN is used to predict net parasitic capacitance and device parameters. The design schematics are converted to hypergraphs and are the inputs to ParaGraph [46], a version of GNN that combines ideas from GraphSAGE and GAT. In Reference [35], ParaGraph is extended to predict parasitic resistance and coupling capacitance. To that end, the ParaGraph version using a GAT gets the node embeddings of source and target nodes. To predict the coupling capacitance and parasitic resistance between those nodes, their embeddings are the inputs to a regression network using bilinear layers. Furthermore, the predicted post-layout parasitics are used for Bayesian optimization.

CircuitGNN [69] uses a GCN to predict magnetic properties per node using resonator components as nodes and edges as their magnetic and electrical coupling. Circuit Designer [59] uses a

Table 2. Summary of GNNs for the Analog EDA Flow

| Section | GNN Model | Task Level | End-to-End Task | Reference |
|---|---|---|---|---|
| **Analog Design** | GraphSAGE GAT | Node Regression | Net parasitic capacitances prediction | ParaGraph [46] |
| | GAT | Node Regression | Net parasitic capacitances and resistances, and coupling capacitance prediction | Ext. ParaGraph [35] |
| | GNN | Graph Regression | Simulation electromagnetic properties of distributed circuits | CircuitGNN [69] |
| | GCN | GCN-RL | Transferring knowledge of transistor sizing | Circuit Designer [59] |
| | GAT | Node Regression | Prediction analog circuit performance due to placement | PEA [32] |
| | GCN | Node Classification | Analog structure recognition for constraint generation | – [47] |
| | GCN | Node Classification | Recognition of different blocks topologies in analog circuits | GANA [30] |
| | GraphSAGE | Node Classification | Recognition of symmetric node-pairs | – [13] |

GCN to extract the node embeddings of a circuit, which are later used as inputs to an RL agent targeting technology-independent transistor sizing. The analog circuit is mapped to a graph, where the nodes are transistors and the edges wires. In Reference [32], a new architecture called Pooling with Edge Attention (PEA) is introduced to evaluate how different placement solutions would affect the analog circuit performance.

In Reference [30], a GCN Based Automated Netlist Annotation (GANA) was presented as a strategy for block recognition. This is habitually performed based on human expertise, and thus, its automation is a fundamental part of an analog design flow with a no-human-in-the loop. GANA transforms flattened transistor-level netlists to bipartite graphs, where nets and elements are nodes, and edges connect only elements to their corresponding nets. Edge features tell how the element connects to its net, i.e., whether through the source, drain, or gate. Node embeddings are passed through one FC layer with a SoftMax function for classification. The output class identifies different variants of larger blocks, such as low-noise amplifiers, operational transconductance amplifiers, and mixers. GANA's outcome can be used to guide the circuit layout optimization.

In the same field as in Reference [30, 47] employs a GCN for primitives structure recognition. In Reference [47] flattened transistor-level netlists are mapped to undirected graphs, where feature nodes include properties such as device model and type, effective width, and channel length. Nets are labeled as power, ground, or signal. The connection type between a transistor to a net determines the net encoding vector, i.e., if the connection goes from the drain, source, or gate terminal to power, ground, or signal nets. In contrast to Reference [30], Reference [47] uses the $k$-means algorithm to find partial labels of the nodes and then, trains the GCN in a semi-supervised manner for predicting the class of the remaining unlabeled devices. The number of clusters $K$ is estimated using a regressor model. The outcomes of this pipeline: regressor model, $k$-means, and GCN are used within a constraint-driven design flow to apply different design rules to each identified structure.

In Reference [13], a GraphSAGE is used for automatic annotation of analog layout constraints. To that end, a GNN is used to classify whether pairs of nodes are symmetric, i.e., they have to be placed symmetrically in the layout. The analog netlist is mapped to a hypergraph, where the nodes are pins and devices such as transistors, diodes, capacitors, and resistors. The hyperedges are the connection between them. The feature vectors are one-hot encoded vectors giving information about the device or pin type. The generated node embeddings are the input to a binary classifier using a bilinear score function, in which pair nodes are classified as symmetric or asymmetric.

## 6 LESSONS LEARNED: APPLICATION RESULTS TO EDA

In this section, we summarize some lessons learned from the above-presented works.

### 6.1 GNNs Superiority

GNNs-based architectures outperform other models and solutions. The above-listed works compare their results against shallow ML, deep learning methods, or task-specific baselines. Beyond doubt, the superiority of GNNs is due to the consideration of topological and feature information, which comes at the price of higher efforts for building training datasets, and higher training times.

*6.1.1 Compared to Shallow ML Methods.* In Reference [42], the GNN-based model solution outperforms classical ML techniques for binary classification such as Linear Regressor (LR), Support Vector Machines (SVM), Random Forest, and Multi-layer Perceptron (MLP). Moreover, when compared against a commercial industrial tool, it achieves an 11% reduction on test points inserted and a 6% reduction on test pattern account without losing fault coverage. In GNN-RE [4], the GNN-based model is compared to an Support Vector Machines (SVM) for a supervised classification task using the same feature vectors per node. But, per definition, the SVM only considers features of single nodes and cannot access neighbor's node features. The training time for SVM was almost 10× less than for GNN-RE [4]. However, classification metrics clearly show the superiority of the GNN. ParaGraph [46] was compared to XGBoost [6] and Linear Regressor (LR) models, as well as with a vanilla GraphSAGE. The prediction accuracy was, on average 77%, 110% better than XGBoost, and 7% better than GraphSAGE.

*6.1.2 Compared to Deep ML Methods.* D-SAGE [55] is compared to an MLP and a vanilla GraphSAGE. The GNN-based models outperform the MLP emphasizing the relevance of structural information for operation mapping. Moreover, D-SAGE [55] resulted in a relative gain of 17% w.r.t. GraphSAGE thanks to the edge direction information. Similarly, the customized ABGNN [22] outperforms vanilla GraphSAGE, GAT, and the Graph Isomorphism Network [62] for both input and output boundaries classification tasks while reducing the inference time by 19.8% and 18.0%, respectively. ABGNN [22] performance is also higher and runtime lower compared to a NN, which uses a *Level-Dependent Decaying Sum-Existence Vector* to represent the circuit topology [11]. In Reference [49], Circuit Designer [59] is compared to a non-graph-based RL architecture. Different experiments show the superiority of GCN-RL, which converges in cases where a non-graph-based RL never does, even with a very high number of simulation steps. The success of Circuit Designer [59] is also attributed to the well-defined features vector considering linear transistor parameters. PEA's [32] superiority was proved against a vanilla GAT and a CNN-based analog performance model, which uses layout images.

*6.1.3 Compared to Task-Specific Baselines.* GNNs superiority is associated with two factors: well-defined initial features and graph learning ability [39]. Well-defined features capture underlying characteristics of the task and provide precious information for graph learning. GNNs capture feature and topological information. This represents a clear benefit against methods that only

Table 3. Types of Input Graphs used for GNNs Applications in the Digital Design Flow

| **Reference** | **Graph Type** | | | |
|---|---|---|---|---|
| | Directed | Undirected | Clique-based | Hypergraph |
| D-SAGE [55] | x | | | |
| GRANNITE [71] | | x | | |
| Edge-GNN [44] | | | | x |
| Net$^2$ [61] | x | | | x |
| PL-GNN [39] | | x | x | |
| – [1] | x | | x | |
| CongestionNet [29] | | x | | |
| – [37] | | x | | |
| – [42] | x | | | |
| GNN-RE [4] | | x | | |
| ABGNN [22] | x | | | |

consider graph connectivity. For instance, in PL-GNN [39], the hierarchy-related features are used to learn which nodes are similar, while the memory-affinity-related help to balance critical paths. PL-GNN [39] is superior to *modularity-based clustering*, which uses only connectivity information. CongestionNet [29] outperforms baseline methods for *a priori congestion estimation* by 29% when predicting congestion for all metal layers and 75% for lower ones. Nevertheless, CongestionNet [29] training required 60 hours on a single GPU. During inference, the runtime gets reduced to 19 seconds for 1.3 million cells, less than the baseline method's runtimes. Edge-GNN [44] is compared to results from the *open source, global placement tool RePLAce*[3] [8] and *manual placement* using an industry-standard EDA tool. After training over 10,000 chip floorplans, Edge-GNN [44] outperforms both methods, giving on average lower or comparable worst negative slack, total area, power, and congestion. Similarly to Edge-GNN [44], DeepPR [9] is around 4× faster than RePLAce while showing effectiveness in public benchmarks. Circuit-GNN [69] could deal with circuits of different sizes and topologies, reducing the simulation time by 4× compared with a *human expert using a simulation tool*. Compared to a *GPU-accelerated conventional probabilistic SAE*, GRANNITE [71] achieves on average 25% lower error at similar runtimes. Finally, GraphSAGE outperforms two baselines methods for symmetric constraint classification using *graph matching* and *signal flow analysis*. GraphSAGE provides a higher true-positive rate and a lower false-positive rate by keeping similar runtimes [13].

## 6.2 Graph Types

The input to all the reviewed applications is a netlist. However, the mapping of those netlists to graphs is different. The type of graph, feature vectors, and scale is defined depending on the task, the design flow (i.e., analog or digital flow), and the abstraction level. Tables 3 and 4 summarize the graph types used in the applications considered in this survey.

## 6.3 Models Depth

The number of layers of the GNN models is a hyperparameter. Adding too many layers over smooths the output embeddings and worsen the end-to-end task accuracy. Thus, deeper models need larger datasets [64].

[3]https://github.com/The-OpenROAD-Project/RePlAce.

Table 4. Types of Input Graphs Used for GNNs Applications in the Analog Design Flow

| Reference | Graph Type | | | | |
|---|---|---|---|---|---|
| | Directed | Undirected | Bipartite | Heterogeneous | Hypergraph |
| ParaGraph [35, 46] | x | | | x | |
| Circuit-GNN [69] | | x | | | |
| Circuit Designer [59] | | x | | | |
| PEA [32] | x | | | | |
| – [47] | x | | | | |
| GANA [30] | | x | x | | |
| – [13] | | | | | x |

In analog applications such as in GANA [30], the number of circuits is limited, and therefore, a two-layer GNN is used and trained with around 1,000 circuits during 2 hours. In GNN-RE [4], experiments sweeping the GNNs depth were conducted, and they reported an over-smoothing problem by increasing the GNN depth to four. For fewer layers, the performance increased at the cost of longer training times. In Net$^2$ [61], the number of layers is found with hyperparameter tuning. As a result, three GNN layers followed by two FC layers are used. In contrast, D-SAGE [55] uses two GNNs and three FC layers. Reference [37] uses two GNNs and two FC layers, for the prestage predictors and the LSTM network. PEA [32] uses four GAT and five FC layers. As the dataset is sufficiently large, ParaGraph [46] goes deeper and uses five GNN and four FC layers. In summary, the number of layers, as well as the filter size in ConvGNNs, are hyperparameters to be fine-tuned. To the best of our knowledge, the number of layers used was always between two and five.

## 6.4 Classical Machine Learning Techniques

Some lessons learned from non-graph ML models are also valid for GNNs.

For instance, GANA [30] uses batch normalization to guarantee all input features are in the same numerical range and dropout to avoid overfitting. Training data are split into training and validation. The latter is used to optimize hyperparameters such as learning rate, regularization, and decay rate. Finally, activation functions such as ReLU and tanh were also evaluated. In GNN-RE [4], a hyperparameter search for different GNNs structures was conducted. The best architecture employs dropout, ReLU as activation function, Adam as optimizer, and Random Walk as sampling module. In Net$^2$ [61], a hyperparameter search was conducted to find the learning rate and momentum factor. Bach normalization was applied after each GNN layer for better convergence and Stochastic Gradient Descent (SGD) was selected as optimizer. D-SAGE [55] also defines a learning rate, a weight decay factor, ReLU as activation function, and Stochastic Gradient Descent (SGD) as optimizer. Moreover, it uses a binary cross-entropy loss for the end-to-end binary classification task and an early stopping strategy on the training loss. Similarly, in Edge-GNN [44], an early stopping mechanism halts the RL training once the policy converges to avoid overfitting.

Other ML techniques are also used in conjunction with GNNs. For instance, in Reference [35], the graph embeddings are the inputs to a Bayesian optimizer leveraging dropout to predict the uncertainty of the predictions. In Reference [37], the graph embeddings are evaluated and visualized by using the t-Distributed Stochastic Neighboring Embedding [56] technique for dimension reduction.

## 6.5 Tools

The widespread use of ML models across different fields can be associated with available free and open source ML frameworks and datasets. Analogously, the exploration of GNNs to solve EDA tasks is encouraged by existing GNNs frameworks, EDA tools, and technologies.

Table 5. Summary of Target Technologies per Application

| Design Flow | Reference | Target Technology |
|---|---|---|
| **Digital** | D-SAGE [55] | *Virtex UltraScale+* |
| | GRANNITE [71] | FinFET 16 nm |
| | Edge-GNN [44] | 7 nm |
| | Net$^2$ [61] | *NanGate* 45 nm |
| | PL-GNN [39] | *TSMC* 28 nm |
| | — [1] | *TSMC* 28 nm |
| | — [37] | *TSMC* 28 nm |
| | — [42] | 12 nm |
| | GNN-RE [4] | *GlobalFoundries LPe* 65 nm |
| | ABGNN [22] | *SAED* 32/28 nm |
| **Analog** | ParaGraph [46] | Sub-10 nm |
| | Ext. ParaGraph [35] | FinFET-16 nm |
| | Circuit Designer [59] | 250 nm, 130 nm, 65 nm, 45 nm, 180 nm |
| | PEA [32] | FinFET *ASAP7* 7 nm |
| | — [47] | 120 nm, 40 nm |
| | GANA [30] | FinFET *ASAP7* 7 nm |

The most popular GNNs frameworks are PyTorch Geometric (PyG)[4] [12] for PyTorch, Spektral[5] [15] for TensorFlow and Keras and the Deep Graph Library (DGL)[6] [48] being framework-agnostic. PL-GNN [39], CongestionNet [29], Net$^2$ [61], and Reference [37] use PyG. While ParaGraph [46], its extension in Reference [35], GRANNITE [71], and ABGNN [22] use Deep Graph Library (DGL). In GNN-RE [4] and GANA [30], TensorFlow was used.

Regarding EDA tools, analog applications such as PEA [32], GANA [30], Circuit Designer [59] and in Reference [13] build datasets using SPICE. The latter also uses a dataset generated with the open source tool ALIGN. Digital applications such as Reference [1], GNN-RE [4], Net$^2$ [61], PL-GNN [39], ABGNN [22], and Reference [37] use Synopsis tools for synthesis such as Design Compiler and IC Compiler. Cadence Innovus is used for placement in Reference [1] and Net$^2$ [61] and RePLAce in Edge-GNN [44] for global-placement. D-SAGE [55] uses Vivado HLS, and Circuit-GNN [69] uses the commercial Electromagnetic simulator CST Studio Suite.

The different applications also target different technologies, as listed in Table 5. Other applications used commercial technologies and did not provide further details. Circuit Designer [59] uses five different technologies to evaluate the results of transfer learning.

## 7 CHALLENGES AND FUTURE RESEARCH

In the following, we present the open challenges for GNNs in general and GNNs in EDA. Finally, we point out some future work directions.

### 7.1 GNNs Challenges

In Reference [72], four main open problems of GNN were presented: robustness, interpretability, pretraining, and complex graph types. GNNs like other NNs, are vulnerable to adversarial attacks

[4]https://pytorch-geometric.readthedocs.io.
[5]https://graphneural.network/.
[6]https://www.dgl.ai/.

and act like a black box. To use GNNs in real-life applications, the security and explainability of the predictions should be guaranteed. However, the training process in GNNs is usually high demanding in terms of time and dataset size. To overcome these, pretraining approaches should be explored as for other NNs. Finally, real-time applications data usually map to complex graph structures such as hypergraphs and heterogeneous and dynamic graphs. These graph categories are orthogonal, and GNNs are expected to adapt to this graph diversity.

We also believe graph scalability is an open challenge for GNNs. The larger the graph, the higher the training time. We expect the use of GPUs and the mentioned computational methods such as sampling, pooling, and skip connections to be extended and alleviate this.

## 7.2 GNNs in EDA Challenges

From the EDA point of view, previous challenges intensify when applying GNNs to the design flows, especially scalability and diversity of graphs [41]. In EDA, the input graphs represent circuit netlists at different abstraction levels with, usually, very-large sizes. Large graphs result in huge, sparse adjacency matrixes and very large node lists, whose computations are time-consuming and computationally expensive. However, EDA objects are intuitively more similar to complex graph types than simple undirected or directed graphs. Intuitively, circuit netlists are directed hyper- or heterogeneous graphs as stated in Reference [39], Data Acyclic Graphs (DAGs) as stated in Reference [22] or Data Flow Graphs (DFGs) as in Reference [55]. To couple with those graph types, new GNN architectures should be formulated, as done in References [22, 55]. Customizing GNNs architectures to EDA objects and improving their performance to deal with complex graph types is still an open research topic.

In practice, we consider data generation is still an open challenge. The EDA toolchain is a compendium of stand-alone tools generating EDA objects, statistics, and reports. Data are available, but there is not a standard data collection infrastructure that could accelerate the data generation and graph mapping. Moreover, the results of EDA tools are used as ground-truth labels during data collection and labeling. As they run using fixed tool hyperparameters, this can lead to non-optimal, sparse, or biased data.

## 7.3 Future Work

We expect future works to address the aforementioned challenges. Based on the listed EDA applications, we also recognize three major directions:

*7.3.1 Exploiting Transfer Learning.* Applications such as CongestionNet [29] are not technology-agnostic, i.e., they build datasets with feature vectors and labels, depending on the technology used. For experiments targeting a new process technology, the dataset should be rebuilt and the model retrained. We expect that future EDA applications evaluate the use of transfer learning techniques for GNNs such as already done in References [19, 32, 44, 59]. For instance, GCN-RL Circuit Designer [59] benefits from the transferability of RL and exploits it for knowledge transfer between five different technology nodes and between topologies.

*7.3.2 Exploiting Feature Information.* As mentioned in Reference [39], a success factor of GNNs is the well-defined feature vectors. Therefore, we believe future research will evaluate different feature nodes to improve the scalability and generalization of current solutions to more advanced technologies and complex tasks. Exploring the impact of new features was also proposed as future work for the Circuit Designer [49, 59] and CongestionNet [29]. For instance, the current Circuit Designer implementation uses only linear transistor parameters as features. Future work should cover non-linear features to support non-linear component types. In Reference [29], an ablation study was conducted to identify the feature importance for the predictions of CongestionNet. As a result, features related to cell type or functionality were more important than features related to cell geometries. Thus, future work proposes to include new features such as pin and edges types. Additionally, they proposed to use directed or hypergraphs rather than undirected graphs. Net$^2$ [61] provides good results in net length estimation. As future work, the feature set should be extended to solve the more complex task of pre-placement timing analysis. Similarly, ParaGraph [46] used for net parasitic capacitance' prediction was extended to predict coupling capacitance and parasitic resistance in Reference [35]. This extended ParaGraph is expected to contribute to the optimization and automation of the analog design.

*7.3.3 Enlarging Datasets.* In some applications, the GNN-based solutions still do not reach complete accuracy. This could be solved using GNNs outcomes as inputs to further post-processing steps, as in GANA [30]. However, we expect future work to enlarge training data and features sets to improve the accuracy, as proposed in GRANNITE [71]. Larger datasets would enable deeper models, which could translate into better results. For the dataset generation, we expect more use of open source EDA tools and technologies. An open and standard infrastructure for data collection would minimize the graph building, mapping, and labeling efforts. Furthermore, it would empower research on GNNs and EDA and enable research results comparison and benchmarking.

## 8 CONCLUSION

To the best of our knowledge, our work is the first that collects seminal papers on the crossing between EDA and modern GNNs. In the presented works, GNNs outperformed the baseline methods. However, as the complexity of circuits in EDA continues growing, scalability and heterogeneity are still open challenges. We expect that the usage of GPUs helps to alleviate this bottleneck. We believe that the high potential of ML combined with GNNs will open the door to many more solutions targeting the EDA flow. We hope that using netlist, layouts, and intermediate RTL directly as graph structures can accelerate the earlier prediction of hardware metrics, and the usage of RL to solve combinatorial tasks in the EDA flow. Finally, we also expect that future work on GNNs targets some open-graph-related challenges such as heterogeneity, scalability, and diversity of graphs.

## REFERENCES

[1] Anthony Agnesina, Kyungwook Chang, and Sung Kyu Lim. 2020. VLSI placement parameter optimization using deep reinforcement learning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'20)*. IEEE, 1–9.

[2] Anthony Agnesina, Sai Surya Kiran Pentapati, and Sung Kyu Lim. 2020. A general framework for VLSI tool parameter optimization with deep reinforcement learning. In *Proceedings of the 34th Conference on Neural Information Processing Systems Workshop on Machine Learning for Systems (NeurIPS'20)*. 1–6.

[3] Tutu Ajayi and David Blaauw. 2019. OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain. In *Proceedings of Government Microcircuit Applications and Critical Technology Conference*. National Science Foundation, 1105–1110.

[4] Lilas Alrahis, Abhrajit Sengupta, Johann Knechtel, Satwik Patnaik, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. 2021. GNN-RE: Graph neural networks for reverse engineering of gate-level netlists. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 1–14. https://doi.org/10.1109/TCAD.2021.3110807

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*. Association for Computing Machinery, New York, NY, 1216–1225. https://doi.org/10.1145/2228360.2228584

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. Association for Computing Machinery, New York, NY, 785–794. https://doi.org/10.1145/2939672.2939785

[7] Tianlong Chen, Kaixiong Zhou, Keyu Duan, Wenqing Zheng, Peihao Wang, Xia Hu, and Zhangyang Wang. 2021. Bag of tricks for training deeper graph neural networks: A comprehensive benchmark study. arXiv:2108.10521 [cs.LG]. Retrieved from https://arxiv.org/abs/2108.10521.

[8] Chung-Kuan Cheng, Andrew B. Kahng, Ilgweon Kang, and Lutong Wang. 2019. RePlAce: Advancing solution quality and routability validation in global placement. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 38, 9 (2019), 1717–1730. https://doi.org/10.1109/TCAD.2018.2859220

[9] Ruoyu Cheng and Junchi Yan. 2021. On joint learning for solving placement and routing in chip design. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS'21)*. 1–12.

[10] Renato Cordeiro de Amorim and Boris Mirkin. 2012. Minkowski metric, feature weighting and anomalous cluster initializing in K-means clustering. *Pattern Recogn'* 45, 3 (2012), 1061–1075. https://doi.org/10.1016/j.patcog.2011.08.012

[11] Arash Fayyazi, Soheil Shababi, Pierluigi Nuzzo, Shahin Nazarian, and Massoud Pedram. 2019. Deep learning-based circuit recognition using sparse mapping and level-dependent decaying sum circuit representations. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'19)*. 638–641. https://doi.org/10.23919/DATE.2019.8715251

[12] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch geometric. arXiv:1903.02428. Retrieved from https://arxiv.org/abs/1903.02428.

[13] Xiaohan Gao, Chenhui Deng, Mingjie Liu, Zhiru Zhang, David Z. Pan, and Yibo Lin. 2021. Layout symmetry annotation for analog circuits with graph neural networks. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC'21)*. Association for Computing Machinery, New York, NY, 152–157. https://doi.org/10.1145/3394885.3431545

[14] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, Vol. 2. 729–734. https://doi.org/10.1109/IJCNN.2005.1555942

[15] Daniele Grattarola and Cesare Alippi. 2021. Graph neural networks in tensorflow and keras with spektral [Application Notes]. *Comput. Intell. Mag.* 16, 1 (2021), 99–106. https://doi.org/10.1109/MCI.2020.3039072

[16] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. Association for Computing Machinery, New York, NY, 855–864. https://doi.org/10.1145/2939672.2939754

[17] William L. Hamilton. 2020. *Graph Representation Learning*. Number 3. Morgan & Claypool, San Rafael, CA.

[18] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, 1025–1035.

[19] Xueting Han, Zhenhuan Huang, Bang An, and Jing Bai. 2021. Adaptive transfer learning on graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'21)*. Association for Computing Machinery, New York, NY, 565–574. https://doi.org/10.1145/3447548.3467450

[20] Sara Hashemi, C.-T. Ho, Andrew B. Kahng, H.-Y. Liu, and Sherief Reda. 2018. METRICS 2.0: A Machine-learning Based Optimization System for IC Design. In *Workshop on Open-Source EDA Technology*. Computer Science, 1–4.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[22] Zhuolun He, Ziyi Wang, Chen Bail, Haoyu Yang, and Bei Yu. 2021. Graph learning-based arithmetic block identification. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'21)*. 1–8. https://doi.org/10.1109/ICCAD51958.2021.9643581

[23] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, Xuefei Ning, Yuzhe Ma, Haoyu Yang, Bei Yu, Huazhong Yang, and Yu Wang. 2021. Machine learning for electronic design automation: A survey. *ACM Trans. Des. Autom. Electr. Syst.* 26, 5, Article 40 (2021), 46 pages. https://doi.org/10.1145/3451179

[24] Michael Kampffmeyer, Yinbo Chen, Xiaodan Liang, Hao Wang, Yujia Zhang, and Eric P. Xing. 2019. Rethinking knowledge graph propagation for zero-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'19)*. 11479–11488. https://doi.org/10.1109/CVPR.2019.01175

[25] Brucek Khailany, Haoxing Ren, Steve Dai, Saad Godil, Ben Keller, Robert Kirby, Alicia Klinefelter, Rangharajan Venkatesan, Yanqing Zhang, Bryan Catanzaro, and William J. Dally. 2020. Accelerating chip design with machine learning. *IEEE Micro* 40, 6 (2020), 23–32. https://doi.org/10.1109/MM.2020.3026231

[26] Thomas N. Kipf. 2020. *Deep Learning with Graph-Structured Representations*. Ph.D. Dissertation. University of Amsterdam.

[27] Thomas N. Kipf and Welling Max. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*. 1–14.

[28] Thomas N. Kipf and Max Welling. 2016. Variational graph auto-encoders. arXiv:1611.07308 [cs.CL]. Retrieved from https://arxiv.org/abs/1611.07308.

[29] Robert Kirby, Saad Godil, Rajarshi Roy, and Bryan Catanzaro. 2019. CongestionNet: Routing congestion prediction using deep graph neural networks. In *Proceedings of the IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC'19)*. 217–222. https://doi.org/10.1109/VLSI-SoC.2019.8920342

[30] Kishor Kunal, Tonmoy Dhar, Meghna Madhusudan, Jitesh Poojary, Arvind Sharma, Wenbin Xu, Steven Burns, Jiang Hu, Ramesh Harjani, and Sachin Sapatnekar. 2020. GANA: Graph convolutional network based automated netlist annotation for analog circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'20)*. IEEE, 55–60. https://doi.org/10.23919/DATE48585.2020.9116329

[31] Kishor Kunal, Meghna Madhusudan, Arvind K. Sharma, Wenbin Xu, Steven M. Burns, Ramesh Harjani, Jiang Hu, Desmond A. Kirkpatrick, and Sachin S. Sapatnekar. 2019. INVITED: ALIGN—Open-source analog layout automation from the ground up. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*. 1–4.

[32] Yaguang Li, Yishuang Lin, Meghna Madhusudan, Arvind Sharma, Wenbin Xu, Sachin S. Sapatnekar, Ramesh Harjani, and Jiang Hu. 2020. A customized graph neural network model for guiding analog IC placement. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*. Association for Computing Machinery, New York, NY, Article 135, 9 pages. https://doi.org/10.1145/3400302.3415624

[33] Zhao Li, Xin Shen, Yuhang Jiao, Xuming Pan, Pengcheng Zou, Xianling Meng, Chengwei Yao, and Jiajun Bu. 2020. Hierarchical bipartite graph neural networks: Towards large-scale E-commerce applications. In *Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20)*. 1677–1688. https://doi.org/10.1109/ICDE48307.2020.00149

[34] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. arXiv:1703.03130 [cs.CL]. Retrieved from https://arxiv.org/abs/1703.03130.

[35] Mingjie Liu, Walker J. Turner, George F. Kokai, Brucek Khailany, David Z. Pan, and Haoxing Ren. 2021. Parasitic-aware analog circuit sizing with graph neural networks and bayesian optimization. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'21)*. 1372–1377. https://doi.org/10.23919/DATE51398.2021.9474253

[36] László Lovász. 1993. Random walks on graphs: A survey. In *Combinatorics, Paul Erdõs is Eighty*, Bolyai Society Mathematical Studies, Vol. 2, 1–46.

[37] Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, and Sung Kyu Lim. 2021. Doomed run prediction in physical design by exploiting sequential flow and graph learning. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'21)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643435

[38] Yi-Chen Lu, Sai Pentapati, and Sung K. Lim. 2020. VLSI Placement Optimization using Graph Neural Networks. In *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS'20) Workshop on ML for Systems*. Computer Science, 1–5.

[39] Yi-Chen Lu, Sai Pentapati, and Sung K. Lim. 2021. The law of attraction: Affinity-aware placement optimization using graph neural networks. In *Proceedings of the International Symposium on Physical Design*. 7–14.

[40] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. *Streaming Graph Neural Networks*. Association for Computing Machinery, New York, NY, 719–728. https://doi.org/10.1145/3397271.3401092

[41] Yuzhe Ma, Zhuolun He, Wei Li, Lu Zhang, and Bei Yu. 2020. Understanding Graphs in EDA: From Shallow to Deep Learning. In *Proceedings of the International Symposium on Physical Design*. Association for Computing Machinery, New York, NY, 119–126. https://doi.org/10.1145/3372780.3378173

[42] Yuzhe Ma, Haoxing Ren, Brucek Khailany, Harbinder Sikka, Lijuan Luo, Karthikeyan Natarajan, and Bei Yu. 2019. High performance graph convolutional networks with applications in testability analysis. In *Proceedings of the 56th Annual Design Automation Conference (DAC'19)*. Association for Computing Machinery, New York, NY, 1–6. https://doi.org/10.1145/3316781.3317838

[43] Ian McLoughlin. 2011. Reverse engineering of embedded consumer electronic systems. In *Proceedings of the IEEE 15th International Symposium on Consumer Electronics (ISCE'11)*. IEEE, 352–356. https://doi.org/10.1109/ISCE.2011.5973848

[44] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. 2021. A graph placement methodology for fast chip design. *Nature* 594, 7862 (2021), 207–212. https://doi.org/10.1038/s41586-021-03544-w

[45] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. Association for Computing Machinery, New York, NY, 701–710. https://doi.org/10.1145/2623330.2623732

[46] Haoxing Ren, George F. Kokai, Walker J. Turner, and Ting-Sheng Ku. 2020. ParaGraph: Layout parasitics and device parameter prediction using graph neural networks. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC'20)*. IEEE.

[47] Husni Habal Rituj Patel and Venkat Reddy. 2021. Machine learning based structure recognition in analog schematics for constraints generation. In *Proceedings of the Design and Verification Conference and Exhibition Europe (DVCON'21)*. 1–8.

[48] Yu Rong, Tingyang Xu, Junzhou Huang, Wenbing Huang, Hong Cheng, Yao Ma, Yiqi Wang, Tyler Derr, Lingfei Wu, and Tengfei Ma. 2020. Deep graph learning: Foundations, advances and applications. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'20)*. Association for Computing Machinery, New York, NY, 3555–3556. https://doi.org/10.1145/3394486.3406474

[49] Kannan Sankaragomathi, Tony Cai, and Qingwei Zeng. 2021. *A Comparative Study of Reinforcement Learning Techniques for Analog Electronic Design Automation*. Technical Report. Stanford University CS229.

[50] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Trans. Neural Netw.* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[51] Johannes Schreiner, Rainer Findenigy, and Wolfgang Ecker. 2016. Design centric modeling of digital hardware. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'16)*. IEEE, 46–52. https://doi.org/10.1109/HLDVT.2016.7748254

[52] Lorenzo Servadei, Jin Hwa Lee, José A. Arjona Medina, Michael Werner, Sepp Hochreiter, Wolfgang Ecker, and Robert Wille. 2022. Deep reinforcement learning for optimization at early design stages. *IEEE Des. Test* (2022), 1–7. https://doi.org/10.1109/MDAT.2022.3145344

[53] Lorenzo Servadei, Edoardo Mosca, Elena Zennaro, Keerthikumara Devarajegowda, Michael Werner, Wolfgang Ecker, and Robert Wille. 2020. Accurate cost estimation of memory systems utilizing machine learning and solutions from computer vision for design automation. *IEEE Trans. Comput.* 69, 6 (2020), 856–867. https://doi.org/10.1109/TC.2020.2968888

[54] Daniela Sánchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Souvik Hazra, Robert Wille, and Wolfgang Ecker. 2021. A survey of graph neural networks for electronic design automation. In *Proceedings of the ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD'21)*. 1–6. https://doi.org/10.1109/MLCAD52597.2021.9531070

[55] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*. Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3400302.3415657

[56] Laurens van der Maaten and Geoffrey E. Hinton. 2008. Visualizing data using t-SNE. *J. Mach. Learn. Res.* 9 (2008), 2579–2605.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*. 1–12.

[58] Lilapati Waikhom and Ripon Patgiri. 2021. Graph neural networks: Methods, applications, and opportunities. arXiv:2108.10733. Retrieved from https://arxiv.org/abs/2108.10733.

[59] Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. 2020. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, 1–6.

[60] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[61] Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Yixiao Duan, and Yiran Chen. 2021. Net2: A graph attention network method customized for pre-p net length estimation. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC'21)*. Association for Computing Machinery, New York, NY, 671–677. https://doi.org/10.1145/3394885.3431562

[62] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? arXiv:1810.00826. Retrieved from https://arxiv.org/abs/1810.00826.

[63] Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial temporal graph convolutional networks for skeleton-based action recognition. arXiv:1801.07455. Retrieved from https://arxiv.org/abs/1801.07455.

[64] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. Association for Computing Machinery, New York, NY, 974–983. https://doi.org/10.1145/3219819.3219890

[65] Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. 2018. Graph convolutional policy network for goal-directed molecular graph generation. In *Advances in Neural Information Processing Systems (NIPS)*, Vol. 31. Curran Associates, Inc., Red Hook, NY, 6410–6421.

[66] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2018. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. International Joint Conferences on Artificial Intelligence Organization, 3634–3640. https://doi.org/10.24963/ijcai.2018/505

[67] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph sampling based inductive learning method. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*. 1–19.

[68] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'19)*. Association for Computing Machinery, New York, NY, 793–803. https://doi.org/10.1145/3292500.3330961

[69] Guo Zhang, Hao He, and Dina Katabi. 2019. Circuit-GNN: Graph neural networks for distributed circuit design. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 7364–7373.

[70] Ruochi Zhang, Yuesong Zou, and Jian Ma. 2020. Hyper-SAGNN: A self-attention based graph neural network for hypergraphs. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*. 1–7.

[71] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2020. GRANNITE: Graph neural network inference for transferable power estimation. In *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218643

[72] Jie Zhou, Ganqu Cui, Shengding Hu, et al. 2021. Graph neural networks: A review of methods and applications. arXiv:1812.08434 [cs.LG]. Retrieved from https://arxiv.org/abs/1812.08434.

Received 10 March 2022; revised 16 May 2022; accepted 27 May 2022